2013

[SOFTWAREQUALITÄT SICHERSTELLEN MIT SONAR]

Autor: Michaela Lutz

Dozent: Michael Theis

FWP-Fach: Aktuelle Technologien zur Entwicklung verteilter Java-Anwendungen

Erklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig verfasst, nur die angegebenen Quellen und Hilfsmittel verwendet habe und Zitate als solche gekennzeichnet wurden. Weiterhin wurde die Arbeit noch nicht zu anderen Prüfungszwecken beim Prüfungsamt eingereicht.

Name: Lutz

Vorname: Michaela

Matrikelnummer: 04165511

Studiengruppe: IB4B

München, 06.06.2013

Inhaltsverzeichnis

1 Einleitung 4

1.1 Was ist SONAR? 4

1.2 Wie ist SONAR aufgebaut und was braucht man um ein Projekt analysieren zu können? 4

1.3 Wie wird eine Analyse gestartet und was liefert sie für Ergebnisse? 6

1.3.1 Analysieren über die Konsole 6

1.3.2 Analysieren mit Eclipse 8

2 Wie stellt man mit SONAR die Qualität von Software automatisch und kontinuierlich während der Entwicklung sicher 9

2.1 Softwarequalität kontinuierlich sicherstellen in Eclipse 9

2.1.1 Eine Violation in Eclipse beheben 9

2.1.2 Erstellen einer Review in Eclipse 10

2.1.3 Quellcode vor dem Commit auf Violations prüfen 11

2.2 Softwarequalität sicherstellen mit dem Sonar-Server 12

3 Die wesentlichen Funktionen im SONAR-Server 12

3.1 Welche Funktionen und Tools werden zur Analyse bereitgestellt? 12

3.1.1 Dashboard 13

3.1.2 Hotspots 15

3.1.3 Reviews 16

3.1.4 Time Machine 17

3.2 Welche Sprachen und Plugins werden von Sonar unterstützt? 18

4 Strategisch relevante Qualitätsmetriken 19

4.1 Metrik – Begriffserklärung 19

4.2 Was für Metriken gibt es in SONAR? 19

4.3 Detaillierte Erklärung der einzelnen Metriken 20

4.3.1 Complexity: 20

4.3.2 Design: 21
4.3.3 Documentation: 23
4.3.4 Duplications: 24
4.3.5 Reviews: 24
4.3.6 Rules: 24
4.3.7 Size: 25
4.3.8 Tests: 27
5 Fazit 28

1 Einleitung

1.1 Was ist SONAR?

Sonar ist ein Tool zum Verwalten und Visualisieren von Qualitäts-Metriken (Messgrößen) in Software-Projekten. Dabei steht die Verwaltung und Verbesserung der Software- und Codequalität im Vordergrund. Man möchte mit diesem Tool den Entwicklern, und damit den Softwarefirmen ein Werkzeug in die Hand geben, mit dem sichergestellt werden kann, dass ein Produkt in einem qualitativ hochwertigen Zustand beim Kunden ausgeliefert werden kann.

Außerdem kann man damit zahlreiche Statistiken erstellen, auswerten und sich anzeigen lassen, die einen Überblick über die Projekte und Arbeitsweise in dem Unternehmen zeigen. Damit kann ein Unternehmen seine einzelnen Projekte nach Fortschritt, Verbesserung etc. auswerten, aber auch alle bisher getätigten Projekten sehen, und damit auch feststellen ob und wie sich das Unternehmen verbessert hat oder nicht.

Sonar dient dazu, den Entwicklern ein organisiertes Qualitätsmanagement für ein Softwareprojekt zur Verfügung zu stellen, mit dem leicht gearbeitet und alle Qualitätsprozesse, die das Projekt durchläuft, nachvollziehbar festgehalten werden können. Um die Qualität eines Softwareprojekts sicherzustellen, deckt das Tool die sieben Säulen der Qualitätssicherung ab: Architektur und Design, Kommentare, Duplikate, Unit tests, Komplexität, Potentielle Bugs und Code-Regeln (Vgl. Sonar). Damit werden alle Bereiche abgedeckt, in denen potentielle Qualitätsrisiken auftreten können, was es zu einem idealen Tool zur Qualitätssicherung von Software macht.

Betrachten wir zunächst den Aufbau von Sonar und welche Voraussetzungen erfüllt sein müssen, bevor ein Projekt in Sonar analysiert werden kann.

1.2 Wie ist SONAR aufgebaut und was braucht man um ein Projekt analysieren zu können?

Sonar besteht aus drei Komponenten, die für die Analyse eines Projektes benötigt werden: Datenbank, Web-Server und Client (Analyser) (Vgl. Installing Sonar), wie in Abbildung 1 zu sehen ist.

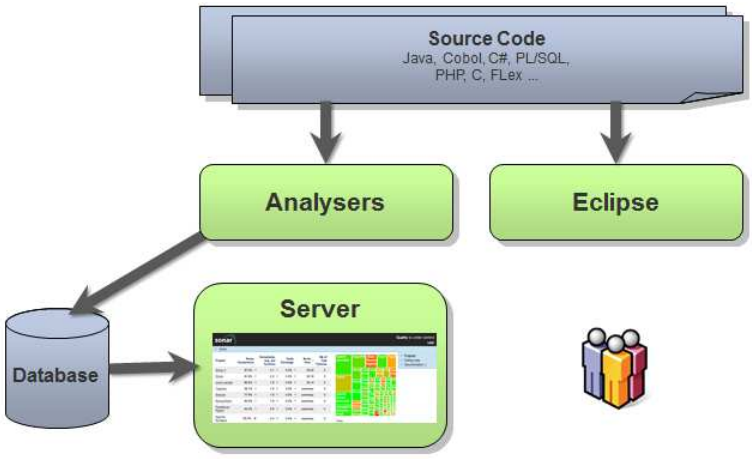

Abbildung 1: Sonar Architektur

In der **Datenbank** werden Konfigurationen und Ergebnisse von Softwareanalysen abgespeichert. Sonar unterstützt mehrere Datenbanken, aus denen man wählen kann. Sonar listet dazu alle Datenbanken incl. Informationen, welche Versionen im speziellen unterstützt werden, auf seiner Internetseite unter „Supported Platforms“ auf. Unterstützt werden Apache Derby, H2, Microsoft SQL Server, MySQL, Oracle und PostgreSQL (Vgl. Supported Platforms).

Im Web-Server, dem **Sonar-Server**, kann man durch die Ergebnisse der Analyse navigieren und Detailinformationen zu allen Metriken, die geprüft wurden, einholen. Der Sonar-Server benötigt mind. 500 MB RAM um arbeiten zu können (Vgl. Requirements).

Die Informationen zu einem analysierten Projekt werden in sogenannten „Dashboards“ angezeigt. Was **Dashboards** sind und welche Informationen sie liefern, wird im nächsten Unterkapitel behandelt. Um den Web-Server darstellen zu können braucht man einen Web-Browser. Unterstützt werden bei Sonar, derzeit alle gängigen, verfügbaren Web-Browser (Vgl. Requirements). Weitere Informationen, welche Versionen unterstützt werden, finden sich auf der Internetseite http://docs.codehaus.org/display/SONAR/Requirements in der auch die Datenbankversionen zu finden sind.

Die Analyse selbst, wird mit einem **Client** gestartet, von dem man wiederum mehrere Auswahlmöglichkeiten von Sonar bereit gestellt bekommt. Der Client kann, aus einer auf der Sonar-Internetseite zusammengestellten Liste, ausgewählt werden. Derzeit sind folgende Clients verfügbar: "Sonar Runner", als bevorzugter „Default Launcher“, "Sonar Ant Task", "Maven", "Gradle" und "CI Engines" (Vgl. Analyzing Source Code).

Desweiteren ist es auch möglich über "**Eclipse**" mit Sonar zu arbeiten. Wie das funktioniert wird später im Kapitel behandelt. Damit wären nun alle Komponenten vorhanden, um mit Sonar arbeiten zu können. Als nächstes schauen wir kurz über den Installationsprozess um anschließend ein Projekt analysieren zu können:

Alle verfügbaren Sonar-Versionen, können bequem als zip.-Datei auf der Internetseite http://www.sonarsource.org/downloads/ heruntergeladen werden. Als nächstes sollte man sich einen, der oben beschriebenen Clients aussuchen und installieren. Sonar empfiehlt "Sonar Runner" als Standard-Client (Vgl. Analyzing Source Code), für unsere Beispiele verwenden wir ab jetzt **Maven** als Client, da viele Projekte mit Maven gemacht werden. Als nächstes muss die eingebettete Datenbank konfiguriert werden. Anleitung dazu findet sich auch auf der Internetseite von Sonar. Anschließend wird im Web-Browser der Sonar Server gestartet. Der Sonar Server hat die Adresse http://localhost:9000 . Mit dem Startbefehl (je nach Betriebssystem anders) der über die Kommandozeile des PC eingegeben wird, startet man den Sonar Server (Vgl. Installing Sonar).

Jetzt sind alle Vorbereitungen abgeschlossen, als nächstes kann ein Projekt von Sonar analysiert werden.

1.3 Wie wird eine Analyse gestartet und was liefert sie für Ergebnisse?

1.3.1 Analysieren über die Konsole

Die Analyse selbst, wird ebenfalls wie der Server, über die Konsole gestartet. Je nach dem welcher Client verwendet wird, lautet der Startbefehl immer anders (Vgl. Analyzing Source Code). Hier in Abbildung 2 findet sich ein Beispiel mit dem Startbefehl für Maven. Bei allen Clients ist gemein, dass der Startbefehl im Basisordner des gewünschten Projekt ausgeführt wird.

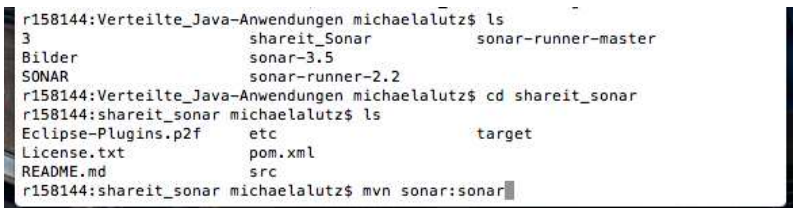

Abbildung 2: Maven-Befehl zum Starten der Analyse über das Terminal

Nach kurzer Rechenzeit liefert das Terminal "Build Success" aus, wie in Abbildung 3 gezeigt.

