BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INFORMATION SYSTEMS
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

INTEGRATION OF STATIC CODE ANALYSIS INTO ISSUE TRACKING SYSTEM
INTEGRACE STATICKÉ ANALÝZY KÓDU DO SYSTÉMU NA SLEDOVÁNÍ PROBLÉMŮ A CHYB

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR
AUTOR PRÁCE

MARKÉTA JANČOVÁ

SUPERVISOR
VEDOUCÍ PRÁCE

Doc. Dr. Ing. DUŠAN KOLÁŘ

BRNO 2018

Brno University of Technology - Faculty of Information Technology

Department of Information Systems   Academic year 2017/2018

Bachelor's Thesis Specification

For: **Jančová Markéta**
Branch of study: Information Technology
Title: **Integration of Static Code Analysis into Issue Tracking System**
Category: Information Systems

Instructions for project work:

1. Study possibilities of API provided by tools JIRA, SonarQube, and Bitbucket.
2. Propose and design JIRA plug-in that provides information about SonarQube evaluation of the code connected with particular issue in JIRA. Consider even other possible information that may be provided.
3. Implement proposal from (2).
4. Suggest and perform suitable testing to verify that the plugin can be used in an enterprise environment.
5. Discuss and evaluate your solution.

Basic references:

- Sonar API: https://docs.sonarqube.org/display/SONARQUBE43/Documentation
- JIRA for developers: https://developer.atlassian.com/jiradev
- Bitbucket API: https://developer.atlassian.com/static/rest/bitbucket-server/5.4.1/bitbucket-rest.html
- Other according to recommendation of the tutor.

Requirements for the first semester:
First two items of this specification.

Detailed formal specifications can be found at http://www.fit.vutbr.cz/info/szz/

The Bachelor's Thesis must define its purpose, describe a current state of the art, introduce the theoretical and technical background relevant to the problems solved, and specify what parts have been used from earlier projects or have been taken over from other sources.

Each student will hand-in printed as well as electronic versions of the technical report, an electronic version of the complete program documentation, program source files, and a functional hardware prototype sample if desired. The information in electronic form will be stored on a standard non-rewritable medium (CD-R, DVD-R, etc.) in formats common at the FIT. In order to allow regular handling, the medium will be securely attached to the printed report.

Supervisor: **Kolář Dušan, doc. Dr. Ing.,** DIFS FIT BUT
Beginning of work: November 1, 2017
Date of delivery: May 16, 2018

L.S.

Dušan Kolář
Associate Professor and Head of Department

Abstract

Static code analysis is a way of improving source code quality. It also helps to reveal bugs before they cause serious runtime problems. SonarQube is a tool that detects findings during periodical analyses and creates appropriate issues that provide essential data to help a developer to fix problems. Jira is an issue tracking system that is used by agile teams all over the world. Integrating static code analysis into issue tracking system should prevent Jira issues to be reopened due to bugs and reduce a number of bugs that are caused by integrating new features.

This thesis investigates ways of integrating SonarQube into Jira and provides possible solution of the integration. The solution is tested, optimized, and another solutions are provided.

Abstrakt

Statická analýza kódu je jedna z možností, jak zlepšit kvalitu zdrojového kódu. Také napomáhá k odhalení problémů před tím, než způsobí závažné problémy za běhu programu. SonarQube je nástroj, který provádí pravidelné analýzy kódu, při nichž detekuje chyby a následně informuje vývojáře o kvalitě kódu a nalezených problémech. Jira je systém na sledování problémů a chyb, který využívají agilní týmy po celém světě. Integrace statické analýzy do systému na sledování problémů a chyb by měla předcházet nutnosti znovuotevření problému kvůli nevhodnému řešení a také snížit množství chyb, které vznikají z důvodu zavádění nových funkcí.

Tato práce zkoumá způsoby integrace SonarQube do Jiry a následně nabízí možnost jejího řešení. Výsledná implementace je otestována, zoptimalizována a jsou poskytnuty návrhy na další vylepšení.

Keywords

SonarQube, Jira, Bitbucket, Atlassian, static analysis, issue tracking, integration

Klíčová slova

SonarQube, Jira, Bitbucket, Atlassian, statická analýza, sledování chyb, integrace

Reference

Rozšířený abstrakt

Kvalita kódu je definována osmi základními charakteristikami: vhodnost, spolehlivost, efektivita, použitelnost, bezpečnost, udržovatelnost, přenositelnost a kompatibilita. Kvalitní kód umožňuje předcházet různým problémům, které vznikají zejména z důvodu nesprávné reakce na neočekávané vstupy, během chybám, jakož i problémům s efektivitou a bezpečností. Přestože je nereálné dosáhnout bezchybného kódu, existuje několik způsobů, jak lze zajistit určitou míru kvality zdrojového kódu. Může se jednat o manuální i automatizované metody. V případě automatizovaných metod nejčastěji hovoříme o statické a dynamické analýze kódu.

Tato práce se zabývá statickou analýzou kódu a způsobem, jak zvýšit přínos jejích výsledků. Jedním z nástrojů pro statickou analýzu kódu je SonarQube. SonarQube provádí pravidelné analýzy zdrojového kódu, během kterých vyhledává kritická místa, jež představují rizikové oblasti pro vznik chyb. Nalezené problémy rozděluje do kategorií podle jejich typu a závažnosti. Druhy nálezů se rozdělují do tří základních skupin:

- Problém, který by mohl způsobit chybové stavy, pády za běhu programu a jiná neočekávaná chování.

- Upozornění na nedostatek bezpečnosti. Hrozí, že aplikace může být využita jiným způsobem, než je očekáváno.

- Informace o špatném návyku programátora. Nález sám o sobě nepředstavuje problém, ale snižuje přehlednost kódu.

Každý nalezený problém je také klasifikován na základě jeho závažnosti. Závažnost je rozdělena do pěti kategorií dle míry dopadu a také pravděpodobnosti, s jakou tohle chování nastane.

Jira je systém na sledování problémů, chyb a požadavků, jehož cílem je usnadnit vývoj aplikací agilním týmům po celém světě. Uživatelé mohou vytvářet projekty a v rámci projektů jednotlivé úkoly, které reprezentují požadavky, které je potřeba splnit. Jira umožňuje přidávání různých doplňků, které realizují chybějící funkcionality a integrace. Cílem této práce je využít této funkce a vytvořit takový doplněk, který umožní zobrazování SonarQube výsledků přímo v Jiře.

Bylo stanoveno několik kritérií, které by SonarQube integrace měla splňovat, jako je například oddělenost od jiných funkcí v Jiře. Implementace by neměla nijak ovlivňovat ostatní prvky Jiry, nevyužívat jejich funkce a umožnit co nejsnadnější manipulaci a nastavení. Samotné zobrazení musí být efektivní a intuitivní.

Na základě požadavků byl navržen způsob integrace, který využívá panel na zobrazování detailů jednotlivých Jira problémů. Integrace je implementována pomocí různých technologií v jazycích Java, JavaScipt a HTML. Strukturu programu tvoří objektový model, který umožňuje zahrnování závislostí a definování komponent.

Hlavní součástí integrace je Analyzér, který je zobrazen na panelech jednotlivých projektových úkolů. Analyzér se skládá ze čtyř sekcí:

- Všeobecný přehled o kvalitě kódu. Tahle sekce se skládá ze dvou statistik, podle typu nebo závažnosti nálezu, mezi kterými lze přepínat.

- Reprezentace jednotlivých nálezů. Reprezentace je realizována pomocí tabulky, kde každý nález představuje jeden řádek. V základním pohledu je zobrazena závažnost a typ problému, v jakém souboru se nachází a odkaz do SonarQube. Každý řádek lze

expandovat, což zobrazí více informací: čas vytvoření, číslo řádku, daný úsek kódu a komu je nález přiřazen.

- Vyhodnocení kvality dle požadavků. Požadavky na kvalitu nastavuje projektový vedoucí ve speciální sekci. Má možnost vybrat konkrétní stupně závažnosti a typy SonarQube nálezů a limit, po který jsou akceptovatelné. Tyto požadavky jsou poté vyhodnoceny v každé analýze. Pokud daný Jira problém nesplňuje požadavky projektového vedoucího, uživatel je na to upozorněn.

- Uzpůsobení výsledků. Výsledky analýzy je možné filtrovat a upravovat v případě, že není žádoucí zobrazovat některé (zejména málo závažné) nálezy.

Realizace spočívá v implementaci RESTové webové služby v Javě, která svými koncovými body realizuje jednotlivé funkce Analyzéru. Implementace koncových bodů spočívá v propojení Jiry a SonarQube tak, aby zvolené koncové body vracely co nejpřesnější výsledky v co nejkratším časovém úseku. Jednotlivé metody jsou volány pomocí asynchronních požadavků, což umožňuje běh více částí Analyzéru současně. Data, která jsou sdílena některými částmi, je nutno získat samostatně při startu Analyzéru a udržovat je na straně serveru, aby později nedocházelo k redundantímu zasílání požadavků.

Pro důkladné otestování řešení byly provedeny tři druhy testů: manuální, jednotkové a testy výkonnosti. Testy ukázaly, že přestože implementované koncové body představují přímou cestu, jak získat požadované informace o kvalitě kódu, právě tento přístup způsobuje největší problémy s efektivitou. Získat všechny informace přímočaře vyžaduje mnoho volání koncových bodů v SonarQube. Na základě výsledků testování byly navrženy následující body optimalizace:

- Najít vhodnější cestu získávání statistik i za cenu toho, že bude nutno lokálně zpracovávat větší obnos dat,

- v každé analýze provést nejdříve test, zda má smysl provádět všechny akce Analyzéru pro případ, že implementace neobsahuje žádné nálezy,

- zvážit dočasné uchovávání dat, která jsou složitě získatelná.

Optimalizace se zaměřovala převážně na způsob, jak lokálně vytvořit statistiku ve stejné kvalitě jako s daty od SonarQube, ale s minimálním množstvím RESTových volání. Byla zvolena metoda, která stahuje informace o všech nálezech v rámci dané SonarQube analýzy, a následně tyhle informace lokálně zpracovává do požadované podoby. Výhodou je, že oba druhy statistik lze nyní získat současně bez výraznější ztráty efektivity. Tato metoda byla zkombinována s uchováním většího objemu dat na straně uživatele za účelem urychlení operací, jako je například filtrování výsledků.

Metody optimalizace umožnily zvýšit efektivitu řešení tak, že se výpočetní čas snížil zhruba na jednu čtvrtinu původního. Prokázalo se, že není efektivní získávat již zpracovaná data pomocí velkého počtu RESTových volání. Vhodnější je stahovat větší obnosy dat v co nejmenším počtu volání koncových bodů a tyhle data zpracovávat lokálně.

Integration of Static Code Analysis into Issue Tracking System

Declaration

Hereby I declare that this bachelor's thesis was prepared as an original author's work under the supervision of Mr. Dušan Kolář. All the relevant information sources, which were used during preparation of this thesis, are properly cited and included in the list of references.

<div align="right">

. . . .
Markéta Jančová
May 13, 2018

</div>

Acknowledgements

I would like express my sincere thanks to my supervisor, doc. Dušan Kolář, for responsible guidance and providing valuable feedback.

I would also like to thank Lukáš Pitoňák for the opportunity to work in Honeywell and for his guidance.

I am also grateful to whole Honeywell Software Tools team for their help with the development.

Contents

Chapter 1

Introduction

1.1 Context

The Atlassian Jira is popular tool that is used by agile teams all over the world. Although it offers many features, there is still missing integration that would allow displaying SonarQube issues directly in Jira. Possibility of showing SonarQube findings could help to predict future bugs that are connected with particular Jira issue. This approach prevents closed issues to be reopened due to a bug that came up from inappropriate task solution or ignoring unexpected states and inputs. SonarQube also informs a developer about his bad practices, it can improve code quality of whole project and prevent a need of source code refactoring because of poor code quality.

The main goal of this bachelor thesis is to find a way how to integrate static code analysis results into Jira and show them in a readable and understandable form. The new Jira plugin should display any SonarQube findings connected with the Jira issue, inform a user about code quality, and provide an information that helps to solve the problem without using other applications like Unity, SonarQube, and Bitbucket.

The thesis is focused on the possibilities of integration. It discusses both functional and visual aspects. It is important to bring the best possible results in acceptable time. There are many ways of possible implementation, it depends on chosen combination of endpoints, that are called, and also on the solution of getting missing essential data. The plugin has to be prepared for all issues of any size, that can consist of many commits, so the amount of data can be large. The result should be well-tested to ensure optimal and appropriate behavior of the plugin.

1.2 Thesis Structure

The thesis is further divided into the following eight chapters.

Chapter 1 Introduction describes context and goals of this bachelor thesis.

Chapter 2 Code Quality provides an overview about code quality in general and describes basic terms that are important for effective development.

Chapter 3 Tools introduces applications, that are necessary for successful implementation, and discusses possibilities of APIs that they provide.

Chapter 4 Application Design discusses functional and user interface requirements, and the concept of the application. These requirements are evaluated to visualize the proposal of the plugin structure.

Chapter 5 Implementation describes tools that were used for implementation, configuration, plugin structure, and the implementation process itself.

Chapter 6 Verification sums up both manual and automated testing results, and discusses possible optimization according to these results.

Chapter 7 Optimization and Future Work describes methods and refactoring that came up from the results of verification, and also provides better solution that could be reached using different versions of tools.

