Version 3.3



Teamscale

The Official User Reference



Copyright 2016 CQSE GmbH

Published by CQSE GmbH

http://www.teamscale.com/

The style of this book is based on http://www.latextemplates.com/template/the-legrand-orange-book. The original layout has been modified.

Image Copyright D. Steidl (CQSE GmbH)

Version 3.3, April 2017



Contents

1 Introduction 11
1.1 Software Intelligence Vision 12
1.2 How (Not) To Control Quality 14
1.2.1 Why Many Fail To Improve 15
1.2.2 Improving Code Quality 16
1.2.3 Ensuring Test Quality 18
1.2.4 Managing Architecture Quality 21
1.2.5 Analyzing Feature Usage 22
1.3 Architectural Overview 23
1.4 Target Audience and Outline 24

1.5 Further Resources 25

2 Quality Analyses 27
2.1 Supported Technologies and Infrastructure 28
2.2 Internal Analyses 28
2.3 Integrated External Analyses 29

3 Common Use Cases 33
3.1 Configuring Projects and Analyses 34
3.1.1 Creating an Analysis Profile 34
3.1.2 Creating a Project Configuration 39
3.2 Getting Personalized Developer Feedback 45
3.2.1 Inspecting the Impact of a Single Change 45
3.2.2 Being Informed Without Leaving Your IDE 48
3.2.3 Push Notifications: Receiving Emails about Changes 48
3.3 Getting an Overview of the Quality Status 49
3.3.1 Creating a Single Project Overview 50
3.3.2 Creating a Multi Project Overview 53
3.4 Monitoring and Assessing Quality Evolution 54
3.4.1 Inspecting Recent Metric Trends 55
3.4.2 Finding the Root Cause for a Trend 57
3.4.3 Analyzing the Findings Churn 60
3.4.4 Performing a Baseline-Delta-Analysis 64
3.5 Identifying Untested Changes 66
3.5.1 Creating a *Test Gap analysis* Dashboard 67
3.5.2 Working with Test Gap Treemaps in Practice 71

4 User Reference 75
4.1 Web Interface 76
4.1.1 General UI Features 76

4.1.2 Dashboard Perspective 80
4.1.3 Activity Perspective 90
4.1.4 Findings Perspective 96
4.1.5 Metrics Perspective 104
4.1.6 Issue Perspective 117
4.1.7 Architecture Perspective 123
4.1.8 Delta Perspective 138
4.1.9 Project Perspective 145
4.2 IDE Integration 157
4.2.1 Eclipse 158
4.2.2 Visual Studio 160
4.2.3 IntelliJ IDEA 164
4.3 Extension and Integration APIs 169
4.3.1 Custom Check API 169

Appendix 175
A Dashboard Widgets 177
A.1 Common Widget Configuration Options 178
A.2 Assessment Pie Chart Widget 180
A.3 Assessment Trend Chart Widget 181
A.4 Code City Widget 182
A.5 Commit Chart Widget 183
A.6 External Content Widget 183
A.7 Findings Churn Widget 184
A.8 Findings Summary Bar Chart Widget 185
A.9 Findings Summary Table Widget 186
A.10 Folder Distribution Pie Chart Widget 187
A.11 Label Widget 188
A.12 Metric Bar Chart Widget 189
A.13 Metric Change Table Widget 190
A.14 Metric Distribution Pie Chart Widget 191

A.15 Metric Distribution Table Widget 193
A.16 Metric File Distribution Widget 194
A.17 Metric Hotspot Table Widget 195
A.18 Metrics Table Widget 196
A.19 Metrics Trend Chart Widget 198
A.20 Numeric Metric Value Widget 200
A.21 Ownership Treemap Widget 201
A.22 Single Metric Trend Widget 202
A.23 Task Summary Pie Chart Widget 203
A.24 Test Gap Overview Widget 204
A.25 Test Gap Treemap Widget 205
A.26 Test Gap Chart Widget 206
A.27 Treemap Widget 207
B Repository Connector Options 209
C Issue Tracker Connector Options 211
D Code Review Tool Connector Options 212
E Entity Selection Predicates 212
F Custom Checks 216
F.1 Custom Checks for ABAP 216
F.2 Custom Checks for C# 223
F.3 Custom Checks for C/C++ 227
F.4 Custom Checks for IEC 61131-3 ST 229
F.5 Custom Checks for Java 229
F.6 Custom Checks for JavaScript 238
F.7 Custom Checks for PL/SQL 239
F.8 Custom Checks for Python 240
F.9 Custom Checks for Xtend 241

Glossary 242

Bibliography 251
Books 251
Online 253



1. Introduction

Teamscale is your platform to ensure high-quality software. It is based on a revolutionary approach to static code analysis: *Teamscale* is a platform not only to collect some metrics, but to combine them with essential data from your development, to aggregate them and to provide solid decision criteria for the fundamental questions during your software development. *Teamscale* addresses your code quality, but also your test quality, your architecture and your usage scenarios. It, thus, supports you throughout the entire life-cycle of your software development.

1.1 Software Intelligence Vision

Ultimately, any software strives to satisfy their users to maximize profit. However, user needs evolve – they change, they expand. Additionally, not only the market situation changes, but also the underlying technology and hardware evolves. Developing software in such agile environments is a challenge: Being able to keep up is not trivial. And it heavily depends on the quality of the software.

To lead a software project to long-term success, software engineers are facing various decisions. From choosing the right underlying hardware and technology to prioritizing which features to implement, decisions have to be made daily. While facilitating continuous software change, managers and developers also have to consider the following: How long will our code base still be maintainable? Is our software sufficiently tested? Does our architecture satisfy current and future requirements? Those decisions are fundamental and have potentially long-term impact on the success (or failure) of the project.

For all these questions, intuition and often prominent gut-feelings are bad decision criteria. Instead, decisions should much rather be made based on solid data from the specific project. Yet, even though the data is technically available, it often remains unused because the data is implicit, hard to extract, hard to visualize or not trivial to interpret. For a profound decision making in software engineering, *Software Intelligence* is required: Just as business intelligence provides a "set of techniques and tools for the transformation of raw data into meaningful and useful information for business analysis purposes" [Wik], software intelligence targets the same purpose: software intelligence uses data mining to provide a solid foundation for decision making in software development.

Today, *Teamscale* is the only Software Intelligence Platform on the market. With *Teamscale*, you can steer the success of your project based on solid and profound data from your software: Integrating various sources such as code repositories, architecture documentations, test executions, issue trackers and runtime execution in production, *Teamscale* provides you transparent answers to the following questions:

- **Code Quality**: Is my code currently more spaghetti-code than clean code?
- **Test Quality**: Did we actually test what we changed or did we miss something?
- **Architecture Quality**: Does my code still match my architecture? Is the architecture suitable?
- **Feature Management**: Is all of my code in fact used by the user?

Being able to control these aspects makes your project more likely to succeed – and also more fun to work in. *Teamscale* will signal red flags early enough when tempting feature pressure deteriorates your software quality. Small quality deviations can be often redeemed quickly when developers discover them early. But at some point, consequences of failing to do so become unbearable. This is not something we just make up. We all have seen software that needed to be rewritten. Also research has shown that software quality – wrt. to code, test, and architecture – gradually decays over time if no effective counter measures are taken [Ede+13; Eic+01; FRJ09; Par94; Ste+14]. So do something before it is too late! And don't waste money for maintaining code that is not even executed by the user. Even though it seems as this code just sits there at no further cost,

do not underestimate the effort which this unused code creates for maintenance [Ede+12].

To make your project a success, *Teamscale* offers a broad range of prospects. It adequately addresses your needs and provides appropriate views of your software's quality regardless of whether you are a developer, a tester, an architect, a product owner or a manager. It adapts flexibly to your chosen development process and also supports feature-branch driven development. In contrast to many existing tools, it does not only analyze the main development trunk, but can be tailored to every single branch.

Further, *Teamscale* can handle common programming languages such as Java, Javascript, C or C++, but also knows how to deal with a bit more exotic languages such as ABAP, Ada, Delphi, Python or PL/SQL – just to mention a few. It also integrates with most common infrastructures such as Subversion, Git or TFS on the version control side oder Jira, Bugzilla and Redmine on the issue tracking side. For an up-to-date and complete list of *Teamscale*'s capabilities, please refer to our website www.teamscale.com.

1.2 How (Not) To Control Quality

Every developer knows the feeling when he looks at code and stunningly asks himself: "Who the hell wrote this code?" Or when he is getting desperate because he wonders: "Yesterday it still worked, didn't it"? It is a common phenomenon in software engineering that lots of code is written under time pressure and, hence, turns into quick 'n dirty written and badly tested code. Code that is very hard to understand and to maintain. In this section, we show you how to get in control of deteriorating quality and what helps to actually improve (as well was what does not).

1.2.1 Why Many Fail To Improve

At some point, many companies realize when gradually decreasing quality becomes an issue. In an attempt to prevent this decay, they install common static analysis tools – tools, which reveal, for example, bad code smells or measure test coverage. However, this is not enough. It is like signing up for the nearest gym and never going there to exercise. Static analyses are very useful to reveal problems in your code. Nevertheless, it is challenging to apply them successfully and to actually improve.

For long-lived software, static analysis tools commonly reveal thousands of problems [SE14]. With respect to code quality, these comprise, for example, badly structured pieces of code or redundant implementations. With respect to test quality, these might be files with very little test coverage. Often, developers look at these findings once or twice, try to fix a few and, ultimately, give up. The number of problems in grown software systems is usually too large to be fixed at once because quality assurance budget is almost always limited. (Dedicated large *clean-up* sessions would not be a good idea anyway because introducing more bugs is far too likely.)

Many existing static analysis tools operate in batch mode, i. e. they often only run once during a (nightly) build. When a developer performs a change, they do not provide feedback till the next build which often takes quite a while. This feedback cycle is simply too long. Who wants to go back to a previous task that seemed already finished? In addition, many existing analysis tools provide feedback only on the *entire* system. However, the developer needs personal feedback. In the moment, he cares only about the code he actually changed himself. Often, instead, he is flooded with all findings from the system and has no easy means to filter the ones relevant for him.

Improving software quality requires more than just measuring some metrics. It requires software intelligence. In the following, we show how to use *Teamscale* to improve code quality (Section 1.2.2), how to ensure test quality (Section 1.2.3) and how to keep your architecture quality (Section 1.2.4). Section 1.2.5 further indicates how *Teamscale* can be used for feature profiling and usage data analysis.

1.2.2 Improving Code Quality

When measuring code quality, there are many metrics out there pretending to provide you some insights. Many of them are useless. Choosing appropriate metrics for your domain requires quite some expertise, see Chapter 2. The default configuration of *Teamscale* offers you commonly used and actionable metrics with respect to the structure of your code (file sizes, method lengths and nesting depths), redundancy caused by copy&paste programming, documentation in term of source code comments, violations of coding conventions and bug patterns derived from data flow analysis. But it can be also easily extended using custom checks specifically tailored for your needs (see Section 4.3.1 and Appendix F).

After choosing appropriate metrics, improving code quality requires a realistic goal, a quality goal. Otherwise, your developers are more frustrated than before. Depending on your project's history, the goal can vary. If you just started to develop a new system on the greenfield, the overall goal of *zero findings* might still be feasible. However, for grown-software, we strongly recommend to accept the current quality status as it is and just start from there. Either choose the goal to keep the status quo and at least not make it worse (i. e. not introducing *new findings*). Or take it one step further and try to improve code that you are working on anyway (i. e. remove findings in code that is currently modified due to a feature change request).

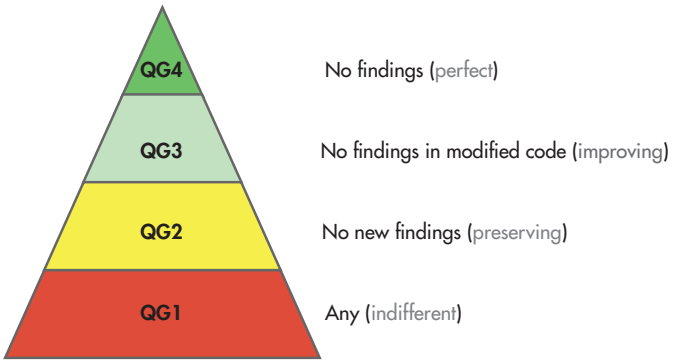


Figure 1.1. Quality Goals

Teamscale helps you to reinforce your quality goal. As many existing tools operate in batch mode, they analyze the entire system at once. While this is lengthy in time, it also omits the ability to provide accurate information which finding is *new* or in *modified* code. *Teamscale*, however, analyzes every single commit to your repository [HHS14]. Hence, it knows exactly when a specific finding was newly introduced to the system. Consequently, *Teamscale* can directly reveal violations of your quality goal, for example, if a developer just created a new finding or if a developer missed the chance to remove a finding in code you touched recently.

Most importantly, developers do not have to wait a day but can fix the problems immediately while they are still working on the subject: *Teamscale* provides the developer feedback within seconds after a commit. It specifically informs the developer about the impact of his change with respect to quality. Even personal notification services can be configured. Personal feedback is crucial to reduce the information flood and to focus on the essentials. It helps to focus only on new problems regardless of the past and to visualize small first steps of success.

In almost all software projects, feature pressure and time constraints are dominating every-day decisions. As a response, many teams have created the role of a *quality engineer* to ensure that quality improvements do not fall off the table. *Teamscale* also addresses the quality engineer's need to monitor the quality trend, to understand outliers of a specific metric and to report to management. Improving quality without management support is like Icarus striving for the sun. Hence, when management is providing resources, creating transparency about its usage is beneficial for both sides. Often, a clearly defined quality control process as exemplary depicted in Figure 1.2 increases satisfaction of all stakeholders and leads to long-term success [Ste+14].

In this process, software intelligence creates transparency and positive quality-awareness. It supports the developer in his daily work and makes the project manager aware of the current quality status and the usage of provided resources. Based on this level of information, he can steer his project more confidently without risking that feature pressure imposes uncontrollable impact on the code quality.

1.2.3 Ensuring Test Quality

In contrast to code quality, test quality is often directly visible to the user – namely, if the software does not work as it should. For long-lived software, most bugs are introduced by changes to existing code or by writing new code. Hence, test managers focus on testing what has changed recently. Nevertheless, untested changes are often released, causing field bugs and potential hot-fixes. Our study has shown that approximately half of the changes are not tested once before being released [Ede+13]. Why is that?

Software errors can have very different root causes. That's why there is usually no single test level, but a pyramid of tests ranging from low-

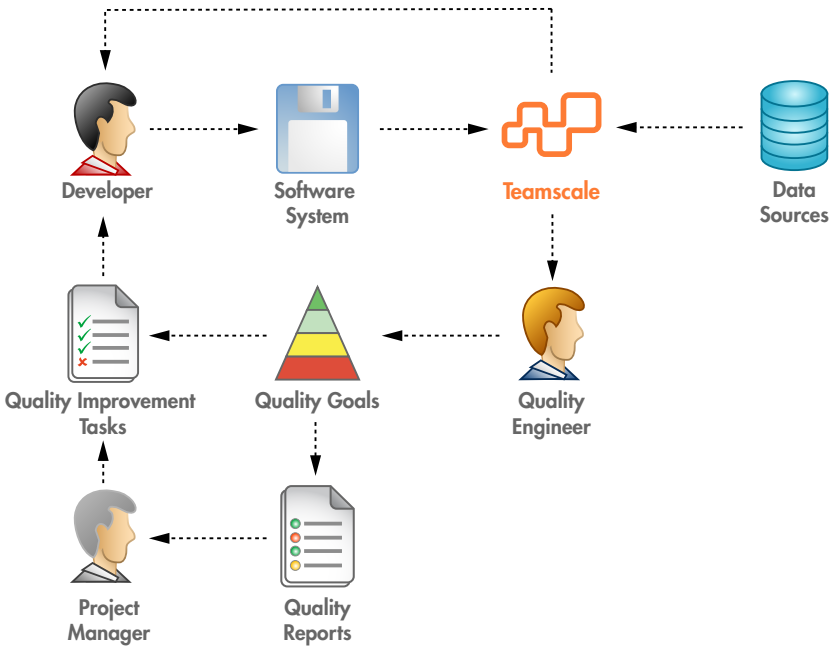


Figure 1.2. Code Quality Control

level unit tests to integration tests and high-level system tests. Each type of test targets to find different types of bugs. Unit tests are often written by the developers themselves. They usually know what changed and what unit tests cover. But testers responsible for integration or system testing often do not know if their test runs cover the actual code changes: Testers are usually driven by change requests resulting from adapting requirements. But the information which code regions were in fact affected by a designated change request is usually not available for them.

Teamscale closes this gap by combining information that is already there: Being connected to the version control system, *Teamscale* knows exactly which parts of the code changed since the last test run. As it also integrates with test coverage tools, *Teamscale* also knows which code was executed during the last test. Combining both sources,

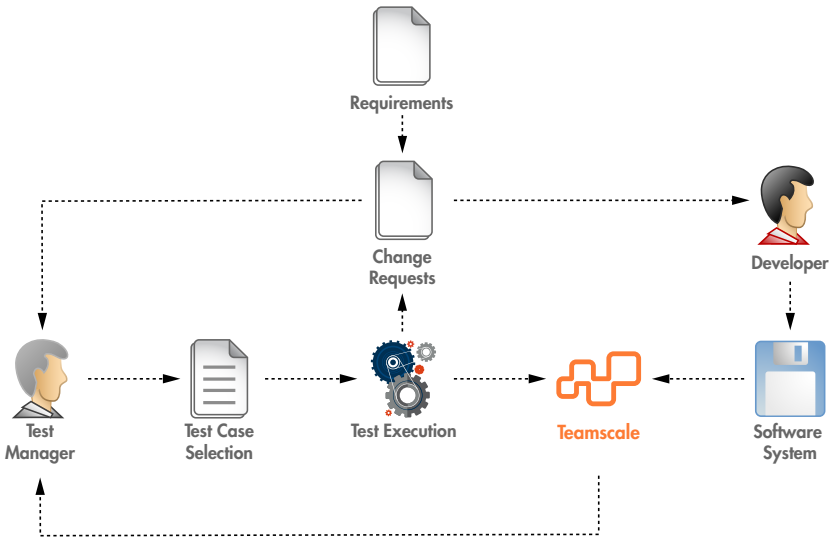


Figure 1.3. Test Quality Control

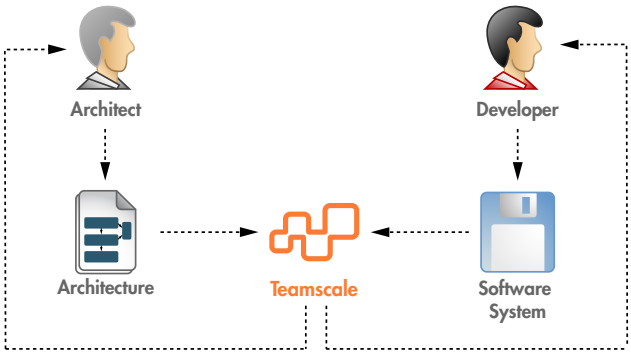


Figure 1.4. Managing Architecture Quality

Teamscale reveals recent changes that have not been tested so far and can provide feedback to the test manager (see Figure 1.3). Based on these existing gaps in the test execution, the test manager can decide which test cases to select for the next test run.

This form of software intelligence combines information from the development and testing process that is already there. But many companies have not been able to combine the information due to missing tool support. With *Teamscale*, the information can be easily combined.

1.2.4 Managing Architecture Quality

In addition to code and test quality, *Teamscale* also helps to manage your architecture quality. Many teams have at least an informal understanding what the architecture of their system – in terms of component diagrams – roughly looks like. But only few have it documented and even fewer have a conforming code base.

Teamscale helps you first to model the architecture. With an easy graphical user interface, you can specify components and wanted dependencies. In a second step, *Teamscale* continuously checks whether your current code base conforms to your specifications. It will notify the developer of unwanted dependencies immediately after a commit. After inspection, it is then up to the developer to clean up the code (if the dependency is, in fact, unwanted) or to adapt your specification (if you can live with the new dependency), see Figure 1.4. *Teamscale* supports you to avoid a commonly seen loss of architecture knowledge [FRJ09] and to keep your implementation conform to the intended architecture.

1.2.5 Analyzing Feature Usage

During any quality assurance process, managers and developers prefer focusing on those parts of the system that users often use. Nobody wants to waste money on code that is not even executed by the user. Even though this code has already been written, it still creates overhead costs for maintenance that should not be underestimated [Ede+12].

On the one hand, *Teamscale* integrates the information from a usage profiler in the system's production environment. On the other hand, it analyzes the development process. Combining both information sources provides valuable input for managing the costs of software maintenance (see Figure 1.5): It supports developers and managers in their decision to delete code that is never executed. It also helps them to make sure only those features are extended that are in fact used.

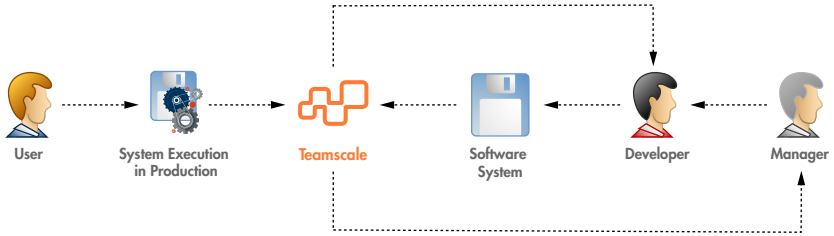


Figure 1.5. Analysis of Usage Data

1.3 Architectural Overview

In this section, we provide a rough overview of *Teamscale*'s internal architecture, see Figure 1.6. *Teamscale*'s core is the incremental analysis engine. It is directly connected to the version control system and, hence, analyzes each commit *incrementally*. This enables *Teamscale* to provide rapid feedback and reveal the root causes on commit-based for emerging problems or deteriorating trends.

With its analysis engine, *Teamscale* provides its own set of code quality analyses, such as clone detection, structural metrics or comment analyses for example (see Chapter 2). It also has its own architecture analysis. In addition, the analysis engine integrates various kinds of data from your software development – from test coverage over usage data to external static code analysis tools that you might wish to use specifically. It also connects to your bug and issue tracking system and can, thus, provide links between change requests, commits and changes to the system's quality status. Integrating all these data, *Teamscale* helps to manage your code, test and architecture quality as well as monitor your feature usages (see Section 1.2). In other words, *Teamscale* becomes the core platform for software intelligence.

The results of the analysis engine are kept in a NoSQL store and made accessible with a REST Service API. The web client, for example, uses

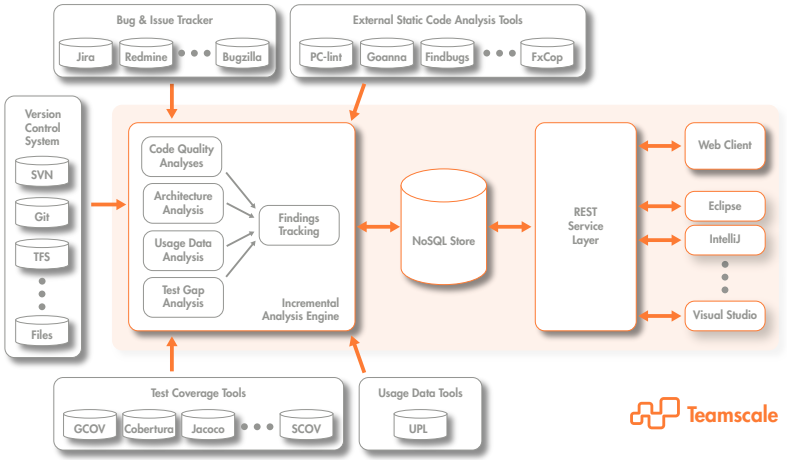


Figure 1.6. *Teamscale* Architecture

this API for the user interface in the browser. In addition, *Teamscale* also provides several IDE clients, for example, plugins for Eclipse, IntelliJ and Visual Studio.

1.4 Target Audience and Outline

This document is targeted at developers, quality engineers and project managers who want to use *Teamscale* in their development projects. This guide assumes that *Teamscale* is already set up in your environment. For details on the installation and configuration, please refer to the Teamscale *Admin Guide* (see also Section 1.5).

The remainder of this document is structured as follows. Chapter 2 introduces the quality analyses provided by *Teamscale*. This chapter is suitable if you want to find out which metrics and findings are computed by *Teamscale*. Chapter 3 describes common use cases for

Teamscale. This chapter is ideal if you want to learn about the capabilities of *Teamscale* and find out how to achieve a specific goal. In contrast, Chapter 4 provides a comprehensive overview of the features of *Teamscale*. This chapter for instance helps you to understand how specific parts of *Teamscale* work in detail. At the end, the glossary explains the underlying terminology used throughout this book.

1.5 Further Resources

Besides this user guide, a number of further resources provide additional information on the usage of *Teamscale*:

- The Teamscale *Admin Guide* available in the `documentation` folder of your *Teamscale* installation provides a documentation on the installation and configuration of *Teamscale*.
- The *Teamscale* Knowledge Base[1] provides FAQs, Getting Started Guides and solutions to recurring problems and common pitfalls.

In case your question is not answered anywhere, please don't hesitate to contact the *Teamscale* support (https://www.cqse.eu/en/products/teamscale/support/).

[1] https://cqse.freshdesk.com/solution/categories



2. Quality Analyses

As a Software Intelligence Platform, *Teamscale* monitors your software's quality and combines it with essential data from your development. It supports various technologies and provides many different quality analyses. This chapter gives you an overview of the supported programming languages and development infrastructures (Section 2.1), *Teamscale*'s own analysis engine (Section 2.2) as well integrated external analysis tools (Section 2.3). In addition to the integrated analyses, *Teamscale* comes with a framework for simple, user-defined analyses based on token streams or simple parse trees. This »custom-check« framework is documented in Section 4.3.1 (Appendix F contains a list of custom checks delivered with *Teamscale*).

2.1 Supported Technologies and Infrastructure

Table 2.1 shows the list of supported programming languages and development Infrastructure tools.

Table 2.1. Supported Technologies and Infrastructure

Programming Languages	
Languages	ABAP, Ada, C#, C/C++, Delphi, Fortran, Groovy, HANA SQLScript, HANA View, IEC 61131,3 ST, Java, JavaScript, Magik, Matlab, OpenCL, PHP, PL/SQL, Python, Rust, Simulink, XML, Visual Basic, Xtend
Integrated Development Infrastructure	
Version Control	Git, Subversion, TFS, File System
IDE Integration	Eclipse, Visual Studio, IntelliJ
Issue Trackers	Jira, Bugzilla, Redmine, TFS
User Management	LDAP, Active Directory, Integrated User Management
Code Review	Crucible

2.2 Internal Analyses

Teamscale's real-time analysis engine enables providing immediate feedback – just seconds after a commit to the version control system.

The engine comprises many different analyses, ranging from code quality analyses (see Table 2.2) to test and architecture quality analyses (see Table 2.3). Please note that not all analyses might work for every language (even though most do).

2.3 Integrated External Analyses

Table 2.4 shows the list of external tools which can be integrated in *Teamscale*. These tools comprise static code analyses, test coverage measures as well as usage monitoring.

Table 2.2. Internal Code Quality Analyses

Code Structure	
File Size	Measures the length of a file.
Method Length	Measures the length of a method.
Nesting Depth	Measures the nesting depth of a method.
Cyclomatic Complexity	Indicates overly complex code.
Redundancy	
Clone Coverage	Measures how much code is affected by code clones.
Code Documentation	
Comment Completeness	Measures how many entities of the public interface are missing a comment. Entities can be configured using entity selection predicates (see Appendix E)
Comment Quality	Indicates comments with insufficient quality.
Task Tags	Indicates ToDo/Hack/FixMe comments.
Code Anomalies	
Bad Practice	Indicates bad programming practices.
Correctness	Indicates potential null pointer dereferences.
General Checks	Indicates commented-out-code and empty blocks.
Unused Code	Indicates unused methods and private fields (Language-dependent.)
Naming	
Naming Convention Violations	Indicates if identifiers violate the naming convention.

Table 2.3. Internal Code, Test and Architecture Quality Analyses

Formatting	
Code formatting	Indicates mutliple statements in same line, multiple declarations in same statement or missing braces.
Text formatting	Indicates Non-ASCII characters or long lines.
Code Reviews [with Crucible]	
Missing Reviews	Measures the number of revisions (commits) where at least one file is not reviewed.
Open Reviews	Measures the number of open reviews over all file revisions.
Test Gaps	
Test Gap Analysis	Indicates changed code that has not been tested.
Architecture Conformance	
Conformance	Indicates dependencies which are not conform to the architecture specification.
Missing Documentation	Indicates types which are not included in the architecture specification.

Table 2.4. Integrated External Tools

Static Code Analysis	
ABAP	SAP Code Inspector
C#	StyleCop, FxCop
C/C++	Clang, PC-Lint
Java	PMD, Findbugs
JavaScript	ESLint, TSLint
Python	PyLint
XML	SpCop
...	This list is extendable! As *Teamscale* provides an interface, any external analysis tool can be integrated.
Test Coverage Analysis	
ABAP	SAP SCOV
C#	XR.Baboon, Visual Studio Test Coverage
C/C++	lcov, Visual Studio Test Coverage, gcov, CTC
Java	JaCoCo, Cobertura
JavaScript	lcov
Python	coverage.py
Usage Analysis	
ABAP	UPL



3. Common Use Cases

This chapter describes five common use cases of *Teamscale*. The first use case explains how to configure a project with a corresponding analysis profile (Section 3.1). The second one explains how to receive feedback from *Teamscale* (Section 3.2). Section 3.3 and 3.4 describe how to get an overview of the current quality status and how to monitor it continuously. The last use case describes how to identify untested changes (Section 3.5).

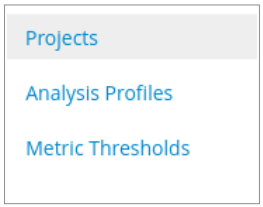


Figure 3.1. Sidebar of the Projects Perspective

3.1 Configuring Projects and Analyses

This use case demonstrates how to configure your project in *Teamscale*. To run *Teamscale* on your code, two steps are required: First, you have to create an analysis profile which configures the analyses that should be run and all their options. Second, you need a project configuration which configures the scope of the analyzed code. Both is done in the **Projects** perspective. The perspective features a sidebar that can be used to navigate to three different configuration pages, Figure 3.1:

- **Projects:** This page allows you to add projects to *Teamscale* that should be analyzed.
- **Analysis Profiles:** This page allows you to configure the analyses that are run for your projects.
- **Metric Thresholds** This page allows you to define a target threshold for all metrics that your projects should abide by.

3.1.1 Creating an Analysis Profile

Teamscale comes with pre-configured analysis profiles for all supported languages (Figure 3.2). These provide sane default values

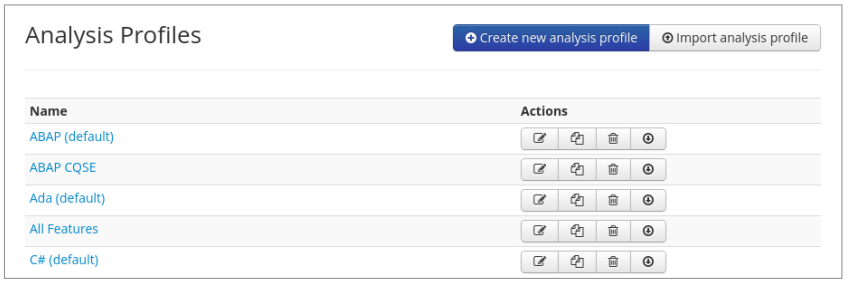


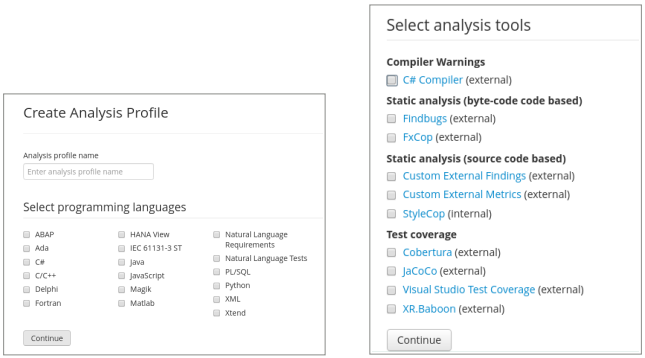
Figure 3.2. The **Analysis Profiles** page

for all analyses. Not all available analyses are enabled by default, though. When starting out, it is usually a good idea to start with the default profile for a language and customize it later to fit your needs. Each project uses exactly one such profile.

To edit an analysis profile, click on next to its name. The other buttons in this row can be used to duplicate a configuration , delete it or download it as an XML file .

You can also create a new analysis profile from scratch by using Create new analysis project or import an existing (previously exported) XML file by using Import analysis profile. Regardless of whether you choose to create a completely new profile or edit an existing one, you will be presented with the analysis profile edit screen (Figure 3.3a).

The first section of this screen allows you to specify a name for the profile and select all the programming languages that should be analyzed. Each project you configure in *Teamscale* can only have a single analysis profile. Thus, if your project contains code written in more than one language, you must select all of these languages in the analysis profile to have them analyzed.



(a) Step 1 (b) Step 2

Figure 3.3. Editing an Analysis Profile

If you click on Continue, the next section will appear (Figure 3.3b). In this section, you can select additional analysis tools that should be integrated with *Teamscale*. These include internal tools, like StyleCop for C#, which are run by *Teamscale* itself and external tools, like FINDBUGS for Java, which must be run by your nightly build, for example. Internal tools are run incrementally (i.e. with each commit), while the results of the external tools are only updated, when you run them yourself (e.g. nightly). You may also select the tools for which you would like to upload test coverage. If you select none, test coverage metrics will be disabled for this project.

After another click on Continue, you will be taken to the quality indicator configuration screen (Figure 3.4). All analyses in *Teamscale* belong to so-called analysis groups which are in turn aggregated into quality indicators, allowing you to combine related groups. A common use case is to combine several formatting-related groups (e.g. spacing and layouting analyses) into one quality indicator called *Formatting*. All findings generated by these analyses will then show up

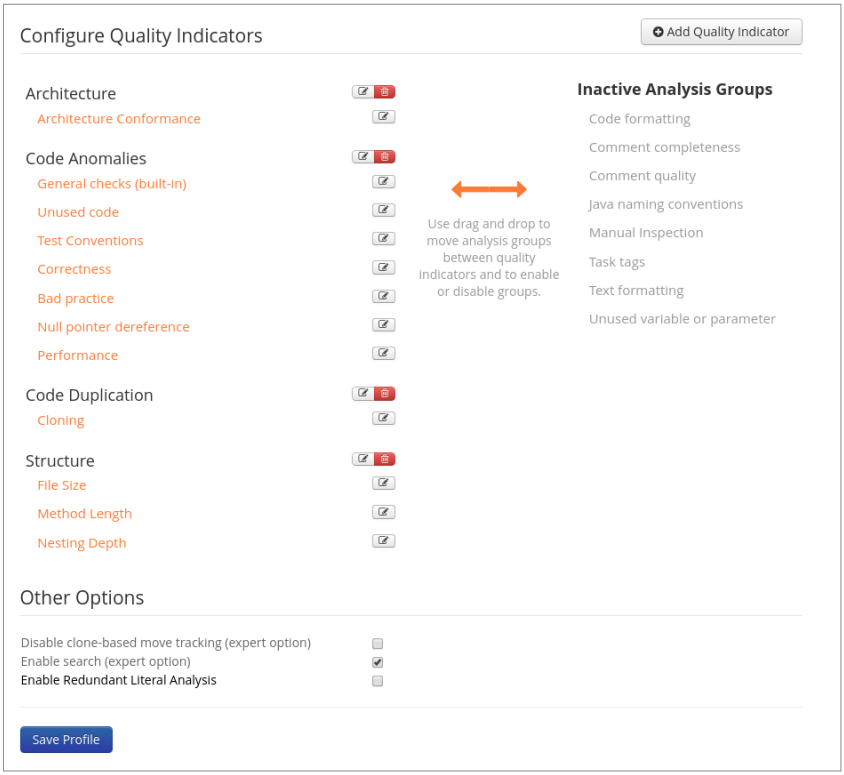


Figure 3.4. Editing an Analysis Profile (Step 3)

under that name in *Teamscale* and can easily be filtered or grouped together.

You may create any number of such quality indicators by using the button Add Quality Indicator. To add or remove analysis groups to/from them, simply drag the group's name between the indicators and the section **Inactive Analysis Groups**.

Groups and indicators all have their own set of options, which can be accessed via the small ✎ next to their name. For a quality indicator, you can configure its name and which metrics should be calculated for this indicator. For analysis groups, you can configure options that determine the behavior of the analyses as well as the severity of the findings produced by the analyses. To match your individual analysis needs, the analysis groups can also contain custom checks. In which analysis group a custom check is placed, depends on the declaration of the programmer within the implementation of the custom check. *Teamscale* comes with a variety of pre-written custom checks for various languages, see Appendix F.