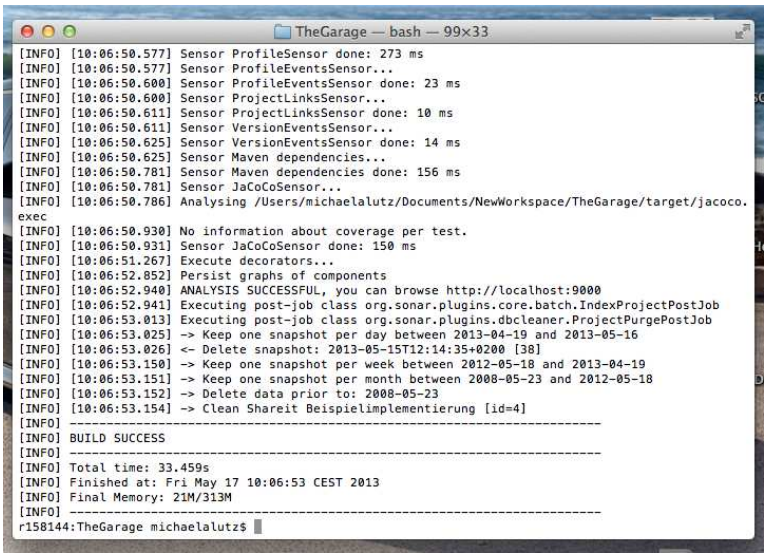

Abbildung 3: Build Success

Jetzt kann man zum Sonar-Server wechseln, auf dessen Home-Ansicht (Abbildung 4), rechts die Projekte angezeigt werden. Auf der Home-Seite (in der Standardversion ohne eigene Anpassungen) werden folgende Angaben vom Projekt angezeigt: Name des Projekts, Version, LOCs (Lines of Codes), RCI (Rules compliance index), das Datum wann die Analyse erstellt wurde und die „Rules Compliance“ (= Regeleinhaltung) in Prozent und graphischer Farbdarstellung, in denen rot für 0 Prozent und grün für 100 Prozent stehen.

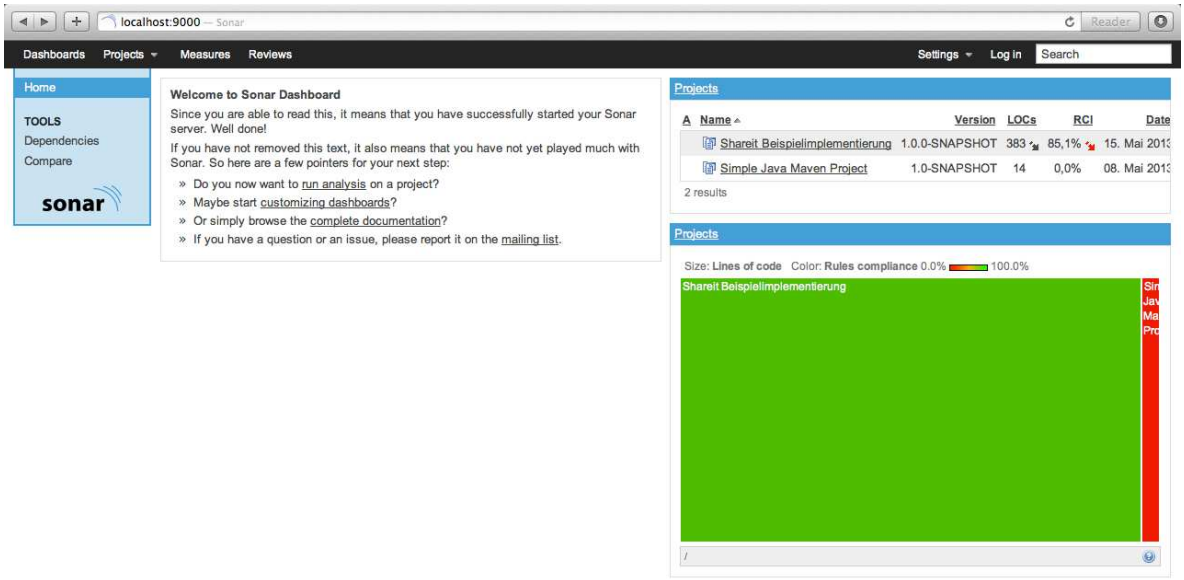

Abbildung 4: Sonar-Server-Homeseite (Standard, noch nicht angepasst)

Mit einem einfachen Klick auf den Namen des Projekts, springt man in die Übersichtdarstellung des Projekts mit der Bezeichnung "Dashboard" (ein Project Dashboard), wie in Abbildung 5 zu sehen ist.

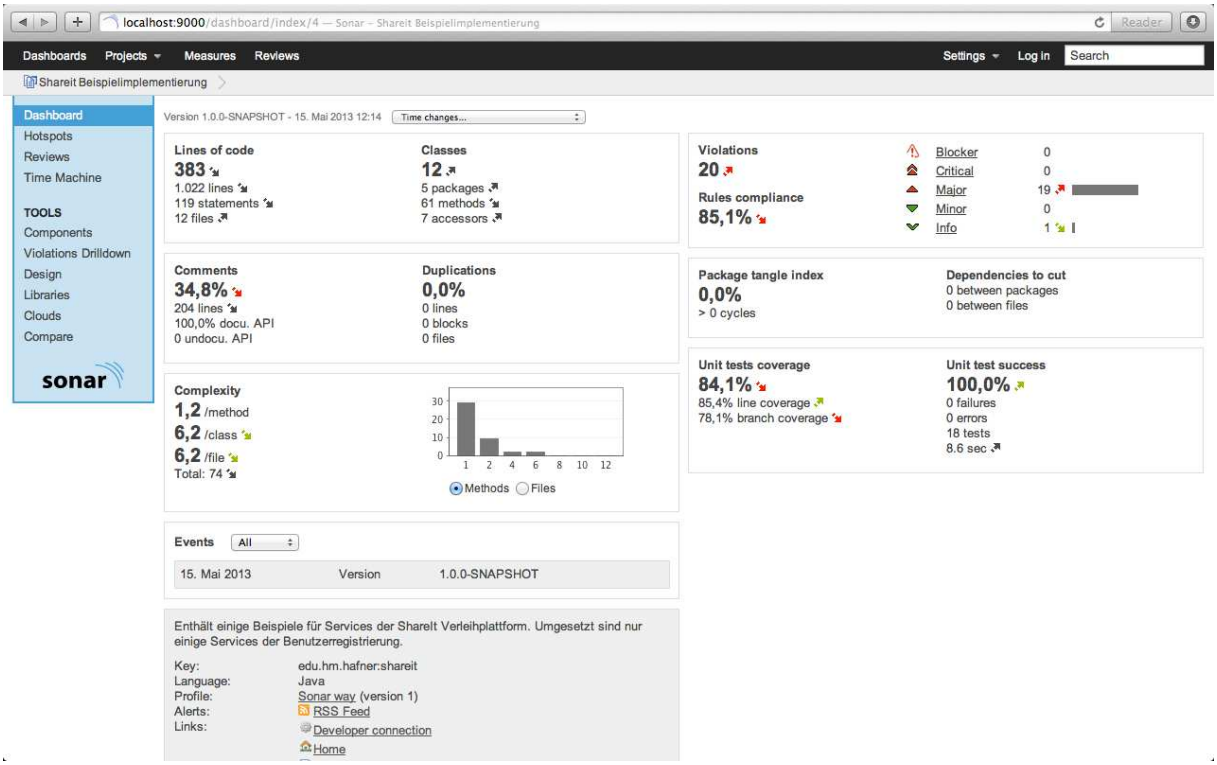

Abbildung 5: Dashboard, (Standardeinstellung - noch nicht angepasst)

Im nächsten Kapitel werden, auf die Arten und Funktionen des **Dashboards**, sowie der weiteren Funktionen von Sonar näher eingegangen.

1.3.2 Analysieren mit Eclipse

Sonar bietet für die Verwendung mit Eclipse ein Plugin an, mit dem Sonar Projekte, die in Eclipse geschrieben wurden, analysiert werden können. Es können Projekte in den Programmiersprachen Java, C/C++ und Python analysiert werden. Der Vorteil, Sonar und damit die Analyse, direkt von Eclipse aus zu starten ist, dass der Entwickler nicht mehr von der Entwicklungsoberfläche auf den Sonar-Server springen muss, um die Informationen über den Code abzurufen und anschließend wieder zurück um den Code zu bearbeiten. Sondern er kann direkt in Eclipse sehen, wo noch Handlungsbedarf besteht und kann sofort den Code ändern oder **Reviews** anlegen (Vgl. Sonar in Eclipse).

Sonar bietet in Eclipse derzeit drei Funktionen an: Managen von auftretenden **Violation** (=Regelverletzungen), **Reviews** durchsehen und Quellcode überprüfen lassen, bevor er committet wird (Vgl. Sonar in Eclipse)

Wie in Eclipse der Quellcode genau gemanagt wird, wird im nächsten Kapitel behandelt.

2 Wie stellt man mit SONAR die Qualität von Software automatisch und kontinuierlich während der Entwicklung sicher

2.1 Softwarequalität kontinuierlich sicherstellen in Eclipse

Wie im vorhergehenden Kapitel schon erwähnt, werden nun die drei Funktionen genauer beschreiben, mit denen man die Softwarequalität direkt in Eclipse kontinuierlich sicherstellen kann.

2.1.1 Eine Violation in Eclipse beheben

Wenn Sonar in Eclipse installiert und konfiguriert worden ist, kann man eine Analyse direkt in Eclipse starten. Wenn man dann durch den Quellcode in Eclipse scrollt, kann man sofort sehen, wo sich **Violations** befinden, da sie hervorgehoben werden. Die Idee dahinter ist folgende: Wenn man sich den Code durchgelesen und ihn verstanden hat, ist der Aufwand um eine Violation zu beheben gering (Vgl. Working with Sonar in Eclipse).

Am schnellsten ist man, wenn man sich in Eclipse gleich den Reiter (Tab) "Problems" hernimmt. In "Problems" werden, wie in Abbildung 6 zu sehen ist, alle Violations angezeigt. Klickt man nun mit Doppelklick auf eine der Violations, wird automatisch in die Quellcodezeile gesprungen, in der der Fehler zu beheben ist. Der Fehler ist im Quellcode hervorgehoben, damit man ihn gleich auf Anhieb sieht (Vgl. Working with Sonar in Eclipse).

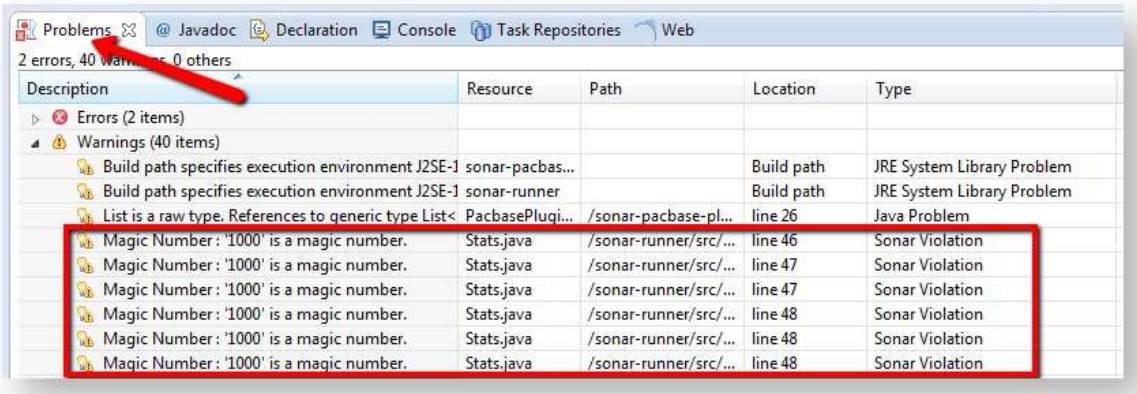

Abbildung 6: Violations in Eclipse

Nachdem man den Fehler behoben hat, muss man nur noch die Fehlernachricht löschen. Dazu wählt man, wie in Abbildung 7 nachfolgend zu sehen, mit Rechtsklick auf den Violation-Icon die Funktion "Delete violation" aus. Dann wird die Nachricht auch automatisch aus dem Reiter "Problems" genommen (Vgl. Working with Sonar in Eclipse).