Chapter 8 Conclusion evaluates the goal fulfillment.

Chapter 2

Code Quality

The ISO/IEC 25010:2011 product quality model categorizes product quality properties into eight characteristics (functional suitability, reliability, performance efficiency, usability, security, compatibility, maintainability, and portability) [6].

- Functional suitability is a degree to which a product provides functions that meet requirements when used under specified conditions. This characteristic consists of functional completeness, functional correctness, and functional appropriateness.

- Reliability is a degree to which a product or its component performs required functions under specific conditions for a particular period of time.

- Performance efficiency is a performance under stated conditions with the consideration of the amount of resources that were used.

- Usability is a degree to which a product can be used to achieve required goals with effectiveness, efficiency, and satisfaction.

- Security degree to which a product protects information and data so that users or other products have the degree of data access appropriate to their types and levels of authorization.

- Compatibility is a degree to which a product can exchange information with other products and perform its required functions, while sharing the same hardware or software environment

- Maintainability is a degree of effectiveness with which a product can be modified without a need of changing other parts.

- Portability is a degree of efficiency with which a product can be transferred from one hardware, software, or other environment to another.

2.1 Source Code Analysis

Some quality attributes can be reached using manual or automated methods of analysis such as static and dynamic code analysis.

- **Manual code review** is the simplest method but also the least effective technique. It requires high programming language knowledge and ability to analyze potentially critical parts of source code. Its time cost is high because of impossibility of any automation. This method usually reveals only the most obvious bugs.

- **Static code analysis** refers to any process of assessing code without executing it. Static analysis is powerful because it allows quick consideration of many possibilities. A static analysis tool can explore a large number of "what if" scenarios without having to go through all the computations necessary to execute the code for all the scenarios [4].

- **Dynamic code analysis** is based on the system execution, often using instrumentation. Its advantage is in ability to detect defects that cannot be revealed in manual or static code analysis.

2.2 Issue Tracking

Issue tracking systems are systems with the main purpose in collecting requirements, their management, and tracking their progression towards resolution. Pieces of work are divided into issues. Issue can be described by its details such as type, status, severity, assignee, date of submission, attempted solutions, and other relevant information. Modern issue tracking system should be able to:

- Share the information across the team,

- have an instant overview of the state of software,

- expertly decide about releasing,

- set and update the importance of individual fixes and adjustments,

- have a recorded history of changes. [8]

Chapter 3

Tools

This chapter is focused on applications that provide important data for integration. Each application is shortly introduced, the description contains basic information about context and characterization of its API. This chapter also defines important terms that will be used in implementation process.

3.1 Jira

Jira is an issue tracking system used by over one million users around the world. It is produced by Atlassian and its main goal is to support agile teams to develop their products more efficiently.

Teams cooperate using Jira projects [10]. A Jira project is a collection of issues that is defined according to organization's requirements. Each project has a name and a key. The project key becomes the first part of that project's issue keys.

Different organizations use Jira to track different kinds of issues [11]. Depending on how organization is using Jira, an issue could represent a software bug, a project task, a helpdesk ticket, a leave request form, etc. Each issue is defined by many attributes like project, key, summary, status, priority, resolution, etc. Jira implicitly defines some issue types, but administrators can manage issue types, modify them, or create a new type. Implicit Jira issue types are:

- **Bug** - a problem that affects functionality of the application,

- **Improvement** - an enhancement to an existing feature or solution,

- **New feature** - a request to develop a new feature of the application,

- **Task** - any piece of work that needs to be done,

- **Custom issue type** - a type optionally set by administrator.

When commiting a code, commit message should contain appropriate issue key. It allows version control system to recognize, which commits belong to a specific issue.

Definitions of Done

A Definition of Done (DoD) is a clear and concise list of requirements that the software must adhere to for the team to call it complete. It is the contract that binds what the Product

Owner (PO) wants to what the Development Team delivers [13]. Completing the DoD list ensures the quality of the product. Each Jira issue should have its own DoD specifications that are usually represented as a custom field or checklist on the view issue screen[1]. DoD can be also global. Global DoD sets the requirements on all issues, both new and old.

Jira Application Links

Jira Application Links is a bundled plugin that allows linking with other applications. Linking two applications allows sharing information and access one application's functions and resources from within the other. Linking Jira to Bitbucket allows viewing commits, branches, and pull requests that correspond to specific Jira issue [9].

3.2 SonarQube

SonarQube is an open source software that performs automatic reviews using static code analysis to detect issues and bad practices in programming. It supports over 20 programming languages but it is mainly used for Java, JavaScript, and C/C++. Reviews are completed during periodical analysis of source code. Whenever any problem is found, SonarQube creates issue that describes the problem by its specific parameters.

Quality Measurements

There is not any optimal solution of project quality measurement for all projects. Some projects need different rules because of stronger quality requirements. Quality profile [21] is a set of rules that can be defined by project leader. Quality gate [20] is a set of boolean conditions that are evaluated to bring information whether project passes quality requirements or not.

3.2.1 Issues

Poor code quality causes a variety of issues that can negatively impact product quality: crashes in production, application decommissioning, security problems, etc. SonarQube analysis automatically detects coding rule break and creates appropriate issue that represents the problem. The issue is stored in database and is displayed in application, so each project participant can check analysis results. User can also create issue manually.

Each issue is defined by its key, type, severity, status, message, assignee, project, file, line of code, flow, time afford, creation date, and update date.

Issue Types

Issue type is an attribute that specifies impact that can be caused by ignoring it. SonarQube defines 3 different issue types:

- **Bugs** are the most serious reliability issues. This category of issues groups everything that might cause unexpected runtime behavior or production crashes.

- **Vulnerabilities** are security issues. This is commonly referred to as vulnerabilities or flaws in programs that can lead to use of the application in a different way than it was designed for [16].

---

[1]The screen that displays information about specific issue.

- **Code smells** are maintainability findings. This category of issues groups everything that has to do with possible difficulties in future updates.

Issue Severities

Severity generally means the measure of impact that could be caused by this issue and also the probability of this behavior. Severities [19] are divided into 5 categories: blocker, critical, major, minor, info.

- **Blocker** is an issue with a high probability to cause some serious problems such as memory leaks.

- **Critical** is an issue with a low probability to cause negative impact to behavior in production or security flaw, for example badly caught exception.

- **Major** is a quality flaw that can highly impact developer's productivity, for example too complex method.

- **Minor** is a quality flaw that might impact developer productivity, for example naming conventions.

- **Info** is an unimportant finding that warns developer about his/her bad practices. The risk is unknown or not well defined yet.

Issue Lifecycle

Each issue flows through a lifecycle that starts after the issue is detected and created. Through whole lifecycle, the issue can take one of the five possible statuses:

- **Open** - set by SonarQube on new issues,

- **Confirmed** - set manually to indicate that the issue is valid,

- **Resolved** - set manually to indicate that the next analysis should close the issue,

- **Reopened** - set automatically by SonarQube when a resolved issue has not actually been corrected,

- **Closed** - set automatically by SonarQube for automatically created issues. [18]

An issue can take "Closed" status only in case of being resolved or rejected. Closed issue is kept in database for 30 days for the case of need to be reopened. There are two possible resolutions: Fixed or Removed. A developer can also manually decide whether an issue should be fixed or not. Doubtful issues can be marked as "false positive" or "won't be fixed" (Figure 3.1).
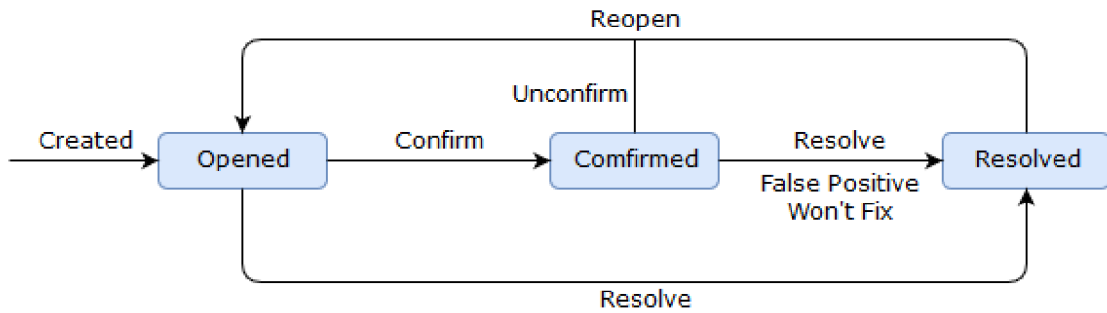
Figure 3.1: SonarQube issue lifecycle

3.2.2 SonarQube API

SonarQube web API contains many REST endpoints [17] that provide an information about user, authentication, components, metrics, issues, permissions, and more.

Authentication

Authentication is a process by which an application confirms user identity. SonarQube requires authentication details in each request on its API. There are two methods of username and password validation that are supported by SonarQube.

- **HTTP Basic access authentication** is the simplest type of authentication. Username and password are passed to SonarQube API as request parameters. Despite its advantage in unnecessariness of any configuration by user, its big disadvantage is in lack of security. Credentials are not secured, they are passed in the same form as they were entered by a user.

- **User token authentication** is the recommended way. It is a safer method than basic access authentication, because user authenticates only when s/he generates a token. The token is a piece of data that contains encrypted credentials and information that is used for token validation. Server encrypts credentials using various complex cryptography techniques. Token is saved locally and is used every time user authenticates.

Authentication is passed to SonarQube using relevant request. SonarQube responses whether username and password are valid authentication details or not.

Request structure
```
GET /api/authentication/validate -u {Username}:{Password}
GET /api/authentication/validate -u {Token}
```

Response
```
{"validate":true}
{"validate":false}
```

SonarQube Components

SonarQube provides REST endpoints that return information about a component. The component can be for example a project, a file, or a directory. Only one component can be

returned at a time. A user needs browse permission to view requested component.

Request structure
```
GET /api/components/show
GET /api/compoentns/tree
```

SonarQube Issues

Information about specific issues is returned by `issues` endpoints. Issues can be filtered by several optional attributes. Filtering allows user to display issues by creation date, update date, assignee, severity, type, and more. User needs browse permissions on the project. Since version 3.6, user can also edit particular issues using REST API.

Request structure
```
GET /api/issues
```

3.3 Bitbucket

Bitbucket is a web-based hosting service developed by Atlassian. It is used for Mercurial and Git version control systems. A version control system is a tool that helps a software team to manage source code changes that came over time. It keeps a track of every modification of source code in a special kind of database. When a mistake is made, the database enables getting the project repository to one of the previous versions to minimize disruption of other team members' work.

3.3.1 Bitbucket API

Bitbucket API contains many REST endpoints [3] that can provide information about projects, repositories, commits, pull-requests, users, groups, permissions, and more.

The preferred authentication methods are HTTP Basic Authentication and OAuth, supported are also HTTP Cookies and Trusted applications. When using Bitbucket API directly from Jira, there is no need to require authentication details. Jira can directly access Bitbucket using application links.

Projects

With appropriate permissions, a user can view and modify his/her projects. This REST endpoint provides information about project itself, and also about project participants, permissions, groups, repositories. User can also browse commits associated with the project using particular repository slug. With modify permission, user can also create, delete and modify his/her repositories.

Request structure
```
[GET|POST] /rest/api/1.0/projects
[DELETE|POST|PUT|GET] /rest/api/1.0/projects/{projectKey}/repos
```

Commits

Bitbucket and Jira existing integration enables searching for commits that are associated with specific Jira issue.

Request structure
```
GET /rest/jira/1.0/issues
```

3.4 Unity

Unity provides a User Management module for easy-to-use project navigation and unified user permissions configuration.

3.4.1 Unity API

Unity API contains endpoints that return associated projects by Unity ID or Jira project key. The ID is unique for every project. All project resources can be obtained using REST endpoint called `forToolProjectIdOrKey`. Endpoint returns two types of responses. When the attribute `crowdGroupId` is included, it returns all projects in both remote and Atlassian applications that are linked to this project through Unity. Attribute exclusion returns Unity project details (Unity ID and project name).

Request structure
```
GET /rest/api/project/forToolProjectIdOrKey
```

3.5 Integrations
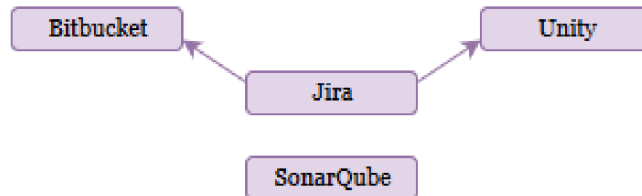
Figure 3.2: Jira integrations

Integration between Bitbucket, Unity, and Jira is already implemented. SonarQube is not integrated with any of these tools (Figure 3.2).

Chapter 4

Application Design

This chapter contains details of given goals and a basic concept of implementation. Goals are split into several requirements in order to make integration interface usable for end users. These details should guarantee that successful implementation will fulfill defined goals.

Integration and interface concept itself is described in section Design. This section provides an overview about main steps that needs to be done to implement working integration and create user-friendly interface.

4.1 Requirements

Final plugin should pass a set of functional and architecture requirements. Functional requirements are focused on general functionality and their correct fulfillment is a guarantee of concept quality. Architecture requirements include set of rules that cannot be broken during integration implementation, for example plugin separation.

4.1.1 Functional Requirements

There are several requirements on application functionality that should bring great value added to project participants. The value is in important information that cannot be directly displayed without using the integration. SonarQube panel should solve following situations:

- A user wants to display concise information about code quality quickly and directly in Jira without searching for information in other applications like SonarQube or Bitbucket.