Since these dialogs often contain many options, you can use the filter text box at the top of the dialog to quickly locate certain options. Be careful with options that are marked as expert options. We recommend to not change these unless you have a deep understanding of the implications. Details for configuration options can be found in Appendix E. Finally, the screen contains a section for **Other Options**. These global options do not affect a single quality indicator but rather the entire analyzed project, e.g. whether to enable searching.

Once you are satisfied with your analysis profile, click Save Profile. You will be directed to a page that shows a summary of the configuration you made. You can now create a project with that profile. Please note that editing an existing analysis profile will not change any set-

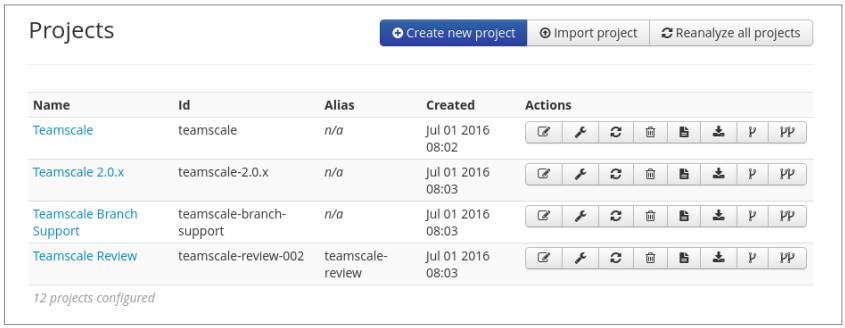


Figure 3.5. Overview of All Configured Projects in *Teamscale*

tings for existing projects that reference this profile. If you want to not only update the profile but also all projects using it, please select to reanalyze those projects when prompted. Note, though, that the reanalysis might take a while, depending on the length of your project's code history!

3.1.2 Creating a Project Configuration

On the **Projects** page (Figure 3.5), you can see a list of all projects configured in *Teamscale*. The following action box (Figure 3.6) shows you which actions can be applied on an existing project.

To create a new project, select **Create new project** button at the top. You will be taken to the project creation screen (Figure 3.7). Here you can enter the name of your project and select the analysis profile (see Section 3.1.1). Next, you can add any number of source code repositories and issue trackers. To do so, select the correct repository or issue tracker type from the corresponding drop-down. A new section (Figure 3.8) will appear where you can configure all options needed to connect to the repository or issue tracker, e.g. credentials,

Project Management Actions

- Opens the edit dialog for the project. The dialog is the same as for project creation except that the project name can not be changed.
- Opens a dialog for advanced configuration options such as a connection to SAP systems and the threshold configuration to use for the project. *Teamscale* defines a default threshold configuration but users can create custom threshold configurations (also discussed in this section).
- Triggers a complete re-analysis of the project.
- Deletes the project.
- Exports the project configuration. The exported file can be imported with the "Import project" button in Figure 4.73.
- Exports a complete backup of the project (including analysis results). The exported file can be imported with the "Import project" button in Figure 4.73.
- Renders a graph of the *Teamscale* analysis process (requires the rendering tool Graphviz).
- Renders a graph of the *Teamscale* analysis process including the database/index layer (requires the rendering tool Graphviz).

Figure 3.6. Actions Available in the Project Overview

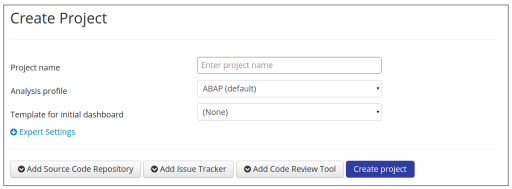


Figure 3.7. Creating a New Project

starting revision, files to exclude, etc. Figures 3.9 and 3.10 explain these options in detail.

Most connectors require you to specify an account that holds shared credentials (user name, password). While any user with sufficient rights to create new projects may also create new accounts, only the system administrator may edit or delete existing accounts in the **Admin** perspective.

You may use ☑ Validate to make sure your settings are correct before clicking **Create Project** to start the analysis. To monitor analysis progress you can have a look at the **System** or **Activity** perspective (Section 4.1.3).

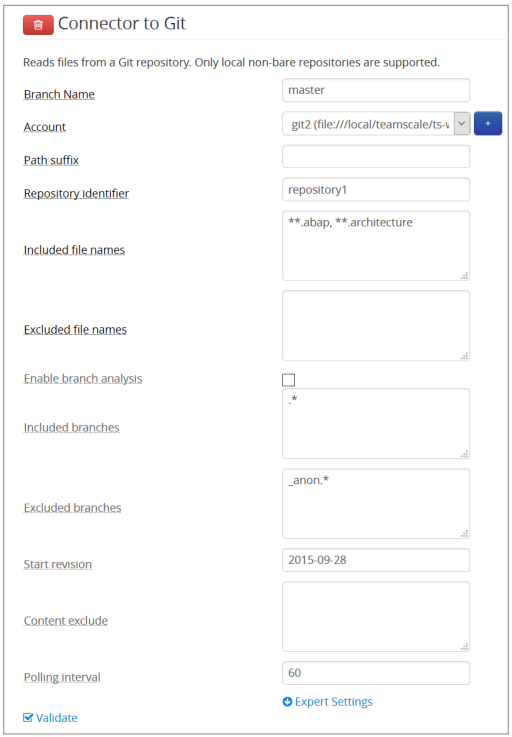


Figure 3.8. Configuring the Connection to a GIT Repository

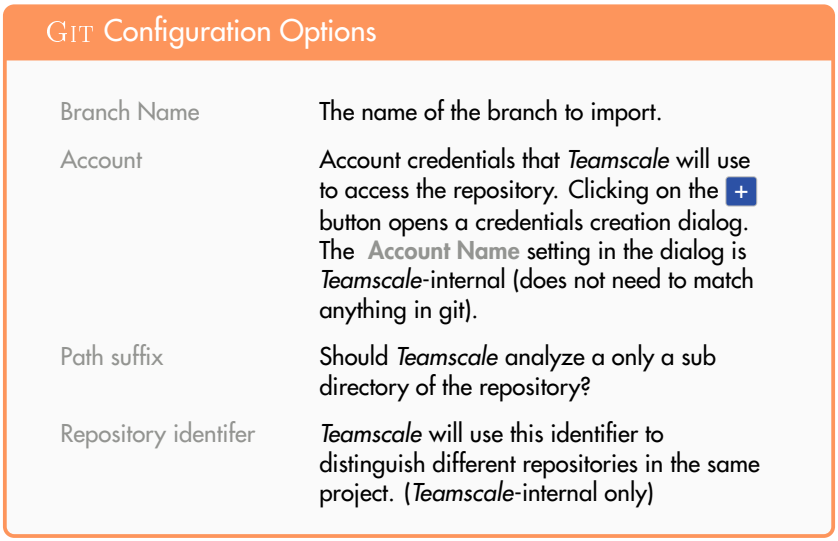


Figure 3.9. Configuring a GIT Repository Connector

GIT Configuration Options (continued)

Included file names	You can use these properties to focus the analysis on a subset of files.
Excluded file names	You can exclude certain files.
Enable branch analysis	This checkbox activates our new branch support feature. If you select it, *Teamscale* will analyze all branches of the repository. If branch analysis is enabled, the **Branch Name** option specifies the main branch (usually »master«).
Included branches	Explicitly control which branches will be analyzed. (Has no effect if **Enable branch analysis** is not selected.)
Excluded branches	Explicitly state which branches to exclude.
Start revision	The first revision that will be analyzed. You can enter a date (which selects the first revision after the date) or a revision hash.
Content exclude	A comma-separated list of regular expressions. *Teamscale* will exclude all files that match one of these filters.
Polling interval	How often should we query the repository and update our findings? (Unit is seconds)

Figure 3.10. Configuring a GIT Repository Connector

3.2 Getting Personalized Developer Feedback

As opposed to many existing static analyses tools, *Teamscale* strives to provide *immediate feedback* for developers. Personal and quick feedback is essential when trying to improve quality (see Section 1.2): We have often seen that the absence of it causes quality improvement campaigns to fail (Section 1.2.1). *Teamscale* employs its analyses commit-based and is, thus, able to provide fine-grained feedback. Feedback which does not overflood the developer but instead focuses on his specific need. And feedback which comes quick enough while he is still working on his task.

Developers can use *Teamscale* in several different ways to get feedback: In the activity perspective of the Web UI, they can inform themselves about the impact of their recent commits (Section 3.2.1). If developers want to avoid context switches, they can stay in their IDEs and use *Teamscale* as a plug-in (Section 3.2.2). Or, they can turn on the email service and receive notifications when something happens (Section 3.2.3).

3.2.1 Inspecting the Impact of a Single Change

In the activity perspective, the developer can browse the commit history of the underlying version control system (see Figure 3.11). He can directly glance at the most recent commits at the top of the stream or use the filter mechanism in the sidebar to view only those changes that were committed by himself, for example. The commit stream provides the technical information about the commit itself (author, message, date and related issue) but also its impact on the system's quality: It shows how many findings were added by this commit (red),

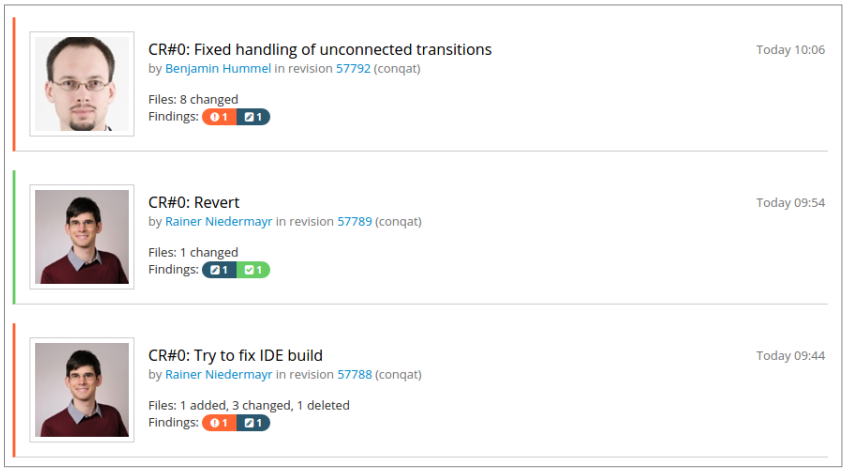


Figure 3.11. Developer Feedback in the Activity Perspective

how many were removed (green) and how many still remain in code that was modified by this commit.

By clicking on the commit, the developer gets more detailed feedback (see Figure 3.12). He can inspect which findings in which file were added or removed and also which impact his commit had on various metrics. He can then either click on the findings message to navigate to the finding's detail page providing more information about the type of the finding and why it is relevant or click on the finding location to directly jump into the source code.

Inspecting the impact of your change in the browser, however, requires the developer to actively pull this information. *Teamscale* also provides push-notification services in your IDE (Section 3.2.2) and via email (Section 3.2.3).

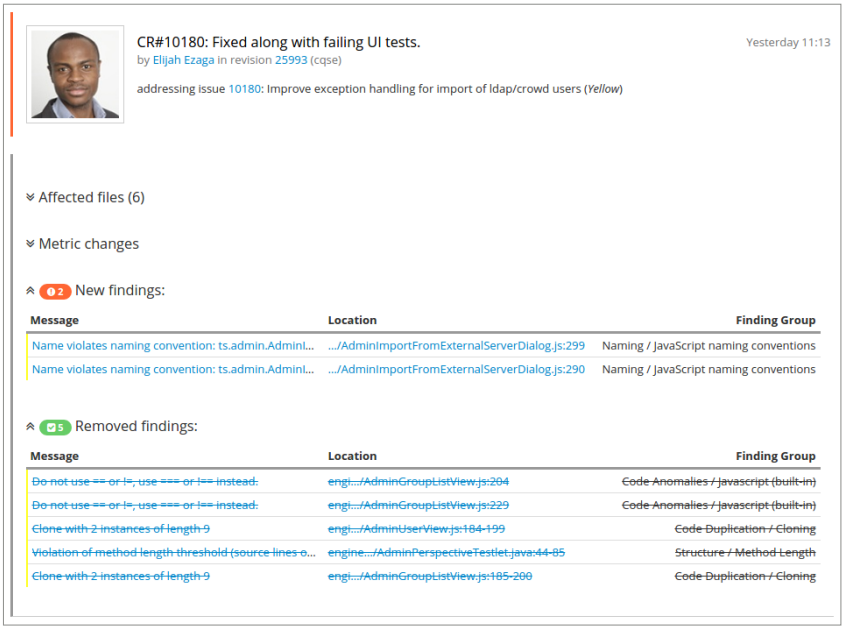


Figure 3.12. Detailed Information about the Impact of a Commit

3.2.2 Being Informed Without Leaving Your IDE

Rather than switching to the browser, developers often prefer to gather all necessary information within their IDE. *Teamscale* integrates in common development environments such as Eclipse, Visual Studio and IntelliJ. With the plug-in, you can inspect the findings directly as markers in the code. To clean up on-the-fly or to inspect newly introduced findings, developers do not have to leave their IDE. Thus, they can avoid distracting context switches. Section 4.2 provides more details about how the IDE integration can be set up.

3.2.3 Push Notifications: Receiving Emails about Changes

Furthermore, you can configure *Teamscale* to notify you via e-mail or on your desktop. It remains up to you, about what kind of changes you want to be alerted.

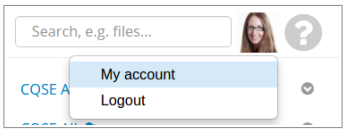


Figure 3.13. Navigating to the Personal Account

To configure your notifications, click on your user symbol in the upper right corner of the browser and select **My account** in the drop down menu. Then you can select the **Notification Rules** page in the right side bar menu (see Figure 3.13). You can activate or deactivate desktop notifications by using the check box **Enable desktop notification**, see Figure 3.14. In terms of email notifications, you can add rules to be notified when a certain type of finding is added, when a metric threshold is violated or when you are related to a task.

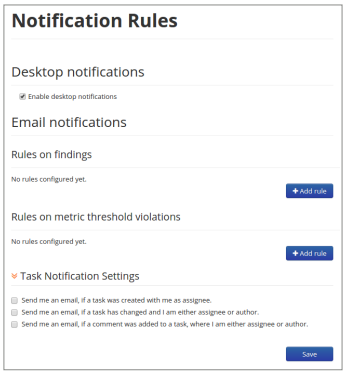


Figure 3.14. Configuring Personal Email Notifications

Figure 3.15 shows exemplary the configuration dialog to add a notification rule for findings. You can configure the projects on which the rules applies as well as committers and events triggering the rule. You can also explicitly sate to not trigger the rule if, e. g. the finding category or the finding group equal your specified string. To look up how your categories are named and which groups you are using, please refer to your analysis profile.

3.3 Getting an Overview of the Quality Status

Often, you as a developer or as a project manager would like to get an overview of the current quality status of your system – or maybe even of your portfolio of systems. In this section, you will learn how to create a single project overview (Section 3.3.1) and a multiple project overview (Section 3.3.2).

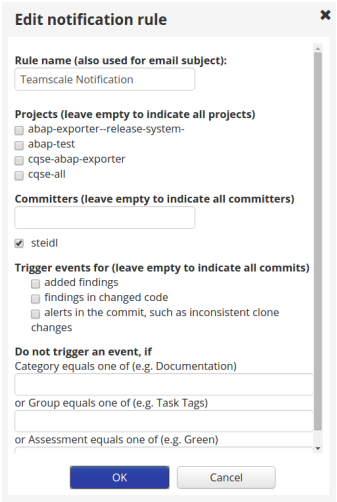


Figure 3.15. Editing a Notification Rule for Findings

3.3.1 Creating a Single Project Overview

To get an overview of the quality status of one single project, there are two different ways – either with the **Metrics** perspective or the **Dashboard** perspective. We will start with the latter.

To configure your personal overview dashboard, click on the **Dashboard** perspective. The **Dashboard** perspective will initially be empty, if no dashboards have been configured yet. It provides three buttons in the right sidebar which are explained in Figure 3.16. Click on ⊕ icon to add a new dashboard and select **Create from template (T)** from the drop-down menu. The following dialog will give you the opportunity to configure the creation of the new dashboard, e. g. give it a name and select a template, e. g. the default overview template. For more details, please refer to the user reference in Section 4.1.2.

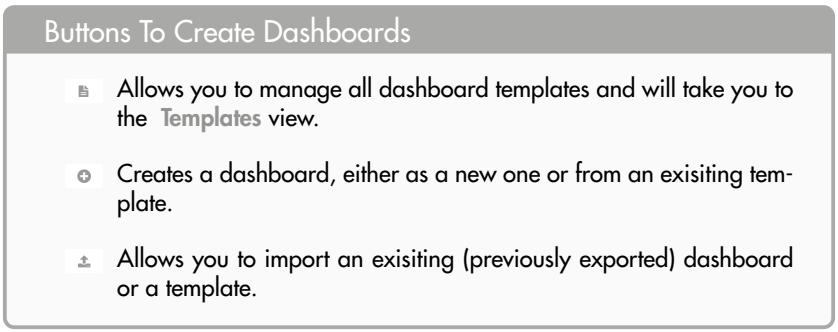


Figure 3.16. Buttons in the Sidebar of the **Dashboard** Perspective

The dashboard now gives you an overview of the quality status of your system, see Figure 3.17. You can modify the dashboard by clicking on next to its name in the right side bar and selecting **Edit dashboard**. Editing allows you to rearrange the widgets and create or delete widgets, Figure 3.18. For more details, please refer to the user reference in Section 4.1.2 or the configuration help for widgets in Appendix A.

The information from the dashboard can be adapted based on (sub-) project and time. To change the project, edit the dashboard as above and configure each widget using the symbol in the top left corner of a widget. By setting the **Path** you can either switch to a different project or select a component or subcomponent of your system.

If you would like to change the time for the dashboard, leave the edit mode and go back to viewing the dashboard. Then click on your projects name in the top right corner (see Figure 3.19). A new row will expand underneath the perspective header, allowing you to click on **Timetravel**. The timetravel allows you to select a calendar time, a

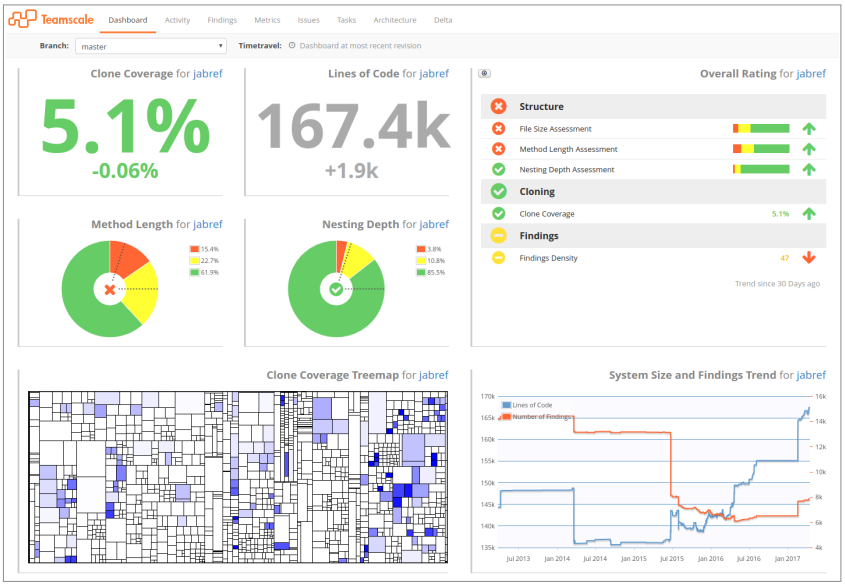


Figure 3.17. Overview Dashboard

revision or a baseline. Then, the dashboard will display the system's quality status at the selected point in time.

As an alternative to the **Dashboard** perspective, you can also navigate to the **Metrics** perspective. Here, you will find an overview of all metrics in the first row of the table of the perspective, see Figure 3.20. The overview contains all numeric metrics and assessment metrics and can also be configured. However, it does not provide all information available in the **Dashboard** perspective. Hence, it is useful for a first glance but not for a personally tailored overview.

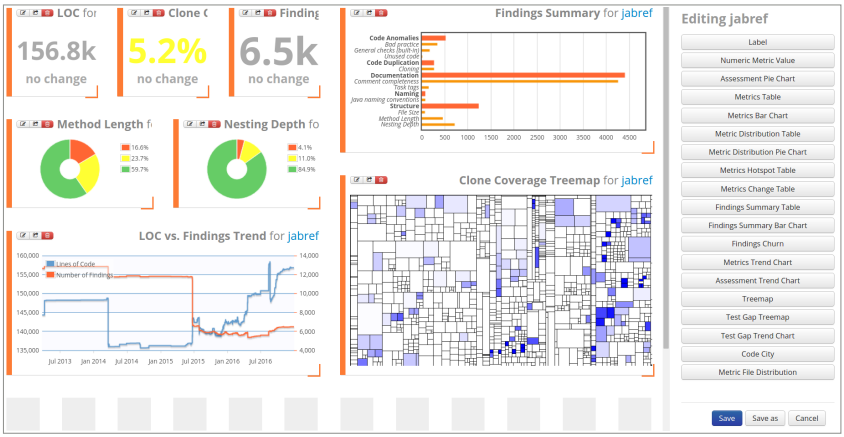


Figure 3.18. Editing a Dashboard

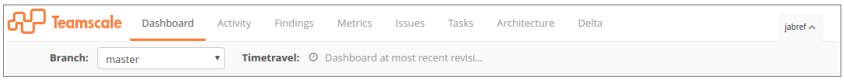


Figure 3.19. Using the Timetravel for a Dashboard

3.3.2 Creating a Multi Project Overview

If your role is being a project manager rather than a developer, you might be more interested in getting an overview of a portfolio of systems. *Teamscale*'s dashboards support combining information from multiple projects. Figure 3.21 shows a very simple example dashboard displaying the lines of code information for six different projects as well as their trend over the last 30 days. You can combine information from all projects. They do not need to be written in the same programming language and they also do not need to be in the same version control system. As long as you access rights, you can combine any of them in one single dashboard.

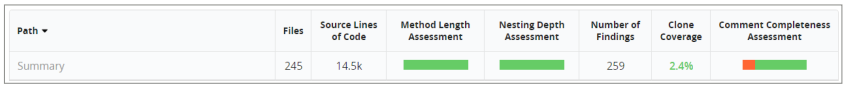


Figure 3.20. Metrics Summary in **Metrics** Perspective Table

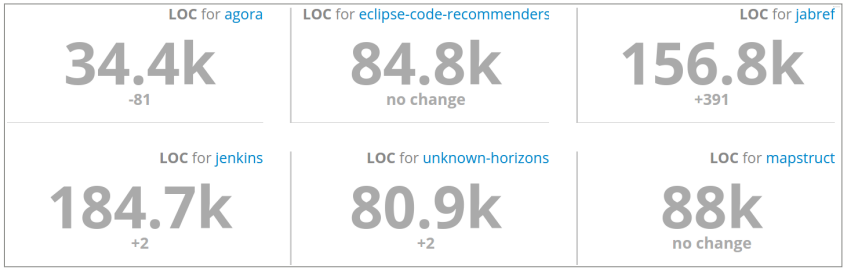


Figure 3.21. Example Portfolio Dashboard

3.4 Monitoring and Assessing Quality Evolution

An essential part of software intelligence comprises monitoring your quality evolution and drawing the right conclusions from it: when something goes wrong, panic and over-the-top decisions are usually the opposite of helpful. Instead, *Teamscale* provides a transparent root-cause analysis. Based on these results, you can derive reasonable and specific counter measures. This section shows how *Teamscale* supports the role of a quality-engineer in a quality-control-like process as depicted in Section 1.2.2. In this context, quality control refers specifically to *code* quality control.

3.4.1 Inspecting Recent Metric Trends

Monitoring quality evolution often starts with the question: In which direction are we heading? Is there a trend to get better or are we doing worse? As *Teamscale* has the entire history available from the underlying repository, it can provide transparent and understandable answers.

For every metric you have configured, *Teamscale* offers you to inspect it as a trend chart. It can compute this trends over any selected period within the history that *Teamscale* has available. To view quality trends for your system, go to the Dashboard perspective and select the button ⊕ in the right sidebar. If you choose **Create From Template**, you can give the dashboard a name and select the overview dashboard template as well as your specific project. The dashboard gives you an overview of your quality as status as already explained in Section 3.3. But the dashboard also shows you in which direction you are heading: Metric that consist only of one numeric values, such as lines of code, clone coverage and findings count are displayed with a numeric metric value widget. This widget shows both the absolute value as well as the trend over the last 7 days as default, see Figure 3.22. If the trend is positive, the delta in the metric value will be marked as green, see the decreasing clone coverage in Figure 3.22. In contrast, if the trend is negative, the delta will be marked in red.



Figure 3.22. Trend Indication for Numeric Metrics

To change the length of the considered trend interval, you can click on the right arrow next to your dashboard's name in the right sidebar

and select **Edit Dashboard**. Each widget displays an edit button in the top left corner. Click on it and the dialog **Edit Parameters** will open in which you can set the duration of your trend interval (see Figure 3.23).

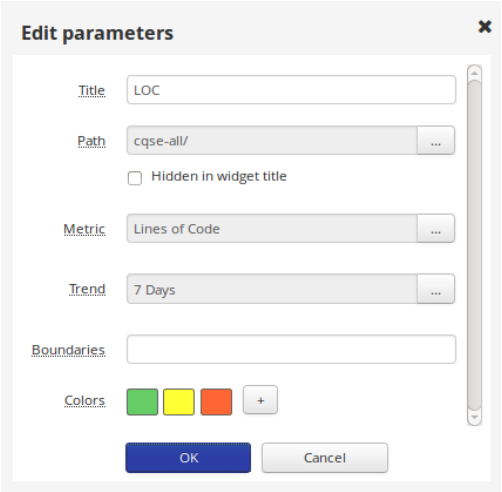


Figure 3.23. Edit Parameters for a Numeric Metric Value Widget

Dashboards can also display trends as graphical trend charts, see Figure 3.24. The template already provides trend chart for the lines of code metric and the number of findings. With the same edit mechanism as above, the widget can be configured to show any numeric metric trend. The default time span of the interval is set to zero days, which means that the entire available history is included. To obtain the graph as depicted in Figure 3.24, we configured the widget to include the clone coverage metric.

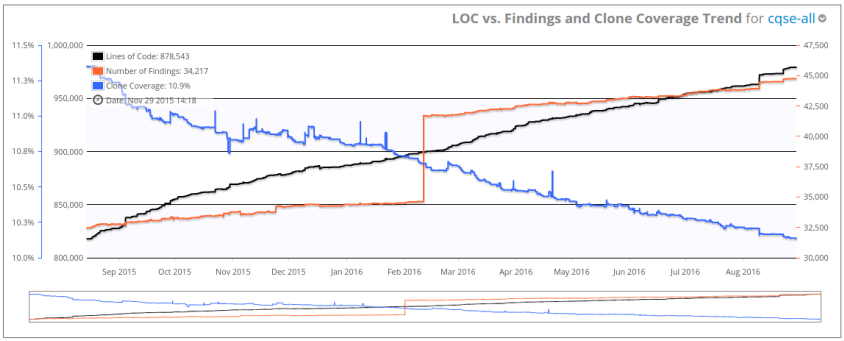


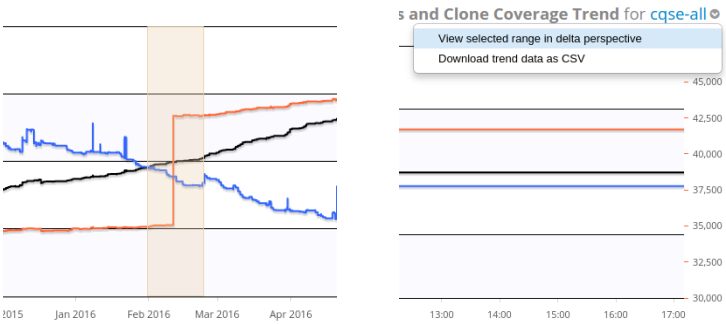
Figure 3.24. Trend Charts for Numeric Metrics

3.4.2 Finding the Root Cause for a Trend

Sometimes, metric trends reveal suspicious drops or increases, for example the sharp increase in the number of findings (red line) in Figure 3.24. This makes us as quality engineers curious to find out what happened. With a specific configuration of the dashboard, you can discover the underlying root cause. Let us show you how: As before, go to the edit mode of our dashboard and open the dialog **Edit Parameters** of this specific metric trend chart widget. Enable the two options **Zooming** and **Action Menu**. Then save the dashboard.

To inspect a suspicious change in a trend, use the mouse to mark the area wich you would like to inspect closer and zoom in, as indicated in Figure 3.25a. Zooming in narrows down the time interval while providing a higher resolution. When releasing the mouse button, the chart will update to represent the zoomed-in area. Then, use the action menu button ⚙ to select **View selected range in delta perspective**, see Figure 3.25b.

While you are redirected to the **Delta** perspective, *Teamscale* computes the so-called delta analysis in the background: It uses the start



(a) Zooming Into a Trend Chart

(b) Selecting the Corresponding Delta

Figure 3.25. Root Cause Analysis For Trend Charts

and the end of the interval that you selected by zooming into the trend chart. It then computes and reconstructs everything that happened in between this start and end date: As a result, it shows you, for example, which files were touched within the time interval or how metrics have changed. In particular, *Teamscale* also shows you the commits to the repository that were made within this time frame. They are often the first starting point to the root-cause analysis.

Select the **Repository Changes** in the right sidebar menu (see Figure 3.26) and scroll through the commits, see Figure 3.27. In our specific example, we want to figure out why the findings trend suddenly increased. Hence, we are looking for commits with a high findings churn. As our selected delta only contains two commits, it is rather obvious which one is responsible.

Click on the commit »inital ABAP architecture« to find out more details about the commit, see Figure 3.28. The commit detail view will show you that an architecture file was added. Resulting from it, more than 7000 findings emerged revealing violations from the newly specified

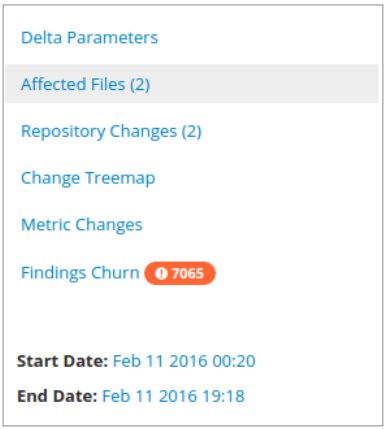


Figure 3.26. The Sidebar of the Delta Perspective.

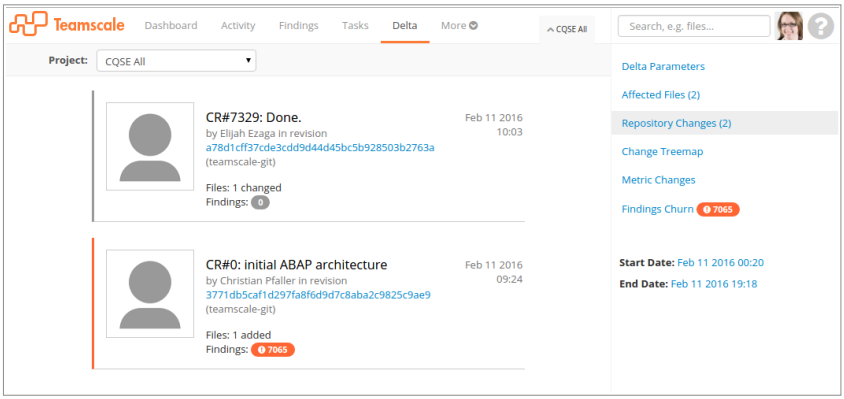


Figure 3.27. The Commits in the Delta

architecture. This is the root-cause for the increase in the findings trend. For more details about the **Delta** perspective, please refer to Section 4.1.8.

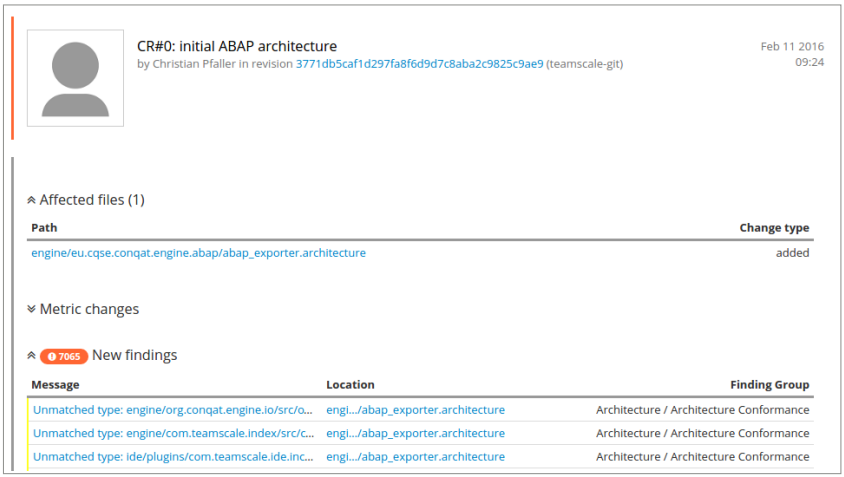


Figure 3.28. The Commit Detail View

3.4.3 Analyzing the Findings Churn

Besides monitoring the quality status in terms of metric trends, a thorough quality assessment also comprises a deeper inspection of the evolution of the findings in a system. You might want to check if *severe* findings were added. You might be curious how many findings were removed. Or if developers became lazy rather than still cleaning up on the fly. To analyze the findings churn, you can use two distinct perspectives: the **Findings Perspective** or the **Delta Perspective**. This section focuses on the first. To learn about the **Delta Perspective**, refer to Section 3.4.4.

The **Findings Perspective** in general provides a navigation page through all findings that exist in your system. But it can be also used to just inspect the findings that were added in a specific time frame, for example within the last month. This gives you the chance to assess the recent quality evolution.

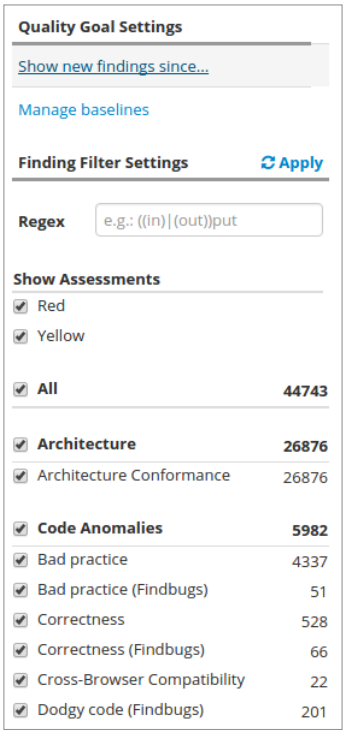


Figure 3.29. Filter Options in Findings Perspective

To focus on *recent* findings, use the right sidebar to configure which findings are shown, see Figure 3.29. For our use case, click on **Show new findings since...**. With the dialog opening up, you can select a date – findings will then be shown only if they were added *after* this reference date. This dialog has different options, see Figure 3.30. Under **Timespan**, you can indicate, for example, that new findings from *within* the last 40 days should be shown. Under **Time**, you can pick your reference date from the calendar or select it as a revision in your repository under **Revision**. The **Baseline** option additionally

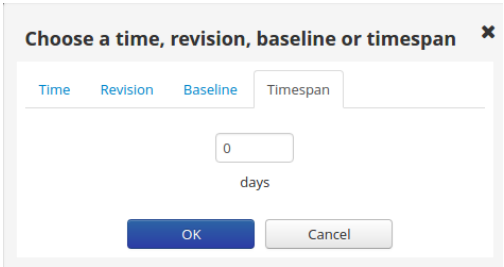


Figure 3.30. Selecting the Baseline

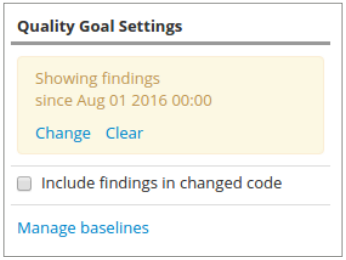
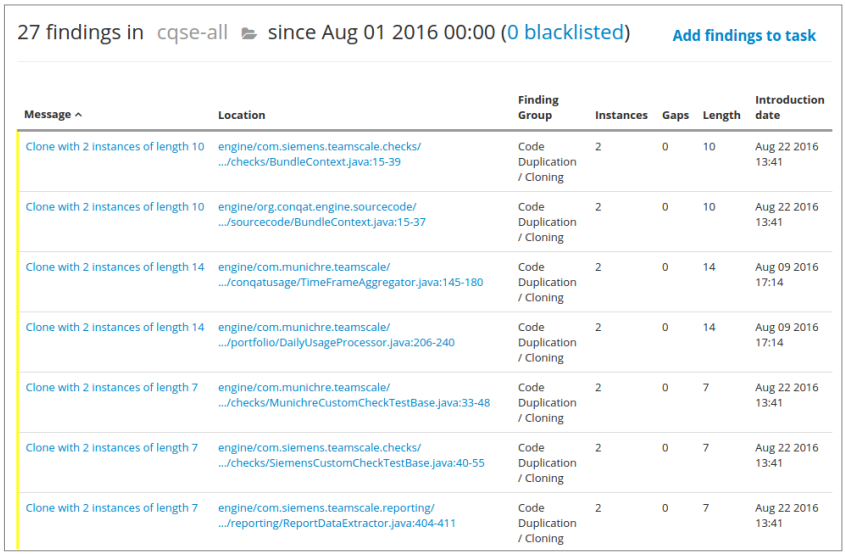


Figure 3.31. Quality Goals Settings in the Findings Perspective

provides you the opportunity to use a pre-configured baseline (or create a new one).