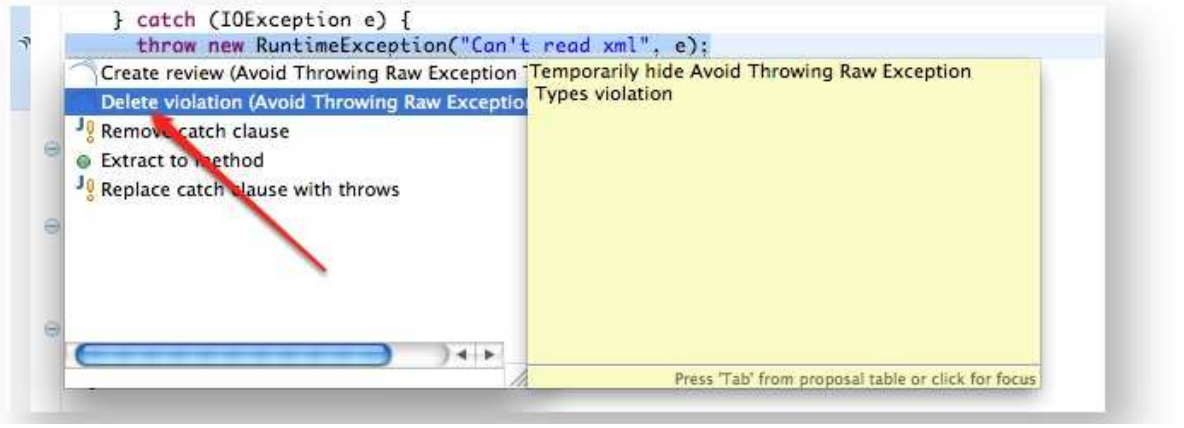

Abbildung 7: Violation löschen

2.1.2 Erstellen einer Review in Eclipse

Wenn man nicht gleich den Fehler beheben kann oder will, hat man die Möglichkeit für den Fehler eine **Review** zu erstellen, um den Fehler dann zu einem späteren Zeitpunkt beheben zu können.
Rechtsklick auf den Violation-Icon und "Create Review" auswählen. In dem sich öffnenden Fenster, wie in Abbildung 8 zu sehen, kann man Termindaten und einen Kommentar einfügen. Mit "Submit" bestätigen. Jetzt kann man sich den Fehler in der Review-Ansicht (Abbildung 9) ansehen und zu einem späteren Zeitpunkt bearbeiten (Vgl. Working with Sonar in Eclipse).

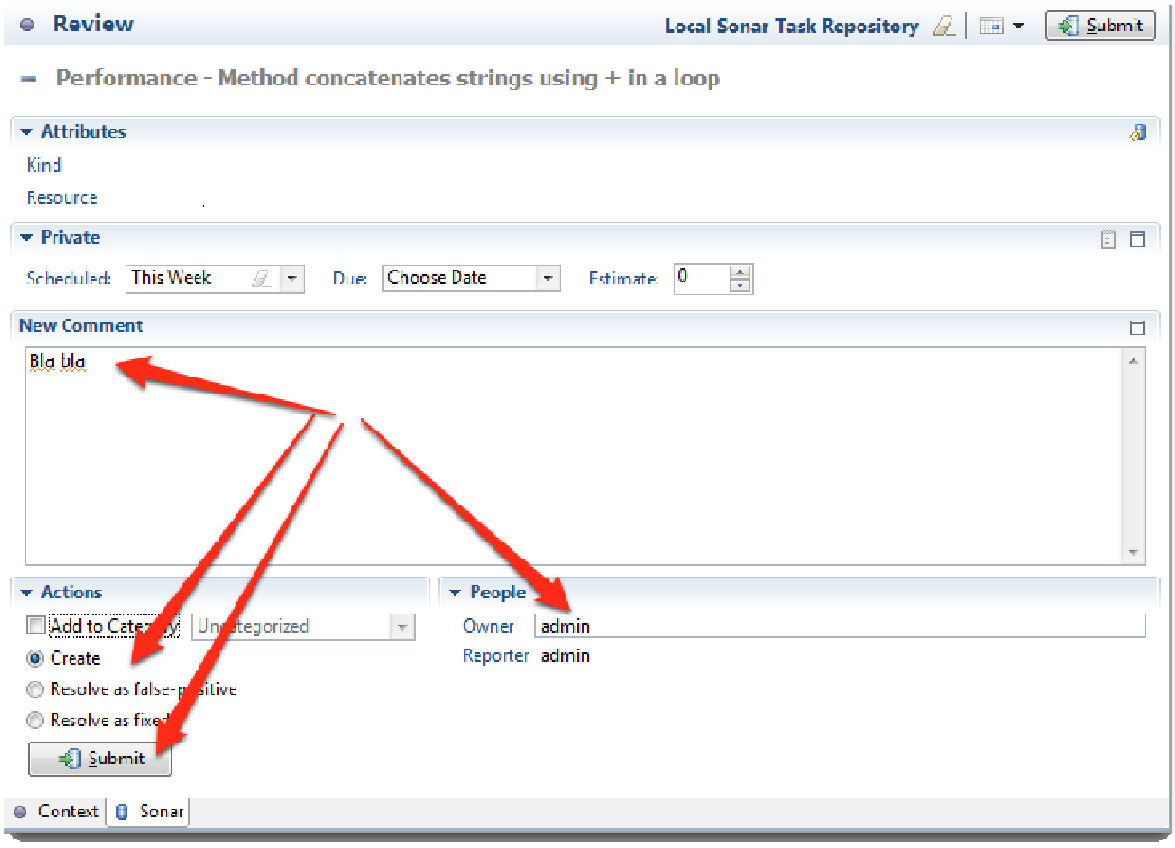

Abbildung 8: Erstellen einer Review

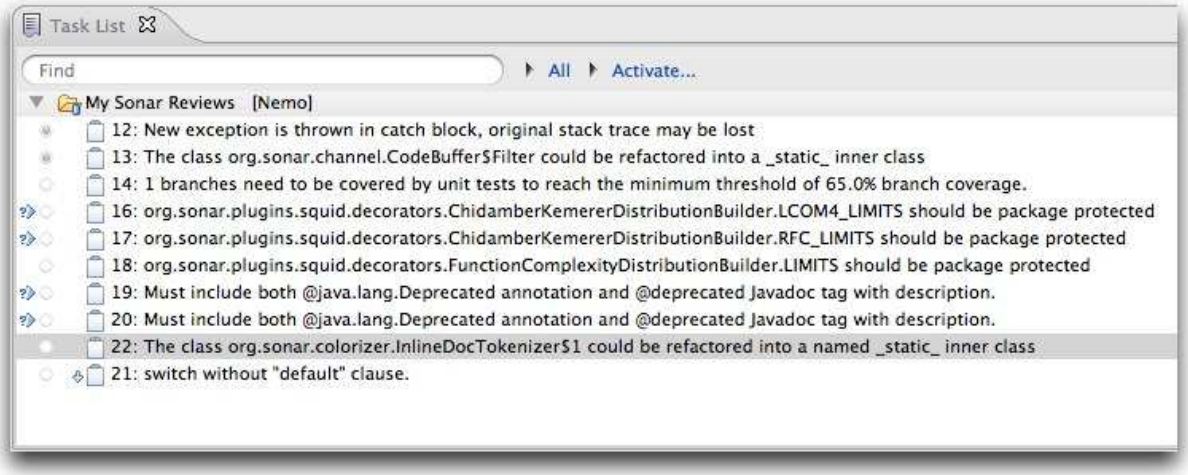

Abbildung 9: Sonar Task List (zeigt alle Reviews)

2.1.3 Quellcode vor dem Commit auf Violations prüfen

In Eclipse zeigt Sonar schon vor dem Commit in ein Repository an, welche Qualität die neu hinzugefügten Codeteile haben. Dabei werden alle Fehler standardmäßig als Error markiert und sind deshalb im „Package Explorer“ zu sehen (durch das rote X am Projekt). Damit kann man schon vor einem Commit problematische Codestellen beseitigen (Vgl. Working with Sonar in Eclipse).

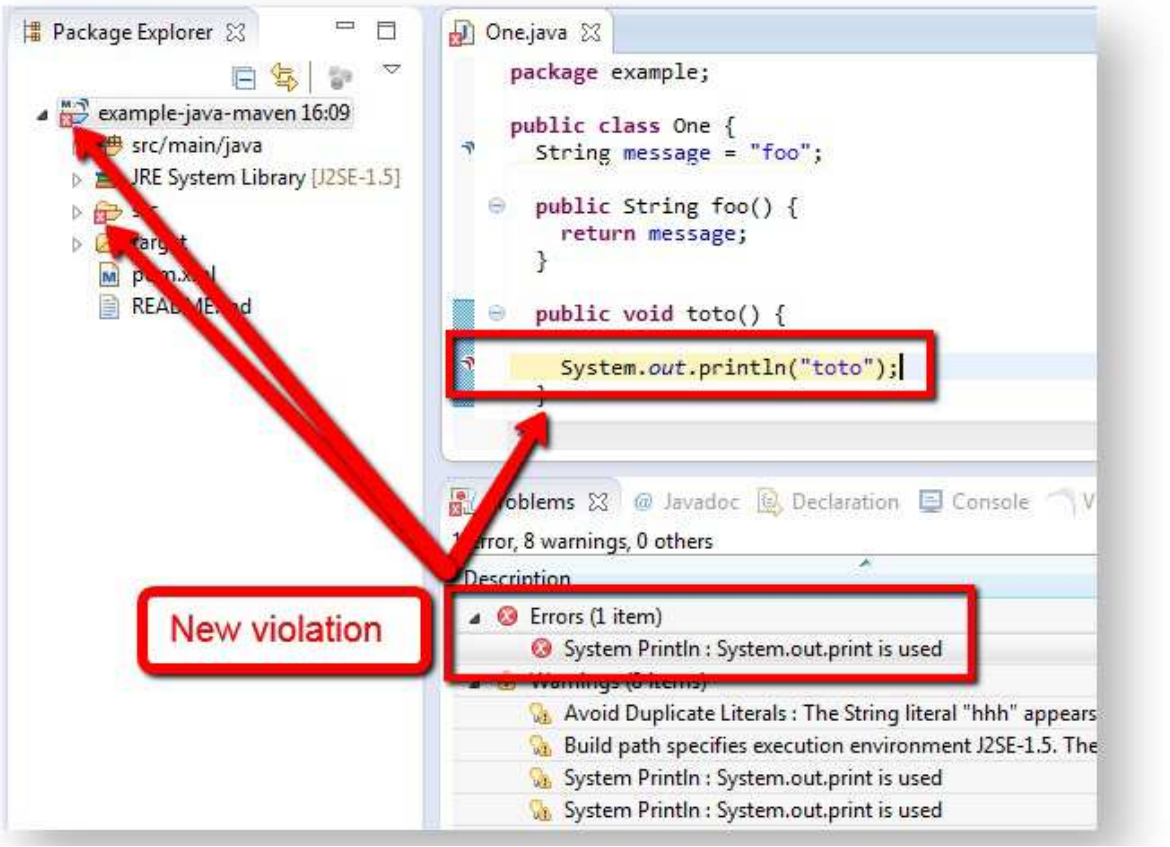

Abbildung 10: Anzeige von Violations vor einem Commit

2.2 Softwarequalität sicherstellen mit dem Sonar-Server

Wenn man nicht mit Eclipse arbeitet , kann man die über das Terminal ausgewertete Analyse über den Sonar-Server betrachten. Das Sonar Webservice-Interface, das vom Sonar-Server bereitgestellt wird, hilft dabei alle Informationen über den Quellcode leicht und schnell einsehen und bearbeiten zu können. Um in die Ansicht des Sonar-Server zu gelangen, muss man nachdem man ein Projekt über das Terminal analysiert hat, den Sonar-Server starten. Das wird ebenfalls mit dem Terminal erledigt. Ist der Server gestartet, ruft man mit dem Webbrowser die Seite http://localhost:9000 auf. Jetzt befindet man sich auf der Standartseite vom Sonar-Server. Um auf die eigene Darstellung zu kommen, muss man sich mit seinen Benutzerdaten anmelden. Jetzt kann man seine analysiertes Projekt betrachten und mit Sonar arbeiten. Welche Funktionen Sonar dazu bereit stellt, wird im nächsten Kapitel behandelt.