- The user already checked quality summary and found out that there are possibly bugs. The user wants to see more information about finding to check whether the finding is a serious bug or some minor finding that does not impact functionality.

User interface should display an overview about issue implementation problems and warn project participants in the case of serious findings or poor code quality. It is also required to display some particular findings so a user does not need to open SonarQube to fix problems. Essential plugin property is the unnecessariness of doing any setup before using the integration. If there is any necessity that needs user operation, it should take as few actions as possible and very low time cost.

Summary of Findings

Summary of Findings is an information that sums up all findings that were found in analyses of changed files. It contains statistics of all analyses related to the Jira issue, what provides general overview about issue quality. This statistical information is displayed in a simple form that is visible and understandable for everyone. Findings summary section should be the response to the questions:

- How many quality problems do I have in my Jira issue?

- How serious are the problems?

Quality Gate

Quality Gate is a set of rules defined by project leader. It is represented as a short message that displays information whether issue passes required quality or not. The message also contains details that summarize what should be improved to fully pass quality requirements. Quality gate is a project specific property set by project leader in admin area. This message should solve the following questions:

- Does the Jira issue pass project owner's requirements?

- Can I close the Jira issue?

Findings

Findings are elements that represent real issues. These elements should be able to visualize all important details that might help developer to resolve the problem without using SonarQube application. Findings help a user with the following problem:

- How can I improve code quality?

4.1.2 Architecture Requirements

Architecture requirements are the requirements that are set on the general concept of integration. The Integration should work on Jira version 7.3.8 and higher and SonarQube version 5.6 and higher.

Application needs to be fully separated from other Jira features, it has to work apart of any functionalities that are not included in Jira core.

There is a strict requirement to do not impact view issue screen, its behavior has to be stable, no matter whether integration is enabled or not. Integration processes cannot slow down any processes of other panels and features.

4.2 Design

An application design describes individual concepts of elements from previous section. This section is divided into two subsections. The first subsection is focused on integration implementation. It describes implementation goals that need to be done to bring important data from SonarQube to Jira. There are also discussed integration problems caused by missing details that are very important for implementation. The second subsection suggests possible user interface mockup that displays all essential data needed to help a user to fix a problem and allows interaction between the user and integrated tools.

4.2.1 Integration Concept

This section discusses possible concept of integration between Jira and SonarQube. The goal is to find the most effective way how to acquire essential data from REST endpoints and how to parse it without any negative runtime impact.

Finding Representation

SonarQube findings are represented in two different ways. One kind of representation is a summary of all findings. The summary is a part of the issue tab panel[1] that contains general statistic of SonarQube findings. Another representation is a table that represents specific findings. The table of findings contains rows that are created according to data from SonarQube. This information can be acquired using SonarQube REST endpoint (Section 3.2.2) that returns specified findings. Result validity is ensured using parameters that are separated with ampersand. Valid filter options are:

- **componentRoots** parameter is a specification of SonarQube project or complete SonarQube path. File path format should correspond with official SonarQube formatting.

- **createdBefore** and **createdAfter** specify creation date filter. Creation date is a time, when an issue was created in SonarQube. This date is not corresponding with particular commit date, but with the time when SonarQube analyzer actually found the issue. Each analysis contains all commits that were posted during the period after the previous analysis. When searching for specific analysis, creation date is guaranteed by two time details: `createdAfter` parameter to specify the time of particular analysis and `createdBefore` parameter to enclose the time period. These two parameters should determine the smallest possible time interval. Both parameters are timestamps represented in ISO 8601 [7] format.

- **ps** and **p** attributes define number of results. The **ps** attribute sets the maximal number of findings in returned content. When searching for statistics, it is not needed to request finding details, because JSON header contains summary of found content. When searching for specific finding details, it is essential to get as many findings as possible in one response to minimalize the number of requests. SonarQube limit of finding details in one response is set to 500. If the number of responses is higher than 500, attribute **p** can be used to get other pages of response.

- **severities** applies a filter that selects only findings of specific severity. Accepted filter options are basic SonarQube severities written with uppercase letters and separated by commas, for example "severities=BLOCKER,MAJOR,INFO". Default setting is searching for findings of all severities.

- **types** enables returning findings of particular type. This filter accepts uppercase comma-separated options that are appropriate SonarQube issue types. Blank spaces are represented with underscore, for example "types=BUG,CODE_SMELL".

- **s** and **asc** attributes define the order of results. Attribute **s** defines an element that is used as a key. The key should be relevant finding attribute written with uppercase

---

[1]The panel that is reached from the tab panel menu on the view issue screen.

letters, for example "s=CREATION_DATE". Attribute `asc` sets the order, value "true" means ascending order, "false" means descending order.

The attributes **severities** and **types** depend on a user choice, which statistic s/he finds the most valuable. Summary displaying statistic by type uses `type` parameter and creates 3 requests (one for each type), and leaves `severities` parameter empty. Statistic that shows counts by severity uses `severities` parameter and leaves `type` empty in every call. It also sends request for each severity, 5 requests in total. Particular issue objects are received by leaving both parameters empty, the response contains all findings of any type and any severity.

Jira does not offer any information about changed files and date. Missing data can be acquired from Bitbucket API (Section 3.3.1). Bitbucket is already integrated with Jira, so it offers information about commits with Jira issue key in commit message. Responses contain all commits associated with the issue. This data provides information about changed files and commit time, represented as timestamp in milliseconds. The information is valuable but not enough for getting findings. There are two problems that need to be solved, because they might cause result inconsistency:

- Changed files are specified by filename and Bitbucket path. SonarQube REST endpoint accepts only SonarQube paths. Obtained paths need to be converted.

- Although date conversion from timestamp in milliseconds to ISO 8601 timestamp is a simple task, the date is an information about commit. SonarQube issues are not specified by commit time but by analysis time. Commit time usage would not probably return any findings, and large interval would return many findings that are not associated with the Jira issue. Analysis time needs to be obtained from SonarQube.

SonarQube offers REST endpoint that returns SonarQube paths. This endpoint can solve the path inconsistency. The following parameters are required:

- `baseComponentKey` specifies SonarQube project. This attribute accepts project key.

- `qualifiers` defines a type of results. Value "FIL" selects files, "DIR" selects directories.

- `ps` sets a limit of results. Value can't be more than 500. If the number of responses is higher than 500, attribute `p` can be used to get next pages.

- `q` attribute defines a filename as "q={filename}" or absolute SonarQube file path "q={project}:{subprojects}:{path}". Only one file can be selected at a time.

The `component` parameter accepts only one name at a time. This fact needs to be taken into consideration because inappropriate solution might cause huge effectivity leak. Leaving this parameter empty causes getting all file paths. This solution might be more suitable. Despite empty parameter returns lots of data that need to be parsed, it is more effective approach than sending a request for each changed file.

Figure 4.1 displays communication of Bitbucket, Jira, and SonarQube. When searching for specific SonarQube issues, it is needed to select appropriate analysis that created these issues. Analysis timestamp is an information about analysis which was the following one after Bitbucket commit. Although SonarQube version 6.3 offers REST endpoint that

returns all analysis dates, version 5.6 does not offer appropriate REST endpoint that provides these results. It causes difficulties in searching for required data. Possible solution is using endpoint that returns specific finding details because finding creation date is a time of analysis. Its big disadvantage is that it only returns analyses that found some issues so there might be some false positives created. This might be caused by small commits that changed a few files. If there is only one or several commits like that in a long time period, there is a possibility that it does not create any new SonarQube issue. Analyses that were successful are not included in results from issue REST endpoint because analysis date is obtained from issue creation date. This inconsistency is accepted and better solution for SonarQube 6.3 is provided in chapter Optimization (Chapter 7).
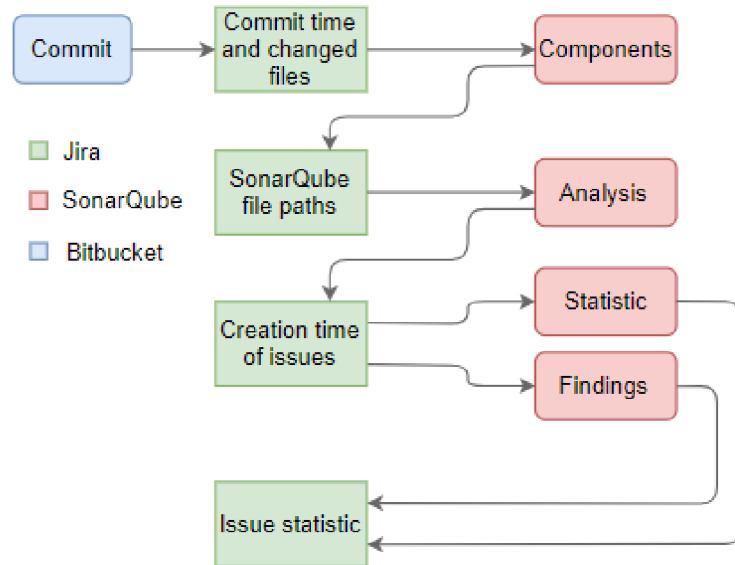
Figure 4.1: Connection of Jira, SonarQube, and Bitbucket

Quality gate

Quality gate is a message that contains information about issue quality. It summarizes quality requirements fulfillment. The requirements are set by project leader in admin area on the project tab panel[2], this feature is project specific so each project has its own quality measurements. The project leader can select custom combination of severities and types to create a new rule up to 10 rules limit. Each rule consists of combination of severities and types, limit of convenient findings and seriousness of rule. After applying the rules, each issue is validated when a user opens SonarQube issue tab panel. Validation result is displayed as a message that describes rule breaks and general quality. General quality is calculated according to percentage of broken rules and their seriousness. There are 3 degrees of quality results that indicate whether issue passes requirements or not.

---

[2]The panel that is reached from the menu of particular project.

Authentication

Authentication is the first action that needs to be done in order to use the integration. SonarQube credentials are validated using basic access authentication, a user enters username and password, and posts them to SonarQube in the same form as they were entered. Despite its lack of security, it is the most suitable way for users because a user does not need to do any settings before s/he uses the plugin. When a user opens SonarQube tab panel on view issue screen, application requests credentials to access SonarQube. After the user enters password and username, details are validated via appropriate SonarQube REST endpoint. If the SonarQube endpoint responds `{"valid":true}`, analysis screen and loading process can start immediately after the response is received, answer that contains `{"valid":false}` alerts user and requests him/her to enter the credentials again. Authentication details are saved in session so the user does not need to repeat authentication because details are kept until s/he logs out.

4.2.2 User Interface

User interface (UI) is a visual part of application through which a user interacts with a software. UI design is usually more important than the application itself.
There are many principles that define good design practices. One of the principles is CRAP[3]. It defines several rules that help developers to consistently deliver effective design [22]:

- **Contrast** - The idea behind contrast is to avoid elements on the page that are merely similar. If the elements (type, color, size, line, etc.) are not the same, then make them very different. Contrast is often the most important visual attraction on the page - it is what makes a reader look at the page in the first place.

- **Repetition** - Repeat visual elements of the design throughout the piece. You can repeat colors, shapes, textures, etc.

- **Alignment** - Nothing should be placed on the page arbitrarily. Every element should have some visual connection with the other elements on the page.

- **Proximity** - Items relating to each other should be grouped close together. When several items are in close proximity to each other, they become one visual rather than several separate units.

Messages

Quality gate and data status messages (Figure 4.2) are displayed at the top of the page. Quality gate message contains the information about quality requirements. This message is the most important element on the screen, so it has to be visible and placed at the top of the panel. Data status message displays an information whether results are actual or outdated. There are 4 types of data status messages:

- Result includes the newest commit and the newest analysis.

- Result includes the newest commit, but the commit has not been analyzed in SonarQube yet. The result might be outdated soon.

---

[3]Contrast, Repetition, Alignment, Proximity

- The issue was never analyzed before.

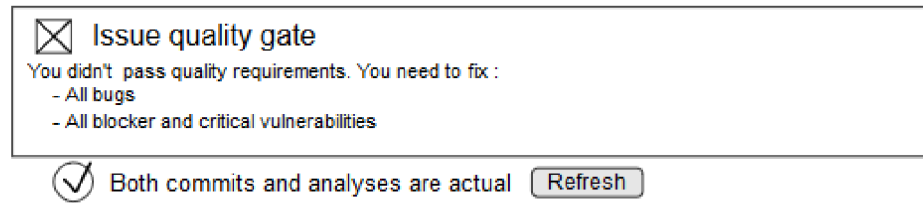- Data status is unknown. A problem occurred while contacting SonarQube or Bitbucket.