After you clicked on the button OK, the findings perspective will show you only findings that were added since the selected date. You can see this from the note in the **Quality Goal Settings** in the right sidebar, see Figure 3.31. Caring only about the *new* findings corresponds to the quality goal *preserving* as outlined in Section 1.2.2. If you would like to assess your quality evolution based on the quality goal *improving*, enable the option **Include findings in changed code**. In this case, the findings perspective shows not only the *new* findings but also findings in code that has been *modified* since your reference start date.



27 findings in cqse-all since Aug 01 2016 00:00 (0 blacklisted) Add findings to task

Message	Location	Finding Group	Instances	Gaps	Length	Introduction date
Clone with 2 instances of length 10	engine/com.siemens.teamscale.checks/ .../checks/BundleContext.java:15-39	Code Duplication / Cloning	2	0	10	Aug 22 2016 13:41
Clone with 2 instances of length 10	engine/org.conqat.engine.sourcecode/ .../sourcecode/BundleContext.java:15-37	Code Duplication / Cloning	2	0	10	Aug 22 2016 13:41
Clone with 2 instances of length 14	engine/com.munichre.teamscale/ .../conqatusage/TimeFrameAggregator.java:145-180	Code Duplication / Cloning	2	0	14	Aug 09 2016 17:14
Clone with 2 instances of length 14	engine/com.munichre.teamscale/ .../portfolio/DailyUsageProcessor.java:206-240	Code Duplication / Cloning	2	0	14	Aug 09 2016 17:14
Clone with 2 instances of length 7	engine/com.munichre.teamscale/ .../checks/MunichreCustomCheckTestBase.java:33-48	Code Duplication / Cloning	2	0	7	Aug 22 2016 13:41
Clone with 2 instances of length 7	engine/com.siemens.teamscale.checks/ .../checks/SiemensCustomCheckTestBase.java:40-55	Code Duplication / Cloning	2	0	7	Aug 22 2016 13:41
Clone with 2 instances of length 7	engine/com.siemens.teamscale.reporting/ .../reporting/ReportDataExtractor.java:404-411	Code Duplication / Cloning	2	0	7	Aug 22 2016 13:41

Figure 3.32. Assessing Clone Findings

To filter findings further, you can also filter them based on their finding group and finding category. Figure 3.29 shows you the selection mechanism. For example, deselect the check box **All**, select only the checkbox **Code Duplication** and scroll to the bottom of the page to press the button ⟳ to **Apply**. The main table of the findings perspective now shows you only recent clones, see Figure 3.32. You can use the table header to sort the set of clones, for example based on their number of instances or based on their length. This way, you can inspect, for example the largest clone recently added or the clone with the most instances. This usually should help you assessing what kind of clones were added to the system and how critical they are.

By clicking on a finding, you navigate to the **Findings Detail Page** of a specific clone, see Figure 3.33. This page gives you more detailed information about the finding. You can inspect the **introduction diff**,

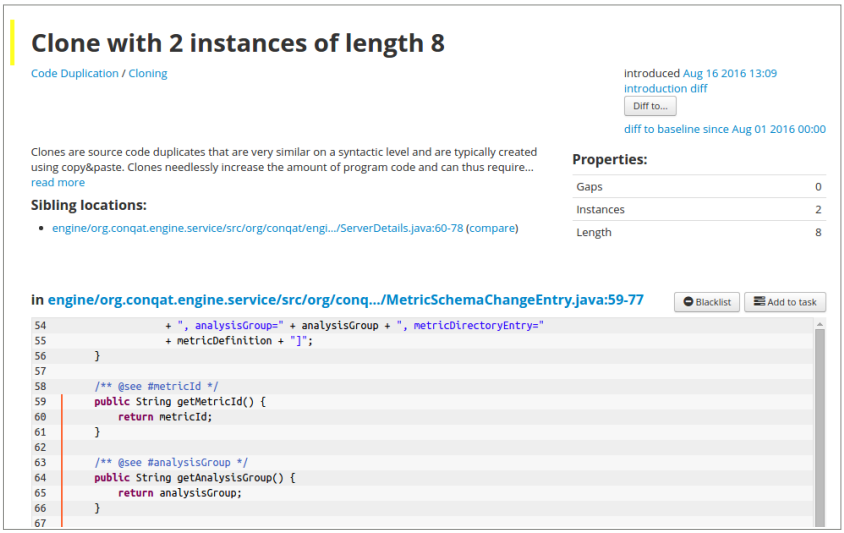


Figure 3.33. Detail Page for a Clone Finding

for example, or jump into the code to see the full surrounding file. In the specific example of code clones, you can also view the siblings, i. e. the other instances of the clone.

3.4.4 Performing a Baseline-Delta-Analysis

Under certain circumstances, you want to compare two snapshots of the system against each other and inspect everything that happened in between. For this use case, *Teamscale* provides the **Delta Perspective**. This perspective allows you to choose an arbitrary start date (the baseline) and an end date and performs the delta analysis in between.

To perform a delta analysis, click on **Delta Parameters** in the right side bar and select your start and end date, see Figure 3.34. You can

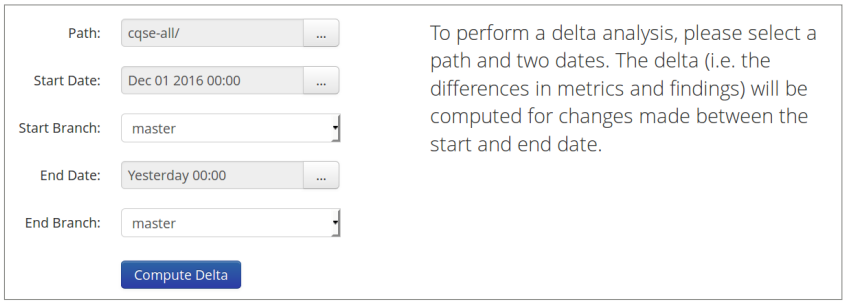


Figure 3.34. Selecting Start and End Date for Delta Analysis

choose both dates in form of a calendar date, a repository revision or a previously defined baseline of your system. In our example, we chose the month of August as underlying time interval for the delta. If branch support is enabled, you can also pick the branch of the underlying version control system. Then, hit the button Compute Delta.

After the delta analysis finished, *Teamscale* will redirect you to the **Affected Files** page within the **Delta Perspective**, see Figure 3.35. It shows you how many and also which files were added, deleted or moved within your project during, in our example, the month of August.

When you click on **Findings Churn** in the right sidebar, you can inspect findings that were either *added* or *removed* within the given time interal. Additionally, this page also shows you findings in code thas was *modified* in this period. Similar as in the **Findings Perspective**, you can navigate and filter these findings (see Section 3.4.3).

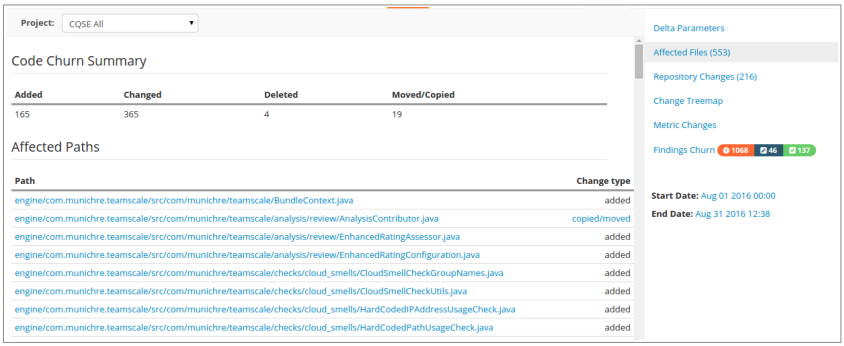


Figure 3.35. Overview of the Affected Files within a Delta

3.5 Identifying Untested Changes

Untested changes are typically error prone and often lead to production defects [Ede+13]. *Teamscale*'s *Test Gap analysis* is an automated way of identifying changed code that was never tested since its last modification. Such areas are called test gaps. Knowing a system's test gaps allows you to make a conscious decision, which code has to be tested before shipping and which gaps are of low risk and can be left untested. This section shows how to identify untested changes with the *Test Gap analysis* feature in *Teamscale*.

To use *Test Gap analysis*, your project should be connected to your source code repository, and you should upload test coverage regularly. Please refer to the Section 3.1 for information on how to set up projects in *Teamscale*.

As *Test Gap analysis* works on *changes*, you need to define a suitable time span, i. e. a baseline and an *end date*. *Baseline* refers to the date from which on to investigate code changes, while *end date* marks the other end of the time frame. All changes that were made to the

system between baseline and end date are considered in the *Test Gap analysis*.

3.5.1 Creating a *Test Gap analysis* Dashboard

In *Teamscale*, untested changes are visualized using dashboards. Therefore, we first briefly describe how to configure a *Test Gap analysis* dashboard, which is needed to identify test gaps. For more detailed information on how to create dashboards and use dashboard widgets in *Teamscale*, please refer to Section 4.1.2.

Switch to the **Dashboard** perspective and create a new dashboard. On the dashboard canvas, add a new **Test Gap Treemap** widget. You will see a treemap, a graphical representation of the source code of your project. Every rectangle on this treemap corresponds to a method in the source code. The bigger the rectangle, the longer the method. Resize the widget by dragging its lower right corner until its size fits your needs. Then hit  at the upper left corner to configure it.

On the configuration dialog, which is shown in Figure 3.36, there are several settings that determine, which information is shown on the treemap. Depending on your situation at hand, i. e., which specific information you are interested in, you will need to make different settings. These are explaind in Figure 3.38.

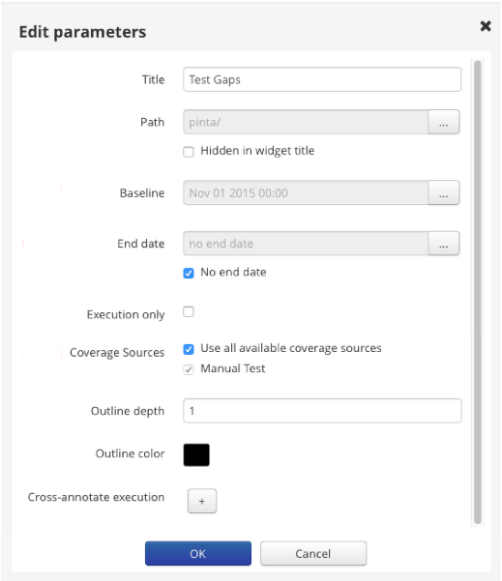


Figure 3.36. Configuration Settings for Test Gap Treemaps

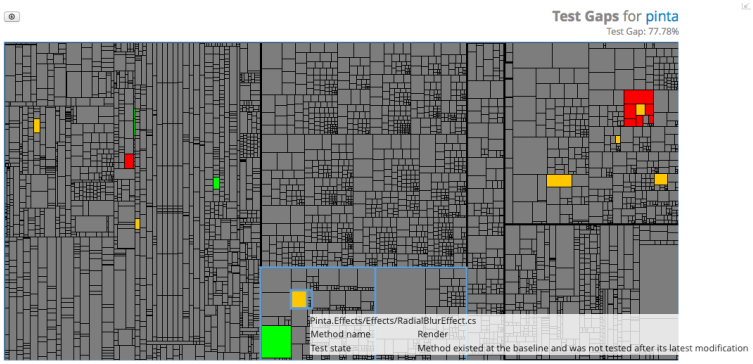


Figure 3.37. Test Gaps on the Test Gap Treemap

TGA Configuration Options

Path	Refers to the part of your project you want to see on the Test Gap treemap. You might want to select your whole project, but you can also choose a smaller subset, say the component you are responsible for. The treemap will then zoom into that area.
Baseline	Refers to the date from which on to consider changes, as described above. Choose a date slightly before the first change that is relevant to you to ensure that no changes are lost.
End date	marks the other end of the time frame. The **No end date** option ensures that also future changes and future test information will be shown, i. e. this option keeps the end date always at the current time.
Coverage Sources	are logical containers for different kinds of coverage, e. g. »Manual Test« or »Unit Test« or »Test Server 3«. Whenever uploading coverage to Teamscale, a coverage source has to be specified. Here you can select the coverage you want to consider in the Test Gap analysis by simply ticking the corresponding check box, or selecting all coverage sources.

Figure 3.38. Configuring a TGA Dashboard Widget

The resulting treemap should look like the one depicted in Figure 3.37. On this treemap, *gray* rectangles correspond to methods that were not changed since the baseline you entered. In contrast, *orange* methods were changed, while *red* methods were added since the baseline. If a method is depicted *green*, it was also executed since its last modification.

Figure 3.39 helps to get a better understanding of how the selected baseline and end date influence what is shown on the Test Gap treemap,

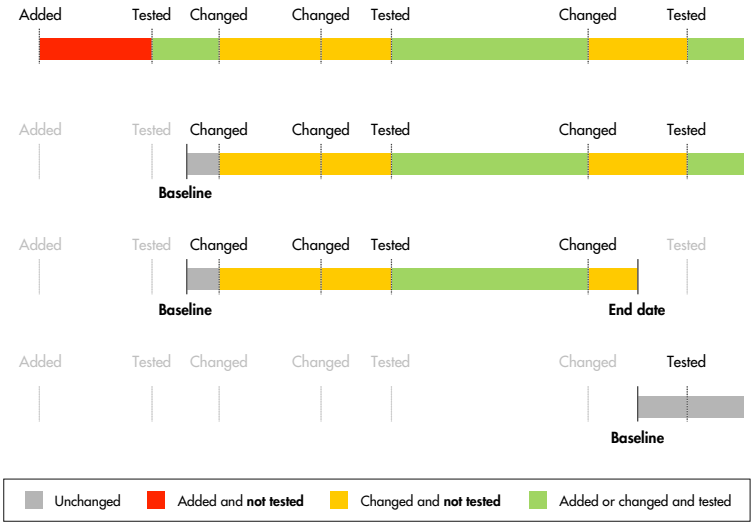


Figure 3.39. Influence of the Selected Baseline on How a Method is Shown on the Test Gap Treemap.

and how methods are color-coded. It shows the history of a single method represented using the previously described colors: The method is first added, then tested, later changed again, and so forth. Because the last version of the method is tested, it will appear green on the treemap.

The second row illustrates, what happens, if a baseline is set at a specific point: The method would still appear green on the treemap.

However, if an end date is set as well, the color might change, as illustrated in row three: Because the end date is set before the last version that was tested, the method would appear orange.

Finally, the last row depicts that a method is shown gray, if the baseline is set after the last change date.

In Figure 3.37, the method `Render` in the class `RadialBlurEffect.cs` is a test gap, because it was changed after the baseline, but not tested since its last modification. If this method is tested in its current version, and the coverage is uploaded to *Teamscale*, the corresponding rectangle will become green, and there will be one test gap less.

3.5.2 Working with Test Gap Treemaps in Practice

A Test Gap treemap answers the question »*which methods have been changed or added since a specific point in time and which of these methods have not been tested yet?*«. These methods, i. e. the untested changes, are called test gaps.

For each test gap, a conscious decision should be made, if the corresponding source code should be tested, or not – e. g. because there

are much more critical parts in the code, and only limited test resources left.

To be able to make this decision, typically two pieces of information are needed: First, which functionality the test gap, i. e. the method, belongs to, and second, which changes were made to this method.

Developers usually get a good understanding of the functionality of a method when they see the source code. As shown in Figure 3.37, hovering over a method in the treemap reveals information about the method under the cursor, such as the class and method name. To see the code behind a method, simply click on the rectangle.

From the source code view, the complete history of the corresponding file can be accessed by pressing the »H« key, as described in more detail in Section 4.1.5. The history of the code shows the corresponding commit messages and, if *Teamscale* is connected to an issue tracker, also points to the change requests from where the changes originate. This information is typically enough to decide how critical a change is.

In our experience, it is most efficient to have a person responsible for development and a person responsible for testing perform this workflow together.

To get just a quick overview of the test status of a system, there is a percentage in the upper right corner of the Test Gap treemap. This denotes the Test Gap ratio for the code shown on the treemap, i. e. the amount of modified or new methods covered by tests divided by the total amount of modified or new methods. In the example in Figure 3.37, there is a Test Gap ratio of over 77 %, meaning that over 77 % of the modified and added methods shown on the treemap have not been tested in their current version.

It is also possible to download a CSV file containing all the methods and their corresponding test state, i. e. if the method is unchanged, was tested, or is a test gap. To this end, click on  left above the treemap. The downloaded CSV file allows for an in-depth inspection of the remaining test gaps.



4. User Reference

This chapter provides a user reference. It ought to provide a step-by-step guidance to facilitate the usage of *Teamscale*. It describes the different perspectives of the web interface (Section 4.1), provides a guide to set up the IDE integration (Section 4.2) and explains *Teamscale*'s different extension and integration APIs (Section 4.3).

4.1 Web Interface

The web interface is designed with different perspectives, represented in the top header, see Figure 4.1. There is e. g. a **Dashboard** perspective, a **Findings** perspective or a **Metrics Perspective**. Each perspective consists of one or several pages, views and dialogs. Below the perspective header, the project can be selected as well as a specific branch (if branch support has been configured).

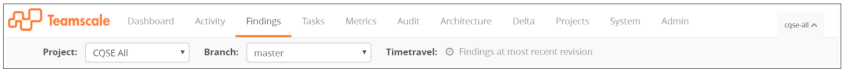


Figure 4.1. Perspective Settings Bar in the Web UI

4.1.1 General UI Features

Based on *Teamscale*'s general philosophy (see Chapter 1), there are several general features independent of the specific perspective. Most perspectives reveal a gray *perspective settings bar* as shown in Figure 4.1. You can use the small arrow in the upper-right corner to hide or show the settings bar.

Project and Branch In the perspective settings bar, you can choose the underlying project for each perspective. Additionally, if branch support is configured, you can select the corresponding branch, e. g. master or a specific feature branch.

Timetravel The perspective settings bar also contains the **Timetravel** feature. As *Teamscale* is connected to the version control system, it has the entire history of a project available. It does not only provide

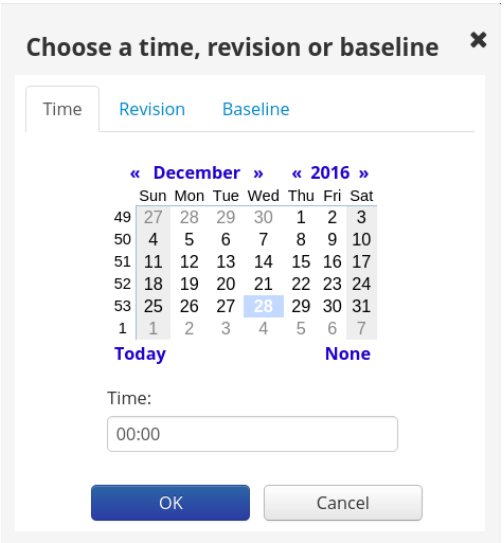


Figure 4.2. Timetravel

the historic version control data but also stores the history of corresponding metrics, findings and other quality data. The **Timetravel** feature allows you to travel back in time and view all available information for a specific date in history. You can select a calendar time, a revision or a baseline, see Figure 4.2. Figure 4.3 describes the different options.

Timetravel Options

Time	You can select a time by picking a calendar date and a time, e. g. December 28th, 2016 at 0:00.
Revision	You can enter a specific revision number from your version control system if you have it at hand.
Baseline	You pick a baseline if you have preconfigured one in *Teamscale*.

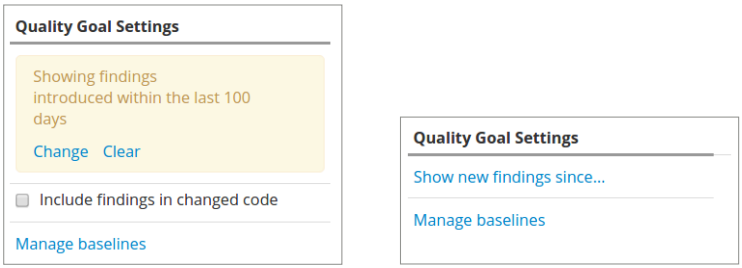
Figure 4.3. Using Timetravel Options

Quality Goals *Teamscale*'s philosophy relies on the concept of quality goals. As the overall set of findings of a grown software system is usually too large to be managed, *Teamscale* provides quality goals as a filter mechanisms reducing the number of findings which are shown in the underlying view or page.

If no quality goal is set, all findings are shown. For the *preserving* quality goal, only *new* findings since a specific *baseline* are shown. For the *improving* quality goal those *new* findings are shown along with existing findings in code which was *changed* since the baseline. For the *perfect* quality goal, all findings are shown. Please refer to Section 1.2.2 to learn more about the use of quality goals in a quality control process.

In the **Findings** perspective as well as in the *Code Detail View* when viewing individual files, the right sidebar provides a configuration for the quality goal. If you want to configure quality goal *preserving*, click on **Show new findings since...** and choose a baseline. To configure quality goal *improving* additionally mark the checkbox **Include findings in changed code**. Note, that this checkbox only ap-

pears after the *preserving* quality goal has already been configured, see Figure 4.4.



(a) Quality Goal On (b) Quality Goal Off

Figure 4.4. Setting a Quality Goal

In the reminder of this section, each perspective is described in detail.

4.1.2 Dashboard Perspective

Use Cases	
	Creating and Viewing Quality Status Dashboards
Views	
	Dashboard View, Dashboard Edit View, Template Management View
Actions	
	Creating Dashboard, Editing Dashboard, Sharing Dashboard, Time Travelling, Managing Templates

After logging in, *Teamscale* puts you into the Dashboard perspective. A dashboard is a way to get a quick overview on a project's current state. Dashboards are configurable collections of widgets, with each widget showing a specific project characteristic, e.g. the system size in lines of code or the distribution of method lengths.

Dashboard View Figure 4.5 shows the **Overview Dashboard** for an example project, depicting various kinds of metrics. The first row in the top left consists of three differently configured numeric metric value widgets, showing the current values for the lines of code, clone coverage, and findings count (Findings Count), respectively, as well as the trend information with regards to the last seven days. Below are two *Assessment Pie Chart* widgets for method length and nesting depth. They show how the code base is distributed into methods that are assessed red, yellow, or green with regards to the respective metric. The *Metrics Trend Chart* widget in the bottom left shows the evolution of metric values over time, namely lines of code and *Number*

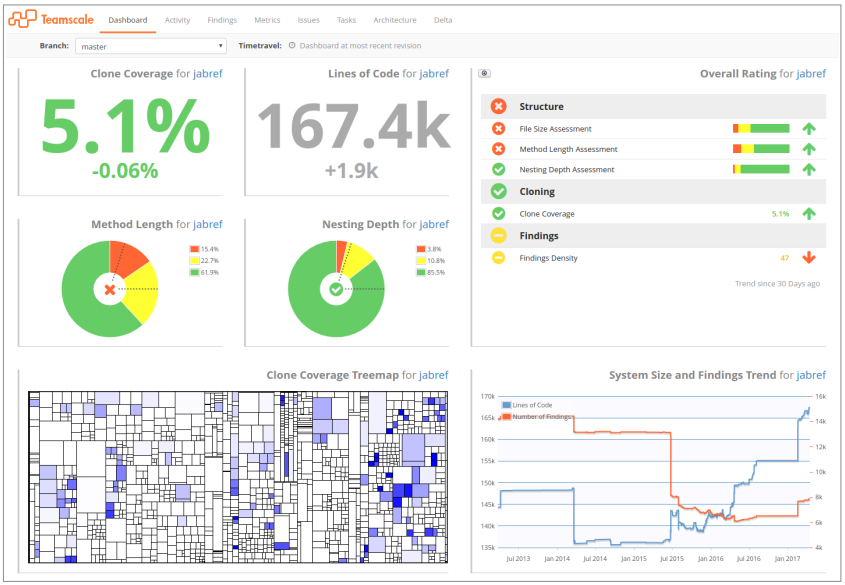


Figure 4.5. Overview Dashboard

of Findings. While there a large variety of Widgets to choose from, *Teamscale* comes with preconfigured templates to provide a starting point for customization.

On the top right, the *Findings Summary Bar Chart* shows how the identified findings are distributed over the available quality indicators and finding groups. Finally, the treemap widget on the bottom right shows how the metric values for clone coverage are distributed over the code base. This makes it very easy to identify hotspots with regard to the selected metric.

4.1.2.1 Action: Creating a Dashboard

To create a new dashboard, you can use the ⊕ button at the bottom of the dashboard sidebar, which allows you to create a new, empty dashboard, or start with a template (Figure 4.6).

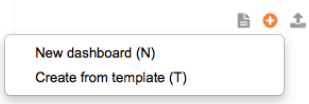


Figure 4.6. Add New Dashboard Menu

You can also import a dashboard, see Figure 4.7. After clicking the ⊕ button, selecting *New dashboard* will present you with an empty dashboard canvas and a list of widgets that can be added to the dashboard. See the following subsection for more information about how to edit a dashboard. Selecting *Create from template* presents you with a dialog to configure a dashboard from template (see Figure 4.8). Here, you can enter the dashboard's name, select the template to create the dashboard from, and select the project and path that should be used by all widgets to display their data for. After confirming with OK, the newly created dashboard will appear in the dashboard sidebar, and you can share or edit it (Figure 4.9).

4.1.2.2 Action: Editing a Dashboard

To edit an existing dashboard, select **Edit dashboard** in the dropdown menu next to the dashboard's name in the sidebar of the perspective. This will lead you to the *Edit Dashboard View*. You will be in the same view right after creating a new dashboard as described in the previous Subsection.

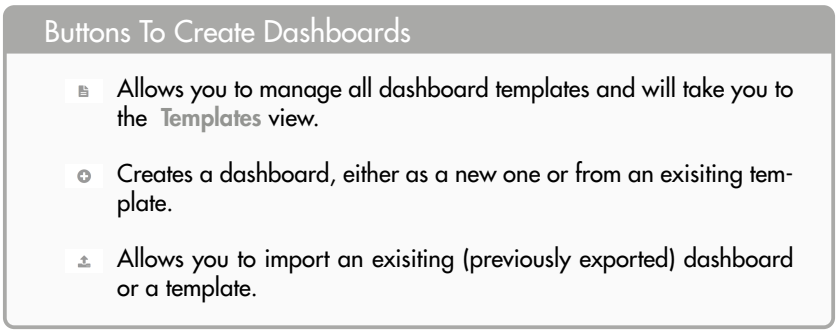


Figure 4.7. Buttons in the Sidebar of the **Dashboard** Perspective

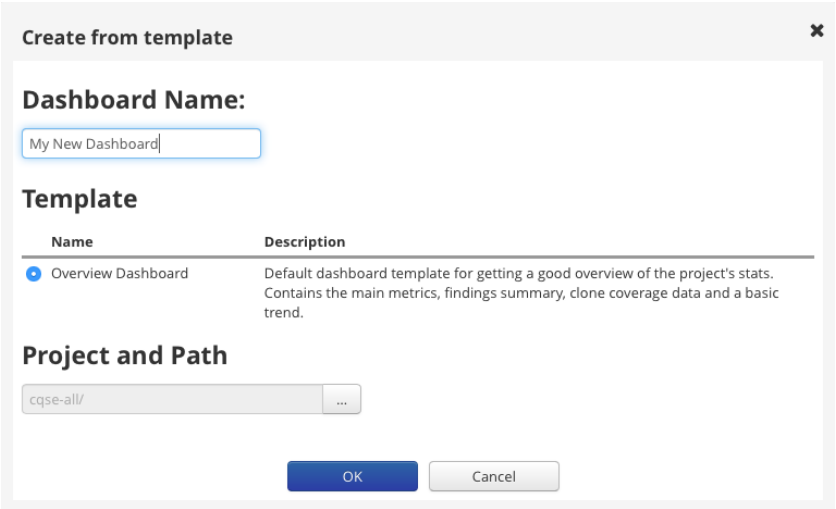


Figure 4.8. Create Dashboard from Template Dialog

Edit Dashboard View Once you are in the dashboard edit mode , the dashboard becomes editable, and the sidebar changes to a list

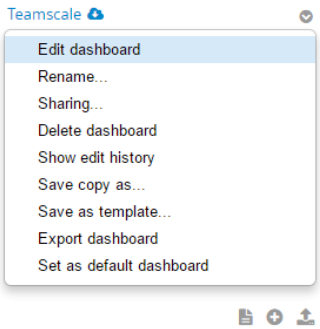


Figure 4.9. Dashboard Sidebar with Context Menu

of buttons. Clicking either of them adds a new widget of the selected kind to the dashboard. As of *Teamscale* 3.0, there are 26 Widgets to choose from, see Figure 4.10. All widgets are also listed in Appendix A. Which widgets to choose strongly depends on the particular use case.

Most widgets are configurable to a large degree, e.g. by selecting which metric(s) to show, which path to consider, the applicable timespan and many more options. Each widget on the canvas is enhanced with a row of buttons to enable configuring, duplicating, or deleting a widget (Figure 4.11 and Figure 4.12). To move a widget across the dashboard, you can drag it via the orange bar on the left. Resizing is possible using the orange corner on the bottom right (see Figure 4.11).

Widgets differ widely in the parameters they provide. Therefore, we illustrate the general concept rather than each individual widget here. Figure 4.13 shows the *Numeric Metric Value* widget's parameters as an example. All widgets share the **Title** and **Path** parameters. These can be used to alter the widget's title and change the project and path for which to display the data. If the path should not be shown on the

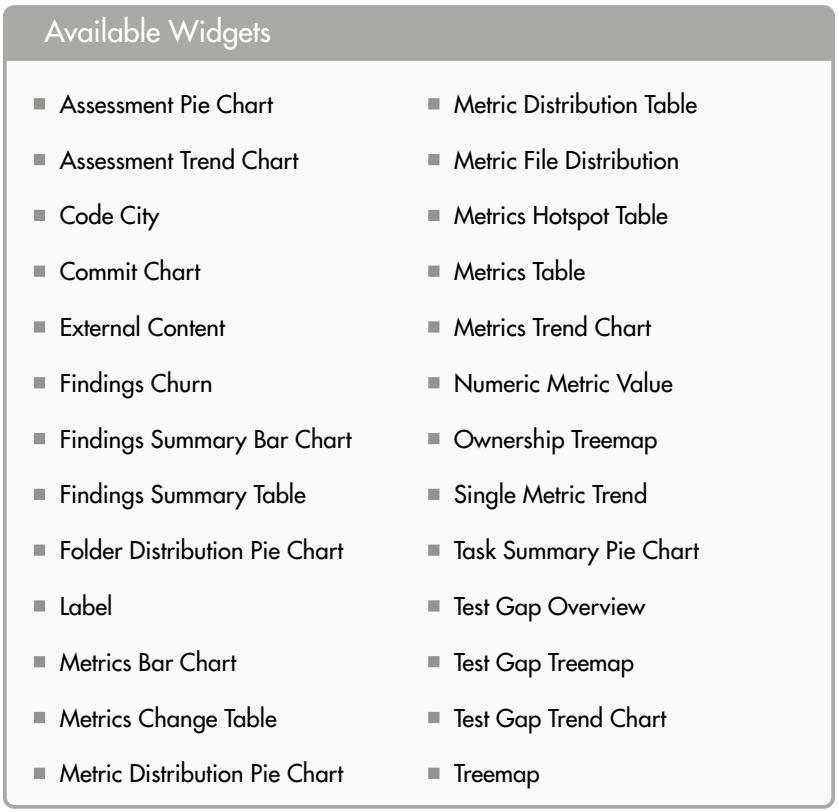


Figure 4.10. List of Widget Types

widget, you can hide it using the check box. Further parameters are provided on a per-widget basis. Usually, they provide tooltips with further information on how to use them when you hover over the parameter name. When you are done editing the widget parameters, OK closes the dialog and applies the changes to the widget.

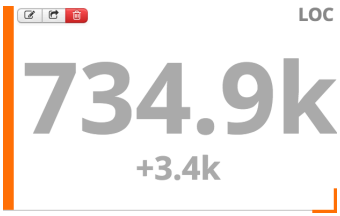


Figure 4.11. Dashboard Widget in Edit Mode

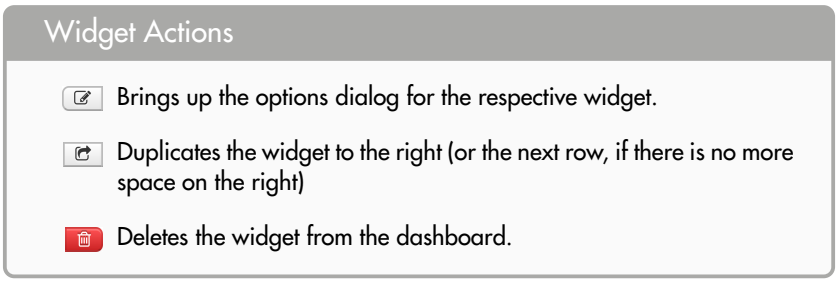


Figure 4.12. Buttons to Edit a Widget

4.1.2.3 Action: Saving and Sharing a Dashboard

When you are done editing a dashboard, you can save it via the Save or Save as buttons on the bottom of the sidebar (Figure 4.14). The former will overwrite the current dashboard, while the latter will create a new dashboard.

Once your dashboard is configured to suit your needs, you may want to share it with you team members. *Teamscale* makes it easy to do just that. A click on Sharing... in the dashboard's context menu in the sidebar (cf. Figure 4.9) brings up the *Sharing View*, where you can configure access rights for individual users or groups. Figure 4.15

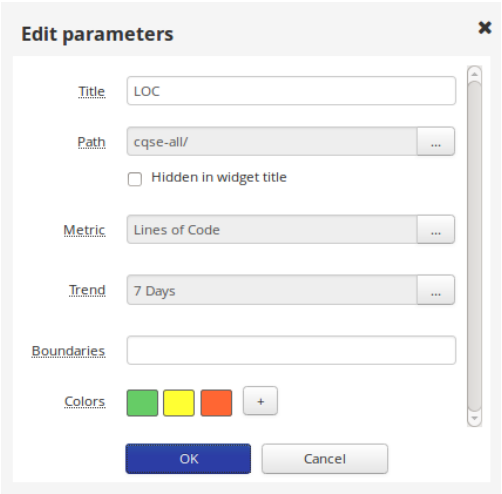


Figure 4.13. Edit Parameters for a Numeric Metric Value Widget

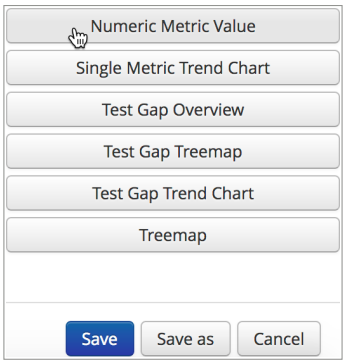


Figure 4.14. Saving a Dashboard

shows the dialog to add a new group. Notice that the input fields for users and groups provide auto-completion as you type, so in order to share your dashboard with a group called »CQSE Developers«, you can just type »Dev« and select the group from the suggestions.

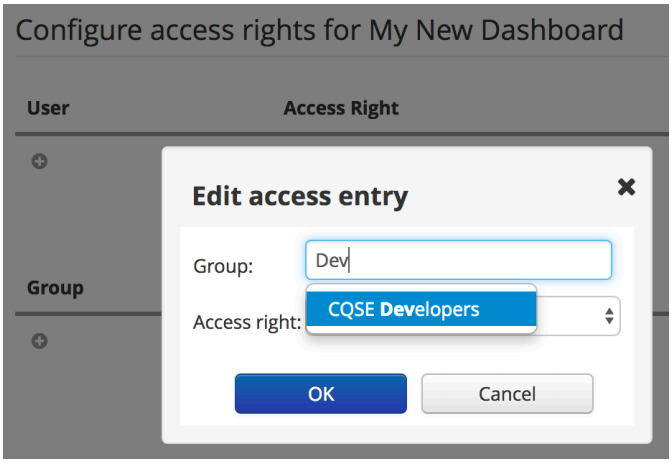


Figure 4.15. Dashboard Sharing

For every user or group you add to your dashboard, you can select whether they should be granted read or write access. When you share your dashboard with colleagues, it will appear in their dashboard sidebar. In your sidebar, private dashboards are shown with , while shared ones have a next to them. Other users' dashboards shared with you appear with a .