3 Die wesentlichen Funktionen im SONAR-Server

3.1 Welche Funktionen und Tools werden zur Analyse bereitgestellt?

Damit alle Funktionen von Sonar übersichtlich bleiben, werden die einzelnen Analysebereiche in Dashboards gegliedert. Als „Dashboard“ wird eine Web-Seite bezeichnet auf der die Daten anzeigt werden, die in einer Datenbank gespeichert sind. Die Daten werden in sogenannten „**Widgets**“ angezeigt, dass sind Boxen in denen die Informationen auf der Seite eingeblendet werden. Es gibt zwei Arten von Widgets: „**Global Widgets**“ zeigen Informationen zu allen Projekten die in Sonar gelistet sind an. „**Project Widgets**“ zeigen nur Daten zu einem bestimmten Projekt an (Vgl. Sonar Architectur). Solange man sich nicht einloggt hat, hat man als anonymer Benutzer die Global Dashboard-Ansicht z.B. die Home-Seite und die voreingestellten „**Project Dashboards**“ (Dashboard, Hotspots, Reviews und Time Machine), zur Verfügung, zu sehen auf Abbildung 11 unter der Rubrik "Left menu".

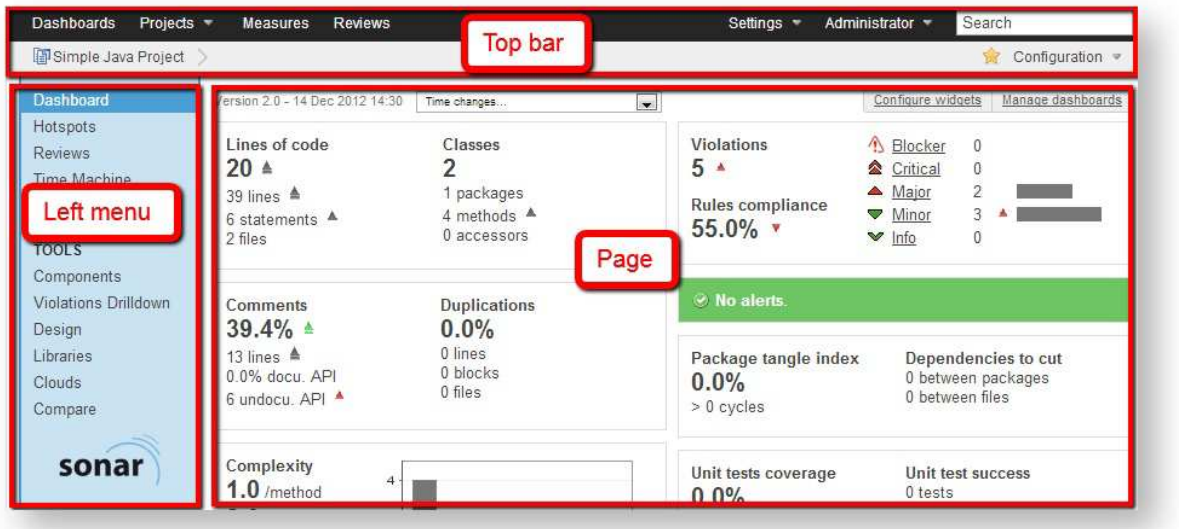

Abbildung 11: Dashboard (Project-Home-Ansicht aufgeteilt in seine Bereiche).

Das „Project Dashboard“ mit der Bezeichnung „**Dashboard**“ markiert den Einstiegspunkt in ein Projekt. Auf der Projekt-Ansicht finden sich Informationen zum Projekt. Sie geben eine Übersicht über das Projekt und enthalten Daten wie Maßnahmen, Violation, Reviews etc. (Vgl. Browsing Sonar). Standardmäßig sind im „Left menu“ folgende Dashboards verfügbar: "Dashboard", "Hotspots", "Reviews" und "Time Machine". Jeder dieser Dashboards stellt unterschiedliche Informationen über ein Projekt zur Verfügung. Nachfolgend wird jetzt geklärt was diese Dashboards im Einzelnen sind und welche Aufgaben und Funktionen sie haben.

3.1.1 Dashboard

Das erste Dashboard im „left menu“ in Abbildung 11 trägt den gleichnamigen Titel. In **Dashboard** finden sich alle Informationen zu einem analysierten Projekt in einer übersichtlichen Darstellung. Es zeigt detailliert alle Informationen den Punkten, wie in Abbildung 11 gezeigt wurde, an: geschriebene Codezeilen und Klassen nach Anzahl, Kommentare in Prozent, Duplikate in Prozent, Komplexität, Verletzungen, Regeleinhaltung in Prozent, Verknüpfungsindex der Packages in Prozent, Abdeckung der Unit tests in Prozent.

Jede Information auf den Project-Dashboards kann angesteuert werden. Es wird in eine Ansicht mit allen Packages (auf der linken) und allen Klassen (auf der rechten Seite) gesprungen, in denen Informationen zur ausgewählten Metrik enthalten sind. Wird eines der Packages ausgewählt, verändert sich die rechte Seite und zeigt nur noch die im Package enthaltenen Klassen an.

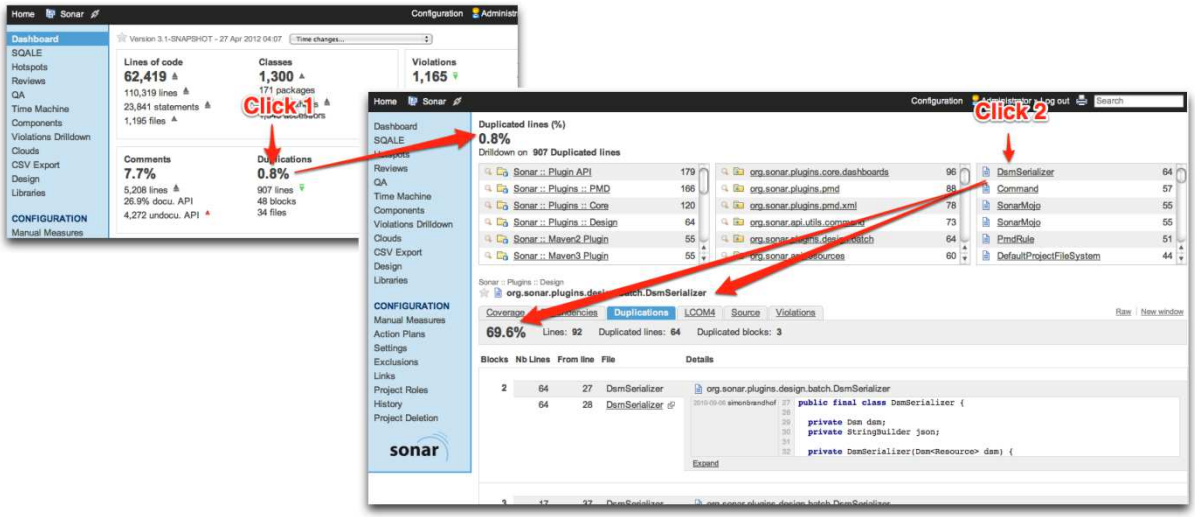

Abbildung 12: Duplication drilldown (als Beispiel für die Struktur)

Jetzt kann man eine der Klassen auswählen um detaillierte Informationen zu erhalten. Die View wird aufgeteilt, in die Bildschirminformationen von der vorherigen View oben und die detaillieren Informationen unten. Durch die Reiter (Tabs) "Coverage", "Dependencies", "Duplications", "LCOM4", "Source" und "Violations" (siehe Abbildung 13 unter „Available tabs“) kann man beliebig wechseln. (Nicht in jedem Fall werden alle Reiter angezeigt, daher kann man nur durch die jeweils vorhandenen Reiter springen.) Jetzt kann man durch den Quellcode gehen und sich die Stellen ansehen in denen Violation auftreten, bevor man den richtigen Quellcode auf die Entwicklungsoberfläche aufruft, um den Code zu bearbeiten.

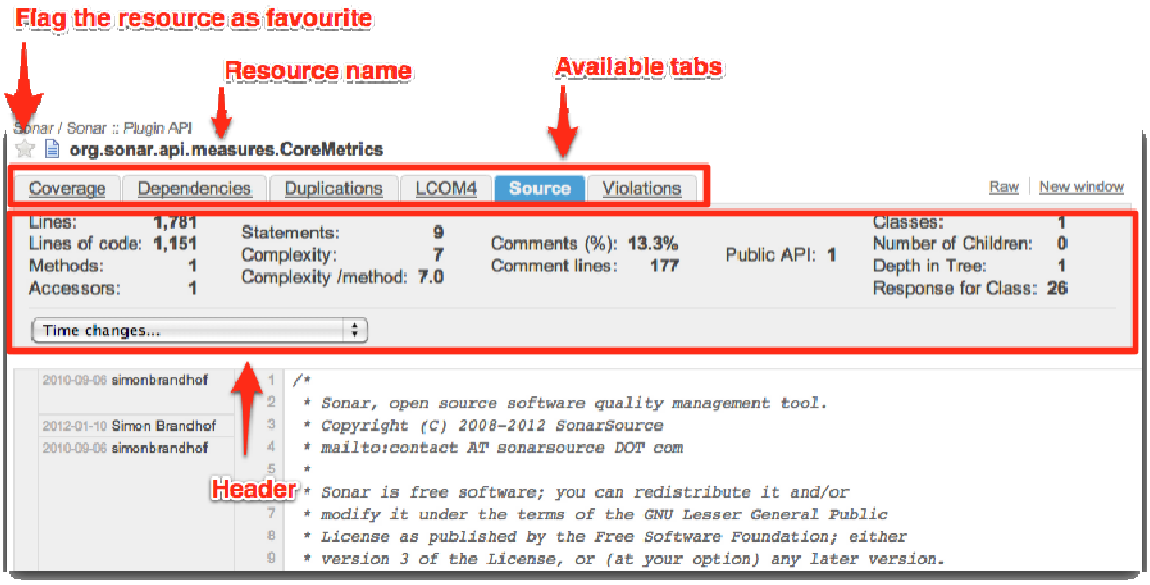

Abbildung 13: Viewing Source Code (View zum Source Code durchgehen)

Man sollte sich wenn einmal in das Tool eingearbeitet, unbedingt seine eigenen Dashboards erstellen, da einem die vorhandenen Project-Dashboards nicht immer genügend oder die gewünschten Informationen bieten. Dabei kann man wählen ob man sich ein Dashboard nach den eigenen Bedürfnissen neu zusammenstellen oder ein vorhandenes neu konfigurieren möchte (Vgl. Dashboards). Beispielsweise könnte man ein Dashboard erstellen, das alle Kennzahlen über die Kosten oder das Rating der Projekte auswertet (Vgl. Sqale). Genauere Informationen wie man sich seine Dashboards erstellt oder verändert, findet man auf der Internetseite http://docs.codehaus.org/display/SONAR/Dashboards . Damit kann man Sonar auch seiner Unternehmens- oder Arbeitsstruktur anpassen, was nicht nur die Arbeit mit dem Tool erleichtert, sondern auch Zeit spart. Einmal konfiguriert, kann man es für alle Projekte hernehmen. Änderungen sind erst erforderlich wenn neue Daten benötigt werden, oder sich Strukturen ändern.