Figure 4.2: Issue tab panel messages

Issue Quality Summary

Quality Summary is an overview about issue code quality. There are two types of summaries that can be displayed:

- Summary by severity

- Summary by type

The summary section is placed under messages on the screen. There are two types of statistic that can be viewed. One of them is summary by severities (Figure 4.3), it displays a number or findings for each severity. The other option is summary by type (Figure 4.4), it displays a number of findings for each type. A user can optionally choose the statistics s/he wants to display, but only one summary, according to types or severities, can be visible at a time. A user can click on appropriate button to swap the view. Every element is interactive. Clicking on blockers displays only findings of severity "blocker", clicking on vulnerabilities selects findings of type "vulnerability" etc.
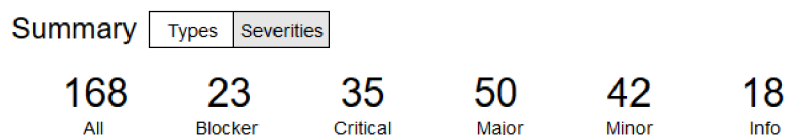
Figure 4.3: Issue tab panel summary by severities
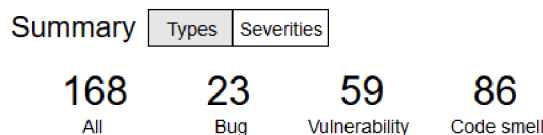
Figure 4.4: Issue tab panel summary by types

Filters

Filtering allows result customization. Findings can be filtered by type, severity, or assignee. Default filtering enables displaying findings of all types, severities, and assignees. If a user wants to change filters, s/he can use the button called "Show filters" which shows filter options and enables result customization. If the user checks for example only vulnerability and unchecks the other issue types, results will be only of type code smell or bug. Unchecked options are also excluded from general statistic, but they cannot be excluded from the quality validation message.

The upper part of filtering section contains information about filters that are actually used. This message is visible in both situations, when filters are showed (Figure 4.5), and also when options are hidden (Figure 4.6).

Figure 4.6: Issue tab panel filters hidden

Figure 4.5: Issue tab panel filters shown

Findings

Findings are elements representing real issues.

This data is displayed as expandable table rows (Figure 4.7). Expanding reveals more information like issue message, assignee, creation date, and line number. There is also a possibility to see the code lines that are specified in finding details. Plugin displays highlighted issue line and two lines around the line. If there are more lines specified, screen displays all lines between boundary lines. The boundaries are highlighted. Only one issue can be expanded at a time to keep the table organized. The limit is set to 10 issues at the start. If a user wants to see more issues, s/he can expand the table. This option is available only when SonarQube returns more than 10 issues.

Admin Area

The plugin also contains an area that serves for integration configuration. There are two modes of quality configuration project tab panel.

One of the modes is the admin mode (Figure 4.9). Admin mode is a view for project administrators and enables custom settings. An admin can select custom specification of a rule that consists of severities, types, and a limit. Each rule also contains a button that removes the rule. An admin can add a rule by clicking on the "Add more" row, which

Figure 4.7: Issue tab panel findings

is the last row of the table. This command generates a new unfilled row. Clicking on the "Save" button saves the selected rules, if the admin does not save the changes, they disappear when the page is left or refreshed.

The other mode is a mode for users (Figure 4.8). A user cannot do any configuration changes, s/he is only allowed to view the selected rules. Any project participant has this browse permission.

Figure 4.8: Configuration project tab panel for a user

Figure 4.9: Configuration project tab panel for a leader

Chapter 5

Implementation

5.1 Technologies

The plugin was implemented using the following front-end and back-end technologies:

- **Java** is a object-oriented programming language used for background logic implementation.

- **JavaScript** (JS) is a interpreted programming language. It is used to make web pages interactive. **jQuery** is a cross-platform JavaScript library. It is designed to simplify working with elements, event handling, and creating Ajax[1] applications.

- **Hypertext Markup Language** (HTML) is the markup language for creating web applications. HTML semantically describes the structure and appearance of a web page.

- **Velocity Template** is a template engine that provides a template language to reference objects defined in backend. It is the main engine used for displaying pages in Atlassian applications [12]. The Velocity Template Language (VTL) provides the way how to create macros and operate with Java objects directly from HTML.

- **Cascading Style Sheets** (CSS) is a style sheet based language used for describing the presentation of a web page.

- **Soy Template** is a templating system for generating re-usable HTML elements that can be used from JavaScript.

5.1.1 Atlassian Software Development Kit

The Software development kit (SDK) is a tool developed by Atlassian. Using SDK, developers can create their own add-ons to extend basic functionality of Atlassian server applications. An add-on is created using command `atlas-create-jira-plugin`. This command prompts a user to define plugin core configuration details and generates a basic skeleton (Figure 5.1) of a new plugin. The file called `pom.xml` contains plugin configuration details and dependencies. The `atlassian-plugin.xml` file describes a plugin to other Atlassian applications. A developer can define elements that create component modules and visual resources.

---

[1]Asynchronous JavaScript and XML

SDK also offers a possibility to locally run Jira instance using the `atlas-run` or `atlas-debug` command in the plugin folder.
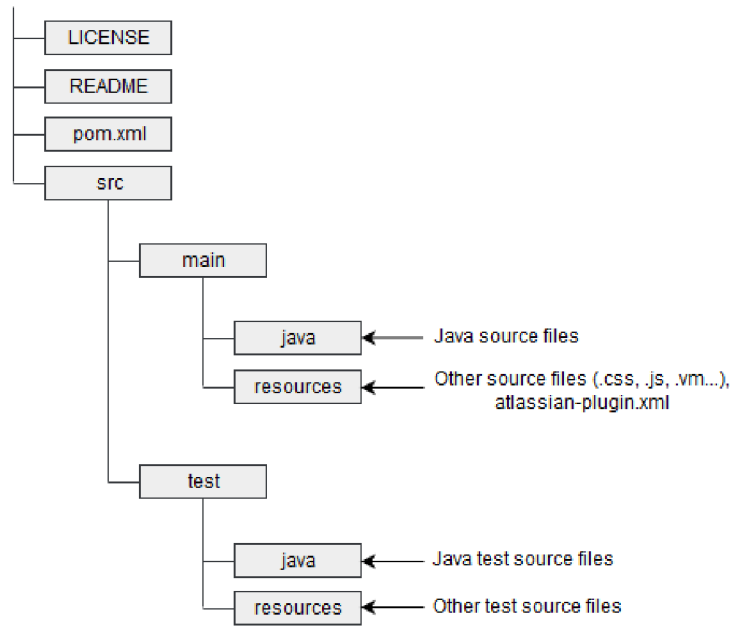

Figure 5.1: Atlassian plugin architecture

5.1.2 Apache Maven

Apache Maven is a project management and comprehension tool, based on the concept of a project object model (POM). The project object model is an XML file that contains information about project and its configuration. This file contains also dependency definitions. Dependencies are external source files that are locally stored in the maven folder. A developer can include a dependency using its ID (Listning 5.1). The `<groupId>` element defines an ID of dependency that can be obtained from the Maven Repository[2].

When a project build is invoked, Maven finds a POM file, reads it, and gets important configuration information. If it is successfully obtained, only then Maven starts executing the goal. Apache Maven can be run using `mvn [options] [<goal(s)>] [<phase(s)>]` command from the plugin directory. Options can be for example `package` for building only current plugin package, `clean` for cleaning the project before installation, `test` for running tests only, etc.

```
<dependencies>
    <dependency>
        <groupId>com.atlassian.jira</groupId>
        <artifactId>jira−api</artifactId>
        <version>${jira.version}</version>
        <scope>provided</scope>
    </dependency>
<dependencies>
```

Listing 5.1: Dependency definition

---

[2]Maven Repository https://mvnrepository.com/

23

5.2 Concept

Integration implementation consists of 5 parts:

- Issue Analyzer on the issue tab panel,

- SonarQube and Unity URL configuration in the administration area,

- SonarQube project settings on the project tab panel,

- SonarQube activation and deactivation on the project tab panel,

- Issue Code Quality Targets on the project tab panel.

Each part is implemented separately and can be changed, removed, or replaced without affecting the other parts. The only connection lies in sharing persistent data (Figure 5.2). The data is represented as active objects (AO).

The Active objects is a layer into Atlassian products, implemented as a plugin into Atlassian applications. It enables easier, faster, and more scalable data access and storage. Active objects has following advantages:

- Real database usage - AO enables accessing real database and working with real data.

- Database independence - AO abstracts all database implementation details.

- Sandboxing - A plugin can access only data that belongs to it.

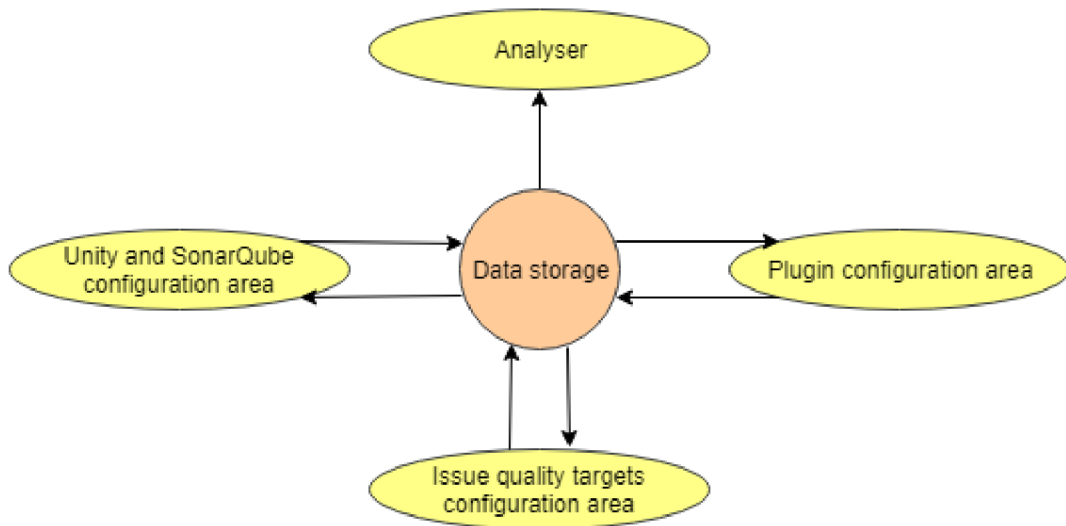- Backup/restore - The backup/restore mechanism takes care of backuping the plugin data. [1]

Figure 5.2: Connection of plugin components

5.3 Integration Implementation

5.3.1 Analyzer

The Analyzer is the main plugin feature, implemented on the issue tab panel. The issue tab panel is a plugin module that allows adding new panels to the view issue screen. It is defined in the `atlassian-plugin.xml` file (Listing 5.2) that was automatically generated by the Atlassian SDK.

```
<issue−tabpanel key="act−sonarqube−Analyzer" name="SonarQube Panel"
      class="act.jira.plugin.issuetabpanel.SonarqubeIssueTabPanel">
   <description>SonarQube issue panel</description>
   <order>30</order>
   <resource type="velocity" location="templates/sonarqube−panel.vm"/>
   <supports−ajax−load>true</supports−ajax−load>
   <label>SonarQube</label>
</issue−tabpanel>
```

Listing 5.2: Issue tab panel definition

The `<resource>` element defines the velocity template that is loaded after a user clicks on the panel. The velocity template can access Jira data using the interfaces `IssueTabPanel` and `IssueAction`, both of them are provided by Atlassian in `issuetabpanel` package. JavaScript files are appended separately using `<web-resource>` element. Web resources are downloadable resources that enable serving additional static JavaScript and CSS files. Each web resource contains file location and optionally other attributes, for example context. A context defines screen type that is enabled to use the content. Default context is `atl.general`, the resource is available everywhere except an area requiring administration privileges. Custom context in unavailable anywhere as default, but can be required from particular velocity template using Web Resource Manager:

$webResourceManager.requireResourcesForContext('atl.sonarqube')

Analyzer Structure

The Analyzer consists of two main parts: Jira endpoints (implemented in Java) and issue tab panel that shows the results (implemented using JavaScript, Velocity Templates, and Soy Templates).

Analyzer is enabled only if the Jira project, that contains the particular issue, has any SonarQube project attached and if the integration is allowed. Before the Analyzer starts loading data from remote sources, it checks the status of SonarQube and Bitbucket. If any of these servers does not respond or responses unexpected status, Analyzer informs the user and successfully ends. Actions that ensure valid start of the Analyzer are displayed in Figure 5.3.

Jira Endpoints

The Analyzer is implemented as Jira endpoints (Figure 5.4) that are called from front-end using AJAX requests:

doInitialAnalysis loads essential data into cache. The data consists of changed files in SonarQube file path format and also times of analyses.
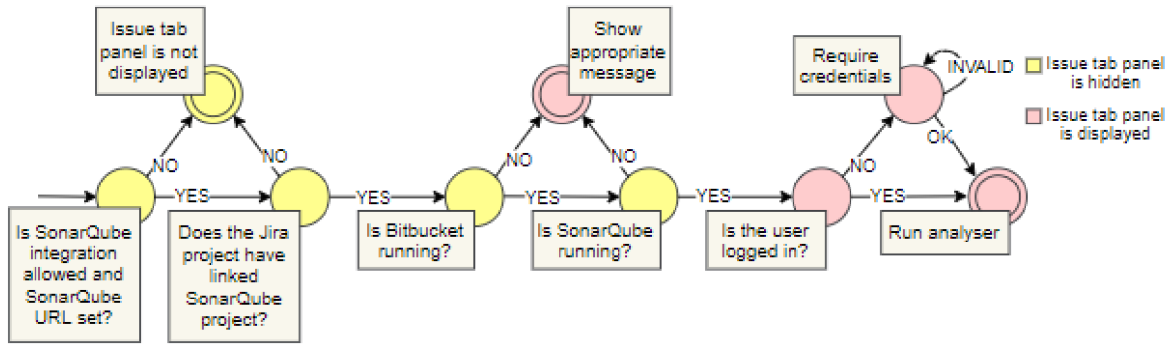
Figure 5.3: Analyzer's actions before start

The first action is loading commits from Bitbucket `/commit` endpoint. The `/commit` endpoint returns all commits, associated with the Jira issue, in JSON array. Commit times and changed files are parsed using GSON[3] library into Java objects.