4.1.2.4 Action: Timetraveling in a Dashboard

In general, dashboards can not only show a system's current state, but can go back to any point in its history. The **Dashboard** perspective has the timetravel feature available as described in Section 4.1.1. After selecting a date in the **Timetravel** dialog, the dashboard will display the system's quality status at the selected point in time. Setting the timetravel will update the content of all widgets in the selected dashboard. It will not be transferred when you switch dashboards.

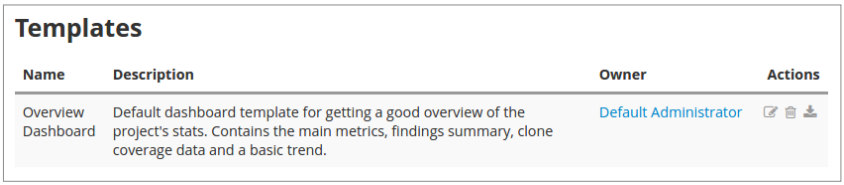


Figure 4.16. Template Management View

4.1.2.5 Action: Managing Templates

The dashboard perspective also allows you to manage all templates. You can click on at the bottom of the right side bar in the *Dashboard View*. This will take you to the *Template Management View*.

Template Management View Figure 4.16 shows the *Template Management View*. It currently contains only one dashboard, the default **Overview Dashboard** along with a short description. To create new templates, you can go back to the *Dashboard Overview View* and use the dropdown menu right next to an existing dashboard's name in the right side bar. This drop down menu has already been shown in Figure 4.9. It contains the option **Save as template** which will create a new template based on the layout from the existing dashboards. The *Template Management View* further allows you to edit, delete and export a template with the buttons next to each template.

4.1.3 Activity Perspective

Use Cases	
	Viewing the History and Inspecting Single Commits
Views	
	Activity View, Commit View, Issue View
Actions	
	Browsing History, Inspecting Commit

For projects with a connector to a version control system, the **Activity** perspective provides an overview of the history. When you open the **Activity** perspective, you will first see the *Activity View*.

Activity View Figure 4.17 shows with an example how the history is presented in the *Activity View*. The view visualizes each commit to the repository and offers seamless scrolling through the history. In the given example, branch support was enabled for the underlying GIT repository. Hence, for the given branch, *Teamscale* shows *all* commits of *all* parents. The history of the branch and all branches that were merged into it is visualized on the left side of the commits.

On the right side, you can click on **Repository Details** to inspect the details of the analyzed history, see Figure 4.18. It shows you the number of changes performed on the repository within the last 24 hours or the last 7 days, for example. Note that only changes performed on the scope of your project are counted. The overall repository might, of course, have more changes. Additionally, the

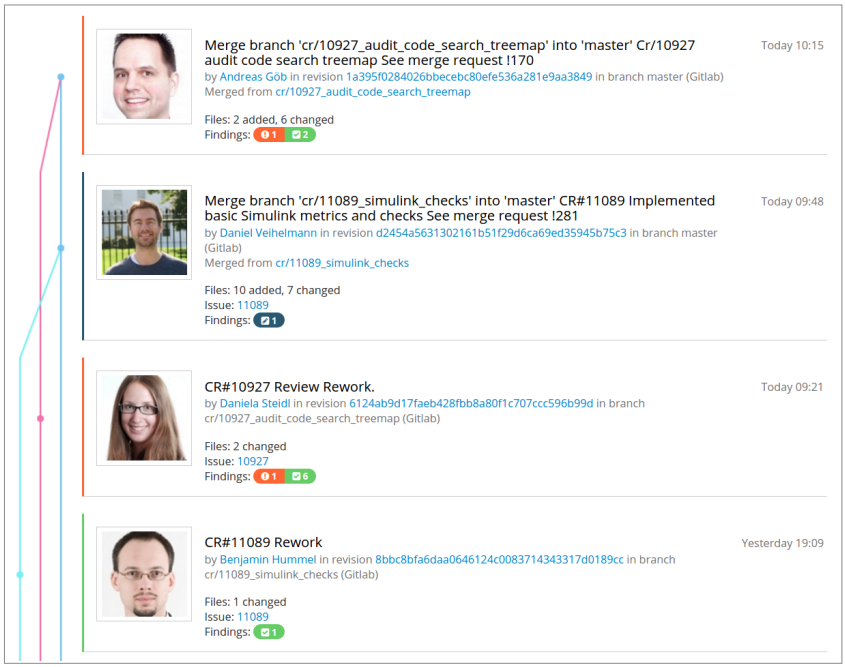


Figure 4.17. History of Commits

time range of the changes is indicated. The **First change** represents the first change within the history time interval as configured in your project configuration. For example, if your project's history dates back to the year 2000 but the project configuration indicates January, 1st 2016 as start revision, then you will see the latter as date of **First change**.

4.1.3.1 Action: Browsing the History

To browse the history, you can either just scroll through the list of commits or you can use the **Timetravel** feature to jump back in time.

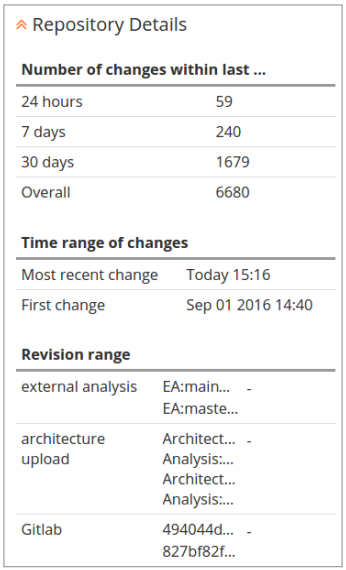


Figure 4.18. Repository Details in the Activity View

If you set your timetravel to January, 1st 2016, then all commits after this date will be hidden and your stream of commits starts with the last commit on January, 1st, 2016. From there on, you can scroll backwards again.

Additionally, several filter options are available to configure which commits are shown or hidden, see Figure 4.19. You can filter commits without findings churn or hide external upload commits. Figure 4.20 shows such external upload commit created by the nightly build to upload test coverage data.

The **User Filter** offers you to enter a user name. After entering the first letter, a drop down menu facilitates selecting the correct user. Afterwards, only commits from the selected user will be displayed.

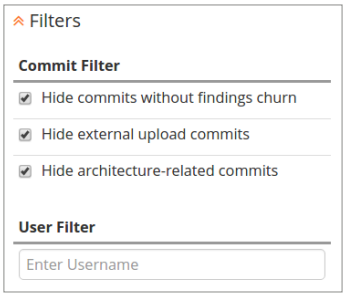


Figure 4.19. Commit Filters

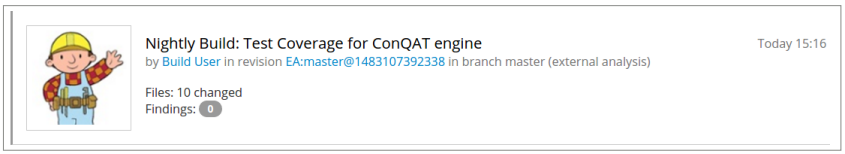


Figure 4.20. External Upload Commit from Nightly Build

4.1.3.2 Action: Inspecting a Single Commit

From the *Activity View*, you can also inspect single commits in more detail. As a short summary, the commit is already annotated with the findings churn, showing the impact of the commit on the system's quality. If you click on the commit message, you will be directed to the *Commit View*.

Commit View The *Commit View* repeats the information about the commit message, the author, the revision number and the branch (if branch support is enabled). Additionally, it links to a connected issue tracking system, if one has been configured. Following the link, you will see the *Issue View* as depicted in Figure 4.21. From there, you can also navigate to the corresponding issue in the issue tracker.

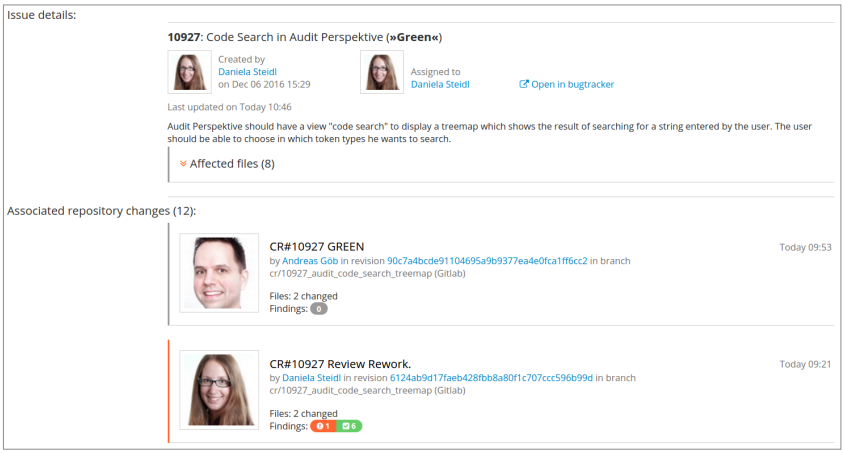


Figure 4.21. Issue View

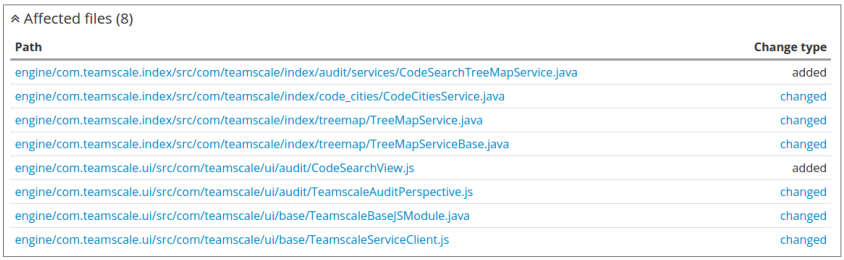


Figure 4.22. Affected Files By a Commit

Furthermore, the *Commit View* provides you information which files were affected by the commit (see Figure 4.22). The **Change Type** indicates you whether the file was **added** or **deleted**. If it was **changed**, you can use the link to jump into the *Code Comparison View*. This view will show you a comparison of the current version of the file and the oldest version prior to the commit. The view itself is described in Section 4.1.5.

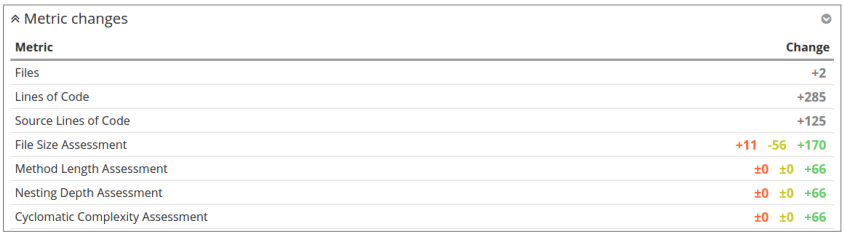


Figure 4.23. Metric Changes for a Commit

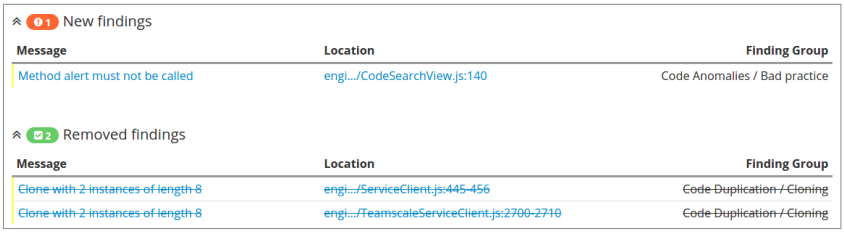


Figure 4.24. Findings Churn of a Commit

Underneath the affected files, you can expand the **Metric Changes** to see the impact of the commit on individual metrics. Figure 4.23 shows the corresponding table. Furthermore, the findings churn of the commit is indicated by a list of new and removed findings, see Figure 4.24. You can either click on the message of a finding to jump into the *Findings Detail View* of the **Findings** perspective or use the link to the source code location to jump into the *Code Detail View*.

4.1.4 Findings Perspective

Use Cases	
	Inspecting and Filtering Findings
Views	
	Findings Overview View, Findings Blacklist View, Findings Detail View, Findings Introduction View, Baseline Management View
Actions	
	Navigating All Findings, Inspecting One Finding, Creating a Baseline, Adding a Finding to a Task

Th main use case of the **Findings** perspective is to inspect the findings of a system. Several filter mechanisms help to reduce the set of findings to a manageable size. For each finding, detailed information about its introduction (and removal) are available.

Findings Overview View When opening the **Findings** perspective, the *Findings Overview View* shows a table with all findings for a *Teamscale* project, see Figure 4.25. For each finding, the table provides information about the finding's message, the finding location, the finding group as well as the introduction date.

4.1.4.1 Action: Navigating All Findings

For grown software systems, the table of the *Findings Overview View* usually contains a very large number of findings. To better inspect and

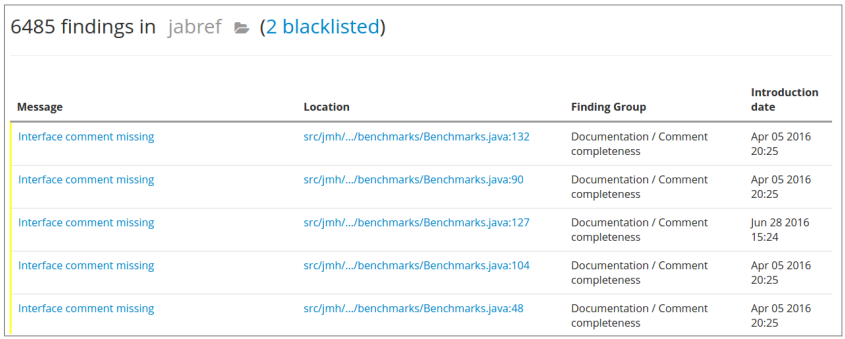


Figure 4.25. Findings Overview Table

navigate them, the sidebar (Figure 4.26) provides several different filter mechanisms and navigation helpers.

First, with the **Quality Goal Settings**, you can configure the underlying quality goal of the project. If a quality goal is set, only findings relevant to the quality goal will be shown in the table. The table will then be annotated, e. g. with **since Dec 06 2016 00:00**. Configuring the quality goal as well as its implication was described in Section 4.1.1. The **Quality Goal Settings** also provide the opportunity to manage the baselines which will be explained in a later action.

Second, the **Finding Filter Settings** give you the opportunity to use an arbitrary regular expression. Hitting »Enter« will apply the regular expression to the finding's message as well as to its location. The table then shows only findings matching the regular expression. For example, entering »test« will show findings with the message »Avoid ignoring JUnit tests«.

Third, **Show Assessments** offers to filter findings based on the finding severity. You can either show only red findings, only yellow findings

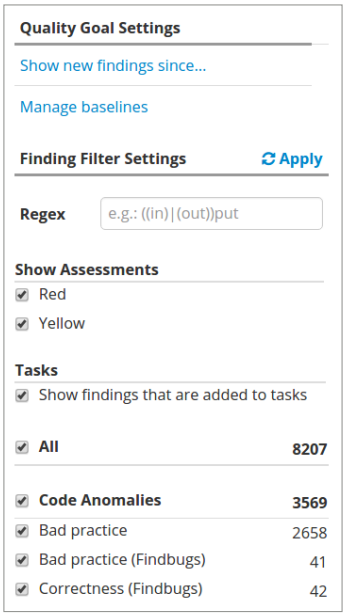


Figure 4.26. Filter Options

or both. Under **Tasks**, you can configure to exclude findings which have been added to a task already.

Fourth, the rest of the sidebar can be used to filter based on finding category and finding group. After configuring the check boxes, you have to use the **Apply** button in order to apply the filters to the table.

Furthermore, the content of the table can filtered based on a path. By clicking on in the table header, you can select subfolders of your project in a dialog opening up . After the selection, the table will show only findings from the specific subfolder. If you are working with an architecture definition, you can also select the architecture and pick a component or subcomponent. Then only findings in files will be shown that map to the selected component.

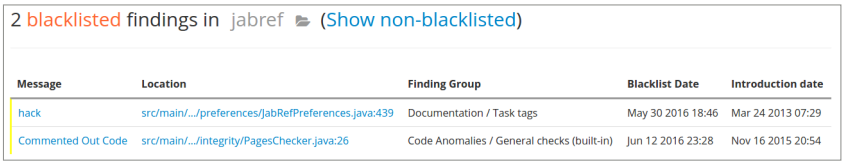


Figure 4.27. Findings Blacklist View

Additionally, the table can be sorted by the values of all four columns. Clicking on the header of one column will sort in descending order, i. e. showing the newest introduction dates or messages starting with »A« first. Clicking the column header again will reverse the order.

Findings Blacklist View Additionally, the table provides a link in its header to the findings blacklist. Clicking on blacklisted will open the *Findings Blacklist View*, Figure 4.27. The view is designed in analogy to the *Findings Overview View*. The table additionally shows the **Blacklist Date** for each finding denoting the date on which the finding was set on the blacklist. In case branch support was enabled for the underlying project, a finding is blacklisted independently of a branch. Hence, if a finding is put on the blacklist, it will not be shown anymore on any branch.

4.1.4.2 Action: Inspecting a Finding

The table provides several possibilities to obtain more information about a single finding. For example, clicking on the finding location will open the *Code Detail View* and show the source code file containing the finding. Clicking on the finding's message opens up the *Finding Detail View*.

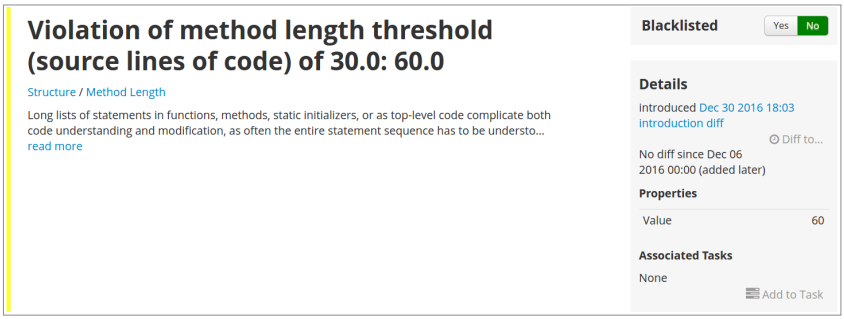


Figure 4.28. Findings Detail View

Findings Detail View This view provides you detailed information about a single finding, see Figure 4.28. It shows you the finding's message again as well as more detailed information about the impact of this type of finding on code quality. On the right, you can set the finding on the blacklist if you consider it to be irrelevant and you wish to no longer see it.

The **Details** provide you more information about the finding. You can view, e. g. its property values or add it to a task which is described in the following in a separate action. Often, you might be curious how a certain finding was introduced. Clicking on the blue date will open up the *Findings Introduction View*.

Findings Introduction View This view shows you how a certain finding was introduced. It comprises a comparison view of the last revision before the finding was introduced on the left an the introducing revision on the right, Figure 4.29. In green, lines are shown which were added. Blue denotes changes to existing lines and red indicates removed lines (not shown in the picture).

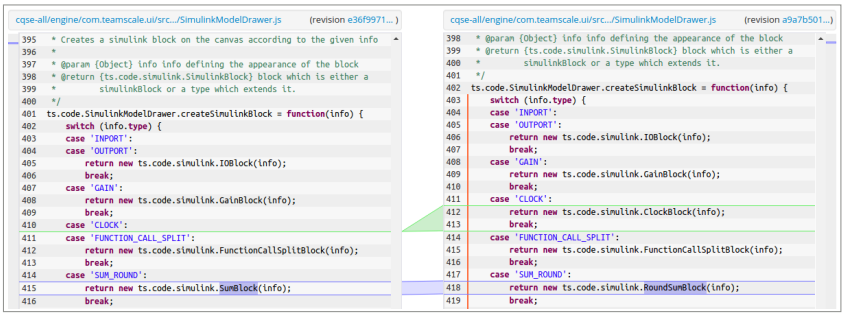


Figure 4.29. Introduction Diff View for a Finding

Back in the *Findings Detail View*, you can also use the Diff to... button and manually choose an arbitrary revision to compare against. This will show you the differences between the revision in which the finding was introduced and the revision you have just chosen.

4.1.4.3 Action: Creating a Baseline

As described in Section 4.1.1, *Teamscale* provides the concept of quality goals to manage potentially large amounts of findings. Quality goals *preserving* and *improving* involve defining a baseline – a point in time after which a finding is considered to be *new*.

To create a new baseline, the sidebar in the *Findings Overview View* provides the link **Manage baselines** under the heading **Quality Goal Settings**. The link opens the *Baseline Management View*.

Baseline Management View This view serves to manage baselines, see Figure 4.30. It contains a table listing all existing baselines. Initially, this table will be empty.

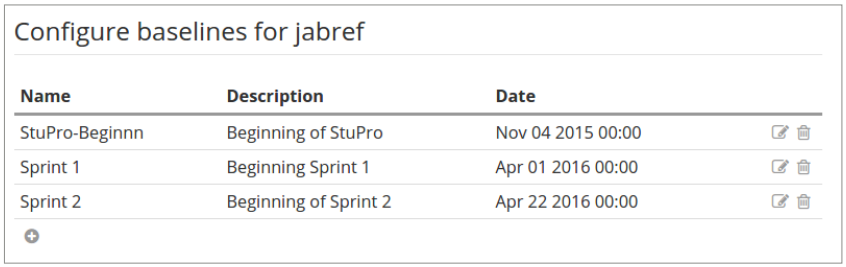


Figure 4.30. Baseline Management View

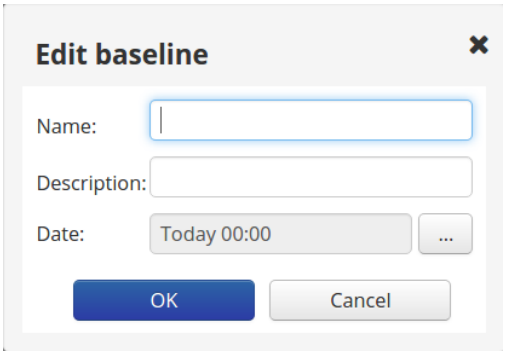


Figure 4.31. Adding A Finding To Task

With the ⊕ symbol, you can add a new baseline which will open up the *Edit Baseline Dialog* as depicted in Figure 4.31. You can create a baseline with a name and a date and a description to it. The same dialog will open up when you use the ✎ icon in the table to edit an existing baseline.

4.1.4.4 Action: Add Finding to Task

Teamscale comes along with its own task management. The **Findings** perspective provides you the opportunity to add findings to a task to track them[1].

In general, you can either add a single finding or a group of findings to a task. The *Findings Overview View* provides the blue link **Add findings to task** on the top right of the findings table. With this link, you can add all selected findings which are currently displayed in the table to a task. Note, that there is a size limit. Hence, you can add not more than 300 findings to a task. The link will open the dialog as depicted in Figure 4.32.

Alternatively, you can navigate to the *Findings Detail View* and use the button Add to Task to add a single finding to a task. This will open the same dialog. In the dialog, you can either create a new task or add the findings to an existing task.

[1] If you are curious how to use *Teamscale* in your task management in addition to your likely already existing issue tracker, please refer to our blog https://www.cqse.eu/en/blog/teamscale-tasks/

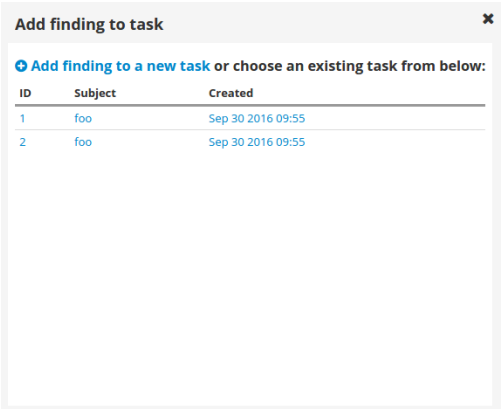


Figure 4.32. Adding A Finding To Task

4.1.5 Metrics Perspective

Use Cases	
	Viewing Metrics of Your System
Views	
	Metrics Overview View, Code Detail View, Code History View, Code Comparison View, Code Dependency View
Actions	
	Navigating the Code Structure, Configuring and Viewing the Metrics Table, Viewing a Code File, Viewing the History of a Code File, Viewing the Dependencies of a Class

The **Metrics** perspective is the central place for metrics of your system. The perspective provides two different aggregation of the metrics. In the *Code* section, the metrics are aggregated based on the file directory hierarchy. In the *Architecture* section, they are aggregated based on architectural components. Additionally, the perspective also shows *Non-Code* metrics, such as build status or test execution.

Metrics Overview View When activating the **Metrics** Perspective, *Teamscale* shows the *Metrics Overview View* (see Figure 4.33). This view contains up to three sections which present two different forms of aggregation and, additionally, the non-code metrics:

- **Code** In this section, you can view metrics aggregated based on the code directory structure.
- **Architectures** For each architecture file under quality control, this section of the perspective shows metrics aggregated based on components from an architectural definition.
- **Non-Code** You can configure *Teamscale* to show non-code metrics like build execution times (see Section 3.1.1). If you are using this feature, this section of the perspective will provide a quick overview of such metrics. Please note that the section will be hidden if there were no non-code metrics configured.

Each of these three sections can be used as a starting point to navigate deeper into the metrics' hierarchy. From the code table, you descend into folders, whereas the architecture section table allows you to descend into subcomponents of your system. Both ways will, at their lowest level, result in an actual code file in the *Code Detail View*. To see this navigation in action, simply click on a folder name of the **Code** section to load the content of this package. In the following paragraphs, we take a closer look at the views you will encounter

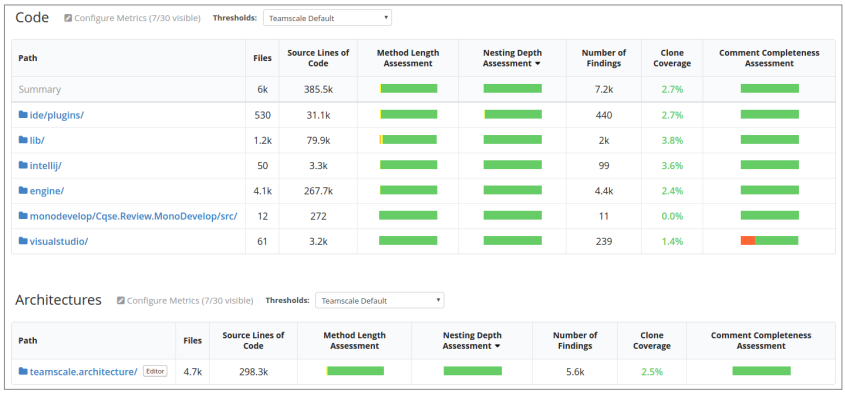


Figure 4.33. The Metrics Perspective

when you are browsing through the metrics. All of the three sections mentioned above work in an analogous way.

In the **Code** section, you can navigate and analyze the code in two dimensions – with respect to directory structure and time. It consists of three main elements:

- **Metrics Overview Table:** The *Metrics Overview Table* consists of one row for each file or directory in the current directory. Furthermore, it consists of several columns showing metric values for these files and directories. The first line of the table contains an aggregated value for the metric in the respective table column (e.g., the total number of files in the directory and all sub-directories).
- **Threshold selector:** The threshold selector lets you choose which of the configured threshold configurations should be used to assess metric values. You can create and edit threshold configurations in the **Project** perspective (see Section 4.1.9).

In addition, when you start to browse deeper into the directory structure of your code base, you will encounter two additional elements:

- **Bread crumbs:** The bread crumbs show the current position in the code hierarchy, i.e. the current directory in a file system-like structure. Clicking them navigates to ancestor folders of the hierarchy. The *Metrics Overview Table* shows the content of the current directory.
- **View Findings Button** A click on View Findings will show you all findings stemming from the files you are currently viewing.

4.1.5.1 Action: Navigating the Code Structure

As mentioned before, each of the tables in the **Metrics** perspective – *Code, Architectures, Non-Code*– can be used as a starting point to navigate deeper into the respective directories. Clicking on a directory will update the *Metrics Overview Table* with the content inside the selected directory. Clicking on a file displays its code in the *Code Detail View*. Navigating back to a top folder can be done by the bread crumbs.

While navigating the code structure, the *Metrics Overview Table* can also be configured. You can click a column head to sort all entries according to the metric values in the corresponding column. For example, you can sort directories and files by their size in lines of code. You can also configure which metrics are shown in the table (see Section 4.1.5.2). Which metrics are available depends on the analysis profile which is configured in the **Project** perspective (see Section 4.1.9).

The table combines the directory structure with change information from its history. The column **Last Change Date** shows you when the respective directory or file was changed last. By clicking the value in

column, you jump to the Activity Perspective (see Section 4.1.3) and the corresponding activity, i.e. the corresponding commit, is shown.

You can also view each directory at a specific point in time. Use the Timetravel feature as displayed in Figure 4.1 to choose a date, baseline or revision. The content of the directory will be updated to the selected timestamp.

When you are browsing through your source code, it is possible to view all findings for the directory currently visible. Simply click View Findings above the table or press »F«. All findings for the current directory will be displayed in the Findings perspective (see Section 4.1.4).

4.1.5.2 Action: Configuring the Metrics Tables

For each table visible in the Metrics Perspective, you can choose which aggregated metrics should be shown. You can configure the visible metrics by clicking on Configure Metrics right above the corresponding table. This will open up a dialog as depicted in Figure 4.34. In addition, you can quickly remove columns by hovering over a column head and clicking the »x« that will appear in the top-right corner. Which metrics are available depends on the analysis profile which is configured in the Project Perspective (see 4.1.9).

For each available metric, you can view its history trend, as well as a treemap that visualizes the metric distribution across all (code) files. To view them, click on a table cell showing a particular metric. This will display the history and the treemap of the metric, see Figure 4.35. For the history trend, a time range can be selected and viewed in Delta Perspective by clicking the arrow and choosing option View selected range in Delta Perspective from the context menu (see the use case description in Section 3.4.2). A click on a

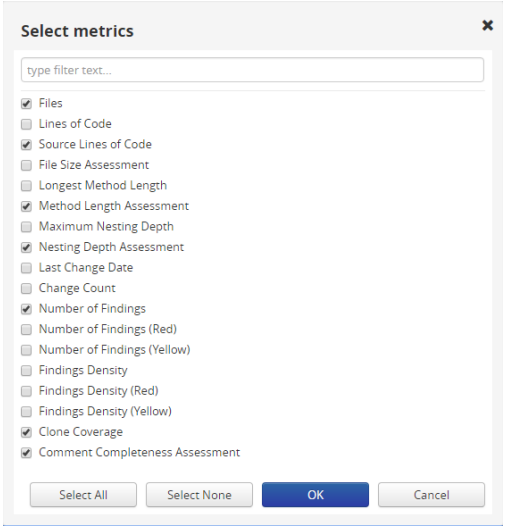


Figure 4.34. Select Metrics Shown in Metrics Table

rectangle in the treemap lets you view the respective file in the *Code Detail View*.

4.1.5.3 Action: Viewing a Code File

Often, browsing a directory or component structure ends when a certain source code file was found. Each code file can be viewed in the corresponding *Code Detail View*.

Code Detail View The *Code Detail View* (see Figure 4.36) appears after clicking on a file in the *Metrics Overview Table* and shows the source code text together with additional information. It consists of the *Bread crumbs* and a More button, the code itself and a sidebar. The sidebar contains the Quality Goal Settings, the Metrics

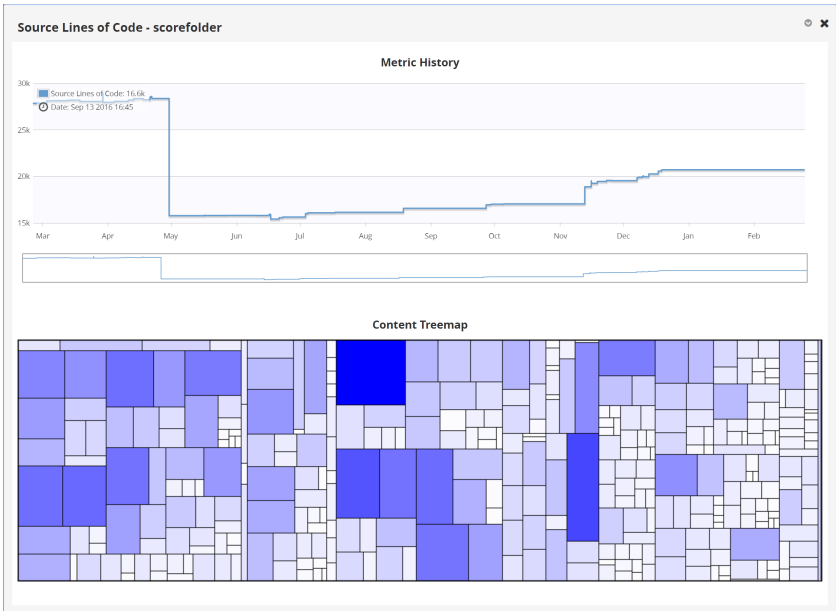


Figure 4.35. History and Treemap for Metric SLOC

Table as well as information about the Recent Committers (not pictured), Coverage Highlighting (not pictured) and a Findings List (not pictured). Each element is described in more detail in the following:

- **Timetravel UI:** It shows the date for which the code is displayed.
- **Bread Crumbs:** It display the position of the current file in the code hierarchy.
- **More Button:** It allows you to perform contextual actions (e. g. to download or pretty-print a file) which are described below.
- **Code Text:** This renders the code of the current file with syntax highlighting. The orange lines at the beginning of a code

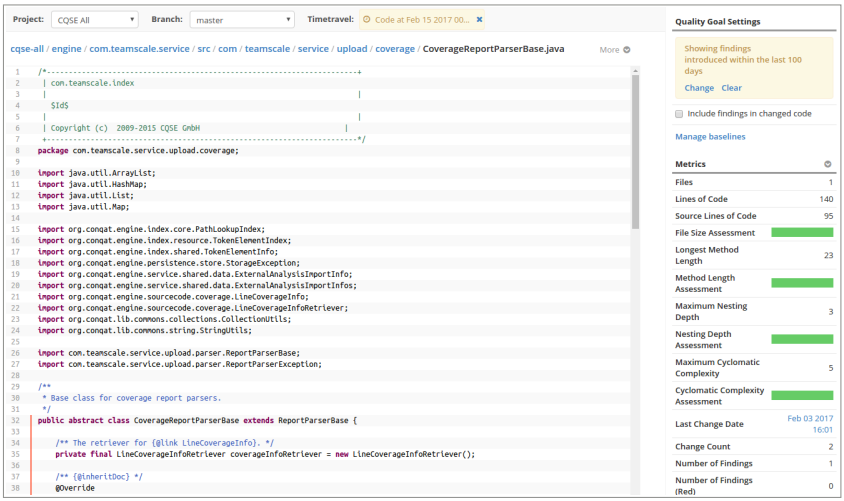


Figure 4.36. Code Detail View

line are finding markers. They indicate that a finding for this line exists. Hovering over the orange line displays a finding description which can be used to jump to the **Findings** perspective (see Section 4.1.4) and see more details.

- **Quality Goal Settings:** The quality goal setting lets you decide which filters to put on findings for code of the current file, i. e. which findings are (not) to be displayed. Please refer to Section 4.1.1 to learn about the impact of quality goal settings on a view.

- **Metrics Table:** The table presents several metrics together with metric values for the current file.

- **Recent Committers:** This list shows which developers recently changed the current file.

- **Findings List:** This list contains all findings for the current file and can be filtered by clicking on **Show filters**.
- **Coverage Highlighting:** If coverage for the current file exists (i. e. is known to *Teamscale*), covered lines are indicated by a green background color. If available, coverage highlighting can be turned on or off by clicking on the check box **Annotate line coverage**.

Each code file can be viewed in all its revisions available in the history. Clicking the **Timetravel** and choosing a particular date, baseline or revision, shows the code version of this point in time.