Um ein Dashboard konfigurieren zu können, muss man als Benutzer eingeloggt sein, da in der Ansicht der anonymen Benutzer keine Änderungen vorgenommen werden können. Deshalb muss erst mal ein Benutzer anlegt werden. Um einen neuen Benutzer anlegen zu können sollte man sich nach der Installation, beim ersten Aufrufen des Web-Servers, mit dem Administrator-Login einloggen. In der Administrator-Ansicht können dann neue Benutzer erzeugt werden. Oder in einem Unternehmen lässt man sich vom Administrator seine Daten für ein Benutzerkonto geben. Der Administrator kann Dashboards erstellen, die als Default-Dashboard von allen Benutzern verwendet werden, diese werden „**Shared Dashboards**“ genannt. Der Benutzer selbst, kann nur Dashboards für den eigenen Gebrauch erstellen (Vgl. Dashboards).

3.1.2 Hotspots

In **Hotspots** sind die Stellen im Quellcode zu finden, an denen sich Violations häufen. Die Ansicht ist (hier ebenfalls in der Standardansicht) in folgenden Bereiche aufgeteilt, wie in der Abbildung 13 gezeigt wird: Regelverletzungen, Hotspots nach Komplexität, Hotspots nach dupliziertem Code, Quellen der Regelverletzungen, Hotspots nach der Komplexität der Methoden und Hotspots nach öffentlicher undokumentierter API.

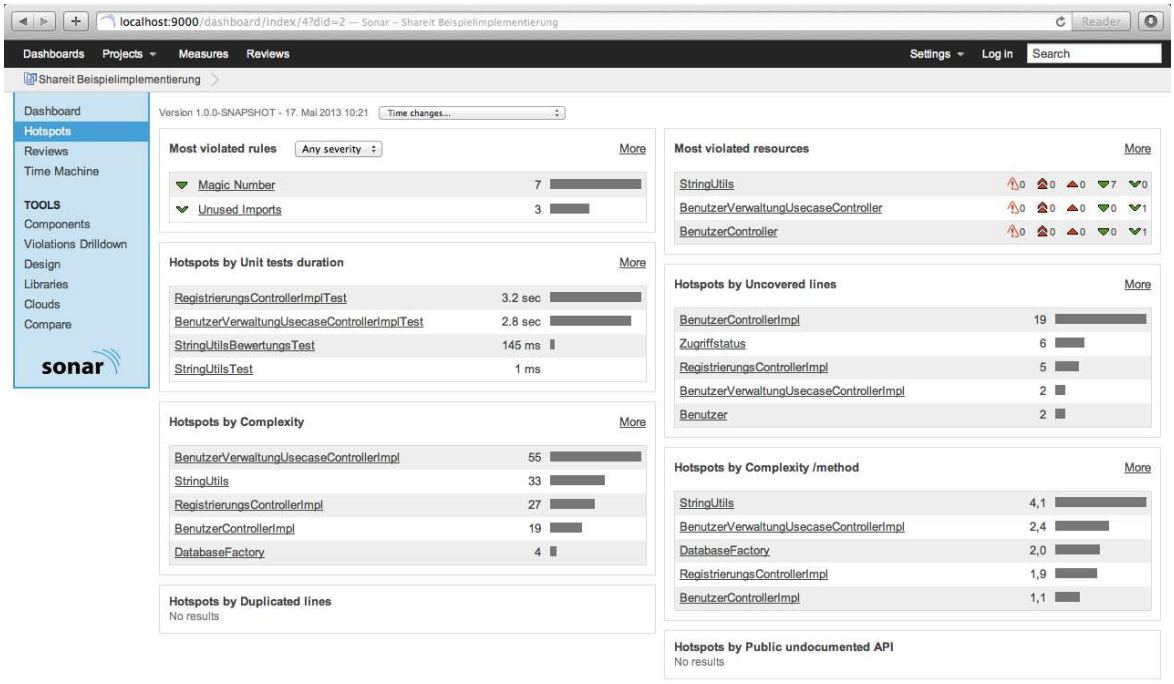

Abbildung 14: Hotspots (Home-Seite, Standardeinstellung (noch nicht benutzerdefiniert))

3.1.3 Reviews

Reviews listet eine Historie aller Regelverletzungen eines Projekts auf. Das heißt, in diesem Dashboard wird die Abarbeitung der ganzen Violations eines Programmes gemanagt. Alle Verletzungen und Bedrohungen werden drei Kategorien zugeordnet: 1. Bedrohungen die sofort behoben werden müssen, 2. Bedrohungen die erst im nächsten Sprint behoben werden, 3. Bedrohungen die im Auge behalten, aber noch nicht behoben werden müssen, da es sich kostentechnisch im Moment nicht lohnt (Vgl. Violations und Reviews).

Wie im Abbildung 15 zu sehen, werden in dieser Ansicht Informationen zu folgenden Punkten angezeigt: Aktive Nachprüfungen, Nicht nachgeprüfte Regelverletzungen, "False Positives", Offene Aktionspläne, geplante Nachprüfungen, ungeplante Nachprüfungen, Aktive Nachprüfungen nach Entwickler.

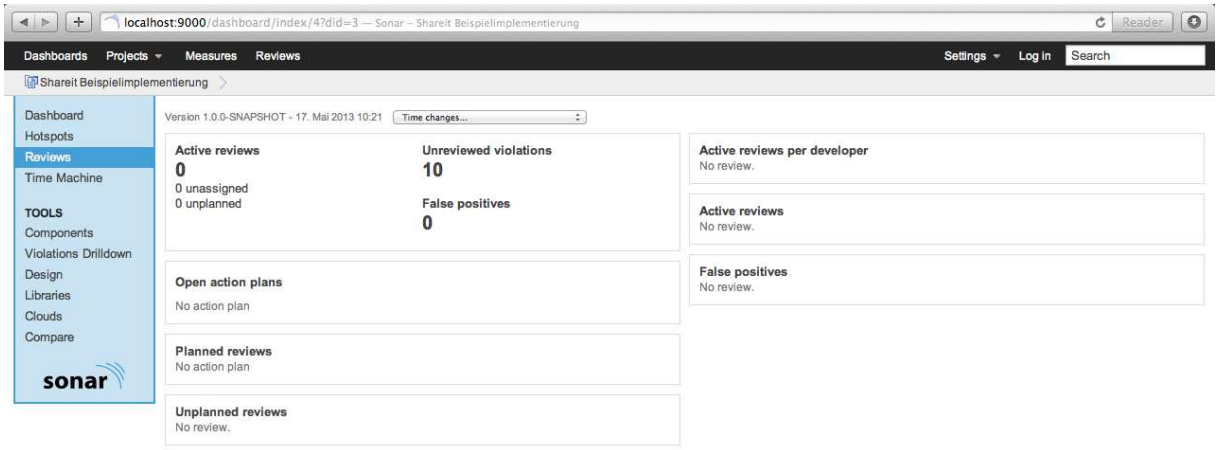

Abbildung 15: Reviews-Home-Ansicht

3.1.4 Time Machine

Wenn man über den Quellcode eines Projekts geht, um einen neuen Aktionsplan für die weitere Vorgehensweise zu erstellen, ist es hilfreich wenn man nicht nur den aktuellen Stand des Projekts sondern auch Vergangenheitswerte zum Vergleich hat. **Time Machine** zeigt alle Informationen von der ersten Analyse und der letzten Analyse (im Standardformat, es können auch Analysen dazwischen angezeigt werden), damit man vergleichen kann, wie sich der Code im Laufe der Zeit verändert hat. „**Tendencies**“ zeigen zusätzlich den Trend an, in welche Richtung sich die Werte der Metriken verändert haben. Sie werden durch einen Pfeil an der Messgröße dargestellt und lassen sich in fünf Kategorien unterscheiden: Starkes Wachstum, Wachstum, Neutral, Abnahme und Starke Abnahme. Alle diese Informationen zeigen dem Programmier, im Hinblick auf neue Aktionspläne, welche Programmteile vor anderen zu bearbeiten sind und hilft somit bei der Abwägung und Entscheidung. (Vgl. Time Machine)

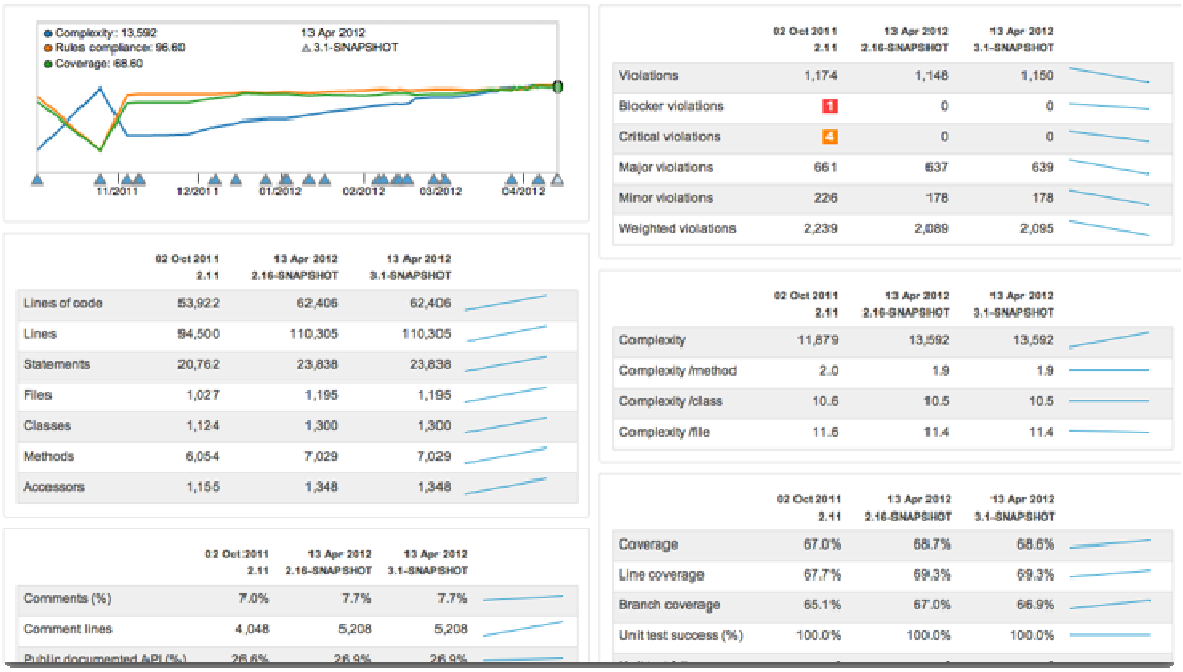

Abbildung 16: Time Machine

Die Ansicht der Time Maschine besteht aus zwei verschiedenen Widgets. Dem „Timeline Widget“ und dem „History Table Widget“. Im **Timeline Widget** können die historischen Daten von bis zu drei Metriken graphisch dargestellt werden. Fährt man mit der Maus über die Linien der Grafik, werden die einzelnen Werte zu den verschiedenen Zeitpunkten angezeigt, wie in Abbildung 17 zu sehen ist. (Vgl. Time Machine)