File paths are obtained using SonarQube `/component` endpoint that returns all files related to the project. This approach solves the problem with inconsistencies between SonarQube and Bitbucket file paths. Response data contains large amount of unused properties that are removed using regular expressions. Final string contains only raw file paths separated by delimiter "#". This approach helps to reduce data to less than one tenth of previous size. Complete paths are then acquired using their parts that were obtained from Bitbucket commits.

Bitbucket commit times are then passed to SonarQube using `/issues` endpoint. Response contains issue objects in ascending order by creation time. Creation time of the first issue the time of the first analysis after the commit.

Collected data about changed files and analysis time are saved into cache under the issue key. All other endpoints use this data. Whenever data is removed from cache, every endpoint is able to invoke these actions again.

This endpoint responses also a part of issue statistics as JSON object that contains finding severities as attribute keys and their counts as values. This statistic is focused only on findings by their severity. The data is collected using the SonarQube `/issues` endpoint and filtering required severity to get its total count. The count is included in SonarQube response header so its not needed to return specific findings as objects. The number is parsed from the response header using GSON library.

statisticTypes creates appropriate requests to get complete overview of findings connected with particular Jira issue. This endpoint calls SonarQube `/issues` endpoint once for each finding type to collect data from response headers. The header is parsed using GSON only to get number of findings. Response contains JSON object with attributes representing each type and values representing number of findings. These data are used to create Summary section by types of findings on the issue tab panel.

customStatistic is used to get statistic with customized values. It can ignore some types, severities, or a combination of both. Selected types and severities are passed as request parameters. This endpoint is used to get new Summary values when a user filters results.

lines returns specific source code lines. The endpoint excepts file name and boundary

---

[3]An open source Java library developed by Google. It is used to serialize and deserialize Java objects to/from JSON

lines as request parameters. These details are passed to SonarQube `/sources` endpoint. SonarQube response contains specified source code lines that are parsed using GSON to remove other properties. Endpoint's response contains JSON object that consists of attributes with line numbers as keys and raw source code lines as values.

findings searches for SonarQube findings, that are connected with particular issue, and collects essential data into objects that represent findings. Findings are obtained using call to `/issues` SonarQube endpoint with analyses' times and file paths (obtained from cache) as request parameters. SonarQube response is parsed using GSON to remove unused data and represent findings in required form. Endpoint's response contains JSON array of objects representing findings with their essential attributes.

customFindings is used to get findings with customized values. It can ignore some types, severities, or a combination of both. Selected types and severities are passed as request parameters. This endpoint is used to get new specific finding objects when a user filters results.

dataStatus returns whether issue analysis is up to date, outdated, or was not analyzed yet. This endpoint calls `/commits` Bitbucket endpoint to get the newest commit time and also `/issues` SonarQube endpoint to get the newest analysis time. If the commit time is newer than the analysis time, results might be outdated because SonarQube did not analyze all commits yet. This endpoint also helps to determine whether cached data is up to date or not.

reload empties cache record to get completely new data for current issue.

qualityInfo verifies issue code quality. It compares statistic results with required quality, that was set by project leader, and returns response that contains number of rule breaks and their type. Each rule is evaluated separately using its specifications and SonarQube `/issues` endpoint. The number of findings in response is compared with the limit of acceptable findings. If the count is higher than the limit, a report is created and attached to the response.
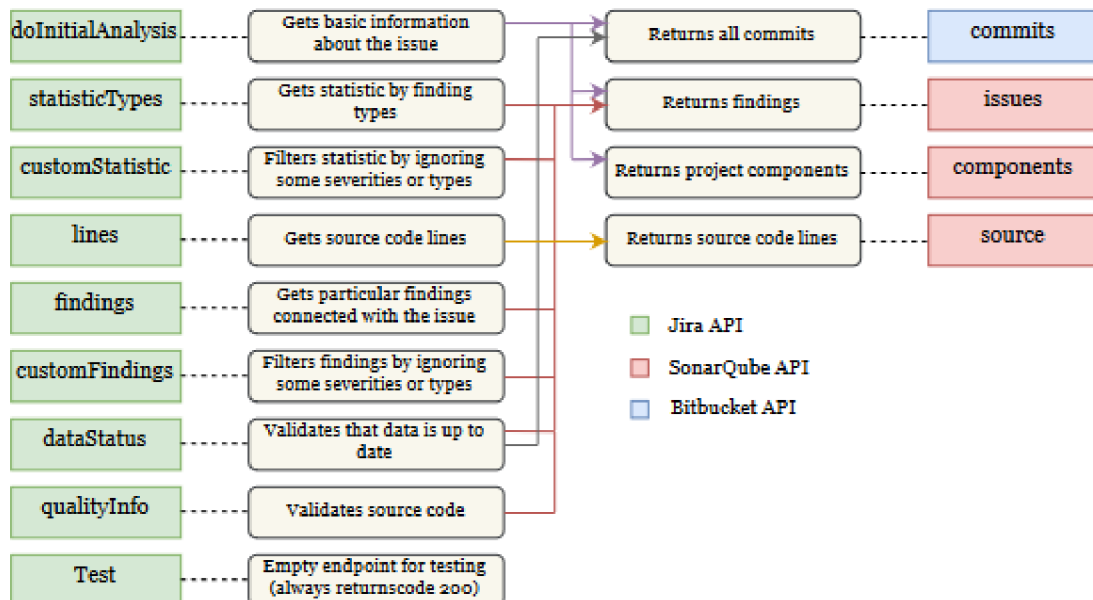

Figure 5.4: Plugin API, Bitbucket API and SonarQube API connection

Data Caching

The Analyzer caches essential data in the Jira instance. Cache is not persistent in database, its content is completely removed any time the plugin is reinstalled or the Jira instance is rebooted. The limit of stored data is set to maximum 500 items. The number of items is controlled by recording the order of items and cache size. The removal algorithm is based on FIFO system. Whenever the cache is full and another issue needs to be stored, the Analyzer takes the oldest record and deletes it.

Cached data is representing SonarQube analyses that were found in connection with particular Jira issue. Analyses are stored because it is an information that is required in every other operation that is performed by the Analyzer. A record is stored under the key of specific Jira issue. Each Jira project member can use a data that another project participant stored by loading the same issue.

Whenever the Analyzer loses cached data for its issue, it invokes completely new issue analysis and creates a new record in the cache. Inconsistent results are prevented by checks before reading from cache and also using locks. A lock is a thread synchronization mechanism [14]. In the case of cache access synchronization, it enables multiple processes to read and write data into cache without any collisions.

Connection of View and Logic

After a user opens SonarQube issue tab panel and successfully authenticates, the loading bar appears and the `doInitialAnalysis` endpoint is called using AJAX. While waiting for response, the Velocity Template that contains Analyzer's elements is parsed. These elements are logically divided according to the endpoints that return data to complete them. After this endpoint returns valid results, `statisticTypes`, `findings`, `dataStatus`, and `qualityInfo` endpoints are called asynchronously using AJAX requests. Figure 5.5 shows these actions in graphical structure. Whenever any of these requests is responded, it parses the JSON response, sets values to its section in template and displays data on the screen. After a data is displayed on the panel, a user is able to operate with them, all sections are independent units.

Table rows that represent findings are parsed using Soy Template. Soy Template helps to render elements repeatably so it is convenient solution for rendering table rows because the Analyzer can display unlimited number of findings, it depends on the number of findings and also how many times a user clicks on the "Show more" button.

5.3.2 Integration Administration

Administration enables providing essential data to make the Analyzer work properly. Administration itself is divided into three sections:

- Configuration project tab panel enables integration activation and also offers possibility to select specific SonarQube projects that are connected to the Jira project,

- SonarQube project tab panel enables issue code quality targets settings,

- Unity and SonarQube configuration items that store links to specific instances of SonarQube and Unity.

Proper settings should provide SonarQube project keys and URL, that are used during analyses, and code quality targets that help to measure issue quality. Permissions are di-
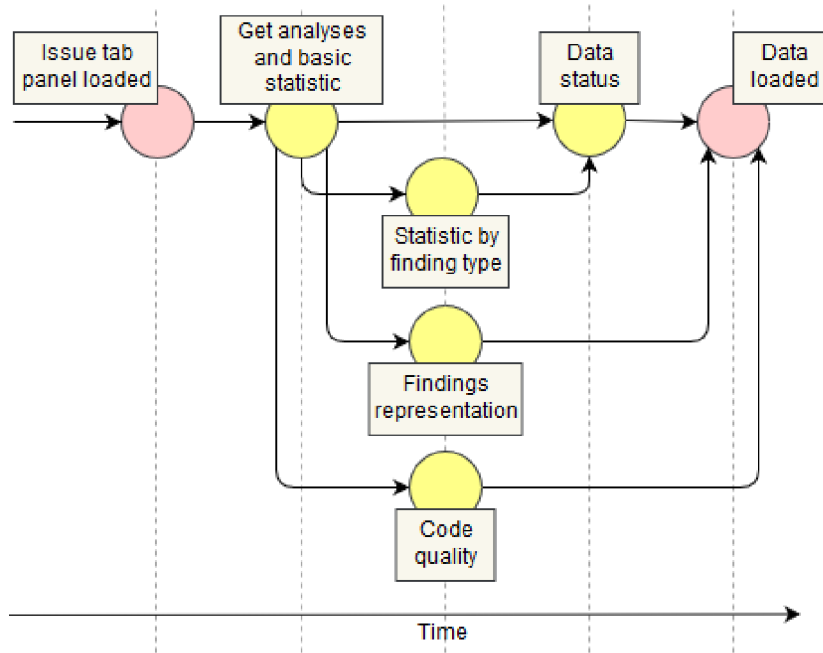
Figure 5.5: Analyzer actions in timeline

vided into Jira administrator, project administrator, and project participant.

Project and quality settings are project specific features, they are separated from the Analyzer, and implemented on the project tab panel. The project tab panel is a module that allows adding new tabs to the main project menu. It is defined using the `<web-item>` and the `<web-panel>` elements in the `atlassian-plugin.xml` file. The `web-item` element (Listing 5.3) is a menu item that contains a link to the appropriate web panel. The `<web-panel>` element (Listing 5.4) represents panel page itself, it is the page that appears when a user clicks on the item in project menu.

```
<web−item section="jira.project.sidebar.plugins.navigation" weight="60"
      key="sonarqube−tab−link">
   <label key="SonarQube"/>
   <link>/projects/{$pathEncodedProjectKey?selectedItem}=
      act.jira.plugin:sonarqube−tab−page</link>
   <condition class="act.jira.condition.ProjectTabPanelCondition"/>
</web−item>
```

Listing 5.3: Menu item definition example

```
<web−panel key="sq−tab−page" location="jira.plugin:sq−tab−page">
   <resource type="velocity" location="templates/tabpanel.vm"/>
   <context−provider class="jira.plugin.tab.SonarContextProvider"/>
   <condition class="jira.plugin.conditions.ProjectTabPanelCondition"/>
</web−panel>
```

Listing 5.4: Project tab panel link definition

The `<condition>` element refers java class that implements `Conditions` interface from Atlassian Web package [2]. The method `shouldDisplayed` returns boolean value that de-

29

cides whether the item should be displayed to a logged in user or not (Figure 5.6). It is used to check the permissions and also whether particular Jira project has attached SonarQube projects. In the case that the Jira project is not connected with any project in Unity or the project leader did not choose any SonarQube project to analyze, the Analyzer and quality configuration sections are fully disabled to avoid plugin inconsistencies.

SonarQube and Unity URL configuration is an instance specific feature. It means that each Jira instance has its own URL settings for all Jira projects. The configuration area is implemented as the `<web-section>` (Listing 5.5) plugin module, referenced by the `<web-item>` (Listing 5.3) in administration menu.

```
<web−section key="sonar_config" location="admin_plugins_menu">
    <label key="SonarQube Integration Configuration"/>
    <condition class="act.jira.condition.AdminCondition"/>
</web−section>
```

Listing 5.5: Administration section
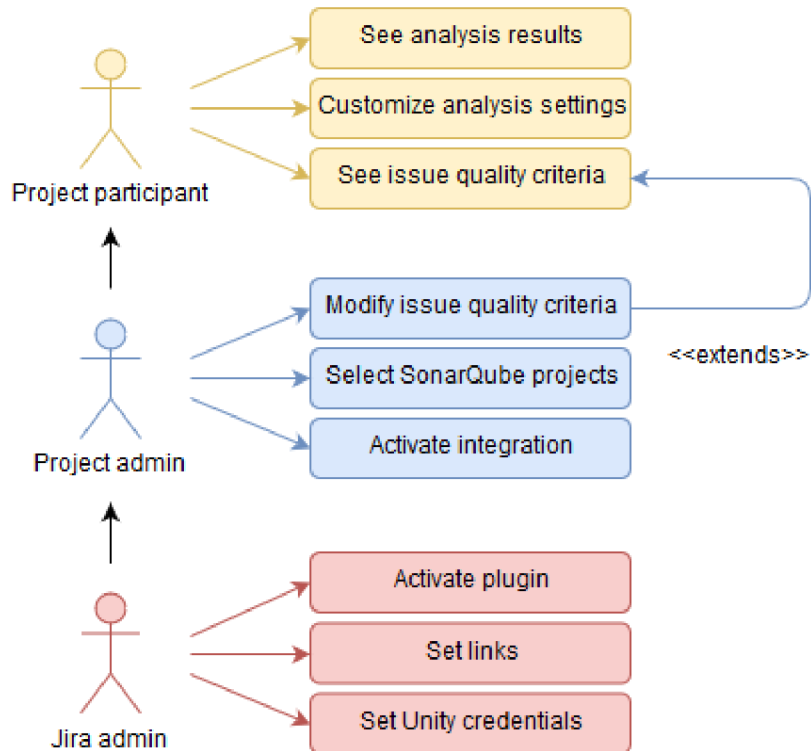
Figure 5.6: Jira permissions division

Configuration Project Tab Panel

Project settings are made in configuration area. The configuration area is implemented as a separate project tab panel that is visible only for project leaders. This behavior is reached using Condition in the *atlassian-plugin.xml* file. This file contains two main sections, one

of them is used to enable or disable SonarQube integration for whole project and the other is used to select SonarQube project.