To read the source code, developers often mark a specific code element to view all its occurrences. Clicking on a specific code element, e. g. a variable, highlights all occurrences of this variable. A marker appears on the right side of the code text visualizing where in the file the code element occurs (see Figure 4.37).

```
if (UserVisibleHint && LastReportedVisibility != ViewStates.Visible)
{
    OnShowFragment();
    LastReportedVisibility = ViewStates.Visible;
}
else if (!UserVisibleHint && LastReportedVisibility != ViewStates.Gone)
{
    OnHideFragment();
    LastReportedVisibility = ViewStates.Gone;
}
```

Figure 4.37. Highlighted Variable Occurrences in Source Code

The **More** button ⚙ provides several more option. Choosing option **Download file** from the context menu, downloads the current file to the local device. Choosing option **Pretty print/ format view** from the context menu or pressing key »P«, pretty-prints the code. This does not change the original code but just the way it is formatted and displayed in the *Teamscale* UI. Clicking **Resource History** (shortcut: »H«) allows you to view the history of the file at hand (see

Section 4.1.5.4). Moreover, **Show dependencies** (shortcut: »D«) lets you view the (outgoing and incoming) dependencies of the current file (see Section 4.1.5.5).

4.1.5.4 Action: Viewing the History of a Code File

In addition to a particular version of a source code file, you can also use *Teamscale* to browse the history of a file. Whenever you are viewing an individual file in the *Code Detail View*, clicking **More** and then choosing the option **Browse history** from the menu (or pressing key »H«) will show you the history of the file in the *Code History View*.

Code History View This view (see Figure 4.38) presents the history of activities for a particular file. It consists of a list of commits which is ordered according to time. Each activity consists of a commit message, an author, a revision and a branch as well as a date.

The *Code History View* provides several possibilities. Clicking **Show content** of a commit, opens the *Code Detail View* and shows the code in the version of the selected commit. Clicking the commit message or the revision number displays the commit of the activity in the *Commit View* (Section 4.1.3). Clicking on **Compare with ...** of two commits opens the *Code Comparison View* and allows to compare two versions and identify differences.

Code Comparison View This view (see Figure 4.39) shows two files – usually the same file in different versions – next to each other and highlights differences between both. It displays the **Comparison Mode** at the top of the view, the older version of the code on left side and the newer version of the code on the right side.

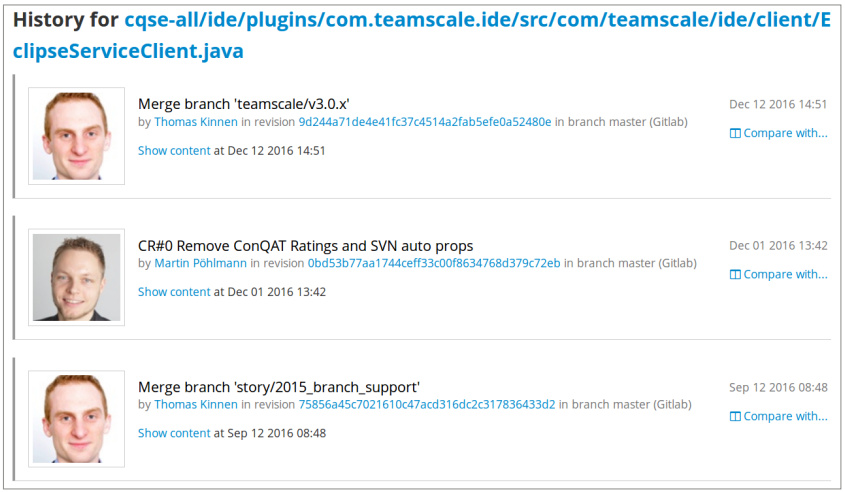


Figure 4.38. Code History View

- **Comparison Mode:** This indicates the current comparison mode. Comparison mode can be line-based, line-based (ignoring white space) and token based. Clicking another radio button changes the comparison mode.
- **Code on the Left:** This presents code of the older file. Additionally, it shows the path and the revision of the older file.
- **Code on the Right:** Similarly, this presents the code, path and revision of the younger file.
- **Visualization of differences:** This shows differences between the code on both sides. Blue differences denote differing code on both sides, green differences denote code only present in the left, older code and red differences denote code only present in the right, newer code.

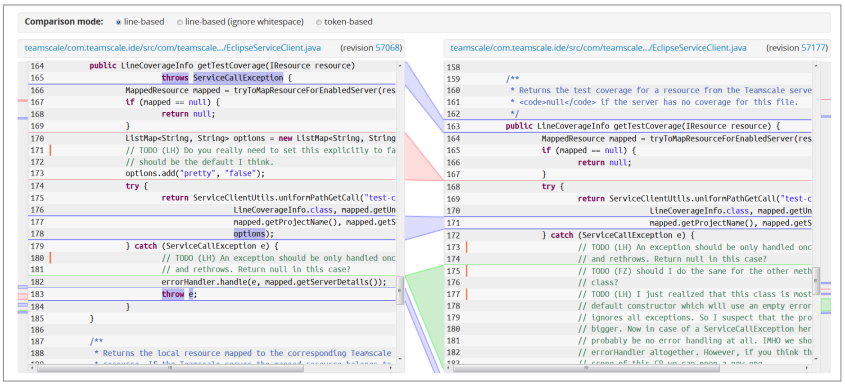


Figure 4.39. Code Comparison View

Scrolling or clicking the visual summary allows to navigate in the code and inspect differences. Clicking the path on top of one side shows only one code version and opens the *Code Detail View*. Clicking the revision of one side shows the revision of in the **Activity** Perspective.

4.1.5.5 Action: Viewing the Dependencies of a Class

In the *Code Detail View*, clicking **More** and choosing option **Show dependencies** from the context menu or pressing key »D«, shows the dependencies of the current file. This will open the *Code Dependency View*. Please note that this only works if architecture analysis is activated in the analysis profile.

Code Dependency View This view (see Figure 4.40) shows outgoing and incoming dependencies of a file on other files as well as frameworks and APIs. Please note that this feature is available only if architecture analysis is activated. The view consists of two lists, namely outgoing and incoming dependencies. Each dependency has



Figure 4.40. Code Dependency View

a path and one or more numbers. The path denotes the file on which the current file depends (in case of outgoing dependencies) or the file which depends on the current file (in case of incoming dependencies). The numbers denote the lines of the current file which refer to the depending file (in case of an outgoing dependency) or the lines in the dependent file which refer to the current file (in case of an incoming dependency).

Clicking on the dependency path opens the dependency file and allows to inspect its code. Clicking on the line number jumps to the code location where a reference is made from or to the dependency file. This will open up the *Code Detail View* with the respective file.

4.1.6 Issue Perspective

Use Cases	
	Evaluating Queries over Issues, Managing Issue Metrics
Views	
	Issue Query View
Actions	
	Query Issues, Display Issue Trend, Manage Issue Metrics

The main purpose of the Issue perspective is to calculate metrics over issues stored in an issue tracker. These metrics can reveal weaknesses in your development process, such as long pending critical bugs, and help you to assess any steps toward process improvement over time. To use this perspective, your project must define at least one issue tracker connector when configuring your project (see Section 4.1.9.2 and Appendix C).

4.1.6.1 Action: Query Issues

The Issue perspective allows you to perform queries against all issues imported into the Teamscale project to obtain lists of matching issues. Hence, you can, e. g. obtain the number of issues matching certain criteria, such as overdue bugs. For this, you navigate to the Issue perspective, which opens the *Issue Query View*.

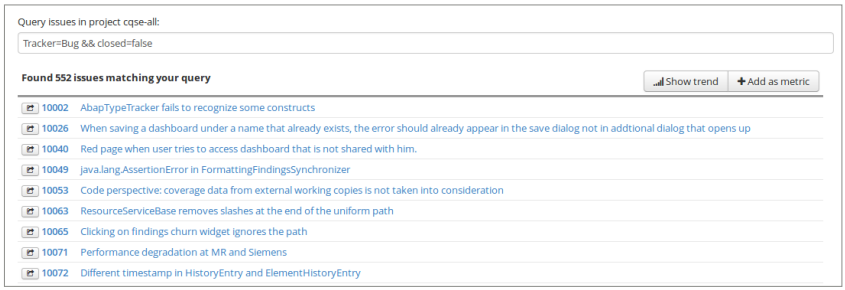


Figure 4.41. Query Area of the Issue Perspective

Issue Query View In this view, you can enter queries (see Figure 4.41). For example, to display all open issues, enter *closed = false* into the query box. After pressing *Enter*, the list of matching issues is displayed below the box. The supported language for these queries is described in Figure 4.42. The input box also provides autocompletion to simplify writing queries.

Each issue in the returned list provides a link to additional details, as well as to the ticket in the originating issue tracker. You should keep in mind, that the default is to query all issues, including closed one. To limit a query to open tickets, add the phrase *closed=false*.

Issue Query Language

Attributes	For each ticket, the default attributes *subject*, *assignee*, *author*, *created*, *updated*, *status*, and *closed* are defined. Additionally, all custom fields defined in the connector configuration are made available as attributes.
Comparison operators	Attributes can be compared to values using the usual comparison operators =, !=, >, <, >=, <=. The order is always attribute, operator, value. You can not compare attributes to each other.
Set query	You can use the keyword *in* together with square brackets to create set queries. An example would be *priority in [high, urgent]*
Logical operators	Comparisons can be connected using the logical operators *&&* (and), *||* (or), and *!* (not). Additionally, parentheses can be used to group subexpressions.
membersOf	The special function *membersOf* can be used together with set queries to check for membership in a Teamscale group (that might again come from LDAP). An example would be *assigne in membersOf('UI Team')*.
State queries	The operator *inState* allows to formulate that a ticket has to be in a given state for a longer time. The operator can contain any subquery and is followed by a comparison (>, <) with a time value. An example would be *inState(closed=false) > 7d* to display all tickets that are open for at least 7 days. The number can be followed by suffixes for days (d), hours (h), minutes (m) and seconds (s). The default when no suffix is given is days.

Figure 4.42. Supported Language for Issue Queries

An example returning all urgent tickets that have not been closed within two weeks would be

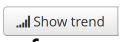
```
isState(closed=false && tracker=Bug && priority in [
    urgent, blocker]) > 14d
```

To limit this to tickets within the UI team, you can change this to

```
assignee in membersOf('UI team') && isState(closed=false
    && tracker=Bug && priority in [urgent, blocker]) > 14d
```

Note that the exact attributes will depend on your issue tracker and the fields that are configured. Also note the subtle difference between including the assignee check in the *inState* part or not. If we would include the assignee check in the inState part, this would cause tickets to not be shown if the assignee changed in the 2 weeks.

4.1.6.2 Action: Display Issue Trend

When an issue query has been executed, you can click on 

in the *Issue Query View*, to display a trend of the number of issues matching the query over time. An example output is shown in Figure 4.43.

4.1.6.3 Action: Manage Issue Metrics

There are certain queries which you might want to execute continuously and see the result regularly without re-formulating the query every time. Therfore, the Issue perspective also allows you to define issue metrics. An issue metric is the numeric result, i. e. the number of tickets matching a query. By defining a metric, the underlying query is saved in *Teamscale* and, thus, can be used continuously.

These issue metrics are displayed in the sidebar, as shown in Figure 4.44. To create such a metric, execute an issue query and click



Figure 4.43. Issue Trend

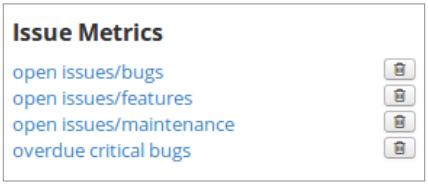


Figure 4.44. Issue Metrics Listed in the Sidebar

+ Add as metric in the *Issue Query View*. After entering a name, the metric will appear in the sidebar. The list of metrics in the sidebar allows you to quickly return to the corresponding query by clicking on its name, or to delete the metric by clicking 🗑. Editing a metric works by recreating a metric with an existing name.

The defined issue metrics are also available in the **Dashboard** perspective (Section 4.1.2) upon configuring dashboard widgets. The configuration of a widget offers you to define the scope of a widget by setting the corresponding path, which usually points to a (sub-) folder or (sub-) component of the system. To use issue metrics instead, use the path *-issues-* within a project. When the metric names

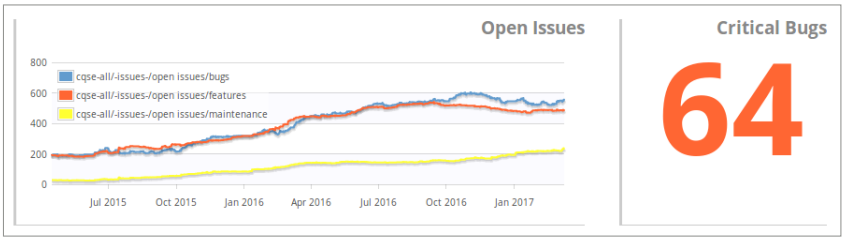


Figure 4.45. Issue Metrics in Dashboard Widgets

include slashes, these are used to defined directories for grouping similar issue metrics. Good widgets for using the issue metrics are the *Numeric Metric Value* and the *Single Metric Trend Chart* widget as shown in Figure 4.45. This allows you to track key indicators for your issue management process and react timely to any changes in the observed trend.

4.1.7 Architecture Perspective

Use Cases	
	Modeling and Analyzing the Architecture of Your System
Views	
	Architecture Overview, Architecture Inspect View, Architecture Edit View
Actions	
	Model Architecture, Save Architecture, View Architecture, Inspect Metrics For Architecture

In the **Architecture** perspective, you can model and analyze the architecture of your system. *Teamscale* provides an architecture conformance analysis to check if the implementation follows the architecture specifications. Therefore, the architecture specifications (.architecture-files) can be modeled using nested components. The specifications can then be committed to the version control system or saved in *Teamscale* itself to be used for continuous conformance checking. To enable the architecture analysis, the architecture quality indicator has to be enabled in the analysis profile of the underlying *Teamscale* project. In the analysis profile, you can also configure if third-party dependencies should be considered in the architecture analysis or excluded from it, see Figure 4.60.

Architecture Overview View When you open the **Architecture** perspective, you will see an overview of existing architecture specifications, see Figure 4.47. This overview will be empty initially if no

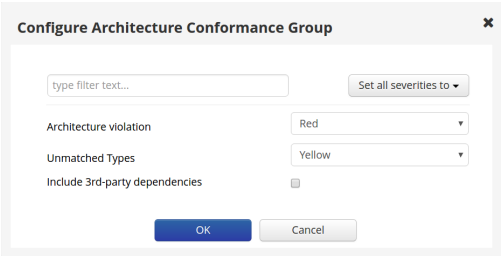


Figure 4.46. Configuring the Architecture Quality Indicator

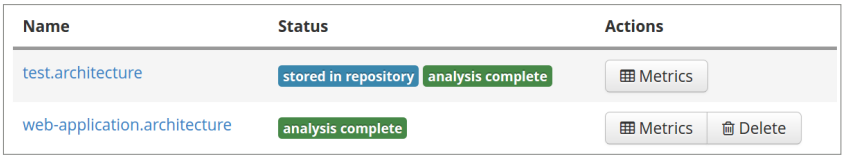


Figure 4.47. Architecture Overview

architecture has been modeled yet. For each existing architecture model, the table shows its status and the available options. For an explanation of all possible status as well as the available action buttons, please see Figure 4.48 and Figure 4.49.

The sidebar provides you four buttons, see Figure 4.50. You can download the dependencies in XML format either as file dependencies or as type dependencies. And you can create a new architecture specification or import an existing one. Before we go into details about how to create a new architecture, we briefly introduce the concept of »architecture« in *Teamscale*, see Section 4.1.7.1.

4.1.7.1 *Teamscale* Architecture Model

The intended architecture can be specified in a simple yet powerful hierarchical component model. The model has few basic concepts:

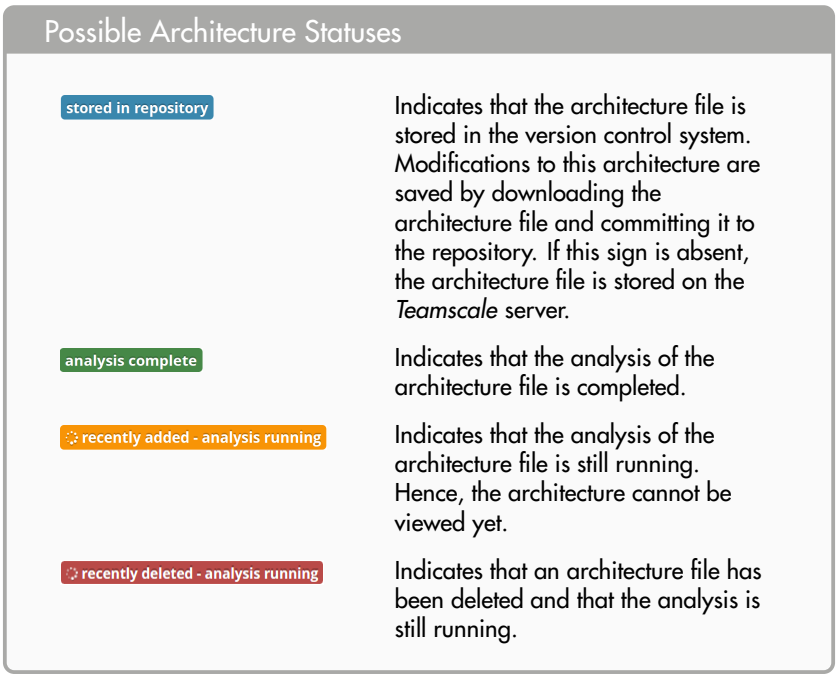


Figure 4.48. Buttons Available in the Architecture Table

Buttons For an Architecture File

Metrics — Shows the metrics for the given architecture similar as in the **Metrics** perspective.

Delete — Deletes the given architecture. This is only an available option if the architecture is stored in *Teamscale*. Architectures stored in the repository can be deleted only by deleting them from the repository.

Figure 4.49. Action Buttons in the Architecture Table

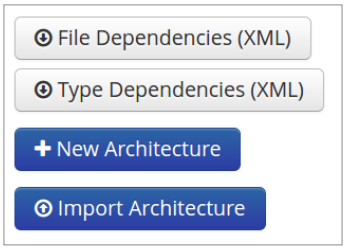


Figure 4.50. Sidebar of the Architecture Overview

- **Components:** The system is modeled as a hierarchy of components. A component has a unique name and mappings and may have subcomponents. Components can have a stereotype PUBLIC indicating that all other components may access this components or a stereotype COMPONENT_PUBLIC specifying that it may be accessed by all other components within the same parent component (e.g. a GUI-library within an UI-layer). This is especially useful for library components that are accessed by many other components.

- **Mappings:** A mapping between code and architecture defines which implementation artifacts belong to which component in the architecture definition. This is specified using Java regular expressions as include and exclude pattern.

- **Architecture Type**: For an architecture, you can choose if the architecture if type- or file-based. For a type-based architecture the full-qualified names of all types (classes, structs, enums etc.) in the code base are used for the mapping. For file-based architectures the path to the source code files are used.

- **Policies:** A policy defines which components may be in a relationship (allow, deny, tolerate) with each other. Each dependency policy has a source and a target component. If components are not connected by a policy-edge, dependencies between them are (implicitly) forbidden. You need to explicitly model allow-policies (green arrows) to allow dependencies. If an allow-policy is specified between two components all of their sub-components may also share dependencies. You can explicitly deny this, by modeling deny-policies (red arrows). Tolerate-policies can be used to accept an explicit set of dependencies between components.

- **Violations**: If additional dependencies are introduced, which contradict the modeled policies, *Teamscale* will mark them as violations of the architecture. Violations can be defined to be treated as regular findings in *Teamscale*. This can be configured while editing the architecture (see Section 4.1.7.2).

- **Orphans**: Types/files that reside within the scope of the architecture, but are not mapped to a component, are regarded as so-called orphans. These orphans can be defined to be findings in *Teamscale*. This is to be configured while editing the architecture (see Section 4.1.7.2).

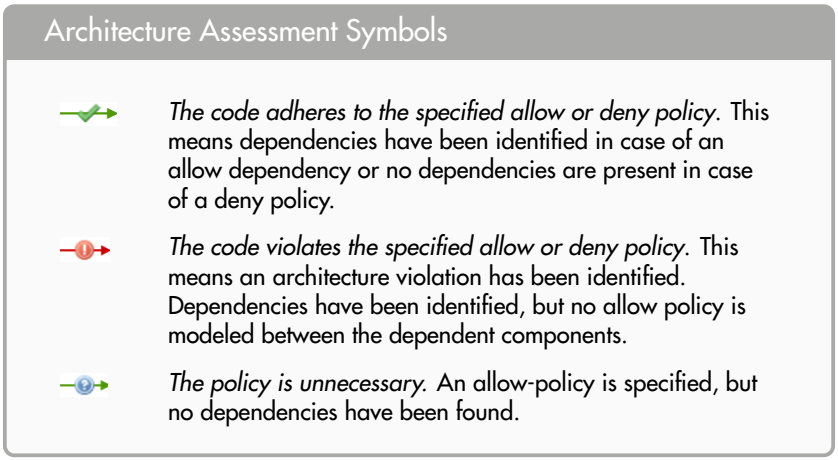


Figure 4.51. Assessment Symbols for Modeled Dependencies

4.1.7.2 Action: Model Architecture

After having gained a basic understanding of the *Teamscale* architecture model in the previous section, we can now start to model our own architecture specification. When you open the **Architecture** perspective and click on New Architecture, the *Architecture Edit View* opens, see Figure 4.52. You will reach the same view upon opening an existing architecture and clicking on Edit Architecture.

Architecture Edit View When creating a new architecture, the editor will be empty. Underneath, a variety of options are displayed on the left and the current orphans are displayed on the right, see Figure 4.52.

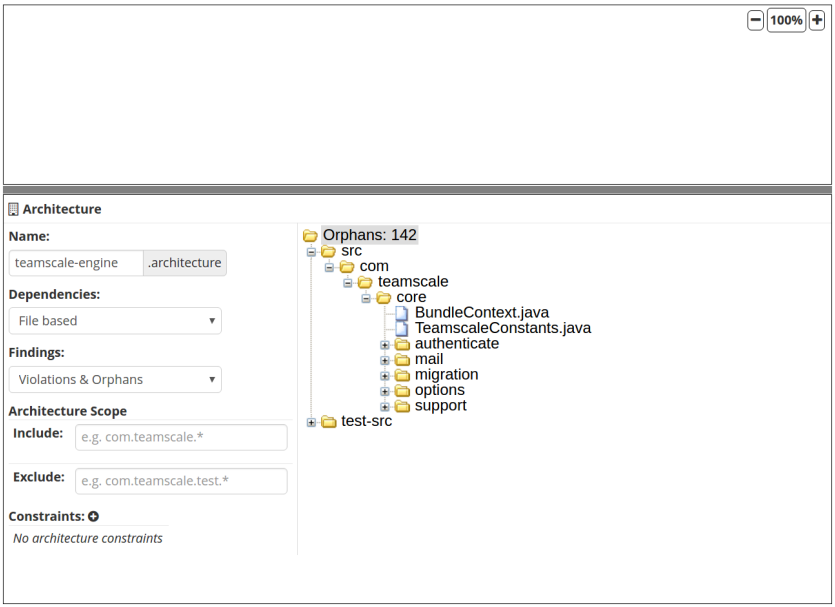
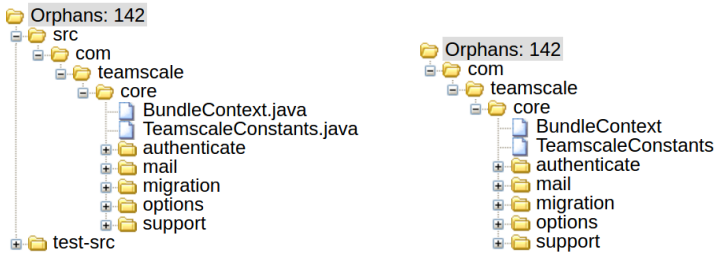


Figure 4.52. Architecture Editor

On the left side, you can edit the name of your architecture specification file and choose, whether the architecture should be based on file dependencies or type dependencies (see our architecture model in Section 4.1.7.1). You can also configure, whether findings should be created for both violations and orphans, or just for violations, or for neither one. Furthermore, the scope of the architecture can be defined. By default, the scope of the architecture matches the scope of the underlying project. It can be narrowed down by defining explicit include or exclude patterns, e. g. to exclude test code.

Additionally, constraints can be configured. A constraint represents a deny policy based on regular expression patterns. For example, in a file-based architecture definition, you may want to add the con-



(a) File-based Orphans (b) Type-based Orphans

Figure 4.53. Differences in Orphan Structure Based On Architecture Type

straint ».$*/src/.* \rightarrow .*/test-src/.*$« to indicate that source code should not depend on test code. In the simplest version, if the regex matches exactly two components, the constraint leads to an identical result as an explicitly defined deny policy between these two components. However, there may be cases, in which policies cannot be matched directly onto components (or it would create overload in creating the specification). Hence, it might be easier to define additional constraints orthogonal to the component definition.

On the right side underneath the editor, the orphans are shown. If no architecture has been created yet, all files or types will be considered orphans. Figure 4.53 shows that in architectures based on file dependencies, orphans are displayed based on the folder structure of the project. For architectures based on type dependencies, in contrast, they are displayed according to, in Java, for example, the package structure.

To start modeling the architecture, the sidebar of the *Architecture Edit View* (Figure 4.54) provides you the necessary tools, which are all described in Figure 4.59. Usually, you start by creating the first component(s). Click on  and then hover over the white (currently still

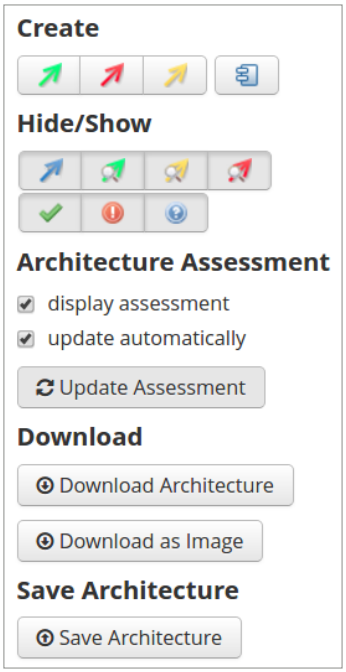


Figure 4.54. Sidebar of the Architecture Editor

empty) editor field to drag&drop your first component. As displayed in Figure 4.55, you can give each component a name.

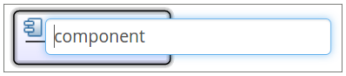


Figure 4.55. Creating a Component.

By clicking on the component, you can edit it and, in particular, add mappings for it, so that source code will be matched to the component, see Figure 4.56. Alternatively, you can use the orphans view and drag&drop folders or packages from the orphans onto the

component. This will create a corresponding mapping automatically. Also, you can drag folders/packages directly from the editor into the model which will create a new component with the respective mapping to the folder/package being dragged. If the checkboxes **display assessment** and **update automatically** in the right sidebar are marked, you will see how many types are matched to the component. Alternatively (again only if **display assessment** is checked), you can also trigger the update of the assessment manually by clicking the **Update Assessment** button.

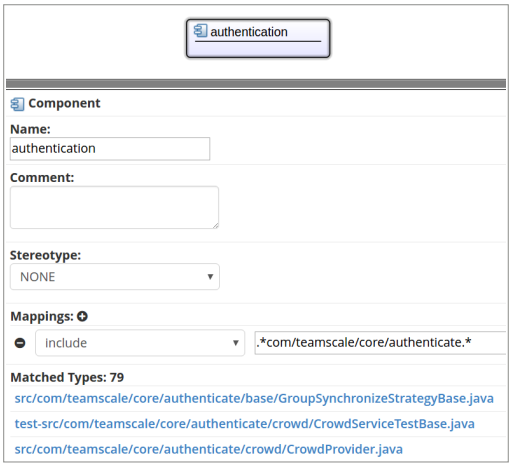


Figure 4.56. Editing a Component.

With the buttons in the sidebar on the right (Figure 4.54), you can model policies between components. For example, click on  to create an allow policy and then first click on the source component and – while keeping the mouse button clicked – drag the upcoming policy arrow till you hover over the target component. Then release the mouse button and a green policy should appear, see Figure 4.57. Similarly, you can create tolerate  and deny  policies.



Figure 4.57. Creating an Allow Policy Between Two Components

While you are modeling your architecture, *Teamscale* will assess the created policies on the fly, if the checkboxes **display assessment** and **update automatically** in the right sidebar are marked. *Teamscale* assesses whether the created policies are violated by dependencies in the source code, see explanations in Figure 4.51. Figure 4.58 shows, for example, that a modeled deny policy was, in fact, violated in the code by one dependency. *Teamscale* also displays source and target of the dependency and allows you to directly jump to the code file by clicking on the respective location link.

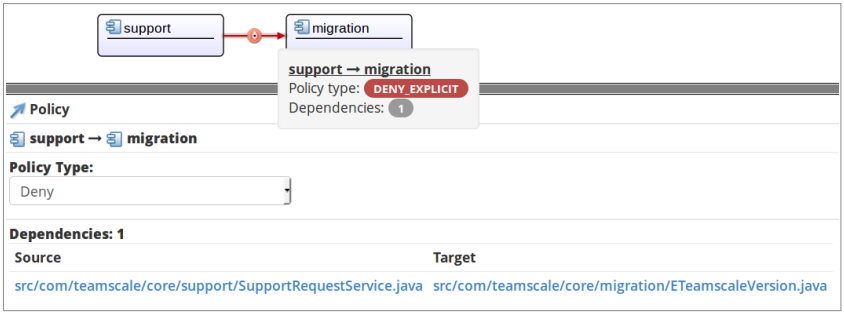


Figure 4.58. Assessment of a Deny Policy.

4.1.7.3 Action: Save Architecture

When you are done creating your architecture, you need to save your changes. When you are in the *Architecture Edit View*, you can either download the architecture file to check it into your repository or you can save the architecture on the *Teamscale* server. Use the corresponding buttons in the sidebar **Download Architecture** or **Save**

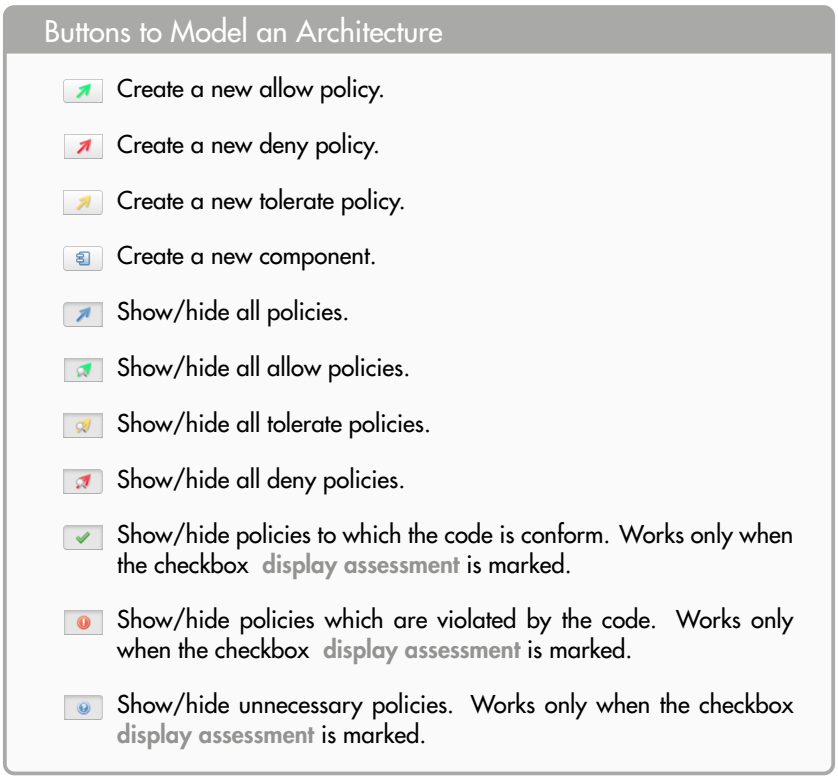


Figure 4.59. Explanations of Buttons in Architecture Edit View

Architecture. In both cases, the change to the architecture file will show up as a commit in the **Activity** perspective (see Section 4.1.3). When you are storing the architecture in your repository, your regular commit to the repository will be displayed. When storing the architecture in *Teamscale*, an artificial commit will be created, see



Figure 4.60. Artifical Commit For Saving an Architecture on the *Teamscale* Server

Figure 4.60. The artificial commit is named »Changed architecture« and also has an artificial revision number.

When saving an architecture on the *Teamscale* server, it will be visible for all users of the underlying project – same as if you would commit it to the repository. Other users can not only read but also modify your architecture.

4.1.7.4 Action: View Architecture

Often, you want to simply view an existing architecture, which was maybe created by your colleagues or by yourself several days ago. When you open the **Architecture** perspective and you already modeled an architecture, you can click on its name in the table of the *Architecture Overview View*, see Figure 4.47. This will open the architecture in the *Architecture Inspect View*. A completely modeled architecture may look as depicted in Figure 4.61.

Architecture Inspect View In this view, you can use the **Hide/Show** buttons in the sidebar, see Figure 4.62 to filter the policies in your architecture view. You can also click on the components to inspect which types were matched or click on the violations to see where they are coming from. Additionally, you can download the architecture

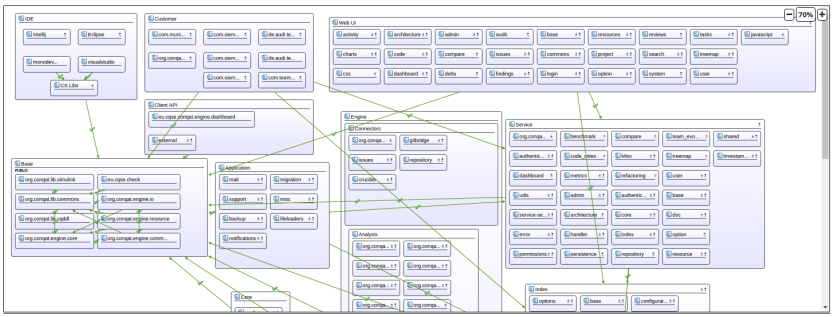


Figure 4.61. Inspecting An Architecture



Figure 4.62. Sidebar for Viewing the Architecture

either as XML file (**Download Architecture**) or as SVG file (**Download as Image**).

4.1.7.5 Action: Inspect Metrics for Architecture

As *Teamscale* measures a variety of metrics for your system, you can also view them with respect to your architecture. When you open

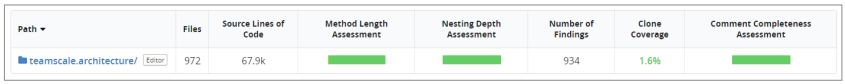


Figure 4.63. Metric Overview Table For an Architecture

the **Architecture** perspective with the *Architecture Overview View*, you can use the *Metrics* button to inspect the metrics. This will provide you a view with a table, in which the metrics are aggregated based on the components of your architecture, see Figure 4.63. The table is the same as you can also find in the **Metrics** perspective, see Section 4.1.5. When you drill into it, you can navigate the various components and descend into sub-components till you reach a mapped code file. Along the way, the metrics in the table are always aggregated to the (sub-) component in which you are currently located. Hence, you can inspect, for example how large a component is in terms on lines of code, if a specific component is missing documentation or whether the test coverage varies greatly between different components.

In addition to the **Architecture** perspective itself, you can use your architecture specification also in other perspectives. For example, the **Dashboard** perspective (see Section 4.1.2) offers to configure a widget for a specific (sub-) folder of your system. In analogy, a widget can also be configured for a (sub-) component of your system. Generally speaking, in *Teamscale*, whenever you can choose a folder path from your project, you can also choose an architectural component definition instead.

4.1.8 Delta Perspective

Use Cases	
	Comparing two Snapshots of a System
Pages	
	Delta Parameters, Affected Files, Repository Changes, Change Treemap, Metric Chages, Findings Churn
Actions	
	Calculate Delta, Inspect Affected Files, Inspect Repository Changes, View Change Treemap, View Metric Changes, Navigate the Findings Churn

The main purpose of the **Delta** perspective is to calculate the delta, i. e. compare two snapshots of a system with each other. A snapshot is defined by a point in time for which *Teamscale* retrieves the corresponding version in the version control system as well as all related quality data. The comparison of two snapshots reveals the differences in terms of code changes, metric changes and findings churn.

4.1.8.1 Action: Calculate Delta

To calculate a delta, you can navigate to the **Delta** perspective which will open up the *Delta Parameters Page*.

Delta Parameters Page On this page you can configure the delta you would like to calculate, see Figure 4.65. The box, see Figure 4.64 describes the available options. If branch support has been enabled

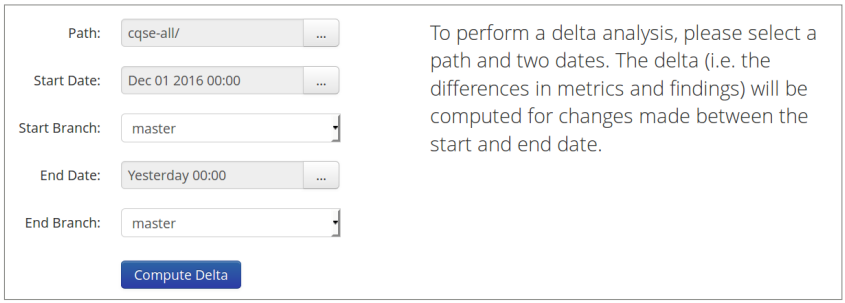


Figure 4.65. Delta Parameters With Branch Support

for the underlying project, you can also compare snapshots from two different branches with each other. Click Compute Delta to start the delta analysis.