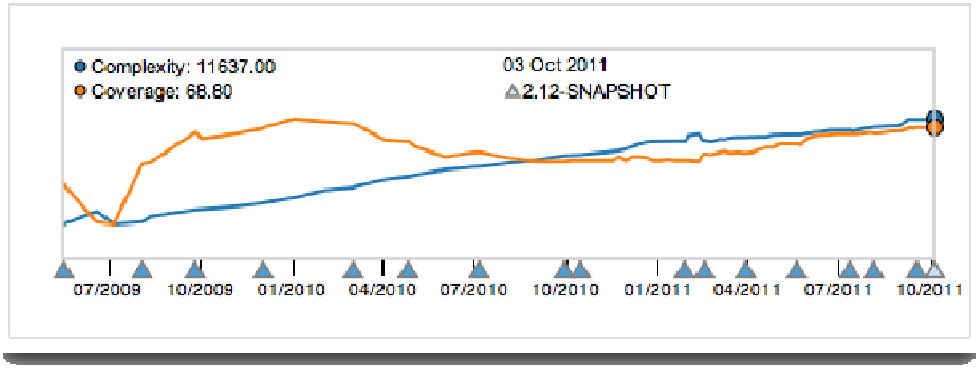

Abbildung 17: Timeline Widget

Im **History Table Widget** kann eine Tabelle angezeigt werden, die von bis zu zehn Metriken, die historischen Daten anzeigen kann. Die Abbildung 18 stellt ein solches History Table Wiget dar:

	16 May 2009	04 Aug 2011 2.10-SNAPSHOT	15 Sep 2011 2.11-SNAPSHOT	03 Oct 2011 2.12-SNAPSHOT	
Complexity	4,351	11,080	11,598	11,637	
Coverage	62.8%	68.2%	68.8%	68.8%	

Abbildung 18: History Table Widget

3.2 Welche Sprachen und Plugins werden von Sonar unterstützt?

Dieses Tool ist nicht auf eine Programmiersprache beschränkt. Sonar kann Analysen auf mehr als 20 Programmiersprachen fahren (Vgl. Sonar). Darunter sind u.a. .NET, C, C++, Java, JavaScript, Python etc. Dazu muss man die gewünschte Sprache auf der Seite http://docs.codehaus.org/display/SONAR/Plugin+Library herunterladen und installieren.

Damit ist das Tool für verschiedene Softwareprojekte egal welcher Sprache einsetzbar. Des weiteren bieten es auch noch mehr als 50 weitere Plugins aus den Bereichen "Developer Tools" , "Integration", "Governance", "Visualisation/Reporting", "Additional Metrics", "Localization", "Exernal Tool Integration"und "Exernal Plugins" (Vgl. Plugin Library). Damit ist Sonar nach den jeweiligen Bedürfnissen anpass- und erweiterbar.

4 Strategisch relevante Qualitätsmetriken

4.1 Metrik – Begriffserklärung

In diesem Kapitel beschäftigen wir uns mit den relevanten Qualitätsmetriken, an denen das Ergebnis einer Analyse hauptsächlich festgemacht wird.

Zunächst einmal klären wir den Begriff der Metrik:

> "Der Begriff Metrik kommt aus dem griechischen und bedeutet Messung. Dabei handelt es sich nach allgemeinem Verständnis um ein System von Kennzahlen oder um Verfahren zur Messung von quantifizierbaren Werten.
>
> In der IT-Technik gilt eine Metrik als "eine standardisierte Messgröße". Die Bezeichnung wird vorwiegend in Verbindung mit Algorithmen verwendet. ... Allen Metriken ist gemein, dass sie eine bestimmende Eigenschaft haben, die sie auszeichnet." (Vgl. Itwissen)

Nun wenden wir uns den Metriken, die in Sonar definiert worden sind, genauer zu.

4.2 Was für Metriken gibt es in SONAR?

Das Ergebnis der Analyse wird hauptsächlich durch die gesetzten Metriken und an den Verletzungen von Code-Regeln festgemacht. Sonar analysiert dabei nur das, was aufgrund der Programmiersprache analysiert werden kann. Im Detail betrachtet kann Sonar folgende Analysen über den Quellcode machen: Sonar kann auf bestimmten Sprachen statische Analysen auch auf dem kompilierten Code ausführen. Dynamische Methoden im Code können nur bei einigen Sprachen analysiert werden (Vgl. Analyzing Source Code).

Sonar zeigt durch die Anzahl von verschiedenen Metriken genau auf, welche Schwachstellen es im Quellcode eines Projektes gibt und wo sie sich befinden. Die wichtigsten Metriken werden dabei in der Projektübersicht, im **Dashboard** angezeigt (Abbildung 5, S. 8).

In Sonar gibt es 8 verschiedene Bereiche in denen die Metriken jeweils unterteilt sind. Diese wären im folgenden: Complexity, Design, Documentation, Duplications, Reviews, Rules, Size, Tests (Vgl. Metrik definitions).Hier ein kleiner Überblick, was jeder Bereich macht. Die einzelnen Metriken werden im nächsten Unterkapitel genauer betrachtet.

Der Bereich **Complexity** vereint alle Messgrößen unter sich, die die Komplexität des Quellcodes messen und einordnen. Ziel ist es, dass die Komplexität im Code möglichst gering ist.

Design ist der Oberbegriff für alle Messgrößen, die sich mit der Architektur und den Abhängigkeiten zwischen Methoden, Klassen und Packages beschäftigt. Ziel ist es, dass das Design möglichst einfach und möglichst wenige Abhängigkeiten und Schleifen zwischen Methoden, Klassen und Packages bestehen.

Die **Documentation** kümmert sich darum, dass die Richtlinien für die Kommentare zum Quellcode eingehalten wurden. Ziel ist es, dass der Quellcode ordentlich und ausreichend kommentiert ist.

Der Bereich **Duplications** vereint alle Messgrößen, die anzeigen wo sich duplizierter Code finden lässt. Ziel ist es am Ende im Code keine einzige duplizierte Zeile zu haben.

Reviews enthält alle Messgrößen, die Anzeigen in welchem Bearbeitungsstatus sich das Projekt in Bezug auf Regelverletzungen befindet. Es werden dort alle offenen, bearbeiteten, neu hinzugekommenen und nicht bearbeiteten Reviews angezeigt.

In **Rules** finden sich alle Messgrößen, die anzeigen wie viele, welche Häufigkeit und welche Prozentzahl die Regelverletzungen am Code einnehmen. Ziel ist es am Ende, keine einzige Regelverletzung mehr im Code zu finden.

In **Size** finden sich alle Messgrößen, die sich mit der Größe des Codes, der vorhandenen Codezeilen, Anzahl der Methoden, Klassen, Dateien und Packages auseinandersetzen.

Tests enthält alle Messgrößen, die sich mit den „Unit tests“ befassen. Sei es die Zeit die Tests brauchen, wie erfolgreich die Tests durchgelaufen sind oder ob zu jeder Methode alle Testfälle vorhanden sind. Ziel ist es eine Coverage-Abdeckung von 100 % zu erreichen.

4.3 Detaillierte Erklärung der einzelnen Metriken

4.3.1 Complexity:

Die Komplexität kann für drei Bereiche gemessen werden: Klassen, Dateien und Methoden. Dabei wird jeweils immer die durchschnittliche Komplexität ermittelt.

Wann immer der Kontrollfluss einer Methode auseinandergeht, erhöht sich auch die Komplexität. In jeder Programmiersprache führen andere Schlüsselwörter zu einer Erhöhung der Komplexität, in einem Codestück (Vgl. Metrik definitions). Hier erläutern wir beispielhaft an der Programmiersprache Java welche das sind und liefern in Abbildung 19 ein Beispiel hierfür.

```
public void process(Car myCar){              <- +1
    if(myCar.isNotMine()){                   <- +1
        return;                              <- +1
    }
    car.paint("red");
    car.changeWheel();
    while(car.hasGazol() && car.getDriver().isNotStressed()){   <- +2
        car.drive();
    }
    return;
}
```

Abbildung 19: Berechnung der Komplexität einer Methode (diese Methode hat eine Komplexität von 5)

Schlüsselwörter, die dazu führen das sich die Komplexität erhöht sind folgende: **if, for, while, case, catch, throw, return** (nur wenn es nicht am Ende der Methode steht), **&&, ||**. Die Schlüsselwörter **else, default** und **finally** führen hingegen zu **keiner Erhöhung** der Komplexität. Eine einfache Methode mit einer **switch**-Anweisung, die einen ganzen Block an **case**-Anweisungen aufweist, kann eine **überraschend hohe Komplexität** aufweisen. (Die Komplexität ist dieselbe, wenn man den **switch**-Anweisungs-Block durch einen äquivalenten **if**-Anweisungs-Block austauschen würde.) **Accessors** (Getter und Setter, dazu mehr später im Bereich **Size**) werden nicht als Methoden betrachtet und erhöhen daher auch nicht die Komplexität (Vgl. Metrik Complexity)

4.3.2 Design:

Die Sparte **Design** beschäftigt sich mit allen Faktoren, welche die Darstellung und Architektur eines Programmes betreffen. Nachfolgend werden alle Metriken, die sich mit dem Design von Programmen beschäftigen, aufgelistet.

Afferent couplings: Dieser Wert misst, wie viele andere Klassen eine spezifizierte Klasse verwenden.

Depth in Tree: Misst die Tiefe der Vererbungsstruktur, misst für jede Klasse ihr Hierarchielevel von der Objekthierarchiespitze aus. Da jede Klasse von Object erbt, beträgt der DIT-Index (Vererbungsindex) mindestens 1.

Efferent couplings: Misst wie viele unterschiedliche Klassen eine spezifizierte Klasse verwenden.

File cycles: Minimale Anzahl von Schleifen, die in einer Datei innerhalb eines gefunden wurden, um unerwünschte Abhängigkeiten festzustellen.

File edges weight: Anzahl der Dateiabhängigkeiten innerhalb eines Packages.