The new endpoints `getSQProjects`, `getSelectedSQProjects`, and `postSQProjects` were implemented to manage SonarQube projects connected with particular Jira project. The `getSQProjects` endpoint is called immediately after the panel is loaded. It calls the Unity endpoint `forToolProjectIdOrKey` to get all projects in other tools that are connected to the Jira project. The response is parsed using GSON to get SonarQube projects only. These projects are passed to fronted and displayed as items of dropdown element. The `getSelectedSQProjects` endpoint is called to find projects that have been selected before. These projects are represented by Active Objects in data storage. If any projects were already connected with the Jira project, they are returned and these projects are checked in project dropdown element. An administrator can select 1-n of these projects, when s/he clicks on the save button, checked items are passed to `postSQProjects` endpoint. This endpoint removes all project Active Objects connected with the Jira project and creates a new ones according to administrator's selection. Created Active Objects, representing SonarQube projects, are then used during issue analyses.

The endpoint called `changeVisibility` enables and disables SonarQube integration for Jira projects. It keeps records as AOs that are always checked before any panel, which is a part of SonarQube integration plugin, appears on the screen. Default state is "not activated" for all Jira projects.

SonarQube Project Tab Panel

The SonarQube project tab panel enables quality settings. Although this panel is visible for all project participants, only project leader permission allows to modify rules. Participants that do not have required permission can display settings only as a plain text.

Quality rules are displayed as a table. Each row stands for one rule. A project leader can add and delete any of rules. Each rule contains severities and types that should be limited and also a limit of these findings that is accepted. Back-end represents rules using Active Objects. Communication is realized using endpoints `getValues`, that finds required AOs and returns them, and `storeValues`, that replaces AOs with new ones. Plugin automatically generates initial rules when integration is activated for the first time, the rules are limiting acceptable bugs and vulnerabilities of all severities to 0. When the integration is deactivated, Active Object keeps quality data for the project, so it can be retrieved later.

Configuration Items

The plugin provides additional configuration items in the administration area. The integration creates two items in a separate configuration area. One item stands for Unity, and the other for SonarQube. Both items contain fields that are required for integration process. SonarQube uses only field URL and Unity uses fields URL, username, and password. This configuration is set up by a Jira administrator. Fields are stored in database using Active Objects.

Chapter 6

Verification

Verification is a procedure used to check whether a product meets requirements and specifications. The ISO 1233 [5] standard defines verification with the following definition:
Verification is the process of evaluating a system or component to determine whether the system of a given development phase satisfies the conditions imposed at the start of that phase.

6.1 Methods

Plugin was tested and verified using one manual and two automated methods:

- Manual testing,

- unit testing,

- performance testing.

Automated testing methods are running daily using the Atlassian Bamboo server.

Atlassian Bamboo

The Atlassian Bamboo is a server for continuous integration, deployment, and delivery. The Bamboo is used to periodically run builds and tests for software projects, and provide reports. It consists of two parts, the Bamboo server and Bamboo agents. The Bamboo server manages whole infrastructure, its main task is to keep agents working properly, manage the list of agents, and assign tasks based on requirements and agents' capabilities. A Bamboo agent is a service that can run the execution of a job or a plan. There are three types of Bamboo agents: local agent, remote agent, and elastic agent [15].

6.1.1 Unit Tests

Unit testing is a testing method that is focused on the behavior of individual classes and their methods. It should ensure a developer that the method passes its functionality requirements. It also helps to reveal potential bugs caused in reaction to unexpected input.

JUnit

JUnit is a unit testing framework for Java applications. A test class consists of set up that is created using `@BeforeMethod` and `@BeforeClass` annotations, test cases that are marked with `@Test` annotation, and optionally tear down that is marked with `@AfterClass` and `@AfterMethod` annotations. Each test case should be focused on one specific functionality and needs to be fully independent from the other test cases.

JUnit allows to simulate component's behavior to get required results using mocks. A mock is a component annotated with `@Mock` that can force a component, that is not the tested object and has complicated structure, to have specific behavior. A behavior of a mock can be specified using `doReturn({OBJECT}).when({MOCK}).{METHOD}`.

The test result is verified using assertions. JUnit offers many assertion methods that can compare any variables of a primitive type or whole objects.

Plugin Unit Tests

The SonarQube plugin is covered by unit tests on 68% of Java source code lines. This coverage is reached using 132 individual test cases in 15 test classes. Each test class is representing one Java class. Tests are mainly focused on API and managers that are working with active objects, parsing server responses, or controlling the Analyzer.

6.1.2 Performance Tests

Performance testing are methods that help to investigate, measure, and verify quality of a system. It is divided into following five categories:

- **Load testing** is used to monitor a behavior of a system under an expected load.

- **Stress testing** is a method that is used to determine the robustness of the system. It measures the performance under extreme load.

- **Soak testing** helps to detect potential memory leaks and performance degradation.

- **Spike testing** is a method that is done using highly increasing and decreasing number of loads.

- **Configuration testing** is a testing created to determine the effects of configuration. It monitors the performance under different conditions.

Performance tests are implemented to monitor the functionality of individual endpoints. The main goal is to measure how quickly individual API endpoints react to a basic request. The reaction is compared with the time that SonarQube needs for one response. This approach helps to optimize only endpoints that have some possible efficiency leaks.

Methods

Plugin was tested using three tools. One of them uses open source software and the other two are parts of the plugin itself:

- The Apache JMeter is an open source software designed to load test functional behavior and measure application performance. The JMeter is designed for both static

and dynamic testing, its main focus is web applications testing. It can be used for example to simulate a heavy load on a server and monitor a behavior of an application, or to analyze overall performance.

- The Java class that monitors the time interval between outgoing request and incoming response from external applications. This approach helps to determine the approximate time that SonarQube requires to return response, and that it takes to reach SonarQube from Jira and the other way around.

- The JavaScript console application that is started from the browser command prompt on the plugin configuration project tab panel. This script sends a request on each integration API and monitors its response time. It requires two parameters, the first is number of requests that should be sent on each endpoint, the second is the key of the issue that should be measured. The script measures only endpoints with responses that are changeable with issue specifications, not with configuration settings.

Test Implementation

Tests are implemented to monitor three real Jira issues that are taken as reference issues. The reference issues' parameters are displayed in Table 6.1.

Issues were chosen according to Jira project standards estimation and research to prepare testing data that have structure, results, and lifecycle close to typical Jira issues. Non-implementation Jira issue usually does not contain any commits. Small bug fix, represented by the row with key "issue-3", usually contains 1-3 commits. Standard bug fix or small feature, represented by the row with key "issue-2", usually consists of around 1-8 commits. Complex and problematic features, represented by the row with key "issue-1", usually has around 20 commits.

The "Findings" column contains SonarQube unresolved findings of all types and severities that were found by the SonarQube plugin Analyzer in connection with the particular Jira issue. The difference in findings counts should strongly impact custom operations like findings filtering. This entry should help to determine the difference in analysis of none and many findings, what brings the important information about the time spent by Analyzer's operations.

The "Changed files" column refers more Analyzer's operations than a SonarQube endpoint response measurement. Each path of changed file needs to be parsed from a large amount of data that is composed by calling SonarQube endpoint multiple times to get all response pages. Number of these calls is always the same for particular project. The difference is in amount of data that needs to be parsed and later composed in the specific structure to append it to each outgoing request.

The "Created" and "Resolved" columns specify the time interval between Jira issue creation and resolution. This entry is important because the more commits in different days issue has, the more different analyses SonarQube typically does. The SonarQube endpoint that returns analyses is called only in case that the difference between a commit and its analysis is higher than between the commit and the analysis of the previous commit. Otherwise the result from SonarQube would be identical, so analysis would be duplicated, what causes higher time cost of other operations.

Issue key	Commits total	Changed files	Findings	Created	Resolved
issue-1	25	23	57	5/26/2017	8/1/2017
issue-2	8	15	1	8/4/2017	10/8/2017
issue-3	3	2	0	3/7/2017	20/7/2017

Table 6.1: Reference issues' parameters

Test Results

The main goal of these tests is to determine approximate time that it takes to find required data and parse them. Tests should reveal some critical parts and possible improvements. Each Jira issue from Table 6.1 was tested separately and with empty cache to get as exact results as possible for each endpoint defined in Section 5.3.1.

The results are displayed in Figure 6.1. The figure displays comparison of endpoints reaction times when taking account of issue size. Results showed that the biggest difference is in `doInitialAnalysis` endpoint and also other endpoints that calculate statistic. Issues are highly impacted by their magnitude.

On the other hand `findings` and `customFindings` endpoints showed that the number of analyses is not the main problem, despite its technique of getting results is based on the same cycle as `doInitialAnalysis` endpoint, its time cost is only around one fifth of `doInitialAnalysis` endpoint's time cost. The most critical part is high number of requests that needs to be done for each of this analysis and number of analysis is multiplying it.

The `qualityInfo`, `dataStatus`, and `validation` endpoints are reaching required results. Their time cost is not influenced by Jira issue properties, it works stable for every issue.

	do Initial Analysis	quality Info	data Status	custom Findings	findings	statistic Types	custom Statistic	Validation	Test
issue-1	34.324	6.324	2.376	8.6	6.124	15.592	24.1871	6.482	2.724
issue-2	19.564	3.446	1.744	3.312	3.458	7.37	9.752	6.253	1.436
issue-3	15.142	4.762	1.702	2.362	2.22	4.616	5.914	6.274	1.398
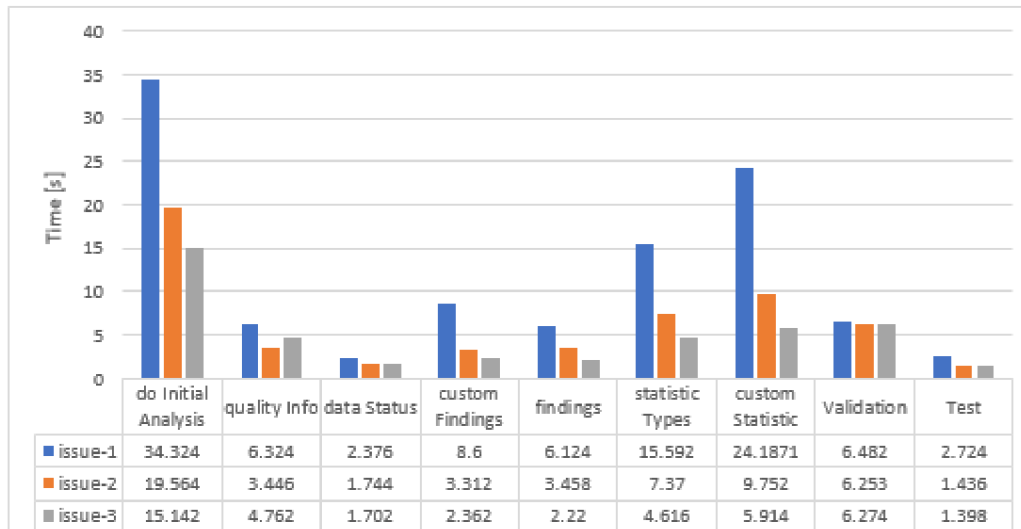
Figure 6.1: Endpoints response times comparison

The smallest reference issue is the **issue-3**. The analysis of this issue showed up that despite the issue does not contain any findings, the Analyzer is still strongly impacted by statistic loading. The other point of view shows that the "findings" endpoint works well for small issues. The actions that are getting statistics can be the initial point for the next

optimization. There might be a build-in system of decision that in a particular part of analysis determines which actions and parts of analysis are essential for an issue with zero findings. This system should be safe and well tested to be sure that it does not cause ignoring any finding.

Average issue size is represented by the **issue-2**. The analysis of this issue showed that the "findings" endpoint, that returns specifications of the issues, works still in acceptable limits. On the other hand, the "doInitialAnalysis" endpoint is perceptibly worse than the previous issue's results.

Large issues are represented by the **issue-1**. This issue has, as expected, the most impacted results by its size. According to the other measurements, the "doInitialAnalysis" endpoint takes only around 9 seconds to do the analysis itself, and then around 20 seconds to get statistics. Similar operations are done by the "customStatistic" endpoint.

The results of analyses were also compared from the point of view of a user. Figure 6.2 shows the results that were calculated to determine total waiting time and time before the first element is loaded.

The total waiting time is a period of time that starts when an authenticated user opens the SonarQube tab panel and ends when the last element is loaded on screen. It takes into consideration asynchronous processes.