Delta Parameters

Path	Select the path for which you want to compute the delta. This can be a a top-level code folder or any subfolder of a *Teamscale* project but also a component of an architecture definition.
Start Date	Select the start date as a time, revision or baseline.
Start Branch	If branch support is activated, you can select the branch for the start date.
End Date	Select the end date as a time, revision or baseline.
End Branch	If branch support is activated, you can select the branch for the end date.

Figure 4.64. Parameters for Calculating a Delta

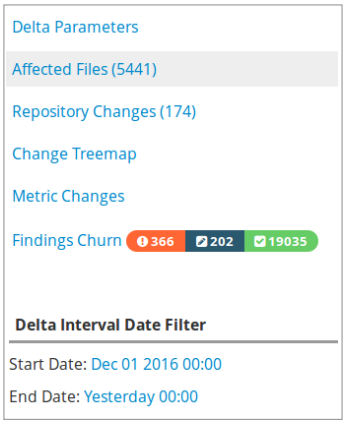


Figure 4.66. Sidebar of the **Delta** Perspective

4.1.8.2 Action: Inspect Affected Files

After calculating the delta, the sidebar of the perspective will appear, see Figure 4.67. It automatically redirects you to the *Affected Files Page*.

Affected Files Page This page gives you an overview which files in the repository were »affected« within the selected interval of the delta. Figure 4.67. The page provides you a **Code Churn Summary** indicating how many files were added, changed, deleted or moved/-copied. The latter is identified with *Teamscale*'s own origin analysis [SHJ14] which reveals if a file was copied from an existing files (either identically or with small changes) or moved between locations. Further more, the page shows you all **Affected Paths**. If you click on a path, it will redirect you to the *Code Detail View* to view the source code (see Section 4.1.5). If a file has been changed, indicated by the **Change type**, you can click on it to open the *Code Comparison View* to view the differences.

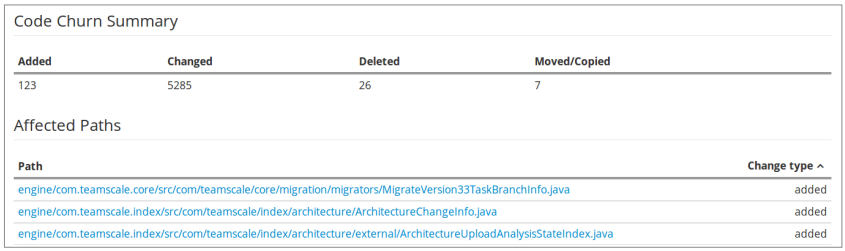


Figure 4.67. List of Affected Files

4.1.8.3 Action: Inspecting Repository Changes

The sidebar provides a link to the *Repository Change Page*.

Repository Change Page This pages provides an overview of all commits which took place within the time interval of the delta. The page contains the activity stream of commits in the same way as the *Activity View* of the Activity perspective (see Section 4.1.3). Browsing the stream of commits is particularly useful if you want to find the root cause of a metric change as described in Section 3.4.2.

4.1.8.4 Action: Viewing the Change Treemap

While the *Affected Files Page* shows the code changes in the delta in a list format, the *Change Treemap Page* provides the same information in treemap form.

Change Treemap Page Figure 4.68 shows such change treemap. Red indicates files which were deleted during the delta, blue indicates changed files and green indicates added files.

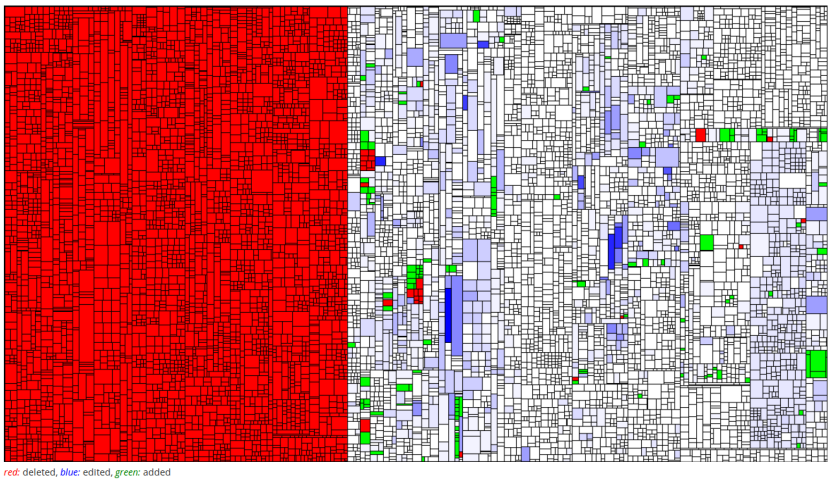


Figure 4.68. Change Treemap for a Delta

4.1.8.5 Action: Viewing Metric Changes

The **Delta** perspective does not only provide information about code churn and the corresponding commits, it also contains the delta with respect to quality data. In particular, it provides the **Metric Churn**. Use the right sidebar to navigate to the corresponding page.

Metric Churn Page . This page shows the delta for each metric, comparing the metric value at the end of the selected time interval with the metric value at the selected baseline. Figure 4.69 shows the table indicating the **Metric Churn**. For each metric, it provides the name as well as the delta change.

Metric Churn

Metric	Change
Files	+99
Lines of Code	-2,686
Source Lines of Code	+9,682
File Size Assessment	+1883 +1558 +6241
Method Length Assessment	+122 +116 +5404
Nesting Depth Assessment	±0 -75 +5717
Maximum Cyclomatic Complexity	+5
Cyclomatic Complexity Assessment	+259 +30 +5353
Last Change Date	Jan 03 2017 15:29

Figure 4.69. Metrics Table for the Delta

4.1.8.6 Navigating the Findings Churn

In addition to code churn and metric churn, the **Delta** perspective also provides the findings churn. Use the right sidebar to navigate to the *Findings Churn Page*.

Findings Churn Page This page gives you an overview of findings which were added (**New Findings**) and which were removed (**Removed Findings** during the selected time interval, see Figure 4.70. Additionally, it also shows you findings in code which was changed during the time interval. This is relevant if you selected quality goal *improving* for your project. Under this quality goals, findings in changed code should be removed on-the-fly. Hence, this page shows you the opportunities you missed.

The overview of the findings churn can be expanded to view the individual findings in each finding category and finding group. Upon expanding, the findings are shown in a table as depicted in Figure 4.71.

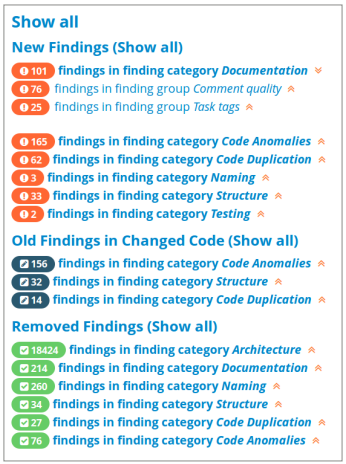


Figure 4.70. Overview Findings Churn

The table provides the message, the finding location as well as the finding group. Clicking on the messages opens the *Findings Detail View* of the **Findings** perspective (see Section 4.1.4), clicking on the location redirects you to the *Code Detail View* (see Section 4.1.5).

Additionally, the sidebar offers the same filter mechanisms to navigate the findings as the **Findings** perspective already described in

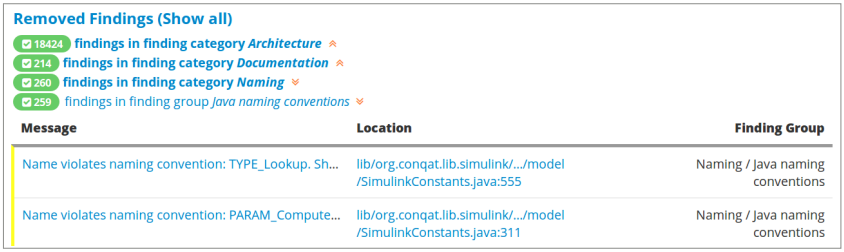


Figure 4.71. Expanding the Findings Churn

Section 4.1.4: You can filter by entering a regular expression or by selecting specific finding categories or finding groups.

4.1.9 Project Perspective

Use Cases	
	Configuring *Teamscale* Projects
Pages	
	Projects, Analysis Profiles, Thresholds
Views	
	Create Project View, Create Analysis Profile View, Create Threshold Configuration View
Actions	
	Managing Projects, Creating a Project, Managing Analysis Profiles, Creating an Analysis Profile, Managing Threshold Configurations, Creating a Threshold Configuration

The *Teamscale* project perspective enables users to manage projects, analysis profiles and metric threshold configurations. This section describes the user interfaces for creating and managing each of these entities.

4.1.9.1 Action: Managing Projects

Projects in *Teamscale* contain links to one or more source code repositories and the findings that *Teamscale* generates and collects for the corresponding source code artifacts.

Projects Page Figure 4.73 shows the **Projects** page of the **Projects** perspective. After the project is set up, it appears in the project list. For each existing project, the project list offers different buttons for project management as shown in Figure 4.72.

4.1.9.2 Action: Creating a New Project

To create a new project, the **Projects** page provides above the list of existing projects the button Create new project. A user needs to set a project name and an analysis profile. In Figure 4.74, we chose the *ABAP (default)* profile as an example. Furthermore, the user can choose a dashboard template.

A project can be configured with various connectors for version control systems, issue trackers and code review tools, see Figure 4.75. In terms of version control systems, *Teamscale* offers connectors to GIT, SVN and TFS. However, it can also read from the file system, either with a simple **File System** or a **Multi-Version File System** connector.

Git Connector To set up a GIT connector, the user has to specify account data (the + button opens an account setting dialog), and a repository identifier (must be unique in the project). Furthermore, the user can specify a start revision to reduce the analysis load and include or exclude files. You might, for example, want to exclude generated code or experimental code from the analysis. Also, you

Project Management Actions

- Opens the edit dialog for the project. The dialog is the same as for project creation except that the project name can not be changed.
- Opens a dialog for advanced configuration options such as a connection to SAP systems and the threshold configuration to use for the project. *Teamscale* defines a default threshold configuration but users can create custom threshold configurations (also discussed in this section).
- Triggers a complete re-analysis of the project.
- Deletes the project.
- Exports the project configuration. The exported file can be imported with the "Import project" button in Figure 4.73.
- Exports a complete backup of the project (including analysis results). The exported file can be imported with the "Import project" button in Figure 4.73.
- Renders a graph of the *Teamscale* analysis process (requires the rendering tool Graphviz).
- Renders a graph of the *Teamscale* analysis process including the database/index layer (requires the rendering tool Graphviz).

Figure 4.72. Actions Available in the Project Overview

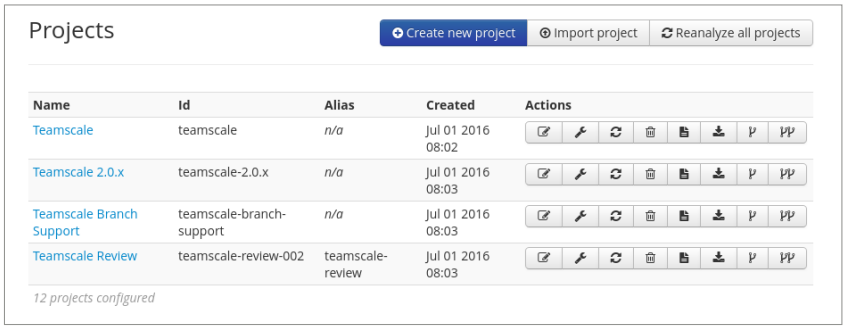


Figure 4.73. List of Configured *Teamscale* Projects

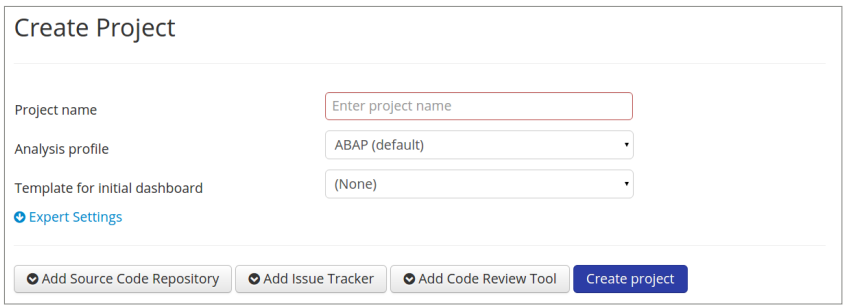
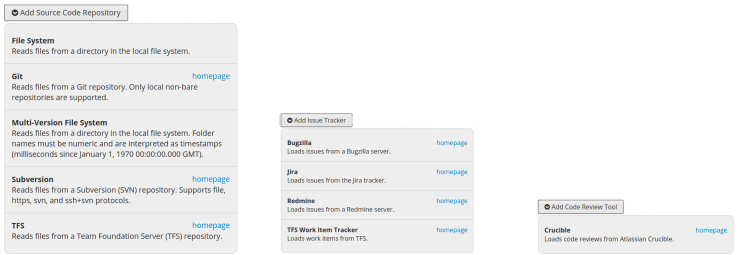


Figure 4.74. Setup View for a *Teamscale* Project

might want to have separate projects for your application and your test code. Note that *Teamscale* should be connected to either a remote repository (*Teamscale* will clone it locally) or a dedicated local clone that is not modified by other tools or users.

Branch Support Teamscale also offers branch support. As default, branch support is not enabled while configuring the connector for a project. For large projects with hundreds of branches, this would dramatically increase the analysis time. You can manually enable branch



(a) Repositories (b) Issue Trackers (c) Code Review Tools

Figure 4.75. Connectors for Creating a Project

support by clicking the corresponding checkbox and configure which branches should be included in the anlaysis by using include/exclude patterns. The patterns are reg-ex based. For example, you can match your release-branches or specific feature branches. Internally, *Teamscale* has a default branch, which maps to »trunk« for SVN and »master« for GIT. Sometimes, this has to be adjusted, however.

Multiple Repository Connectors You can also use multiple repository connectors for one project. For example, you can read both from GIT and SVN or from GIT and the file system or from multiple GIT repositories. If you have two connectors read the same files and/or commits, *Teamscale* will merge these inputs into a unified view. In the **Metrics** perspective and the subsequent views, each file will only appear once. In the history of a file (*Code Detail View*) and in the *Activity View*, each commit for each repository will be shown separately. If there are multiple commits with the same timestamp but with different content (different changes), *Teamscale* will decide which of them to display as the »latest« commit. This choice also affects computation of metrics in the dashboard as the metric values might differ depending on the order of the commits (e.g., added lines of code). A

situation where such overlapping commits are loaded can easily be generated by using multiple multi-version file-system connectors.

Branch Support For Multiple Repository Connectors If you use multiple repository connectors and have branch support enabled, *Teamscale* will assume that branches with the same name can be mapped onto each other. Hence, *Teamscale* will only show one branch with the data from all connectors. However, if the branching structure is not consistent between connectors, this will not work, e. g. if you have a SVN connector followed by a GIT connector. For this case, *Teamscale* provides expert options in the connector configurations. The option **Branch transformation** allows you to provide a manual mapping between branches. In our example, you would indicate »trunk → master«, for example.

Multi-Version File-System Connector A multi-version file-system connector reads multiple versions of a repository that are stored in separate directories. The connector expects an input directory that contains one subdirectory per repository version. The names of the subdirectories must be numeric and are interpreted as timestamps (milliseconds since January 1, 1970 00:00:00.000 GMT). Figure 4.76 shows an example of this directory layout. The multi-version file-system connector expects that each subdirectory contains a full snapshot of the repository at the given timestamp. Each subdirectory is represented by a commit in *Teamscale*. If a file is missing in a subdirectory, it is treated as if it was deleted.

All options for repository, issue tracker or code review tool connectors are described in detail in Appendix B, C and D.

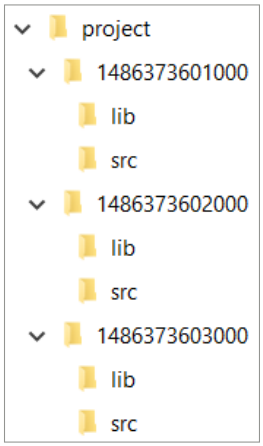


Figure 4.76. Directory layout for a multi-version filesystem connector

4.1.9.3 Action: Managing Analysis Profiles

In addition to configuring projects, the project perspective enables users to configure analysis profiles that can be used in different projects. An analysis profile configures the programming languages that will be analyzed, external tools which can be included in the analysis results, and which metrics are computed and displayed. *Teamscale* already offers many analysis profiles which are usually tailored to specific programming languages (e.g., **ABAP (default)** or **Java (default)**) but users can create and manage their own ones.

By clicking on **Analysis Profiles** in the right sidebar of the **Projects** perspective, the *Analysis Profile Page* will be shown. The page shows all existing default profiles as well as individually created ones, see Figure 4.77. For each profile, the list offers different buttons for management, see Figure 4.78.

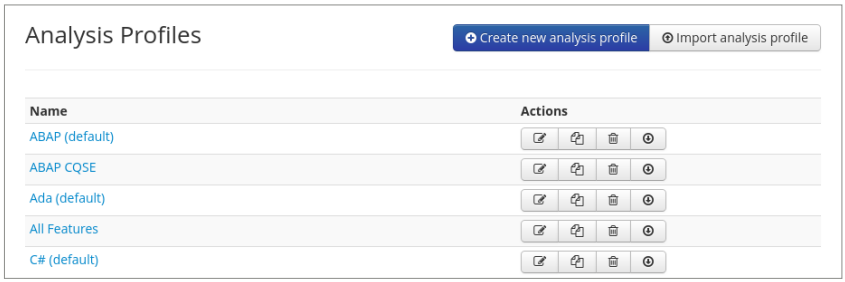


Figure 4.77. Analysis Profile Page

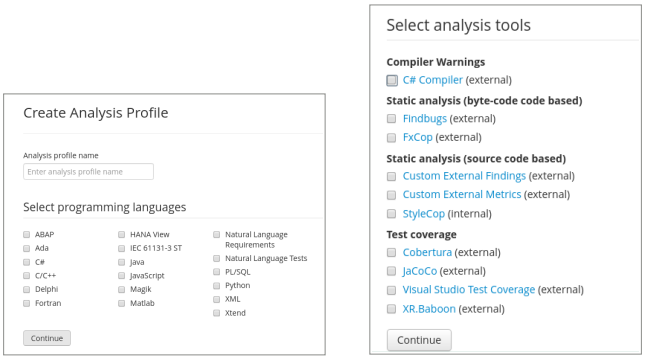
Actions for Analysis Profiles

- Opens the edit dialog for the analysis profile. The dialog is the same as for analysis profile creation except that the profile name can not be changed.
- Opens a dialog for copying the analysis profile.
- Deletes the analysis profile.
- Exports the profile as a file. Exported profiles can be imported with the Import analysis profile button above the analysis profile list.

Figure 4.78. Actions for an Analysis Profile

4.1.9.4 Action: Creating an Analysis Profile

The Analysis Profile page offers a button to create a new analysis profile. Creating and modifying an analysis profile has three steps illustrated in Figures 4.79a–4.80. In the first step, the user sets a profile name and selects the analyzed programming languages. The choice



(a) Step 1 (b) Step 2

Figure 4.79. Configuration Dialog for Analysis Profiles

of programming languages determines which options are available in the next steps since some metrics are not available for all languages.

In the second step, the user selects which external tools are queried for analysis results. If, for example, the **Findbugs** check box is chosen, *Teamscale* will display findings generated by the Findbugs tool in addition to its own code metrics. The **Custom External Findings** and non-code metrics features enable users to upload their own findings using a python API.

In the third step of the analysis profile setup, users can choose which code metrics *Teamscale* will compute and display. *Teamscale* groups code metrics in **analysis groups** and each group is part of a **quality indicator**. In the analysis profile setup (Figure 4.80), the user can configure which analysis groups should be shown in which indicators, and which analyses groups should not be computed. To match your individual analysis needs, the analysis groups can also contain custom checks. In which analysis group a custom check is placed, depends on the declaration of the programmer within the im-

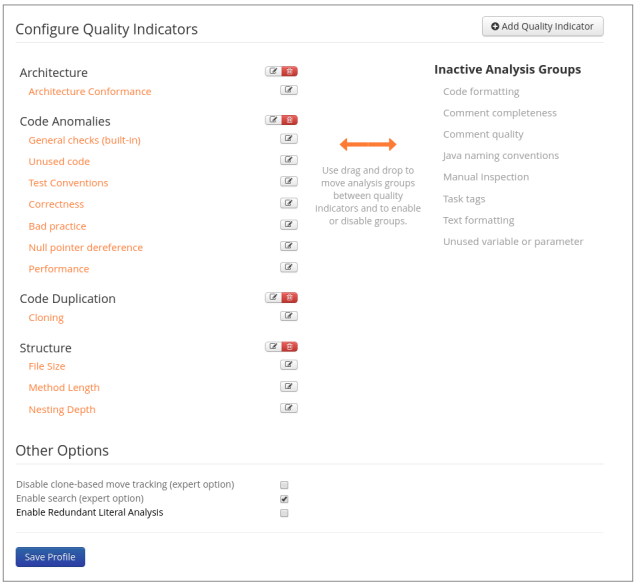


Figure 4.80. Configuration Dialog for Analysis Profiles (Step 3)

plementation of the custom check. *Teamscale* comes with a variety of pre-written custom checks for various languages, see Appendix F. In the analysis group options, each custom check can be individually switched off or on by setting its severity to yellow or red.

Below the analysis profile *Teamscale* shows a set of expert options which you can use to fine-tune the analysis behavior. Some of these options depend on the chosen languages, so some options might be missing in your analysis profile dialog.

Expert Options for Analysis Profiles

Disable clone-based move tracking	You can select this check box to disable tracking of moving or renaming files.
C/C++ predefined macros	Here you can specify macros which are not defined in the project source code. If those macros are not defined, *Teamscale* might issue warnings about undefined macros.
Enable search	Should *Teamscale* enable users to search in the project sourc code, commits, and issues?
Enable Redundant Literal Analysis	This checkbox enables the analysis of redundant literals in the source code. If enabled, teamscale generates findings for often-occurring string literals.

Figure 4.81. Configuring an Analysis Profile

After an analysis profile is set up, it is shown in the analysis profile list.

4.1.9.5 Action: Managing Threshold Configurations

A threshold configuration specifies threshold values for metric. It influences how metric values are displayed, for example, in the dashboard. A threshold configuration is identified by its name and consists of thresholds structured in groups. Threshold configurations are project-independent. The **Thresholds** page gives an overview of all existing threshold configurations, see Figure 4.82.



Figure 4.82. **Thresholds** Page

4.1.9.6 Action: Creating a Threshold Configuration

To create a new threshold configuration, you have to select **Thresholds** in the projects perspective sidebar. This will lead you to the **Thresholds** page which offers the Create new Thresholds button. This opens the *Create Threshold Profile View* as shown in Figure 4.83.

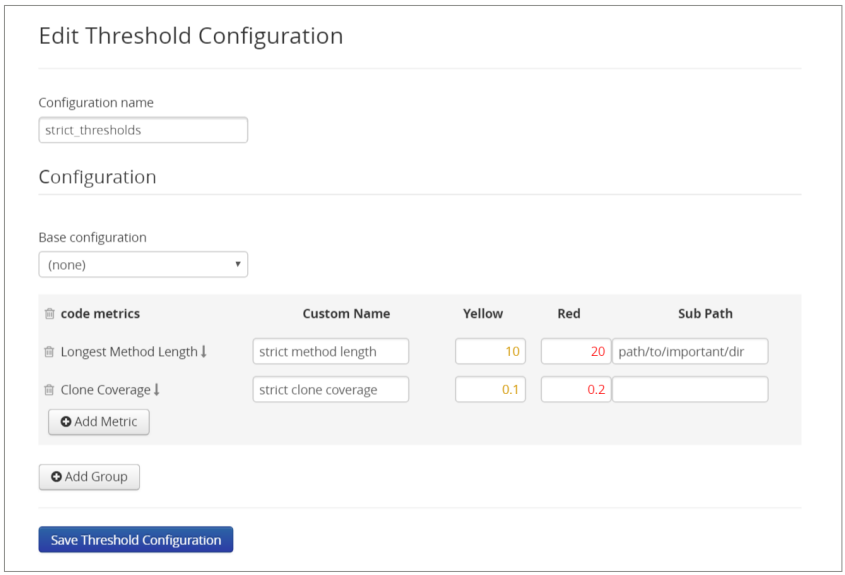


Figure 4.83. Setup View for an Example Threshold Configuration

In this dialog, the you have to choose a name for the threshold configuration and add metric groups to the threshold configuration. Next, you can add metrics to the metric groups. Each metric is shown in one line. The arrow next to the metric name explains in what direction the metric is assessed. If the arrow points down, low values indicate good values and vice versa.

The field **Custom Name** allows you to use another name than the metric's name when displaying the metric, for example, in the dashboard. Unless two rows with the same metric exist, the custom name is optional and the metric's name will be displayed in case the custom name is not specified.

The fields **Yellow** and **Red** enable you to specify thresholds. Depending on the metric, the fields can be either filled with an arbitrary numeric value or a percentage value (as number between 0 and 1). It is also possible to specify only one of the thresholds or none (in which case no thresholds will be displayed). Note that some metrics are not supposed to be assessed (such as »Lines of Code«); it is not possible to specify thresholds for those.

The field **Sub Path** allows restricting the assessment and validity of the thresholds to a certain part of a project. The sub path can be used to assess different parts of a project differently (e.g. restrict the threshold to the production code by excluding test code). It is also possible to use a path that points to an architecture component.

4.2 IDE Integration

Teamscale provides an integration with three major IDEs: Eclipse, IntelliJ IDEA and Microsoft Visual Studio. The *Teamscale* IDE plug-ins fetch the analysis results from the *Teamscale* server and annotate the

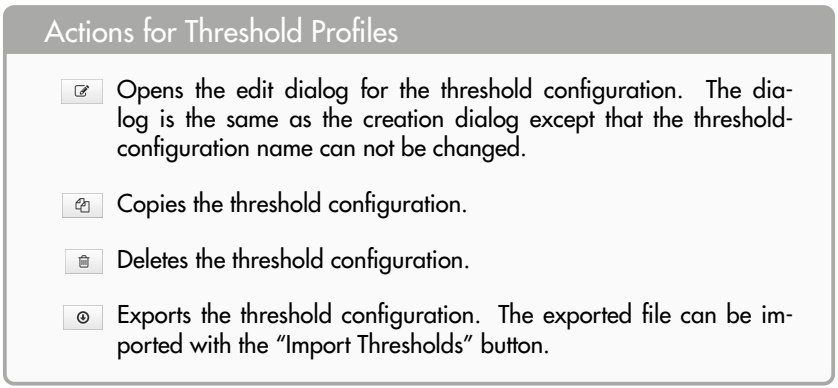


Figure 4.84. Actions Available in the Threshold Configuration Overview

source code with quality-related information (e. g. findings, test coverage) directly within the editor. This avoids a context switch for developers giving them access to quality data right in their accustomed working environment.

In the following we describe the installation and configuration for each of the *Teamscale* IDE plug-ins.

4.2.1 Eclipse

The Eclipse plug-in for *Teamscale* is available in the *Eclipse Marketplace*[2]. The Marketplace page also states the current system requirements (especially supported versions of Eclipse). To install the plug-in, drag the *Install* button to your running Eclipse workspace. Follow

[2] https://marketplace.eclipse.org/content/teamscale-eclipse-plugin

the instructions of the installation wizard and restart Eclipse when prompted.

After installing the plug-in, you can configure a *Teamscale* server via the following steps:

1. Open the Eclipse preferences dialog, navigate to the *Teamscale* settings (*Window* → *Preferences...* → Teamscale) and click on *Add*
2. Enter a descriptive name and the URL of the *Teamscale* server
3. Enter your user name and IDE access key (you can generate an access key via the Web UI on your personal profile page accessible via the user icon in the upper right corner)
4. Test the connection by clicking on *Validate*.
5. Close the dialog by pressing *OK*.

To connect an Eclipse project to *Teamscale* and obtain quality data such as findings, perform the following steps:

1. Open the project properties by right-clicking on the project and selecting *Properties*.
2. On the left side, switch to the Teamscale entry
3. Click on *Enable* Teamscale *for this project* and select the appropriate server
4. Click on *Automatically configure* Teamscale which will automatically identify the correct mapping to the code in *Teamscale*.

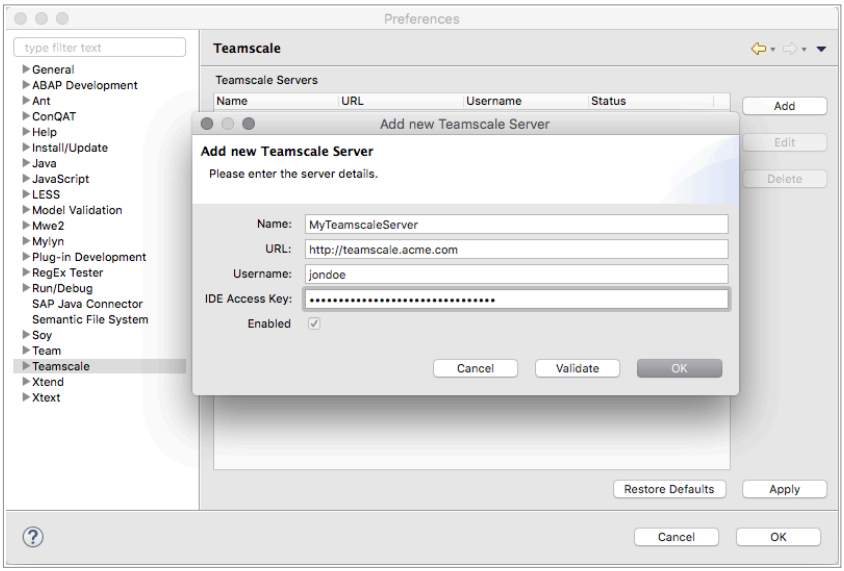


Figure 4.85. *Teamscale* Preferences in Eclipse

After connecting a project to *Teamscale* you can enable the display of findings in the menu bar via Teamscale → *Load findings for open files*.

4.2.2 Visual Studio

In addition to the Eclipse plugin, *Teamscale* also provides an integration within Visual Studio.

4.2.2.1 Download & Installation

You can download the Teamscale Visual Studio plugin by using the integrated extension manager in Visual Studio by searching for Team-

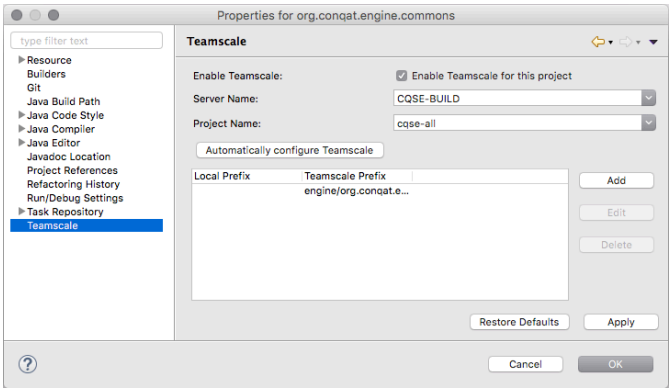


Figure 4.86. *Teamscale* Project Properties in Eclipse

scale or by downloading the Plugin from the Microsoft Visual Studio Gallery[3].

Installation via the extension manager has the added benefit of automatically being notified about new updates, as they get published.

In case you downloaded the extension manually you can install it as follows:

1. Double click on the Teamscale.vsix Extension file
2. In the dialog of the VSIX Installer, click Install
3. Restart Visual Studio

[3] https://visualstudiogallery.msdn.microsoft.com/1b425e5b-014b-449c-a327-b1dc54af733f

4.2.2.2 Configuring a Teamscale server

Next you have to configure a Teamscale server, so that the plugin knows from where to fetch the findings.

1. Click Tools → Teamscale Options...
2. Click Add
3. In opened dialog enter the Teamscale Server ID (this name is later references from project mappings), the Teamscale Server URL, your username and password/api key.

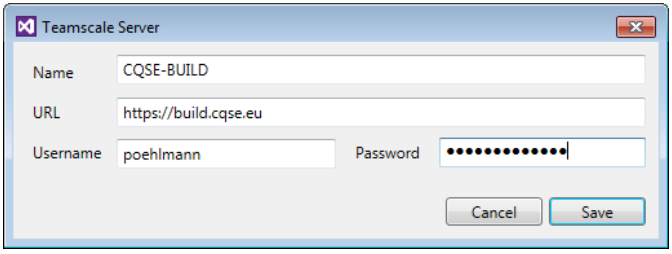


Figure 4.87. *Teamscale* Teamscale Server Configuration in Visual Studio

4. Click Save. The dialog should close if validation of the connection was successful.

4.2.2.3 Configuring the project mapping

Next you have to supply a project mapping, so that the plugin knows which Teamscale Project belongs to which code on the hard drive.

1. Click Tools → Teamscale Options...

2. On the left hand side select Teamscale → Project Mapping
3. Click Add
4. In the opened dialog select the Teamscale Server and Project . Choose the local folder you want to map the Teamscale Project to and optionally specify a sub-path of the project on the server (this can be left blank in most cases).

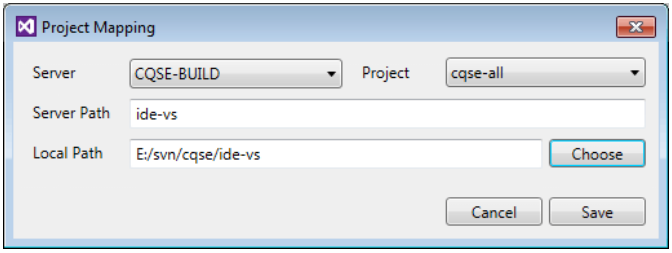


Figure 4.88. *Teamscale* Project Mapping in Visual Studio

5. Click Save
6. Reopen the currently opened Solution if it corresponds to the configured project

4.2.2.4 Inspecting findings

After the previous steps have been completed successfully you should now be able to explore findings in your code.

1. Findings in the opened document should be marked with a red bar on the left margin
2. Hovering with the mouse over the margin will show a tooltip containing the description of the finding

3. To show all findings of a file, either right click in the current editor or the Solution Explorer and select Teamscale → Show Findings

4. In the findings list one can double click to jump to the finding in the code

4.2.3 IntelliJ IDEA

Teamscale also integrates with IntelliJ IDEA and many other IntelliJ IDEs, such as WebStorm and PyCharm.

4.2.3.1 Download & Installation

The IntelliJ plugin has been tested and used on IntelliJ IDEA, WebStorm and PyCharm IDEs, but should also work on other IntelliJ IDEs. To download and install the *Teamscale* IntelliJ IDEA plug-in, perform the following steps:

1. Inside an IDE browse the Jetbrain Plugin Repositories via *File* → *Settings* → *Plugin* → *Browse Repositories*.

2. In the dialog box, search for »Teamscale IntelliJ Plugin«. On the right view, click the install button.

3. Restart the IDE to conclude installation.

4. If the plug-in installed properly, you should see a Teamscale menu entry in the main menu bar.

4.2.3.2 Configuring a *Teamscale* Server Connection

You need to perform the following actions to configure a *Teamscale* server, see Figure 4.89:

1. Open the IntelliJ IDEA Settings via the menu *File* → *Settings*.
2. Expand *Tools (1)* and select Teamscale *(2)* on the left hand side.
3. Use the + button to add a new *Teamscale* server (3).
4. Fill the dialog (4) with the appropriate values. Note: For security reasons you may not use your *Teamscale* password for authentication. Instead, an IDE Access Key is required. You can generate a personal access key with the *Teamscale* web interface. To do so, navigate to your user profile page by clicking on the user icon in the upper right corner of and navigating to the tab *IDE Access Key*.
5. Click the *Validate Connection (5)* button to ensure successful connection the *Teamscale* server.
6. Click *Apply* and *OK* to save the settings.