File dependencies to cut: Anzahl der Dateiabhängigkeiten die gekappt wurden, um Schleifen zwischen den Packages zu entfernen.

File tangle: Identisch zu Suspect file dependencies.

File tangle index: File tangle in %. Man berechnet den Idex wie folge: 2 * (File tangle / File edges weight) * 100.

LCOM4: Fehlen von Zusammenhängen in Methoden(Vgl. Cohesion of Methods)

Number of children: Anzahl der erbenden Klassen, egal ob direkte oder indirekte Vererbung von einer anderen Klasse.

Package cycles: Minimale Anzahl von Schleifen die in einem Package gefunden wurden, um unerwünschte Abhängigkeiten festzustellen.

Package dependencies to cut: Anzahl der Package-Abhängigkeiten die gekappt wurden, um Schleifen zwischen den Packages zu entfernen.

Package tangle index: Level der Verwicklungen in einem Package. Der beste Wert 0 % bedeutet, dass es überhaupt keine Verwicklungen in dem Package gibt, der schlechteste Wert 100% sagt aus, dass die Packages wirklich stark miteinander verwickelt sind. Der Index wird nach dieser Formel berechnet: 2 * (File dependencies to cut / Number of file dependencies zwischen Packages) * 100.

Response for class: RFC misst die Komplexität einer Klasse nach den Aufrufen der Methoden. Für jede Klasse zählt er die Anzahl der Methoden und für jede Distinct Methode nochmal zusätzlich eins hoch. (Vgl. Abbildung 20)

```
1  public class ClassA
2  {
3    private ClassB classB = new ClassB();          // call (constructor of class B) => +1
4    public void doSomething(){                     // method declaration => +1
5      System.out.println ( "doSomething");         // call (System.out.println) => +1
6    }
7    public void doSomethingBasedOnClassB(){        // method declaration => +1
8      System.out.println (classB.toString());    // call (System.out.println) => 0 because already counted
9  on line 5 + call (toString) => +1
10   }
11 }
12
13 // default constructor of ClassA => +1
   // RFC = 6
```

Abbildung 20: RFC (Response for class) Beispiel

Package edges weight: Anzahl der Dateiabhängigkeiten zwischen mehreren Packages.

Suspect file dependencies: Datenabhängigkeiten, die gekappt wurden um Schleifen, die zwischen Dateien in einem Package auftreten zu entfernen. (Anmerkung: Schleifen zwischen Dateien in einem Package sind aber nicht immer schlecht für die Architekturqualität)

Suspect LCOM4 density: Dichte von Dateien die eine LCOM4-Dicht größer 1 aufweisen.

(Vgl. Metrik definitions)

4.3.3 Documentation:

Der Bereich **Dokumentation** enthält die Metriken für die Kommentare in dem Quellcode. Auch hier gibt es unterschiedliche Messgrößen.

Blank comments: Anzahl nicht erheblicher Kommentarzeilen. Dazu gehören leere Kommentarzeilen, Kommentarzeilen die nur spezielle Zeichen enthalten, etc.

Comment lines: Zeilenanzahl die ein Kommentar enthält.

Comments (%): Zeilenanzahl eines Kommentars in Prozent. Hier wird die Dichte der Kommentarzeilen in Bezug zu den Codezeilen gesetzt. 50% bedeutet, dass das Verhältnis von Code und Kommentaren ausgeglichen ist. 100% bedeutet, dass eine Datei nur aus Kommentaren besteht.
Die Formel zur Berechnung der Kommentardichte ist: Kommentarzeilen / (Codezeilen + Kommentarzeilen) * 100.

Comments in Procedure Divisions: Diese Messgröße gibt es nur in der Programmiersprache Cobol und misst die Prozedurteile der Kommentare

Public documented API (%): Kennzahl für die Dichte der öffentlich dokumentierten API in Prozent. Die Berechnung ist wie folgt: (Öffentlich dokumentierte API - Nicht-Öffentlich dokumentierte API) / Öffentlich dokumentierte API * 100

Public undocumented API: Öffentlich dokumentierte Kommentare ohne Header.

4.3.4 Duplications:

Dieser Bereich misst den Anteil an dupliziertem Code. Dabei werden nach diesen Metriken unterschieden:

Duplicated blocks:	Anzahl an duplizierten Zeilenblöcken.
Duplicated files:	Anzahl der Dateien in denen duplizierter Code auftaucht.
Duplicated lines:	Anzahl an duplizierten Zeilen.
Duplicated lines (%):	Anzahl an duplizierten Zeilen in Prozent. Ermittelt die Dichte an duplizierten Zeilen wie folgt: Duplizierte Zeilen / Zeilen * 100

(Vgl. Metrik definitions)

4.3.5 Reviews:

Unter **Reviews** sind alle Metriken aufgeführt die sich mit Regelverletzungen und deren Behandlung auseinandersetzen. Folgenden Metriken werden dabei unterschieden:

Active reviews:	Anzahl der noch offenen Reviews. (Noch nicht bearbeitet und noch nicht geschlossen.)
False-positive reviews:	Anzahl der Reviews die zur späteren Bearbeitung markiert wurden.
New unreviewed violations:	Anzahl der neu hinzugekommenen, noch nicht überprüften Regelverletzungen.
Unassigned reviews:	Anzahl der noch nicht zugewiesenen Reviews.
Unplanned reviews:	Anzahl der Reviews die noch nicht einem Aktionsplan zugeordnet wurden.
Unreviewed violations:	Anzahl der noch nicht überprüften Regelverletzungen.

(Vgl. Metrik definitions)

4.3.6 Rules:

New violations:	Anzahl neuer Regelverletzungen
New xxxxx violations:	Anzahl neuer Regelverletzungen mit Schweregrad xxxxx. Dabei steht das xxxxx für die Kategorien in die Regelverletzungen eingeteilt werden: Blocker, Critical, Major, Minor oder Info.
Violations:	Anzahl der Regelverletzungen.

xxxxx violations: Anzahl aller Regelverletzungen mit Schweregrad xxxxx (Blocker, Critical, Major, Minor oder Info).

Weighted violations: Alle Regelverletzungen aufsummiert, gewichtet nach dem Koeffizienten der mit dem jeweiligen Schweregrad verknüpft ist. Die Berechnung der Weighted violations lautet wie folgt: (Summe(xxxxx_Regelverletzungen * xxxxx_Gewichtung)).
Um die Gewichtung der einzelnen Schweregraden einstellen zu können, muss man sich als Administrator einloggen und folgendem Pfad folgen Settings > Configuration > General Settings > General und bei „Rules weight“ die Gewichtung einstellen.

Rules compliance: Rules compliance index (RCI) = 100 - (Weighted violations / Codezeilen * 100). Misst die Prozentzahl der vorhandenen Regelverletzungen.

(Vgl. Metrik definitions)

4.3.7 Size:

Der Bereich **Size** beschäftigt sich mit allen Metriken zum Thema Größe. Dabei wird unter anderem die Größe von Methoden, Dateien, Klassen und Verzeichnissen gemessen.

Accessors: Anzahl der verwendeten Getter und Setter im Code zum zugreifen auf Klassenattribute.

Abbildung 21: Accessors (Getter und Setter)

Classes: Anzahl aller Klassen incl. der verschachtelten Klassen, Interfaces, Enums und Annotations.

Directories: Anzahl der Verzeichnisse.

Files: Anzahl der Dateien.

Generate Lines:	Anzahl der Zeilen die mit z.B. CA-Telon in Cobol generiert wurden.
Generated lines of code:	Anzahl der Codezeilen die mit z.B. CA-Telon in Cobol generiert wurden.
Inside Control Flow Statements:	(Nur in Cobol), Anzahl der internen Kontrollfluss Anweisungen (GOBACK, STOP RUN, DISPLAY, CONTINUE, EXIT, RETURN, PERFORM paragraph1 THRU paragraph2).
Lines:	Anzahl der physikalischen Zeilen (Anzahl der carriage returns)
Lines of code:	Anzahl der physikalischen Zeilen, die mindestens einen „Character“ enthalten, der weder eine Leerzeile noch ein Tabulator oder ein Teil eines Kommentars ist. (In Cobol: Generierte Codezeilen oder datenverarbeitende Anweisungen (SKIP1, SKIP2, SKIP3, COPY, EJECT, REPLACE) werden nicht zu den Lines of Code hinzugezählt.
LOCs in Data Divisions:	(Nur in Cobol.) Anzahl der Codezeilen in Dateneinheiten. Generierte Codezeilen werden nicht dazu gezählt.
LOCs in Procedure Divisions:	(Nur in Cobol.) Anzahl der Codezeilen in Prozesseinheiten. Generierte Codezeilen werden nicht dazu gezählt.
Methods:	Anzahl der Methoden. (In Cobol : Anzahl der Paragraphen, in Java: Accessors werden als Methoden gezählt wenn der Parameter sonar.squid analyse.property.accessors auf false gesetzt wurde. Konstruktoren werden auch als Methoden gezählt. In VB.NET werden Accessoren nicht als Methoden gewertet.)
Outside Control Flow Statements:	(Nur in Cobol.) Anzahl der äußeren Kontrollfluss Anweisungen (CALL, EXEC CICS LINK, EXEC CICS XCTL, EXEC SQL, EXEC CICS RETURN).
Packages:	Anzahl der Packages.
Projects:	Anzahl der Projekte in einer Ansicht.
Public API:	Anzahl der öffentlichen Klassen + Anzahl der öffentlichen Methoden + Anzahl der öffentlichen Attributen (Properties). In Java werden keinen Attribute angezeigt die mit public final static ausgewiesen sind.
Statements:	Anzahl der Anweisungen. Beispiele: **In Cobol** : move, if, accept, add, alter, call, cancel, close, etc. **In Java**: if, else, while, do, for, switch, break, continue, return, throw, synchronized, catch, finally. Klassen, Methoden, Felder, Annotationen, Definitionen, Package declarations und Import declarations werden nicht hinzugezählt. In PL/SQL: Nur nicht nach außen sichtbare Anweisungen mit PL/SQL Blöcken.

(Vgl. Metrik definitions)

4.3.8 Tests:

Branch coverage: In jeder Codezeile, in der ein boolscher Ausdruck ausgewertet wird, überprüft die **Branch Coverage** ob sowohl die false-Seite als auch die true-Seite ausgewertet wurde. Das richtet sich nach den möglichen Ereignissen, die während der Ausführung der unit tests auftreten können. Der **Branch Coverage** berechnet sich wie folgt:

```
Branch coverage = (CT + CF) / (2*B)

where

CT = branches that have been evaluated to 'true' at least once
CF = branches that have been evaluated to 'false' at least once

B = total number of branches
```

Abbildung 22: Branch coverage - Berechnung

Coverage: Eine Mischung aus **Branch coverage** und **Line coverage**. Hier wird überprüft wie viel Quellcode durch die unit tests abgedeckt wurde.
Der **Coverage** berechnet sich wie folgt:

```
Coverage = (CT + CF + LC)/(2*B + EL)

where

CT = branches that have been evaluated to 'true' at least once
CF = branches that have been evaluated to 'false' at least once
LC = covered lines = lines_to_cover - uncovered_lines

B = total number of branches
EL = total number of executable lines (lines_to_cover)
```

Abbildung 23: Coverage - Berechnung

Line coverage: Zeigt ob jede Zeile während der Ausführung der unit tests aufgerufen wurde. Der **Line coverage** wird wie folgt berechnet:

```
Line coverage = LC / EL

where

LC = covered lines (lines_to_cover - uncovered_lines)
EL = total number of executable lines (lines_to_cover)
```

Abbildung 24: Line coverage - Berechnung

Lines to cover: Anzahl der Codezeilen die durch die unit tests abgedeckt werden. (Leere Zeilen oder Kommentarzeilen werden nicht dazu gewertet.)