The time period before the first element is shown is a period from the start of analysis to the time when the first result is shown on screen (despite the other elements might not be loaded yet).
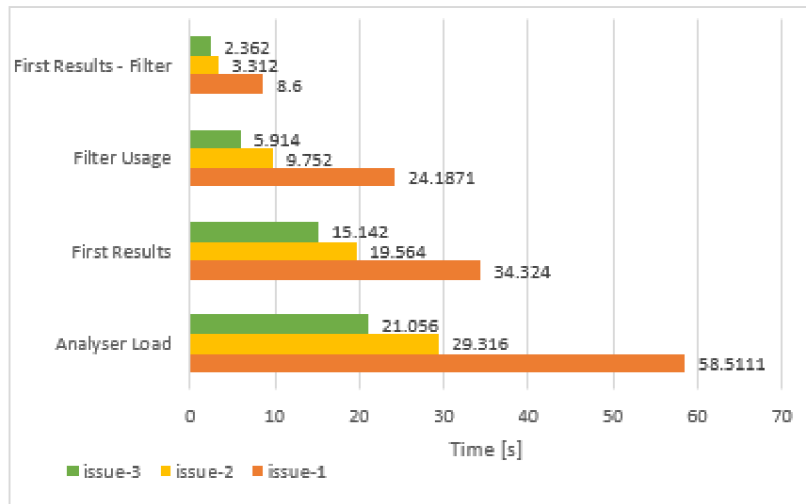
Figure 6.2: Comparison from the point of view of a user

Manual Testing

The manual testing method was selected due to instance restrictions that currently do not allow involving integration or behavior-driven testing. Manual tests were created to cover both configuration sections and the Analyzer with tests that verify plugin functionality. The manual testing plan consists of 15 test cases that are focused on specific functionalities using different project roles with distinct permissions, and under variety of conditions. Manual tests are divided into two sections.

The first part is focused on configuration testing. It consists of 7 test cases that verify

36

whether a configuration change appropriately impacts the plugin behavior according exceptions. These tests are also focused on invalid plugin settings and how plugin reacts to these conditions.

The second section tests the interface of the Analyzer itself. It includes 8 individual tests that verify the reaction of the Analyzer on user's input and requirements.

The test results showed following bugs and inconsistencies:

- A Jira project without any SonarQube project attached in Unity does not show the message that informs project leader in configuration area. The problem was fixed, despite the data was loaded properly, the message was kept hidden.

- The issue tab panel reacts inappropriately to invalid settings. The problem is caused because of the API of SonarQube, it requires credentials to get its status, so the problem is to determine how to validate SonarQube status without credentials, or how to validate credentials without knowing the status. The problem is solved by the authentication endpoint, it validates credentials only when server is running properly so the valid response is also the code 401 (not authenticated).

- When quality requirements are changed, and an issue is currently loaded, quality results do not refresh when a user clicks on the "Reload" button. The problem was created by missing invoker when an issue is refreshing.

Conclusion

The plugin was tested by three types of testing. Tests that are focused on general functionality did not reveal any serious bugs. On the other hand, the performance tests showed that there are some leaks of efficiency that should be taken into consideration in optimization. The optimization task is to improve general logic that the Analyzer uses to get issue statistics. Current solution requires many REST calls that can be potentially replaced with different logic. This problem is divided into following sub-tasks:

- The most of Jira issues are not connected with any SonarQube finding and some Jira issues, usually testing tasks, do not even have any commits. The Analyzer should be able to determine whether it is worth to do all actions despite the issue has not any findings or commits.

- It takes a long time before the Analyzer returns the data from initial loading processes. This data brings essential information that other REST calls also need. The Analyzer's part that does the initial loading of analyses should be divided into two separate sections. It is required to first bring only data that is important for all later actions.

- The Analyzer actually searches for finding statistics using direct requests, that are also returning very specific data. The task is to inspect the other possible ways of getting issue statistics. There might be a way that does not return results that are not as specific as from the previous solution, but creating own statistic can work much more effectively itself.

Chapter 7

Optimization and Future Work

7.1 Latest Versions of SonarQube

The SonarQube version 6.3 offers features that are enhancement for the SonarQube and Jira integration. One of the problems that complicates Analyzer's functionality is that SonarQube 5.6 does not have any endpoint that directly returns analyses. The current solution had to use a way that is not beneficial for the plugin's efficiency.

The update to version 6.3 offers endpoint that directly returns all SonarQube analyses. It is possible to save noticeable amount of whole Analyzer's time cost only using this endpoint instead of the current solution. Request structure is defined as following:

GET api/project_analyses/search

This endpoint responses the time of the analysis and also the results. The most essential property is that it requires only one or few calls for whole issue. Another advantage is the analysis results because it enables displaying code quality results directly from SonarQube.

Branching is a new feature of SonarQube version 6.6. Standard name convention of naming branches according to Jira issues enables removing Bitbucket integration from this plugin. SonarQube offers endpoint that can return all branches for a single project. The request structure is defined:

GET api/project_branches/list

The response contains list of branches that are connected with the SonarQube project. Using the name conventions properly, there is a possibility to filter a required branch to get essential data. This endpoint might enable reaching issues directly according to SonarQube branch. Unfortunately, even the newest version 6.7 does not provide important information that could fully replace Bitbucket and possibly also some SonarQube endpoints.

SonarQube version 6.0 offers an extension to the authentication endpoint. It is possible to login and logout user through the endpoint. This endpoint could solve the basic authentication security problem because it removes the need of sending credentials in every request. The structure is defined:

POST api/authentication/login
POST api/authentication/logout

All these enhancements should be taken to the consideration in the future work because it can improve performance and also the results.

7.2 Optimization Based on Test Results

The performance test results showed that the weak part of the Analyzer is getting statistical data. Because of these results, optimization was focused mainly on the solution that could help to bring the essential data in the same quality of results but more effectively.

At first, the `doInitialAnalysis` endpoint that gets information about the issue and initial statistic was divided into two separated endpoints. The reason is that having it in a single endpoint slows down other processes because they start after the initial loading finishes. Because of this reason, the initial loading should be as effective and small as possible. The second part of the previous `doInitialAnalysis` endpoint brings a part of statistical data, so it was merged with the `statisticType` endpoint that returns statistical information findings by their type. These two endpoints work with the same data so there is no need to create redundant calls. The new endpoint, created by merging a part of `doInitialAnalysis` and `statisticType` endpoint, is called `statistic`.

The endpoint called `statistic` obtains essential data for both statistic by severities and types at the same time using a new logic that is based on getting large amount of data in a few calls instead of getting small data in many calls. The plugin was reaching statistical information by creating a request for each severity and type of findings, this solution was refactored to create a request for all findings and create own statistic by parsing their parameters locally. This approach allows returning up to 500 findings for one REST call. The Analyzer is able to store up to 75% of a time in total by collecting the data by itself. The Figure 7.1 shows the results of the optimization. Test data was collected using issues from the Table 6.1.

	do Initial Analysis	quality Info	data Status	custom Findings	findings	statistic	custom Statistic	Validation	Test
issue-1	11.174	6.072	2.486	5.722	5.938	5.776	6.634	6.025	1.674
issue-2	8.11	3.214	1.742	3.018	3.164	3.088	3.044	6.57	1.513
issue-3	7.025	2.058	2.018	2.122	3.392	2.562	2.216	6.142	2.114
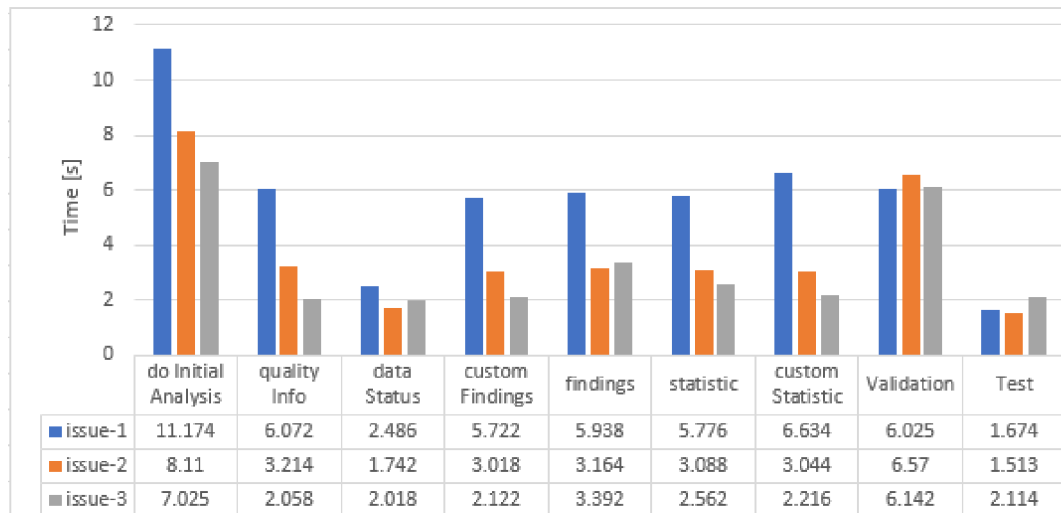
Figure 7.1: Optimization results

Improvement

The results showed that differences between small and large issues got smaller. The time cost is now dependent more on the number of analyses and also on the number of findings but is able to work much more effectively. Dividing the `doInitialAnalysis` endpoint, that gets analysis and basic statistics from SonarQube, into two separated endpoints `doInitialAnalysis` and `statistic` grants big benefit for other parts of the Analyzer that

are able to work asynchronously. The `statistic` endpoint now collects complete statistics, not only its part "by finding type" as before.

The reference issues were analyzed again and the results were compared with the previous statistics. Figure 7.2 displays loading time of the Analyzer in global. Comparison with Figure 6.2 showed that time that is needed to load complete analysis of particular issue was reduced to one third (Figure 7.3). The time period between the time when the first results are shown and when whole analysis is displayed is noticeably shorter, all Analyzer's parts load within a few seconds. Another improvement is filter usage. Despite "First result" appears in almost same time as before, the time period before filtering is completely done was reduced to one fourth.
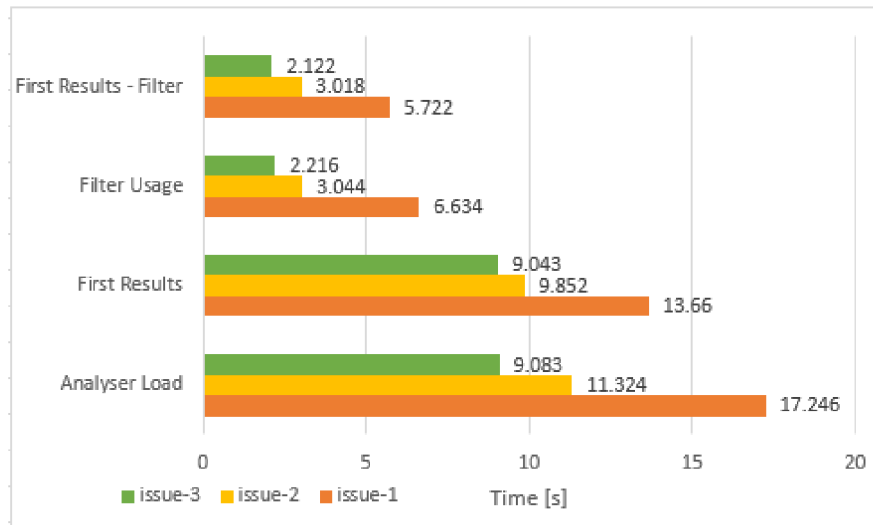
Figure 7.2: Statistic from the point of view of a user
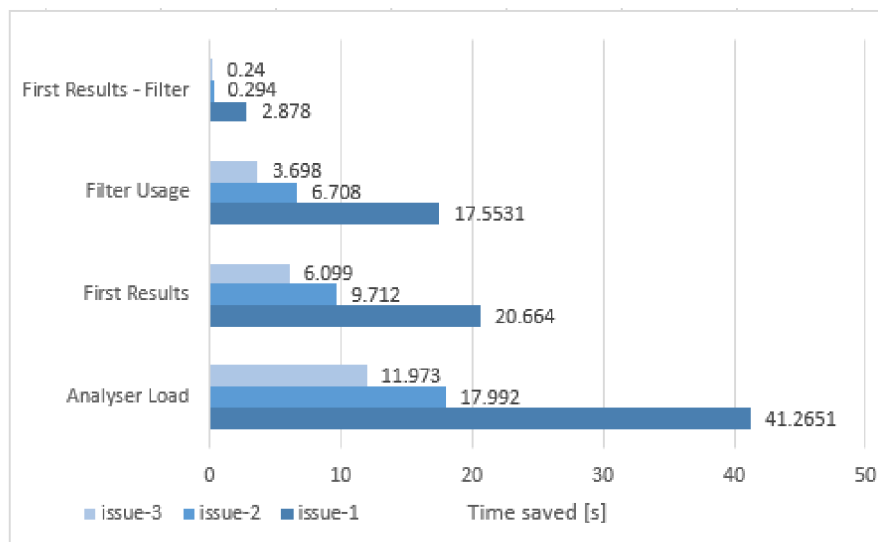
Figure 7.3: Time saved using optimization

7.3 Optimization of Finding Operations

This type of optimization is mainly focused on data loading invoked by a user. It includes for example issue filtering and changing result view. All actions, including non-trivial operations with findings, are processed in back-end. Although this approach produces correct results in acceptable time, it is not sufficient solution for showing many findings and doing operations with them. All operations are impacted by number of findings, which are loaded, so the general effectivity decreases. Because of this problem, the goal of this optimization is to discuss finding loading possibilities and rework current solution to improve usability of the Analyzer interface.