4.2.3.3 Connecting an IDE Module or Project to a *Teamscale* Project

In order to fetch findings from the *Teamscale* server for an IntelliJ IDEA module, a Webstorm or PyCharm project, the *Teamscale* IntelliJ IDEA plug-in needs to know to which *Teamscale* project the IDE module or project corresponds. This mapping is done in the mapping settings page. To access the mapping settings page on IntelliJ IDEA , you have to perform the following steps:

1. Right click on the module and select *Open Module Settings*.

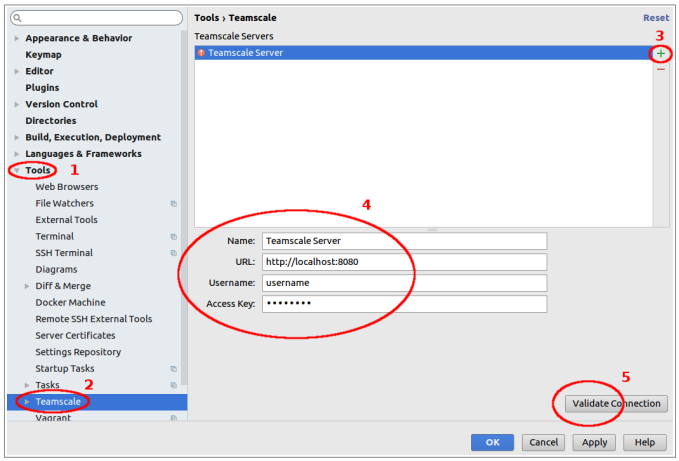


Figure 4.89. Configuring a *Teamscale* Server Connection

2. Select the Teamscale tab on the right hand side.

In Webstorm or PyCharm IDE,

1. Click *File* → *Settings*.
2. Expand *Tools*. Underneath, expand Teamscale and click Project Mappings.

On the mapping settings page (see Figure 4.90), you have to carry out the following steps:

1. Check the option »Enable Teamscale get quality analysis data for this module/project«.
2. Select the appropriate *Teamscale* server and project values.
3. Check the option »(Re)create prefix mappings for...« for automatic creation of mappings. The following combobox controls

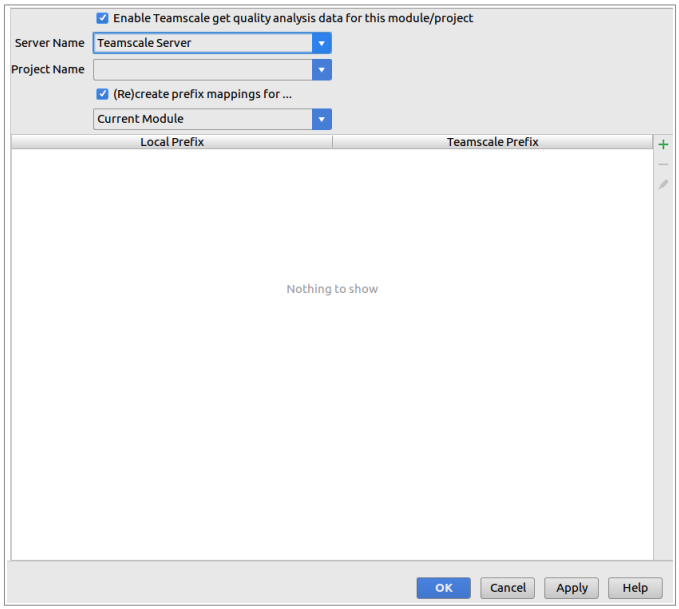


Figure 4.90. Creating Module or Project Mappings

the modules which will have mappings created. IntelliJ IDEA can have more than one module. To create mappings for all at once, select »All modules«.

4. Click *Apply* and *OK* to save settings.

File Mapping Under normal circumstances, the plugin automatically determines how to map the module files to the files in the *Teamscale* project. If the automatic mapping failed, a message is shown. To manually configure mappings between the local and the server's structure, you can add them in the module settings:

1. Open the mapping settings page as mentioned above.

2. Uncheck the option »(Re)create prefix mappings for«.

3. Create a mapping by clicking the + sign on the right side. Enter the mappings prefixes in the upcoming dialog. The Local Prefix is a part of your local directory structure. The *Teamscale* Prefix is a part of the project directory structure on the *Teamscale* server.

To check if everything is configured correctly, open the *Teamscale* Findings Tool Window via the menu (*View* → *Tool Windows* → Teamscale *Findings*), see Figure 4.91.

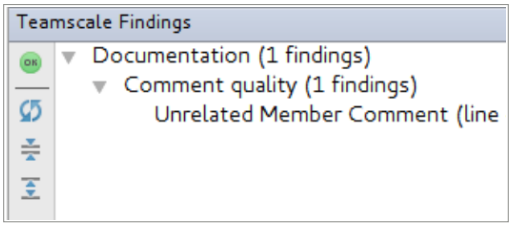


Figure 4.91. *Teamscale* Findings Tool Window

Now you can select any resource that has findings and wait for findings to appear in the *Teamscale* Findings Tool Window. The symbol on the top of the left hand side of the Tool Window will indicate the server's reachability state. Hover the mouse over the icon to get additional information. A click on the icon opens up the *Teamscale* settings.

Refresh Findings In order to get an updated view of the findings of a selected folder or a file after changes in a file system, for instance by switching to a new repository or branch, click the button in the *Teamscale* Findings Tool Window.

4.3 Extension and Integration APIs

This section describes the interfaces provided by *Teamscale* for extension and integration.

4.3.1 Custom Check API

The »*Teamscale* Custom Check API« allows users to extend *Teamscale* by writing custom analyses that create findings. These *custom checks* are executed within *Teamscale* 's incremental analysis engine and, thus, provide real-time feedback to developers on every commit. Consequently, *Teamscale* treats the findings created by custom checks equal to all other findings, such that all finding-related features, like blacklisting, visualizations, filtering etc., can be used just as usual.

A custom check is a *local* analysis in *Teamscale*, i.e. its implementation is given a single source file as an input and the check can create findings for code locations within this file. A check can use the source code of the file as plain text, the sequence of tokens or the abstract syntax tree (AST) of the file or a combination of these for its analysis. Each *Teamscale* distribution already contains many custom checks which we list in Appendix F.

4.3.1.1 Implementing a Custom Check

Setting up the Development Environment It is recommended to use the Eclipse development environment to implement Custom Checks. To set up a development workspace, please perform the following steps:

1. Download an up-to-date version of the Eclipse IDE and unzip it.

2. Launch Eclipse and create a new workspace.
3. Download the Custom Check Example project from the following URL, create an Eclipse project as described there and import it into your Eclipse workspace.
   https://github.com/cqse/teamscale-custom-check-sample

Writing a custom check To implement a custom check, create a new Java class deriving from `CheckImplementationBase` that is annotated with `@ACheck`. The class CheckImplementationBase contains the analysis context (ICheckContext) which you can use to get information about the analyzed source code file. This includes the textual representation of the code, the stream of tokens in the file (the output of the scanner/lexer), as well as the abstract syntax tree (AST, output of the parser). The only method your Custom Check needs to implement (overwrite) is named execute(). The baseclass offers you helper methods to select certain elements in the AST using XPath expressions and to create findings easily. However, in many cases your Custom Check class does not need to use this rather general base class but a more specific subclass of CheckImplementationBase. E.g. one of the following:

- EntityCheckBase: for checks that select a set of entities via XPath and inspect each of these.
- EntityFindingCheckBase: for checks that select a set of entities via XPath and create a finding for each selected entity.
- EntityTokenCheckBase: for checks that select entities via XPath and process the tokens of each selected entity.

In the @ACheck-annotation you must specify meta-data about your check. This meta-data is used by *Teamscale* mainly for configuration

of the analyses (quality profile) and the presentation of the findings your check will produce. The meta-data includes the following information:

- name: Name of the check
- description: A description of the check that explains developers why the findings of the check are a problem and how they should fix them.
- groupName: The analysis group the check should be contained in (e.g. General Checks)
- languages: The programming languages for which the check can be used.

Furthermore, a Custom Check must specify on which representation of the code it performs the analyses (using the parameters-argument of ACheck). E.g., the check may only access the AST if the option ECheckParameter. ABSTRACT_SYNTAX_TREE is specified or type information will only be present if type resolution is explicitly switched on by using ECheckParameter.TYPE_RESOLUTION. *Teamscale* needs this information to cluster the checks, so that representations that are not needed are not calculated at all.

The following code listing shows a simple Custom Check:

Listing 4.1. A simple check that selects abstract types with public constructors and emits a finding for every hit.

```
1 @ACheck(name = "Abstract types should not have
      constructors (CA1012)",
2       description = "Constructors on abstract types can
          be called only by derived types. Because public
           constructors create instances of a type, and
          you cannot create instances of an abstract type
```

```
        , an abstract type that has a public
        constructor is incorrectly designed.",
3     groupName = "Language misuse",
4     languages = { ELanguage.CS },
5     parameters = { ECheckParameter.
        ABSTRACT_SYNTAX_TREE })
6
7 public class
    AbstractTypesShouldNotHaveConstructorsCheck extends
     EntityFindingCheckBase {
8
9   /**
10   * Returns an XPath selection string that
11   * selects public abstract types with public
12   * constructors.
13   */
14  @Override
15  protected String getXPathSelectionString() {
16    return "//TYPE[modifiers('public', 'abstract')]/
        METHOD[subtype('constructor') and modifiers('
        public')]";
17  }
18
19  /**
20   * Returns the message for the created finding.
21   */
22  @Override
23  protected String getFindingMessage(ShallowEntity
      entity) {
24    return "An abstract type must not have a public
        constructor.";
25  }
26 }
```

Structure of the AST and selection via XPath The AST consists of so called shallow entities. These entities are the nodes of the AST (e.g. types or methods). They are called shallow because the AST is only shallow in the sense that it is constructed down to the state-

ment level but does not have detailed information on expressions within statements. The enum EShallowEntityType documents the list of possible shallow entity types. These are also used by the XPath notation to make a selection on AST nodes. For example, the XPath expression //METHOD would select all shallow entities that denote a method. Shallow entities also have a subtype, which further specify the entity. For instance a method entity can have the subtype 'constructor' denoting the special case of a constructor. Subtypes may be specific for a programming language. For a list of possible values, refer to the class SubTypeNames. Selection functions can be used to further filter shallow entities. As an example, the XPath expression //METHOD[modifiers('public')] selects all public methods. For a list of possible selection functions, please refer to the package eu.cqse.check.framework.core.xpath.functions.

Testing Your Custom Check The *Teamscale* Custom Check Framework provides a Unit-Testing framework you should use to write tests for your checks. The easiest way to test your check is using data-driven tests. This way you do not have to write any testcode but only provide test data in terms of code listings that contain the findings your check should identify and a file describing the expected results. To define a data-driven test you need to perform the following steps:

1. Create a file (name should be the name of your check-class followed by the typical file extension used for the programming language, e.g. <myCheck>.cs or <myCheck>.java) in a folder named 'test-data'. Put some code into the file on which your check should (not) detect certain findings.

2. Create a second file in the test-data folder and use the same name followed by '.expected', e.g. <myCheck>.cs.expected. This file will be used to compare the findings detected by your check with the expected findings. For every finding your check

should emit, put a line into this file using the following format: Offsets <fromOffset>-<toOffset> (lines <fromLine>-<toLine>): <findingMessage> The offsets here are character offsets in the source code. Please specify the following information corresponding to your test-data file: fromOffset/toOffset: Offsets to the tokens the finding should be annotated. fromLine/toLine: The line number that should be marked with a finding of your check. findingMessage: The message that should be displayed for your finding.

3. To be able to execute the tests, you need to create a single test class deriving from CheckTestBase that will automatically create a test suite from the test data (this class is already available in the test-src folder of Custom Check Example project):

Listing 4.2. Test class for data-driven custom check tests

```
1  /** Main class for testing checks */
2  @RunWith(Parameterized.class)
3  public class CheckTest extends CheckTestBase {
4
5    /** Constructor. */
6    public CheckTest(File referenceFile, Map<String
         , CheckInfo> checkInfoBySimpleClassName) {
7      super(referenceFile,
           checkInfoBySimpleClassName);
8    }
9
10   /** Generate Test Parameters. */
11   @Parameters(name = "{0}")
12   public static Collection<Object[]>
         generateParameters() throws IOException {
13     return CheckTestBase.generateParameters(new
           CheckTest(null, null));
14   }
15 }
```

4. To run the tests, launch the test class as a JUnit-Test in Eclipse and inspect the results (information about the actual results of your check are written to the directory test-tmp).

Deploying your Custom Check Package your custom check into a JAR file using the JAR Export Wizard of Eclipse. You do not need to re-package any of the included libraries. The check classes are sufficient. Place the JAR in the custom-checks subfolder of your *Teamscale* installation and restart *Teamscale*. The checks then become available when configuring analysis configurations.



Appendix

A Dashboard Widgets

Teamscale offers a large variety of highly configurable widgets to create personalized dashboards. These widgets and their configuration options are displayed in the following. Some options are the same for all widgets. Thus, they are described only once at the beginning.

A.1 Common Widget Configuration Options

The following table denotes configuration options that are commonly shared by many widgets. As an example, Figure App.92 shows the beginning of a configuration dialog containing some of these option.

Title	Each widget can be given its own custom title. Teamscale suggests a default title.
Path	The path can be used to select the scope for which the content of the widget should be displayed. The scope can either be an entire project or you can select any folder or component (if an architecture file has been modeled). You can also configure the branch if branch support is enabled and use the timetravel feature.
Trend	Many widgets can display a trend indicating the evolution of a metric. You can configure the time interval, e. g. if the trend should be considered for the last five days.
Zooming	Allows you to zoom into the chart.
Hide Y-Axes	Removes labeling of the y-axes.
Baselines	Shows the baselines of the underlying project as vertical blue lines in the chart.
Action menu	Adds a drop down menu with option to view a selected zoom range in the Delta perspective and to download the trend as csv.

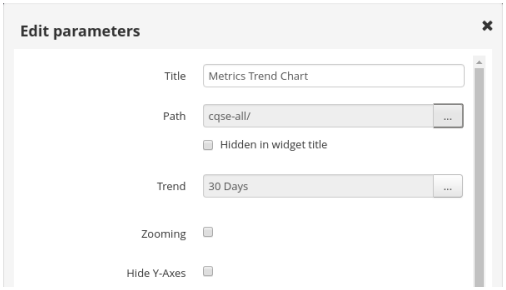


Figure App.92. Excerpt of Widget Configuration Dialog

A.2 Assessment Pie Chart Widget

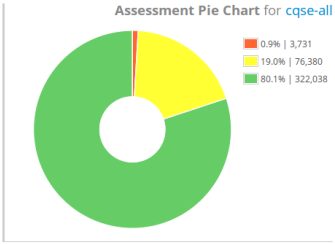


Figure App.93. Assessment Pie Chart Widget

Purpose	Shows an assessment metric as a pie chart.
Metrics	Lets you choose from the available assessment metrics. The thresholds of those metrics are configured in the analysis profile, not in this widget.
Legend	Displays a legend for the widget.
Absolute values	If checked, absolute values are shown. For structure metrics, these absolute values are LOC or SLOC depending on the analysis profile. (For the comment completeness, these values are the number of (un)documented code entities.) If unchecked, only relative percentage values of the assessment metric are shown.
Assessment thresholds	Shows target threshold lines. Should be of format »0.05,0.25«, for example.

A.3 Assessment Trend Chart Widget

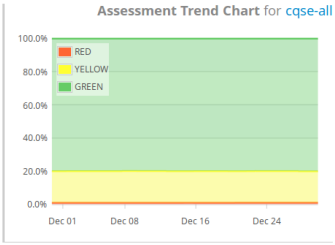


Figure App.94. Assessment Trend Chart Widget

Purpose	Shows the evolution of an assessment metric over time.
Assessment	Lets you choose from the available assessment metrics as configured in your analysis profile.
Show ratios	Displays a legend.

A.4 Code City Widget



Figure App.95. Code City Widget

Purpose	Shows the system as a code city.
Area Metric	Metric used to determine width and depth of a code city skyscraper.
Height Metric	Metric used to determine the height of a code city skyscraper.
Color Metric	Metric used to determine the color of a code city skyscraper; can be different than the height metric.
Color	Color used to color the skyscrapers. When using an assessment metric as color metric, the assessment colors red, yellow and green are always used.
Included files	Only files matching this regular pattern are displayed in the code city. The regular pattern is provided as a Java Regex. To include only Java files, enter »`.*\.java`«. You can add multiple include patterns, too, by clicking on `+`. This will provide you additional input fields.
Excluded files	Files matching this regular pattern will be excluded from the code city.

A.5 Commit Chart Widget

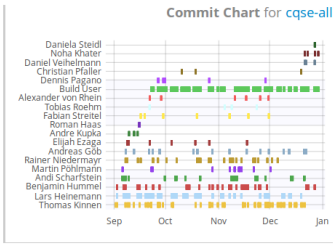


Figure App.96. Commit Chart Widget

Purpose	Shows the commit frequency of the development team over time.
Min. commits per developer	Number of commits a developer must have made at least in order to appear in the chart.
Order	Indicates ordering in which the authors are sorted. You can either select by time, by number of commits or alphabetically by the account name of the authors.

A.6 External Content Widget

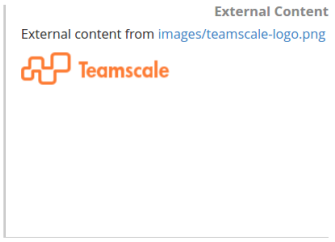


Figure App.97. External Content

Purpose	Provides a link to external content and embeds it as an iframe.
URL	Url for the external content.
Allow scrollbars	Allows scrolling within the widget.

A.7 Findings Churn Widget

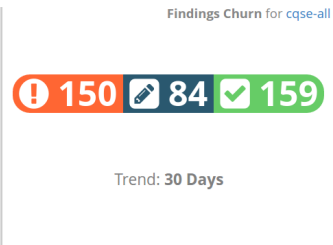


Figure App.98. Findings Churn Widget

Purpose	Shows the findings churn within the configurable time interval. Red indicates added findings, blue indicates findings in modified code (based on quality goal *improving*, see Section 1.2.2), and green indicates removed findings.
Show trend text	Indicates if a label with the trend should be displayed within the widget.
Filters	Allows to select the findings to be shown in this widget based on, for example, finding group and finding category.

A.8 Findings Summary Bar Chart Widget



Figure App.99. Findings Summary Bar Chart Widget

Purpose	Provides a bar chart as summary of all finding categories. Yellow and red indicates the severity of the findings as configured in the analysis profile.
Findings since	Selects a baseline for the findings. Only findings introduced after the baseline are shown.
Detail level	Choose for which level a bar should be created; you can pick »only categories«, »categories and groups« or »categories, groups, and rules«.
Assessment filter	Choose if all, only red or only yellow findings should be displayed.
Show timespan text	Choose whether the timespan since the baseline should be displayed.
Selected categories	Select which finding categories should be considered in this widget. The available categories depend on the underlying analysis profile.
Horizontal orientation	Indicates whether bar chart should be oriented horizontally or vertically.
Sort by findings count	Indicates whether chart should be sorted by number of findings or by category names.

A.9 Findings Summary Table Widget

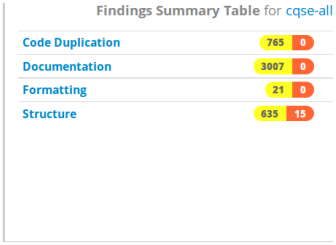


Figure App.100. Findings Summary Table Widget

Purpose	Provides a table with a summary of all finding categories. Yellow and red indicates the severity of the findings as configured in the analysis profile.
...	Please refer to the Findings Summary Bar Chart Widget in Appendix A.8 which has the same configuration options.

A.10 Folder Distribution Pie Chart Widget

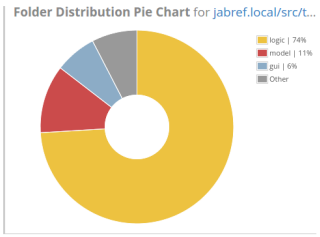


Figure App.101. Folder Distribution Pie Chart Widget

Purpose	Shows metric distribution for subfolders and files (or subcomponents and files) inside a folder (component). For example, you can use this pie chart to see the size distribution in lines of code of a folder's content.
Metric	Select the metric based on which the subfolders or subcomponents are compared.
Legend	Add a legend to the widget.
Labels with % and Values	Creates the legend both with relative as well as with absolute values.
Combine slices below %	Combine subfolders/files into »other« which make up less than the specified amount of % of the overall distribution.

A.11 Label Widget

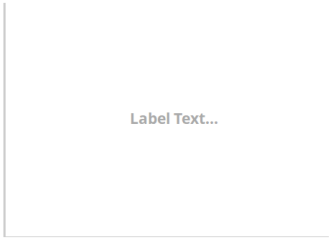
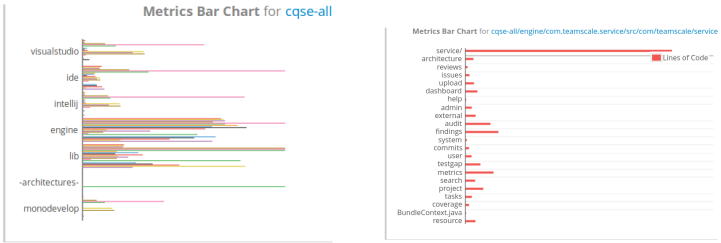


Figure App.102. Label Widget

Purpose	Adds a label to the dashboard.
Text size	Choose between small, medium and large.
Text color	Pick the label color.
Bold text	Uses a bold font instead of regular.
Wrap text	Uses more than one line if necessary.

A.12 Metric Bar Chart Widget



(a) Displaying multiple metrics. (b) Comparing lines of code.

Figure App.103. Metric Bar Chart Widget

Purpose	Displays metrics for folders or components as bar chart. Can be used, for example, to compare subfolders by size (LOC).
Metrics	Select the metrics you want to display as bars. Available are only numeric metrics. As default, all are selected.
Metric Threshold Configuration	Select your threshold configuration profile.
Hide Threshold Configuration Markers	If unchecked, thresholds from the selected threshold configuration will be shown as markers in the bar charts.

Additional Marker Threshold Configuration	For each numeric metric, you can enter an additional threshold. For example, for the clone coverage, enter »0.05« if you want to set a 5% threshold.
Metric bar colors	Choose the colors of the bars. Colors are assigned to the metrics based on the ordering of the metrics.
Horizontal orientation	Indicates whether bar chart should be oriented horizontally or vertically.
Include parent node	Include the folder/component you selected as path for the widget.

A.13 Metric Change Table Widget

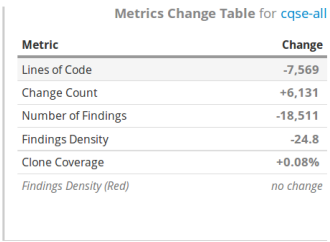


Figure App.104. Metric Change Table Widget

Purpose	Shows metric changes within a configurable time interval in a table.
Metrics	Select the metrics from your analysis profile to be shown in the widget.

A.14 Metric Distribution Pie Chart Widget

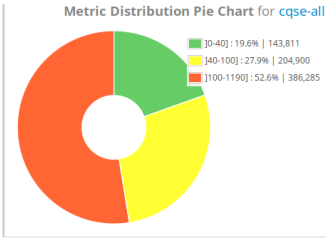


Figure App.105. Metric Distribution Pie Chart Widget

Purpose	Shows how code distributes over user-defined intervals of a metric. Examplary questions that can be answered with this widget are: How much of my code is in overly long files? How much of my code is in files with very high clone coverage? How much of my code is in files with sufficient test coverage?
Primary Metric	Based on the values of the primary metrics, files are classified into different categories. The categories are described by the boundaries. If you select clone coverage as a primary metric, you can display, for example, how much of your code is in files with very high clone coverage.
Boundaries	Sets the thresholds for the primary metric. Based from this thresholds, intervals and corresponding categories are derived. For example, entering two values »0.1 , 0.2« defines three intervals: $[0, 0.1],]0.1, 0.2],]0.2, \infty]$, which could be interpreted as »low«, »medium« and »high« clone coverage.
Colors	For each category defined by the boundaries, you can define a color. Colors are assigned in the same order as the interval/categories are derived.

Aggregation Metric	After each file is classified based on the boundaries of its primary metric, the aggregation metric is used to calculate the distribution. Choosing lines of code, for example, will give you the percentage of your code (in lines of code) that resides within files with a low/medium/high clone coverage.
Legend	Adds a legend.
Boundaries in Legend	Adds a legend and the configured boundaries.
Show absolute values	Shows also the absolute values of the aggregation metric in addition to the relative distribution.

A.15 Metric Distribution Table Widget

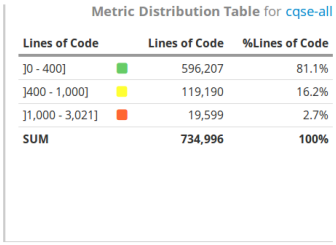


Figure App.106. Metric Distribution Table Widget

Purpose	As the previous *Metric Distribution Pie Chart* widget, this widget shows how code distributes over user-defined intervals of a metric. It only differs by showing the result in a table rather than a pie chart.
...	Please refer to the configuration of the Metric Distribution Pie Chart Widget in Appendix A.14.

A.16 Metric File Distribution Widget

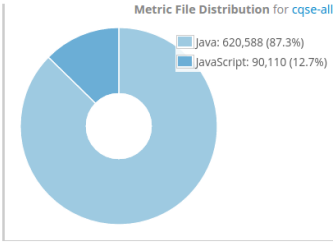
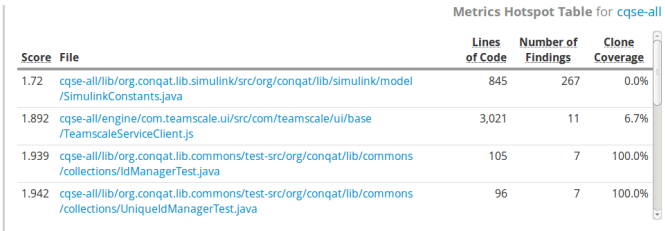


Figure App.107. Metric File Distribution Widget

Purpose	Shows how a size metric distributes over files matching configurable regular expressions. Can be used, for example, to display how the amount of code (in LOC) distributes over Java and JavaScript files.
Metric	Choose between LOC, SLOC and Files (Number of Files) to measure the amount of code matching one of your regular expressions.
File regexes	Enter the regular expression patterns using Java Regex syntax for which you want to know the code distribution. Examples can be »`Java:.*\.java$`« and »`JavaScript:.*\.js$`«, assigning the labels »Java« and »JavaScript « to the two categories of java and javascript files.
Colors	Specify a color for each category of files. Colors are assigned in the same order as the categories are defined.
Legend	Include a legend to the widget.

A.17 Metric Hotspot Table Widget



Metrics Hotspot Table for cqse-all

Score	File	Lines of Code	Number of Findings	Clone Coverage
1.72	cqse-all/lib/org.conqat.lib.simulink/src/org/conqat/lib/simulink/model/SimulinkConstants.java	845	267	0.0%
1.892	cqse-all/engine/com.teamscale.ui/src/com/teamscale/ui/base/TeamscaleServiceClient.js	3,021	11	6.7%
1.939	cqse-all/lib/org.conqat.lib.commons/test-src/org/conqat/lib/commons/collections/IdManagerTest.java	105	7	100.0%
1.942	cqse-all/lib/org.conqat.lib.commons/test-src/org/conqat/lib/commons/collections/UniqueIdManagerTest.java	96	7	100.0%

Figure App.108. Metric Hotspot Table Widget

Purpose	Shows a table with hotspots for the selected metrics. Given the selected metrics, an internal score is calculated to find and rank the hotspots. The smaller the score, the »hotter« the hotspot.
Metrics	Select one ore multiple numeric metrics.
Number of hotspots	Enter how many hotspots (files) should be displayed in the table.
Score cutoff	Files with this score or higher will not be shown in the table.
Show score	Select whether or not to show the score in the table.

A.18 Metrics Table Widget

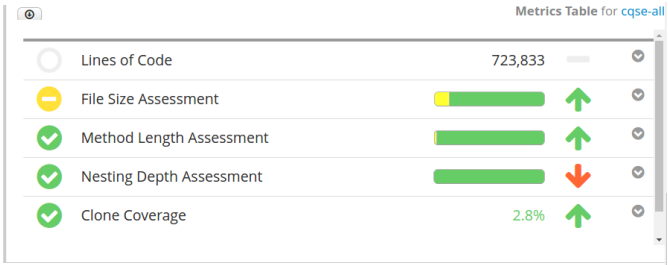
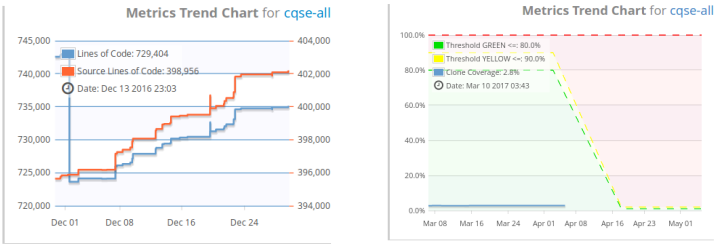


Figure App.109. Metrics Table Widget

Purpose	Shows a table with metrics, along with their trend and assessment based on the underlying threshold configuration.
Metric Threshold Configuration	Select your threshold configuration.
Metrics	Choose the metrics to be shown in the table. Only metrics configured in the threshold configuration are available.

Flatten Single Group	Hides the group title if selected threshold configuration contains only one single group.
Show actions	Shows an action button for each metric to view the history trend of the metric as well as its treemap.
Show values	If checked, shows the values of the metric.
Show icons	If checked, shows the assessment icons on the left (red, yellow or green circle)
Show trend	Shows the trend icons on the right (red or green vertical arrow)
Number of days in future	This option can be used if »Show actions« is checked. The actions provide the opportunity to view the history trend of a metric in a new pop-up dialog. In this dialog, the trend chart can include a certain number of days in future. This can be used for a burn-down chart, when certain target thresholds are set for a metric in the future.

A.19 Metrics Trend Chart Widget



(a) LOC and SLOC Trends. (b) Treshold Configuration Trends.

Figure App.110. Metric Bar Chart Widget

Purpose	Shows the trend chart for one or multiple metrics over time.
Colors	Choose the colors for each metric. Colors are assigned in the order as the metrics are sorted.
Show Thresholds	Shows the thresholds. This works only if you selected exactly one single metric and for this metric, there is a corresponding threshold configured in the selected threshold configuration.
Metrics	Select one ore multiple numeric metrics.
Stacked	Stack all metrics on top of each other. Needs »Include 0« and »One scale« to be enabled.
Include 0	Include 0 in the y-Axis.
One scale	If checked, all metrics are shown on the same scale with just one axis. Otherwise multiple y-Axis with different scales are used.

Number of days in future	Includes future days in the chart with the main purpose to see thresholds configured for the future. Enabling this feature works only, if you selected exactly one single metric and for this metric, there is a corresponding threshold configured in the selected threshold configuration. The right screenshot in Figure App.110 shows an example of a trend that also displays days in the future. This can be seen as the blue line indicating the clone coverage stops on April, 5th. The threshold configurations in the example were set up such that they decline over time.

A.20 Numeric Metric Value Widget

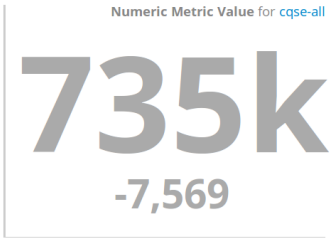


Figure App.111. Numeric Metric Value Widget

Purpose	Shows the value of a numeric metric.
Boundaries	Boundaries are a light-weight version of a threshold configuration. They allow you to configure thresholds to assess a metric. For example for the metric of clone coverage, you can enter the values »0.05,0.10 «. Using the default colors of green, yellow and red, this configuration would display clone coverage values less than 5% with green color, between 5 and 10% with yellow color and above 10% with red color.
Colors	The input for boundaries defines a certain number of intervals. For example, entering two numbers results in three intervals, as explained above. For each interval, you have to pick a color. The color is used as font color of the label when the metric value resides within the corresponding interval.

A.21 Ownership Treemap Widget

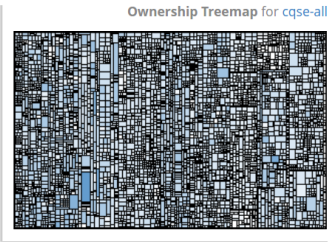


Figure App.112. Ownership Treemap Widget

Purpose	Shows code ownership information for each file in a treemap, i. e. how many developers have changed the corresponding file. The widget reveals, e. g. code areas that only one or few developers are familiar with.
Exclude merge commits	Exclude merge commits to calculate ownership information.
Exclude Teamscale import	Excludes the initial *Teamscale* import commit to calculate ownership information.
Ownership color	Pick the color to display ownership information. The darker the color, the more developers have changed the corresponding file. Areas, that only one developer is familiar with, are displayed in white.

A.22 Single Metric Trend Widget

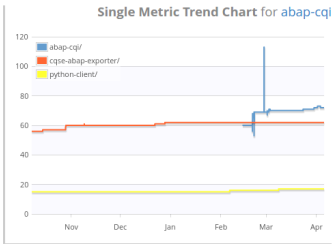


Figure App.113. Single Metric Trend Widget

Purpose	Shows the trend chart for a single metric over time. It can show, however, the same metric for multiple projects.
Additional paths	In addition to the normal path which every widget has, you can add additional paths for this widget. For each configured path, the selected metric will be shown as a separate trend line. Hence, you can, for example, show the lines of code for three different projects in one chart and compare their evolution.
...	For the remaining configuration options, please refer to the very similar *Metrics Trend Chart* widget described in Appendix A.19.

A.23 Task Summary Pie Chart Widget

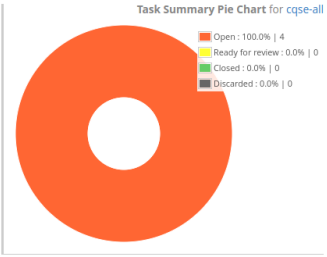


Figure App.114. Task Summary Pie Chart Widget

Purpose	Shows a summary of *Teamscale* tasks by giving a distribution of open, ready for review and closed tasks.
Legend	Include a legend.
Show absolute values	Shows the absolute number of tasks in the legend in addition to the percentage values.
Show discarded tasks	Includes discarded tasks as separate category in the summary.

A.24 Test Gap Overview Widget

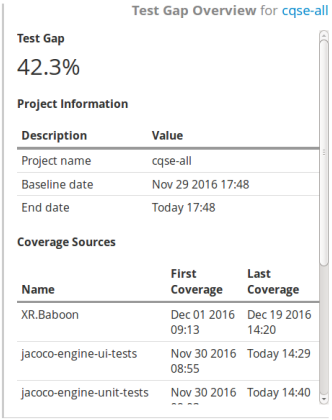


Figure App.115. Test Gap Overview Widget

Purpose	Gives an overview of the results from the test gap analysis.
End date	The point in time until which changes are considered.
Cross-annotate execution	Use additional coverage information from more projects than specified in the given path. This might be useful in setups where you have different testing environments for different branches of the same system.
Coverage sources	Selected which coverage sources should be considered.

A.25 Test Gap Treemap Widget

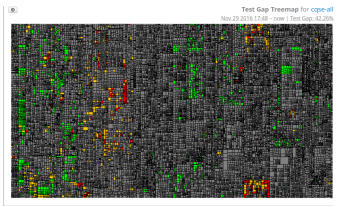
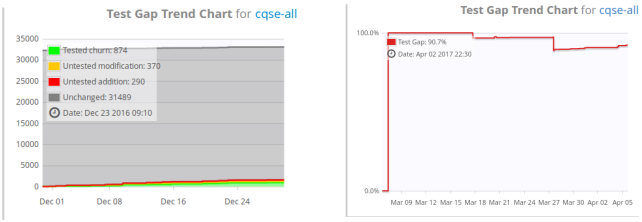


Figure App.116. Test Gap Treemap Widget

Purpose	Shows the results from the test gap analysis in a treemap, in which every rectangle represents a single method.
...	Some configuration options have already been described in the *Test Gap Overview* widget in Appendix A.24.
Execution only	Shows only the execution information from the test runs (without combining it with change information).
Outline depth	Determines how many level of packages should be outlined. Outlined packages will be surrounded by a thicker line.
Outline color	Color used to outline packages.
Issue id	Only methods changed during the implementation of a specific issue are shown in the treemap. If set, baseline and end date parameters will be ignored.

A.26 Test Gap Chart Widget



(a) Complete trend. (b) Only percent trend.

Figure App.117. Test Gap Chart Widget

Purpose	Shows test gap analysis results as a trend chart.
...	Some configuration options have already been described in the *Test Gap Overview* widget in Appendix A.24.
Focus on changes	Rescale the y-Axis to eliminate the big gray and uninformative areas to focus on where the trend actually changes.
Show only percent trend	Shows the test gap analysis information as one single numeric metric (»test gaps«) and displays its trend. This option cannot be used, if the previous one is enabled and vice-versa.
Execution only	Show the execution information only (without combining it with change information).