New branch coverage: Identisch zum **Branch coverage**, aber beschränkt auf den neu hinzugekommen oder geupdateteten Code.

New line coverage: Identisch zu **Line coverage**, aber beschränkt auf den neu hinzugekommen oder geupdateteten Zeilen.

New lines to cover: Identisch zu **Line to cover**, aber beschränkt auf den neu hinzugekommen oder geupdateteten Zeilen.

New uncovered lines: Identisch zu **Uncovered lines** , aber beschränkt auf den neu hinzugekommen oder geupdateteten Zeilen.

Skipped unit tests: Anzahl der übersprungenen Tests.

Uncovered branches: Anzahl der Wege, die nicht durch unit tests abgedeckt wurden.

Uncovered lines: Anzahl der Zeilen, die nicht durch unit tests abgedeckt wurden.

Unit tests: Anzahl der Unit tests.

Unit tests duration: Die Zeit die benötigt wurde um alle unit tests durchlaufen zu lassen.

Unit test errors: Anzahl der fehlgeschlagenen Unit tests.

Unit test failures: Anzahl der Unit tests die mit einer unerwarteten Exception fehlgeschlagen sind.

Unit test success density (%): Prozentzahl der Erfolgsquote der gesamten Unit tests. Die Berechnung ist wie folgt: (Unit tests - (Unit test errors + Unit test failures)) / Unit tests * 100

(Vgl. Metrik definitions)

5 Fazit

Damit wären alle Bereiche angesprochen, die nötig sind, um mit dem Tool Sonar arbeiten zu können. In dieser Arbeit haben wir uns angeschaut, welche Voraussetzungen man erfüllen muss, um mit dem Tool arbeiten zu können. Es wurde gezeigt, wie eine Analyse sowohl über das Terminal als auch auf der Entwickleroberfläche Eclipse gestartet wird. Und wir haben uns den Sonar-Server mit seinen Dashboards und alle Messgrößen zum Auswerten eines Projekts genauer angeschaut.

Grundsätzlich handelt es ich bei dem Tool Sonar um ein nützliches und hilfreiches Instrument zur Verwaltung von Violations und Reviews im Quellcode, was es zu einem geeigneten Tool macht, um mehrere Entwickler zu managen. Die Installation ist einfach und die Oberfläche des Sonar-Servers ist aufgeräumt und einfach gehalten. Die vielen Metriken zu verstehen, ist Übungssache und kann etwas Zeit in Anspruch nehmen.

Bei er Installation in Eclipse können Integrationsprobleme auftreten, wenn man schon viele anderen Plugins zur Softwareüberprüfung laufen hat. Aber grundsätzlich ist die Handhabung in Eclipse einfach und schnell.

Meiner Meinung nach finde ich das Tool zum Verwalten von Software bequem. Ich konnte mich schnell einarbeiten und sofort Analysen durchführen. Einzig die Vielzahl von Metriken nimmt etwas Zeit in Anspruch bis man alle kennt. Die wichtigsten sind aber schnell verinnerlicht.

Das Tool eignet sicher meiner Meinung nach für Entwickler oder Unternehmen die nicht allzu groß sind oder nicht zu viele Projekte haben. Bei vielen Projekten könnte es irgendwann unübersichtlich werden(z.B. beim Vergleich der Projekte miteinander), da ich bei einer Recherche keine Möglichkeit entdeckt habe, Projekte wieder zu entfernen.

Das Sonar in Eclipse integriert werden kann, finde ich sehr praktisch. Vor allem da man nicht mehr vom dem Sonar-Server in den Quellcode springen muss und umgekehrt.

Mitnehmen aus dieser Arbeit sollte man, das Sonar ein sinnvolles und einfaches Tool zum Verwalten von Softwarequalität ist und es sich durchaus für den einen oder anderen lohnt damit zu arbeiten. Ob man damit arbeiten möchte ist jedoch letztendlich Geschmacksache.

Für weiterführende Informationen steht die Internetseite www.sonarsource.org zur Verfügung.

Quellenverzeichnis

[Analyzing Source Code] Codehaus; http://docs.codehaus.org/display/SONAR/Analyzing+Source+Code; Stand: März 2013 (Zugriff 21.04.2013)

[Browsing Sonar] Codehaus; http://docs.codehaus.org/display/SONAR/Browsing+SonarStand: Januar 2013 (Zugriff 24.04.2013)

[Cohesion of Methods] Codehaus; http://docs.codehaus.org/display/SONAR/LCOM4+-+Lack+of+Cohesion+of+Methods; Stand: Februar 2013 (Zugriff: 17.05.2013)

[Dashboards] Codehaus; http://docs.codehaus.org/display/SONAR/Dashboards; Stand: Dezember 2012 (Zugriff: 24.04.2013)

[Installing Sonar] Codehaus; http://docs.codehaus.org/display/SONAR/Installing+Sonar; Stand: Mai 2013 (Zugriff 21.04.2013)

[Itwissen] http://www.itwissen.info/definition/lexikon/Metrik-metric.html; Stand: 16.05.2013 (Zugriff: 16.05.2013)

[Metrik Complexity] Codehaus; http://docs.codehaus.org/display/SONAR/Metrics+-+Complexity; Stand: Februar 2013 (Zugriff : 14.05.2013)

[Metrik definitions] Codehaus; http://docs.codehaus.org/display/SONAR/Metric+definitions; Stand: Feburar 2013 (Zugriff : 14.05.2013)

[Plugin Library] Codehaus; http://docs.codehaus.org/display/SONAR/Plugin+Library/; Stand: April 2012 (Zugriff 21.04.2013)

[Requirements] Codehaus; http://docs.codehaus.org/display/SONAR/Requirements; Stand: Mai 2013 (Zugriff: 17.05.2013)

[Sonar] Codehaus; www.sonarsource.org; Stand: April 2013 (Zugriff: 20.04.2013)

[Sonar Architectur] Codehaus; http://docs.codehaus.org/display/SONAR/Sonar+Concepts; Stand: Februar 2013 (Zugriff 21.04.2013)

[Sonar in Eclipse] Codehaus; http://docs.codehaus.org/display/SONAR/Using+Sonar+in+Eclipse; Stand: Dezember 2012 (Zugriff 19.05.2013)

[Supported Platforms] Codehaus; http://docs.codehaus.org/display/SONAR/Requirements#Requirements-SupportedPlatforms Stand: April 2013 (Zugriff: 21.04.2013)

[Sqale] Codehaus; http://nemo.sonarsource.org/dashboard/index/48569?did=19; Stand: Juni 2013 (Zugriff: 06.06.2013)

[Time Machine] Codehaus; http://docs.codehaus.org/display/SONAR/Historical+Information; Stand: April 2013 (Zugriff: 21.04.2013)

[Violation und Reviews] Codehaus; http://docs.codehaus.org/display/SONAR/Violations+and+Reviews; Stand: März 2013 (Zugriff (24.04.2013)

[Working with Sonar in Eclipse] Codehaus; http://docs.codehaus.org/display/SONAR/Working+with+Sonar+in+Eclipse, Stand: März 2013 ; (Zugriff (19.05.2013)

Bildquellenverzeichnis

[Abbildung1] SonarArchitectur;http://docs.codehaus.org/display/SONAR/Sonar+Concepts#SonarConcepts-SonarArchitecture (Zugriff 21.04.2013)

[Abbildung 2] Analyse Project Maven Befehl Bildschirmfoto vom 17.05.2013

[Abbildung 3] Analyse Project Build Success Bildschirmfoto vom 17.05.2013

[Abbildung 4] Localhost Homeseite Bildschirmfoto vom 17.05.2013; http://localhost:9000

[Abbildung 5] Dashboard Bildschirmfoto vom 17.05.2013; http://localhost:9000/dashboard/index/4

[Abbildung 6] Eine Violation beheben; http://docs.codehaus.org/display/SONAR/Working+with+Sonar+in+Eclipse; (Zufgriff 19.05.2013)

[Abbildung 7] Violation löschen; http://docs.codehaus.org/display/SONAR/Working+with+Sonar+in+Eclipse; (Zufgriff 19.05.2013)

[Abbildung 8] Eine Review erstellen; http://docs.codehaus.org/display/SONAR/Working+with+Sonar+in+Eclipse; (Zufgriff 19.05.2013)

[Abbildung 9] Reviews ansehen; http://docs.codehaus.org/display/SONAR/Working+with+Sonar+in+Eclipse; (Zufgriff 19.05.2013)

[Abbildung 10] Fehler vor Commit ansehen; http://docs.codehaus.org/display/SONAR/Working+with+Sonar+in+Eclipse; (Zufgriff 19.05.2013)

[Abbildung 11] Layout Project-Home; http://docs.codehaus.org/display/SONAR/Browsing+Sonar (Zugriff: 17.05.2013)

[Abbildung 12] Duplication drilldown; http://docs.codehaus.org/display/SONAR/Browsing+Sonar (Zugriff: 17.05.2013)

[Abbildung 13] Viewing Source Code; http://docs.codehaus.org/display/SONAR/Resource+Viewer (Zugriff: 22.05.2013)

[Abbildung 14] Hotspots Homeansicht Bildschirmfoto vom 17.05.2013; http://localhost:9000/dashboard/index/4?did=2

[Abbildung 15] Reviews Homeansicht Bildschirmfoto vom 17.05.2013 http://localhost:9000/dashboard/index/4?did=3

[Abbildung 16] Time Machine; http://docs.codehaus.org/display/SONAR/Historical+Information; Stand: April 2013; (Zugriff: 20.05.2013)

[Abbildung 17] Timeline Widget; http://docs.codehaus.org/display/SONAR/Historical+Information; Stand: April 2013; (Zugriff: 20.05.2013)

[Abbildung 18] History Table Widget; http://docs.codehaus.org/display/SONAR/Historical+Information; Stand: April 2013; (Zugriff: 20.05.2013)

[Abbildung 19] Metric Complexity; http://docs.codehaus.org/display/SONAR/Metrics+-+Complexity; Stand: Februar 2013
(Zugriff : 14.05.2013)

[Abbildung 20] Response for class; http://docs.codehaus.org/display/SONAR/RFC+-+Checking+Coupling; (Zugriff 22.05.2013)

[Abbildung 21] Accessors;http://docs.codehaus.org/display/SONAR/Metrics+-+Accessors; (Zugriff 17.05.2013)

[Abbildung 22] Branche Coverage; http://docs.codehaus.org/display/SONAR/Metric+definition; (Zugriff 22.05.2013)

[Abbildung 23] Coverage; http://docs.codehaus.org/display/SONAR/Metric+definition; (Zugriff 22.05.2013)

[Abbildung 24] Line Coverage; http://docs.codehaus.org/display/SONAR/Metric+definition; (Zugriff 22.05.2013)