A.27 Treemap Widget

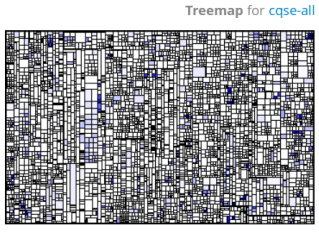


Figure App.118. Treemap Widget

Purpose	Shows a metric with a treemap in which every rectangle represents a file.
Area Metric	Metric used to determine the area size of a rectangle representing a file. Only size metrics are available. If the files metric is chosen, all rectangles will have the same (default) size.
Color Metric	Metric used to color a rectangle. The higher the value for a file, the darker the color for the corresponding rectangle.
Color	Color for the treemap.
Included files	Only files matching the provided regular pattern are displayed in the code city. To include only Java files, use the Java Regex syntax and enter »`.* \.java`«. You can add multiple include patterns, too.

Excluded files	Files matching this regular pattern will be excluded from the treemap.
Enable cushion	Enables a cushion effect on the treemap.
Minimum value	Only for numeric metrics. All values equal or below this value are colored with white. Use -1 to automatically use the lowest metric value as minimum value.
Maximum value	Only for numeric metrics. All values greater or equal to this value are colored with the specified color. Use -1 to automatically use the largest metric value as minimum value. You might want to use this option if you have a single but very large value – an outlier – in your data set which almost dismisses the gradient in the coloring in the treemap.

B Repository Connector Options

To create a project, *Teamscale* needs to be connected to your version control system. To this end, it offers several different repository connectors. We describe the options for each repository connector. Setting up the other connectors to source code repositories, issue trackers, and review tools is very similar to the git connector setup. Many connectors share general options such as a **polling interval** and therefore, we only describe the options (if there are any) for first connector that uses it. If there are no additional options in a connector, we don't document the connector separately.

GIT Configuration Options

Branch Name	The name of the branch to import.
Account	Account credentials that *Teamscale* will use to access the repository. Clicking on the + button opens a credentials creation dialog. The **Account Name** setting in the dialog is *Teamscale*-internal (does not need to match anything in git).
Path suffix	Should *Teamscale* analyze a only a sub directory of the repository?
Repository identifer	*Teamscale* will use this identifier to distinguish different repositories in the same project. (*Teamscale*-internal only)

GIT Configuration Options (continued)

Included file names	You can use these properties to focus the analysis on a subset of files.
Excluded file names	You can exclude certain files.
Enable branch analysis	This checkbox activates our new branch support feature. If you select it, *Teamscale* will analyze all branches of the repository. If branch analysis is enabled, the **Branch Name** option specifies the main branch (usually »master«).
Included branches	Explicitly control which branches will be analyzed. (Has no effect if **Enable branch analysis** is not selected.)
Excluded branches	Explicitly state which branches to exclude.
Start revision	The first revision that will be analyzed. You can enter a date (which selects the first revision after the date) or a revision hash.
Content exclude	A comma-separated list of regular expressions. *Teamscale* will exclude all files that match one of these filters.
Polling interval	How often should we query the repository and update our findings? (Unit is seconds)

File System Connector Options

Input directory	The path of the source-code directory.

Options for a Connector to a Team Foundation Server

Branch lookup paths	The path in the repository where *Teamscale* will search for branches. Leave the option empty to search in the entire repository.
Branch path suffix	The sub path (relative to the branch directories) that should be analyzed. Leave empty to analyze all code within each branch directory.

C Issue Tracker Connector Options

Options for a Connector to Bugzilla

Issue ID pattern	A regular expression used to filter issue IDs from commit messages.
Projects	Names of the projects in the issue tracker that should be associated with the *Teamscale* project.

Options for a Connector to Redmine

API access key	An API key used for accessing the redmine server.

Options For Team Foundation Server Work Item Tracker	
Areas	A list of full area paths to extract work items from.
Include Sub-Areas	If checked, issues from sub-areas are retrieved as well, otherwise just issues exactly matching one of the area paths are retrieved.

D Code Review Tool Connector Options

Options For a Connector to Crucible	
Crucible Projects	A comma-separated list of Crucible projects to poll (leave empty for all projects).

E Entity Selection Predicates

Teamscale provides special predicates for the selection of code entities. For example, you can use these predicates to define methods that require a documentation comment in the analysis profile.

An entity selection predicate consists of primitives that can be connected using negation »!«, conjunction »&«, disjunction »|«, and parentheses. The primitives are listed in the following configuration boxes. Obviously, not all predicates are applicable to all programming languages.

An example that selects all public simple getters and setters would be

```
public & (simpleGetter | simpleSetter)
```

A good starting point for the Java and C# languages is

```
public & (type | method | property) & !(simpleGetter |
    simpleSetter | annotated(Override) | override)
```

And for C/C++ you might want something along these lines, if you only document the headers and not the C files. But as this depends a lot on the team, you might have to adjust this.

```
type | (public & (attribute | (method & declaration)))
```

Selection Predicate Primitives (Part 1)

language(name)	Selects entities in files with the given language. If the given language name is invalid, no entities are matched. Valid names are for example JAVA and CPP.
module	Selects namespaces and modules.
type	Selects types, such as classes or structs.
method	Selects methods and functions.
declaration	Selects declarations of types and methods.
internal	Selects entities that are internal (C# keyword).
attribute	Selects attributes, fields and global variables.
property	Selects all properties: defined attributes that have only method children.
statement	Selects all statements.
meta	Selects all meta information (defines, annotations, etc.).
public	Matches all public entities as well as methods inside interfaces, which are considered public in most languages.
protected	Matches all protected entities.
override	Matches all entities marked with override keyword.
private	Matches all private entities.

Selection Predicate Primitives (Part 2)

default	Matches all entities with default visibility.
final	Matches all final entities.
static	Matches all static entities.
primitive	Matches all primitive entities (i.e. those without children).
subtype(name)	Matches entities by their subtype. The name parameter could be for example *if* for if statements.
typeSubtype(name)	Matches entities by their parent type's subtype (e.g. to select all members of interfaces).
name(name)	Matches entities by their name. For example a method name.
nameRegex(regex)	Matches entities by their name using regular expression.
annotated(name)	Matches all entities that are annotated with an annotation of given name (excluding the »@« sign).
simpleGetter	Matches simple getters, i. e. methods starting with »get« or »is« and containing at most one statement.
simpleSetter	Matches simple setters, i. e. methods starting with »set« and containing at most one statement.
simpleMethod(prefix)	Matches simple methods whose name starts with the given prefix. A method is simple, if it contains as most one statement.

F Custom Checks

Custom checks can be used to easily define rules checked by *Teamscale*. The *Teamscale* distribution already comes with about 100 custom checks for different languages. Users can write their own custom checks as documented in Section 4.3.1. Each custom check can be enabled or disabled separately for each analysis profile that contains one of the languages supported by the custom check. A check (and its options) can be configured in the configuration dialog of its analysis group in the analysis profile configuration (Figure 4.80).

The following sections list the available custom checks, ordered by their supported languages and by the analysis groups in which they can be configured. The analysis groups are: Unused code, Test Conventions, Exception Handling, Correctness, Performance, Security, Bad practice, Javascript (built-in), and Cross-Browser Compatibility.

F.1 Custom Checks for ABAP

F.1.1 Bad practice

Assert commands should have ID

Assert should always have an ID.

No automated code manipulation

Programs should not create, delete, or change coding.

No static breakpoints in code

Coded breakpoints should not be used.

No call to external form routines

External form routines should not be called.

No complex logic in event block

Complicated statement blocks should not be implemented directly in event blocks but in appropriate modularization units.

No complex logic in macro

A macro should not contain complex logic.

No data declarations in functional code blocks

Data declarations should not be placed in functional code blocks.

No database operation in macro

A macro should not contain database operations.

No non-class-based exceptions via RAISE statement

Non-class-based exceptions should not be thrown using RAISE statement. Use 'MESSAGE Ennn with <…> RAISING <exception>' or class based exceptions via 'RAISE EXCEPTION <…>' instead.

No use of task handler functions

Task handler functions (FuGr THFB, THFB2, THFB_ALV, THSCHED, THTX) should not be used.

No write access to SAP standard tables

Write accesses to SAP standard tables by customized program components should not occur.

No write access to system fields

System fields (Sy-xxxx, fields of structure SYST) should only be read (as they are set by the runtime environment).

Unescaped WHERE ... LIKE

SELECTs with non-literal WHERE …LIKE should always specify ESCAPE to avoid unintended wildcards.

F.1.2 Security

Suspicious call of function modules RFC_REMOTE_EXEC or RFC_REMOTE_PIPE

The function modules RFC_REMOTE_EXEC and RFC_REMOTE_PIPE allow to execute operating system commands. In custom code, these function modules should not be used. The SXPG framework should be used instead.

Suspicious usage of CALL system function

The internal statement CALL *cfunc* executes a system function *cfunc*. Some system functions, e.g. 'SYSTEM' or 'ThWpInfo' allow to execute any command of the operating system and should be avoided. The SXPG framework should be used instead.

Control flow depending on system variable (SY-...)

A system variable, e.g., SY-UNAME (user name of current user) or SY-SYSID (SAP system id) is misused to control the program flow. This should should be avoided, since it may lead to different behavior depending on logon data or platform.

Critical Statement

Critical statements should be avoided since the danger to introduce security flaws is high. Usually these statements should be never used in production code. These statements might be necessary e.g., for code generators or other development related activities, but then the containing objects should be placed in packages which will not be transported to the production system. Usages of critical statements should properly document why the statement is needed.

The statements which are regarded as critical can be specified in the analysis profile.

Statements which are considered as critical by default are:

- EDITOR-CALL - allows to modify ABAP programs regardless if development authorization is granted to the executing user
- GENERATE SUBROUTINE POOL - generates FORM routines at runtime, security critical code could be obfuscated in this way. A severe vulnerability exists if the code is generated from an external input.
- INSERT REPORT - generates programs (REPORTS) at runtime, as with GENERATE SUBROUTINE POOL, security critical code could be obfuscated in this way. A severe vulnerability exists if the code is generated from external input.

Cross-client database access

The addition CLIENT SPECIFIED for the Open SQL statements SELECT, INSERT, UPDATE, MODIFY, DELETE and OPEN CURSOR allows access to data of an other client than the client to which the current user is logged in. Since there is no authority check performed, a user could possibly access data in a client where he has no authorization.

Note that there is no need to add a condition to the WHERE clause for restricting the selection to the current logon client (e.g. MANDT = SY-MANDT) together with CLIENT-SPECIFIED. Without CLIENT SPECIFIED this is ensured by the Open SQL statements.

See also ABAP Security Note on Cross-Client Database Access.

Hard-coded password

Hard-coded passwords should be avoided, since hard-coded credentials like user name or password are a potential security risk. Anybody who has access to the source code (regardless if on the development or production system) would be able to read the password.

Hard-coded user name

Hard-coded user names should be avoided, since hard-coded credentials like user name or password are a potential security risk. In any case it should be avoided that the program flow depends on the name of the currently active user.

Missing authority check at CALL TRANSACTION

CALL TRANSACTION executes an transaction with the given transaction code. Until SAP_BASIS release 7.40 no authority check was performed for CALL TRANSACTION, from SAP_BASIS release 7.40 on, a authority check is only performed, if the addition WITH AUTHORITY CHECK is used.

To solve this issue, the addition WITH AUTHORITY CHECK should be added (if SAP_BASIS version is 7.40 or higher)or the function module AUTHORITY_CHECK_TCODE should be called before.

Missing AUTHORITY-CHECK in reports/programs

Executable programs (reports)—introduced with statement REPORT or PROGRAM—should contain an explicit AUTHORITY-CHECK before executing any statement. This is especially relevant since programs may be executed not only via transaction codes but by other programs using the SUBMIT statement. In these cases no implicit authority check is performed by the system.

The AUTHORITY-CHECK is expected to be placed at the beginning of the INITIALIZATION or START-OF-SELECTION event block, or before the first executable statement if none of these blocks is explicitly specified.

Custom authority check procedures which encapsulate the AUTHORITY-CHECK statement can be specified in the analysis profile as alternative authority check statements to avoid false positives.

Missing check of SY-SUBRC return code after authority check

Improper authorization: The statements AUTHORITY-CHECK and CALL FUNCTION 'AUTHORITY_CHECK_TCODE' require an check of the return code SY-SUBRC to be effective. This check is missing and has to be added.

Suspicious usage of OPEN DATASET

When used with the addtion FILTER, OPEN DATASET allows to execute any operating system command. Thus, the FILTER addtion should be avoided in general. The SXPG framework should be used for OS calls instead.

Suspicious Statements

Suspicious statements are statements which may introduce security risks if not used properly.

Examples of such statements are:

- CALL TRANSACTION - executes the stated SAP transaction code, authorization of the user is only check if stated explicitly.
- EXEC SQL - executes native SQL

The list of relevant statements is specified in the analysis profile.

Usage of system variables (SY-...)

A system variable, e.g., SY-UNAME (user name of current user) or SY-SYSID (SAP system id) is accessed. System variables which contain user, installation or client specific values should be avoided, since these may lead to different behavior depending on logon data or platform.

F.2 Custom Checks for C#

F.2.1 Bad practice

File must have a valid copyright header

Ensures that each file has a valid copyright header.

Avoid using ternary operators

Ternary operators make debugging difficult and represent code that is hard to read.

Usage of goto

The usage of goto is discouraged as it leads to code that is hard to read.

Switch/case without default

Switch/Case statements must have a default case.

Abstract types should not have constructors

Constructors on abstract types can be called only by derived types. Because public constructors create instances of a type, and you cannot create instances of an abstract type. An abstract type that has a public constructor is incorrectly designed.

Avoid partial classes

Partial classes allow the class definitions to be broken up across multiple files. This is considered as a bad practice, because you will not be able to see the bigger picture of the class anymore and may not realize the side-effects you are introducing to the other partial files. By only viewing and working on a part of the class, you will not see all the internal fields that need updating, thus leading to incompatible changes and higher possibility of breaking code or dependencies in the rest of the class.

Methods should not have too many parameters

Methods should not have too many parameters in order to be maintainable.

Do not declare virtual members in sealed types

A public type is sealed and declares a method that is both virtual (Overridable in Visual Basic) and not final. This rule does not report violations for delegate types, which must follow this pattern.

Pointers should not be visible

A public or protected System.IntPtr or System.UIntPtr field is not readonly.

Use params for variable arguments

A public or protected type contains a public or protected method that uses the VarArgs calling convention.

Usage of the var keyword

The usage of the var keyword is discouraged as it makes the code harder to read.

F.2.2 Correctness

Override GetHashCode on overriding Equals

A public type overrides Object.Equals but does not override Object.GetHashCode or vice versa.

Test for NaN correctly

Creates a finding if an expression tests a value against Single.Nan or Double.Nan.

F.2.3 Exception Handling

Exception stacktrace is lost

If you catch an exception and then throw a different exception, you should preserve the original stacktrace by passing the caught exception to the constructor of the thrown exception. Failing to do so will cause the original stacktrace to be lost, which will hinder debugging the root cause.

F.2.4 Performance

Use literals where appropriate

A field is declared static and readonly and is initialized with a value that is computable at compile time. Change the declaration to a const field so that the value is computed at compile time instead of at runtime.

F.2.5 Unused code

Avoid unused private fields

This rule is reported when a private field in your code exists but is not used by any code path.

Avoid unused private methods

This rule is reported when a private method exists but is never called by any code path.

Remove empty finalizers

Creates a finding if a type implements a finalizer that is empty, calls only the base type finalizer, or calls only conditionally emitted methods.

F.3 Custom Checks for C/C++

F.3.1 Bad practice

File must have a valid copyright header

Ensures that each file has a valid copyright header.

Avoid using ternary operators

Ternary operators make debugging difficult and represent code that is hard to read.

Usage of goto

The usage of goto is discouraged as it leads to code that is hard to read.

Methods should not have arrays as arguments.

Having arrays as method argument should be avoided. This check reports an extra finding if 'sizeof' is used on an array argument. Using 'sizeof' on the array argument returns the size of the pointer instead of the byte size of the array as they are treated as pointers by the compiler.

Exception not caught by reference

Exceptions should always be caught by reference to avoid unnecessary copies.

Do not use forbidden types (C++)

Reports findings if a type from a user-defined list is used.

Methods should not have too many parameters (C++)

Methods should not have too many parameters in order to be maintainable.

Multiple classes in one file (C++)

Defining more than one class in a file makes finding the file for a given class very hard. Therefore each file should contain only one top-level class.

Public class attribute (C++)

Classes should not have public attributes as this violates encapsulation. Instead access should be provided using public methods. This rule does not apply to constants and to attributes in structs.

Top-level class name (C++)

Reports if the top-level class name does not match the file name.

F.3.2 Correctness

Missing header protection

Header files must protect from multiple inclusion using the preprocessor. The macro variable used for this must match the filename.

Missing virtual destructor

If a class has no virtual destructor and some other class inherits from it, this can lead to memory not being freed correctly.

F.4 Custom Checks for IEC 61131-3 ST

F.4.1 Bad practice

Assert uppercase keywords and operators

All keywords and operators in Structured Text should be written in uppercase.

F.5 Custom Checks for Java

F.5.1 Bad practice

File must have a valid copyright header

Ensures that each file has a valid copyright header.

Avoid using ternary operators

Ternary operators make debugging difficult and represent code that is hard to read.

Switch/case without default

Switch/Case statements must have a default case.

Avoid inverted conditions

To improve readability, it is desirable to have an assertFalse statement when checking for conditions to be unsatisfied (and vice versa)

Never check for more than one condition in assert statements

Assert statements should never test more than one feature. This way, failed tests can be linked to errors more easily.

Bad assignments

Using assignments in places where comparison operators are frequently used can lead to unreadable code and thus should be avoided.

Transformation of byte sequence into String must consider encoding

When transforming byte sequences into Strings the encoding must be considered. Using the platform encoding may result in interoperability issues.

Avoid leaving deprecated classes / methods / fields

Deprecated classes / methods / fields

Do not compare this and null

This must not be compared with null, because this cannot be null.

finalize() may not be overwritten

The method finalize() may not be overwritten.

hashCode() and equals()

A class implementing one of equals() or hashCode() should also implement the other.

Avoid ignoring JUnit tests

JUnit tests should not be ignored for longer periods of time.

Inner type visiblity

Inner types must have private or protected visibility. If the type must be public, move it to its own file.

Field could be final

Fields that can be made final, should be marked as such to avoid accidental assignment to them.

Import from java.lang

Types from java.lang do not need to be imported explicitly as they are imported by default.

Methods should not have too many parameters

Methods should not have too many parameters in order to be maintainable.

Java unwanted method calls

Certain methods like console or debugging methods should be avoided in code because they could introduce unwanted side effects or reduce performance.

Specify new Logger instances with current class

The logger should be specified with the class it operates in. This way, the log output can be linked to said class.

Maximum type name length

Type name should not be longer than a given number of characters (default=30).

Multiple assignments

Multiple assignments in one statement should be avoided as they hamper readability.

Multiple top-level classes

Multiple top-level classes in a single file are possible in Java but make them hard to find for developers. Better use separate files, inner classes or anonymous classes.

No star imports

Star imports must not be used, because they clutter the local namespace.

Too many constructors

Classes should not contain more than the configured number of constructors (default=5). Too many constructors make the class difficult to use and should thus be avoided.

Too many imports

The number of imports should not be too big. Too many imports are an indication of too many concepts being used in one class.

Number of method parameters

Methods should not take more than 5 parameters. Using a large number of parameters is an indication of high complexity and should be avoided.

Too many methods

Classes should not contain more than then configured number of methods (default=32) as this indicates a high complexity that is difficult to understand.

Modifiers must be in the right order

According to the Java Language Specification the modifiers on methods and fields should be ordered as follows: public/protected/private, abstract, static, final, transient, volatile, synchronized, native, strictfp.

Package name mismatch

Package names must start with one of the configured prefixes.

Call to printStackTrace()

Avoid use of printStackTrace() calls.

Private methods must not be final

Since private methods cannot be overridden, declaring them as final is unnecessary.

Public class attribute

Classes should not have public attributes as this violates encapsulation. Instead access should be provided using public methods.

Method must not be named like constructor

A method should not be named like its containing type's constructor, because this can lead to confusion.

Same package import

Must not import types from own package, as they are accessible without import.

Properly initialize static variable

Static variables that are not protected or private should be initialized directly or in a static initializer.

Suspicious methods

If methods are named similar to ones from Object, this might be a typing error or can lead to confusion. Those methods should be renamed.

Case Blocks Length

Each case block in a switch statement should not be longer than the configured number of statements (default=10).

Unused imports

Unused imports are unnecessary.

Unwanted constructor calls

Constructors of some types, e.g., primitive wrappers and String must not be called, because they only create extra objects which take up more memory and reduce performance, without any other functional effect.

Java unwanted imports

Creates a finding for unwanted imports.

F.5.2 Correctness

Method returns null but return type is Optional

Optional.empty() should be returned instead of null.

F.5.3 Exception Handling

Exception stacktrace is lost

If you catch an exception and then throw a different exception, you should preserve the original stacktrace by passing the caught exception to the constructor of the thrown exception. Failing to do so will cause the original stacktrace to be lost, which will hinder debugging the root cause.

Catch of generic exception

The exceptions java.lang.Throwable, java.lang.Error, java.lang.RuntimeException, and java.lang.Exception must not be handled by a catch block.

Custom exception class

Creates a finding for each custom exception class.

Throw of generic exception

The exceptions java.lang.Throwable, java.lang.Error, java.lang.RuntimeException and java.lang.Exception must not be thrown by methods. Appropriate subclasses should be used instead.

F.5.4 Performance

Calling contains on list

Calling contains() on lists is inefficient. Use hashsets instead.

F.5.5 Test Conventions

Annotation in JUnit 4 test class missing

Public, non-annotated methods in JUnit 4 test classes may be an indicator for a missing @Test annotation.

Test class naming convention

Test classes should be distinguishable from other classes by their name. Reports test classes whose name does not match a configured pattern.

Unit test classes should be located in the same package as the class under test

Unit test classes should be located in the same package as the class under test.

F.5.6 Unused code

Avoid unused private methods

This rule is reported when a private method in your code exists but is not called by any code path.

Avoid unused private fields

This rule is reported when a private field in your code exists but is not used by any code path.

F.6 Custom Checks for JavaScript

F.6.1 Bad practice

File must have a valid copyright header

Ensures that each file has a valid copyright header.

Avoid using ternary operators

Ternary operators make debugging difficult and represent code that is hard to read.

Javascript unwanted method calls

Certain methods like console or debugging methods should be avoided in code because they could introduce unwanted side effects or reduce performance.

Use consistent form for String literals

Use consistent form for String literals, i.e. only one of ″ or ″″

F.6.2 Cross-Browser Compatibility

Internet Explorer compatibility check

Some method calls on global objects are not be supported by all browsers. Please refer to online resources like the Mozilla Developer Network (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference) for details.

Google Closure Method Check

For optimal cross-browser compatibility, certain JavaScript method calls should be replaced with their matching Google Closure methods. Please note that this check uses a heuristic which might lead to false positives.

F.6.3 Javascript (built-in)

Do not use == and !=

The comparison operators == and != should be avoided. Their behavior is inconsistent and the performance of === and !== is better.

F.7 Custom Checks for PL/SQL

F.7.1 Bad practice

Plain SQL SELECT column count

In case you have to select lots of columns, a cursor and FOR loop is the better way as you don't need to add so many local variables.

Unclosed cursors

An opened cursor is not closed within the same procedure.

F.7.2 Performance

No SELECT *

Never use 'SELECT * FROM xxx'—only select the columns you need.

Prefer JOIN over Get_

Do not use Get-functions if it's not absolutely needed—join the tables instead.

F.8 Custom Checks for Python

F.8.1 Bad practice

Do not compare None with == or !=

None should not be compared with == or !=. Instead 'is' or 'is not' should be used.

Do not shadow built-ins

Built-in names should not be used as identifier for variables, parameters, methods and types.

Enclose doc-strings in triple quotes

Documentation strings should be enclosed in triple quotes.

Redundant parentheses check

Reports redundant parentheses within if, while, for and return statements.

F.9 Custom Checks for Xtend

F.9.1 Bad practice

Switch/case without default

Switch/Case statements must have a default case.

Bad assignments

Using assignments in places where comparison operators are frequently used can lead to unreadable code and thus should be avoided.

Do not compare this and null

This must not be compared with null, because this cannot be null.

hashCode() and equals()

A class implementing one of equals() or hashCode() should also implement the other.

Import from java.lang

Types from java.lang do not need to be imported explicitly as they are imported by default.

No star imports

Star imports must not be used, because they clutter the local namespace.

Private methods must not be final

Since private methods cannot be overridden, declaring them as final is unnecessary.

Same package import

Must not import types from own package, as they are accessible without import.

F.9.2 Unused code

Avoid unused private methods

This rule is reported when a private method in your code exists but is not called by any code path.



Glossary

analysis profile An analysis profile comprises a set of analyses and their configurations. For each language, there is a default analysis profile, but *Teamscale* users can also configure their individual profiles tailored to suit their needs. 33, 34, 39, 48, 115, 123, 145, 146, 150, 180, 181, 185, 186, 190, 247

assessment metric An assessment metric is a metric which divides the system into parts belonging to the green assessment, the yellow and the red one. Example: The metric File Size is an assessment metric. Green files are, for example, those with less than 300 SLOC, yellow are those with less then 750 SLOC and the remaining ones are red. 52, 180, 181, 247

baseline A baseline is a point in time defined by the user. A baseline is often used as reference point to compare the current quality status against, for example in the delta analysis. In Teamscale, a baseline is defined with a calendar date and a corresponding label. It is thereby independent of the underlying branch in case branch support is enabled. 51, 60, 64, 66, 76, 77, 78, 97, 101, 142, 178, 185, 244

blacklist To blacklist a finding means that this finding will be put in the blacklist and not shown directly anymore. In the *Code Detail View* or in the IDE integration, it will be not shown at all. It can, however, still be inspected in the blacklist view. 99

branch support If enabled, the branch support feature allows *Teamscale* to not only analyze the »master« or »trunk« of a repository, but also its branches. 64, 76, 90, 93, 99, 138, 139, 148, 150, 178, 244, 250

clone coverage The clone coverage indicates the percentage of code lines which are covered by at least one clone. It can be interpreted as the probability that a random (evenly distributed) change to a code line needs to be propagated to at least one other clone instance. 55, 80, 81

component In an architecture specification, a system is modeled as a hierarchy of components. A component has a unique name and may have sub-components and mappings. 124, 248

custom check A custom check is an analysis implemented by the user which can be integrated in *Teamscale*. *Teamscale* also comes with a set of pre-written custom checks. 27, 38, 153

delta The delta between two snapshots contains the changes from the repository between a start date and an end date as well as differences in the quality status. It also contains the differences

in metric values and assessments as well as the finding churn, for example. Often, the start date is referred to as baseline for the delta. 138, 140, 141, 245

delta analysis The delta analysis computes the delta. 57, 64, 138

dialog A dialog is a UI element of *Teamscale*'s web interface. It dynamically pop ups if the user executes a certain task and closes when the user is done with the task. The user has to press a button in order to close a dialog. 76

external upload commit These commits are not present in the version control system of the project, but artificially added to *Teamscale*. They provide data from external analysis tools, such as test coverage tools, for example. Often these commits are created by the nightly build system. 92

file dependency In a file-based architecture, the path to the source code files are used for dependencies. 124, 128, 130, 247

finding A finding refers to a specific region in the code, that is likely to hamper software quality. The region in the code is referred to as finding location. Generally, a finding belongs to one finding group. A finding can occur when a metric and its corresponding threshold are violated, for example, when a file exceeds the size limit of 300 SLOC. Then this overly long file becomes a finding. Alternatively, a finding can directly results from a quality analysis. The code clone detection, for example, reveals clones as findings. These clones do not result from a threshold violation of the corresponding clone coverage metric, but are findings themselves. Each finding as a introduction date in *Teamscale*. 15, 16, 24, 36, 55, 59, 76, 96, 127, 143, 249

finding group A finding group describes a single analysis or a group of analyses creating findings. For example, there are three dif-

ferent finding groups »File Size«, »Method Length« and »Nesting Depth« which represent one analysis each. The finding group »Comment quality« comprises several analyses, detecting unrelated member comments or empty interface comments, for example. A finding group is always associated with one finding category. 48, 62, 96, 98, 143, 144, 184, 245, 246

finding category A finding category combines several finding groups. For example, the finding category »Structure« contains the three groups »File Size«, »Method Length« and »Nesting Depth«. A finding category represents a quality indicator. 48, 62, 98, 143, 144, 184, 185, 186, 245

finding location The location can span several files (e. g. a code clone), a single file (e. g. a overly long file), a method (e. g. an unused method) or a single line (e. g. a naming convention violation). 46, 96, 99, 143, 245

findings count This metric denotes the number of all findings for a project. Naturally, it depends on which analyses are configured. 80

findings churn A finding churn indicates for a given time internal, how many findings were added to or removed from the system plural. 58, 59, 92, 94, 138, 142, 184

introduction date The introduction date refers to the revision in which a finding was first detected by *Teamscale*. For internal analysis, this refers to the revision with which the developer introduced the finding in the code base. For external analysis, which are integrated via a nightly build, this refers to the revision with which the finding was first uploaded to *Teamscale*. Hence, the introduction date then refers to the first detection time, not the actual introduction time. 96, 245

issue metric An issue metric is the numeric result of an issue query which can be saved in *Teamscale* and, thus, be used continuously. 120

lines of code This metric counts all lines of code of a file as displayed to the developer. The count includes empty lines and comments. 52, 55, 80, 107, 136, 187, 189

mapping Mappings between code and architecture define which implementation artifacts belong to which component in the architecture definition. If the architecture is based on file dependencies, a mapping specifies which files belong a component. For type dependencies, it specifies which types map to a component. 126, 131, 244

method length This metric is an assessment metric and denotes the distribution of code in short (green), long (yellow) and very long (red) methods. The length of each method category is configured by a threshold in the analysis profile. 80

metric A metric captures an automatically measurable aspect of software quality. In can either be a numeric metric or an assessment metric. 24, 76, 142

nesting depth This metric is an assessment metric and denotes the distribution of code over methods with low (green), high (yellow) and very high (red) nesting. Each nesting category is configured by a threshold in the analysis profile. 80

non-code metric A metric that cannot (directly) be derived from source code, e. g. the build status of your system. *Teamscale* offers a flexible mechanism to integrate show such custom metrics. 105, 153

numeric metric A numeric metric is a metric that consists of a single numeric value. Example: The clone coverage is an example of

a numeric metric. Its value could be 20%. Another example is the metric number of files. Its value could be 4000. 52, 55, 56, 80, 189, 195, 198, 200

orphan If a file or a type are not mapped to any component in an architecture specification, they are marked as orphan. 127, 128, 130

page A page is a UI element of *Teamscale*'s web interface. If a perspective has a sidebar on the right hand side, then each entry in this sidebar represents a page. 76, 78, 250

perspective A perspective is the main UI element in *Teamscale*. Toplevel, *Teamscale* consists of serveral different perspective which can be navigated in the top header row of the web UI. 76, 248

policy A policy in an architecture specification determines which components may be in a relationship (allow, deny, tolerate) with each other. 127, 132, 250

quality indicator A quality indicator denotes a certain quality aspect. For example, »Structure« or »Code Duplication« represent quality indicators. 36, 38, 123, 246

quality goal A quality goal represents the target line for the quality status of a system. It can have one of the following four values: perfect (no findings at all), improving (no new findings and no findings in modified code), preserving (no new findings) or none (any number of findings). 16, 62, 78, 97, 101, 111, 143, 184

section A section is a part of a view that is independent of the other content of that view, e. g. one of the main tables of the **Metrics** perspective. 105

severity The severity of a finding is expressed by either a yellow or a red color. It can be configured in the analysis profile and used to filter findings. 38, 97, 153, 185, 186

size metric A size metric indicates the size of a system. It can either count the number of files, lines of code, or source lines of code. 194, 207

task A task in *Teamscale* is a concept to schedule findings for removal and keep track of the process. Multiple findings can be grouped together to one task. 97, 100, 102, 203

Test Gap ratio This metric denouts the amount of modified or new methods covered by tests divided by the total amount of modified or new methods. 72

test gap A test gap is a method that has been changed but not tested ever since. 65, 71, 204, 205, 206

threshold A threshold can be used to derive findings from metrics. For example, the metric »File Size« has two thresholds, set to 300 (yellow) and 750 (red) as default. Hence, files which are longer thann 300 SLOC, become a yellow finding. Files longer than 750 SLOC become a red finding. 34, 247

threshold configuration An threshold configuration comprises a set of thresholds for the analyses. 106, 145, 155, 189, 196, 198, 200

timetravel The timetravel is a feature of *Teamscale* which makes it possible to show all of *Teamscale*'s content in a historized form, i. e. to display all information at any point in time of the project's history. 51, 88, 91, 178

treemap Treemaps visualize individual metrics in relation to the source code structure and file sizes. Therefore, directories or source

code files of the analyzed system are drawn as rectangles. The area of the rectangle corresponds to the number of lines contained in the file (or for directories all the files included in the directory). The position of the rectangle follows the directory structure, i. e. rectangles for files in the same directory are drawn next to each other. The shading of the rectangles indicates the hierarchical nesting of the files and directories. 67, 81, 108, 141, 196, 201, 205, 207

trend A trend indicates the evolution of a metric over time. A trend can be calculated both for a numeric metric as well as for an assessment metric. In this first case, the trend is just a simple function in the mathematical sense. In the second case, the trend chart indicates three values per time: the green, the yellow, and the red value and shows how the system distribution over these three categories evolves. If branch support is disabled, the history of the system and, hence, a metric trend, is linear by default. If branch support is enabled, however, the history of a branch is not necessarily linear. To display a linear trend, the »first-parent« stragety is used to pick one parent for each commit. 52, 54, 80, 108, 196

type dependency In a type-based architecture, the full-qualified names of all types (classes, structs, enums etc.) in the codebase are used for dependencies. 124, 128, 130, 247

view A view is a UI element of *Teamscale*'s web interface. In constrast to pages, it cannot be references by the sidebar but appears and disappears as the user navigates and performs certain tasks. 76, 78, 248

violation If a dependency contradicts a modeled policy, it will be treated as an architecture violation. 127, 128



Bibliography

Papers

[Ede+13] Sebastian Eder, Benedikt Hauptmann, Maximilian Junker, Elmar Juergens, Rudolf Vaas, and Karl-Heinz Prommer. "Did We Test Our Changes? Assessing Alignment between Tests and Development in Practice". In: *Proceedings of the Eighth International Workshop on Automation of Software Test (AST'13)*. 2013 (cited on pages 13, 18, 66).

[Ede+12] Sebastian Eder, Maximilian Junker, Elmar Jürgens, Benedikt Hauptmann, Rudolf Vaas, and Karl-Heinz Prommer. "How

much does unused code matter for maintenance?" In: *Proceedings of the 34th International Conference on Software Engineering (ICSE'12)*. 2012 (cited on pages 14, 22).

[Eic+01] Stephen G. Eick, Todd L. Graves, Alan F. Karr, J. S. Marron, and Audris Mockus. "Does Code Decay? Assessing the Evidence from Change Management Data". In: *IEEE Software*. 2001 (cited on page 13).

[FRJ09] Martin Feilkas, Daniel Ratiu, and Elmar Jürgens. "The loss of architectural knowledge during system evolution: An industrial case study". In: *Proceedings of the 17th IEEE International Conference on Program Comprehension (ICPC'09)*. 2009 (cited on pages 13, 22).

[HHS14] Lars Heinemann, Benjamin Hummel, and Daniela Steidl. "Teamscale: Software Quality Control in Real-Time". In: *Proceedings of the 36th ACM/IEEE International Conference on Software Engineering (ICSE)*. 2014 (cited on page 17).

[Par94] David Lorge Parnas. "Software aging". In: *Proceedings of the 16th International Conference of Software Engineering (ICSE)*. 1994 (cited on page 13).

[Ste+14] Daniela Steidl, Florian Deissenboeck, Martin Poehlmann, Robert Heinke, and Baerbel Uhink-Mergenthaler. "Continuous Software Quality Control in Practice". In: *Proceedings of the IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 2014 (cited on pages 13, 18).

[SE14] Daniela Steidl and Sebastian Eder. "Prioritizing Maintainability Defects by Refactoring Recommendations". In: *Proceedings of the 22nd International Conference on Program Comprehension (ICPC)*. 2014 (cited on page 15).

[SHJ14] Daniela Steidl, Benjamin Hummel, and Elmar Juergens. "Incremental Origin Analysis of Source Code Files". In: *Proceedings of the 11th Working Conference on Mining Software Repositories (MSR'14)*. 2014 (cited on page 140).

Online

[Wik] Wikipedia. *Business Intelligence*. https://de.wikipedia.org/wiki/Business_Intelligence (cited on page 12).