The problem might be solved by loading group of findings at the beginning and keeping them to be able to access data directly from front-end without using API and data caching in back-end. It is important to ensure that this approach does not negatively impact the UI, so it should keep as small amount of data as possible. Estimated appropriate data size is one SonarQube response, so up to 500 findings in total. The UI operations and filters are reworked to manipulate only with cached data, without using plugin API. This approach is able to load 500 findings in several seconds and then to operate with them with negligible time cost. The efficiency of operations with issues that contain less than 500 findings is highly increased.

This optimization creates a new problem that needs to be solved. Issues with more than 500 findings always show the most serious findings, but there also needs to be a possibility of showing less important findings of severity minor or info. Partial solution is comparing number of loaded findings with the number in statistic. This approach works well when searching for findings of specific severity or type. When a user filters issue property, the Analyzer looks into cache and searches for all appropriate findings and compares the number of results with the number of findings in statistic. If the counts match, results can be displayed, otherwise the Analyzer calls plugin API to gain required missing data (Figure 7.4). More complicated situation is when a user searches for a combination of severities and types. Its harder to decide whether the results are complete, so the Analyzer needs to calculate a prediction based on statistic and verify result completeness.
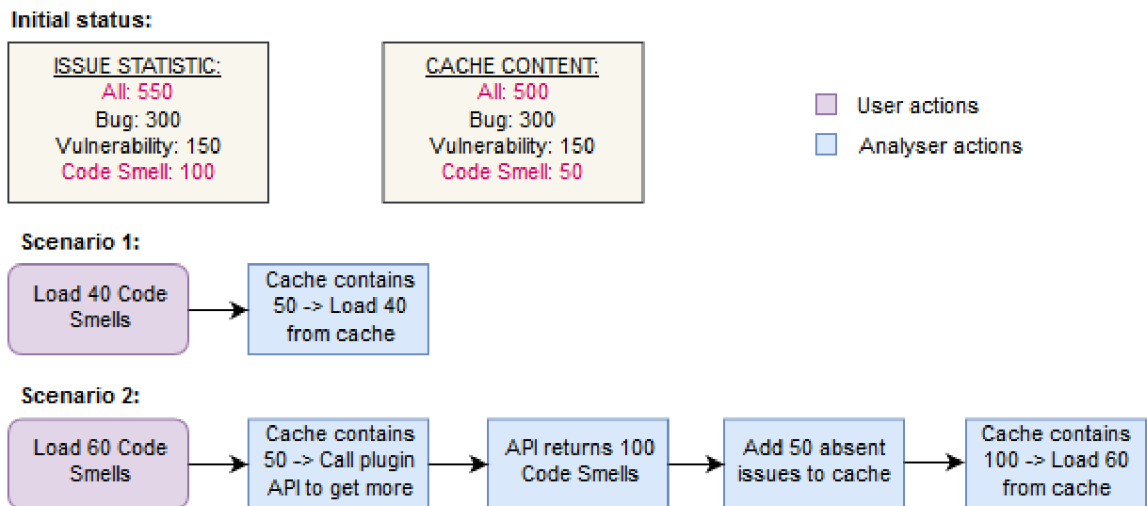
Figure 7.4: Analyzer's issue loading actions

Improvement

This optimization helped to minimize time cost of result filtering. Compared to the statistic after implementation (Figure 6.2) and also the optimized version (Figure 7.3), this optimization helped to make whole interface more appropriate for common usage. The time cost of managing results was reduced to negligible value. Figure 7.5 shows saved time after the filtering optimization, chart shows plugin statistic with both optimizations, including optimization of statistic calculation from previous section, compared to the first version of plugin.

Before the optimization, it took up to 10 second to filter small issues, filtering of complex issues could last up to 25 seconds. Optimization based on test results decreased the duration to 2 seconds for small issues and around 6 seconds for large issues. Despite this time cost is acceptable, caching of findings reduced the time cost to less than 1 second for all issues with less than 500 findings.
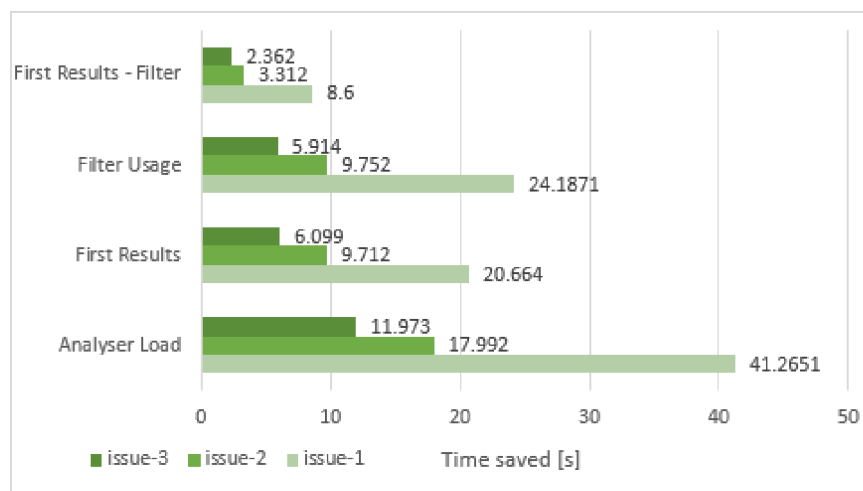
Figure 7.5: Improvement from the point of view of a user

7.4 Future Work

There are many possibilities of plugin improvement and expanding its functionality. This section is focused on the concept of a new feature that enlarges plugin utilization integrating function that provides overview about Jira project version. The feature analyzes all issues that are connected with particular project version, visualizes the results and provides detailed statistic.

The Version Analyzer uses Issue Analyzer, that was described in previous chapters. At first it loads all versions that are connected with specific Jira project. When a user selects a version and starts analysis, the Version Analyzer finds all Jira issues that are connected with required version. Because the version is usually associated with many issues, it is convenient to use threads in order to analyze as many issues at the same time as possible. The result is calculated using shared cache that collects results from threads that analyzed issues.

The process of version analysis can be slowed because of many threads that are running at the same time. The negative impact of slow calculation can be solved using contin-

uous result delivery. While issues are analyzed, the Version Analyzer periodically sends request to read the cache content. It returns all results of threads that finished their job. This approach enables visualization of the results before all issue analyses are completed. The Version Analyzer's structure is displayed in (Figure 7.6).
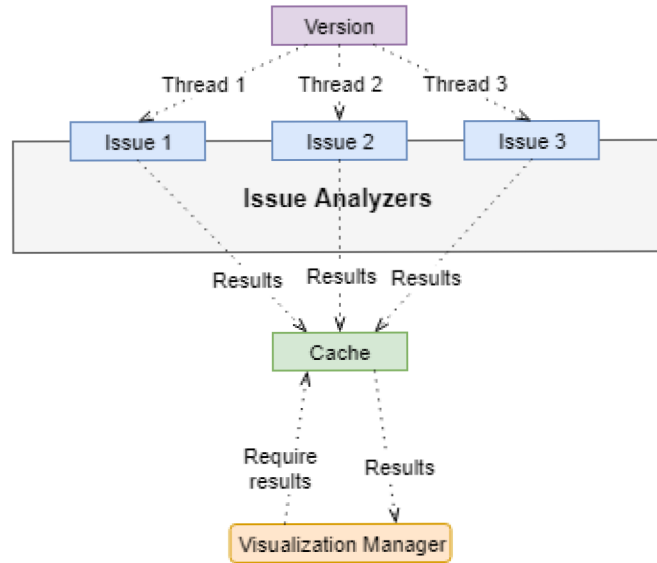
Figure 7.6: Version analysis structure

Chapter 8

Conclusion

The SonarQube plugin was implemented on the issue tab panel that is shown on the view issue screen. Because of a need to configure some properties, there were also implemented two project tab panels and one configuration section.

The main part of the plugin is the Analyzer that allows a user to check the results of the source code analysis, which are connected with the specific issue, directly in Jira. The Analyzer consists of three main parts: quality summary, issue statistic, and particular findings. The Analyzer provides all data that are considered as essential to fix the finding. Whole user interface is interactive, it contains tool tips to navigate user through the interface options.

A project leader can modify project settings and also set up quality requirements that are used to evaluate general issue quality. Plugin contains specific sections for this purpose. The settings are project specific, so they are applied to all issues from a particular project.

The plugin was tested with three types of software testing: unit tests, performance tests, and manual tests. Tests revealed some minor bugs, that are connected with the reaction on unexpected error status of tools, and showed efficiency leaks, which are caused by one specific part of the Analyzer. The bugs were fixed to meet the requirements of appropriate reaction to error states. The efficiency leaks were discussed and the solution was proposed.

The new solution completely changes the approach to statistic loading and filtering of findings. These changes did perceptible step to improve general efficiency of the Analyzer. The loading time was noticeably reduced and operations with analysis results became more user-friendly. In all cases it appears to be more profitable to do more operations locally with large data and use cache instead of creating many REST calls to get required data directly. Although the solution returns stisfactory results in acceptable time, there are several aspects that are important to keep the Analyzer running properly:

- Having specific issues,

- having periodical analyses,

- using conventions in commit messages.

Bibliography

[1] Atlassian: Active Objects. December 2017. [Online; visited 19.3.2018].
Retrieved from: https:
//developer.atlassian.com/server/framework/atlassian-sdk/active-objects/

[2] Atlassian: Conditions. August 2017. [Online; visited 28.3.2018].
Retrieved from:
https://developer.atlassian.com/cloud/jira/platform/conditions/

[3] Bitbucket: REST Resources Provided By: Bitbucket Server. [Online; visited
3.4.2018].
Retrieved from: https:
//docs.atlassian.com/bitbucket-server/rest/4.5.1/bitbucket-rest.html

[4] Chess, B.; West, J.: *Secure Programming with Static Analysis*. Addison-Wesley
Professional. June 2007. ISBN 0-321-42477-8.

[5] IEEE: Guide for Developing System Requirements Specifications. ISO 1233. 1998.

[6] IEEE: Systems and software engineering – Systems and software Quality
Requirements and Evaluation (SQuaRE) – System and software quality models. ISO
25010. 2011.

[7] IEEE: Data elements and interchange formats — Information interchange —
Representation of dates and times — Part 1: Basic rules. ISO 8601-1. 2016.

[8] Janák, J.: *Issue Tracking Systems*. Masaryk University Brno. 2009.

[9] Jira: Using AppLinks to link to other applications. [Online; visited 26.10.2017].
Retrieved from: https://confluence.atlassian.com/adminjiraserver071/using-
applinks-to-link-to-other-applications-802592232.html

[10] Jira: What is a Project. [Online; visited 26.10.2017].
Retrieved from: https:
//confluence.atlassian.com/jira064/what-is-a-project-720416135.html

[11] Jira: What is an Issue. [Online; visited 26.10.2017].
Retrieved from:
https://confluence.atlassian.com/jira064/what-is-an-issue-720416138.html

[12] Kohler, S.: *Atlassian Confluence 5 Essentials*. Packt Publishing. June 2013. ISBN
978-1849689526.

[13] Meyer, D.: 8 steps to a definition of done in Jira. October 2013. [Online; visited 26.10.2017].
Retrieved from: https://www.atlassian.com/blog/jira-software/8-steps-to-a-definition-of-done-in-jira

[14] Oracle: Interface Lock. [Online; visited 28.3.2018].
Retrieved from: https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/locks/Lock.html

[15] Peťovský, S.: *Continuous Integration and Code Quality Enhancements*. Masaryk University Brno. 2015.

[16] SonarQube: Code Quality. [Online; visited 17.12.2017].
Retrieved from: https://www.sonarsource.com/why-us/code-quality/

[17] SonarQube: Web API. [Online; visited 3.4.2018].
Retrieved from: https://next.sonarqube.com/sonarqube/web_api

[18] SonarQube: Issue Lifecycle. April 2016. [Online; visited 25.10.2017].
Retrieved from: https://docs.sonarqube.org/display/SONAR/Issue+Lifecycle

[19] SonarQube: Metric Definitions. January 2018. [Online; visited 3.4.2018].
Retrieved from: https://docs.sonarqube.org/display/SONAR/Metric+Definitions

[20] SonarQube: Quality Gates. January 2018. [Online; visited 3.4.2018].
Retrieved from: https://docs.sonarqube.org/display/SONAR/Quality+Gates

[21] SonarQube: Quality Profiles. March 2018. [Online; visited 3.4.2018].
Retrieved from: https://docs.sonarqube.org/display/SONAR/Quality+Profiles

[22] Williams, R.: *The Non-Designer's Type Book*. Peachpit Press. second edition. October 2005. ISBN 0-321-19385-7.

Appendix A

Mockups

Figure A.2 shows complete mockup of the first plugin version that was defined in Section 4.2.2. This mockup has separated messages for quality results and data status. Filters are also separated from the other parts of the panel. This structure was considered to be too complicated and unconnected, what causes negative impression. Figure A.1 displays latest mockup that was designed. It differs in many aspects, for example messages are represented by labels, filters are completely hidden and replaced by the configuration icon, and findings contain direct link that is also reachable by clicking on the link icon.

Figure A.1: Final mockup of the plugin

Figure A.2: Complete mockup of application design

Appendix B

Final Application

The final plugin (Figure B.1) was designed according to proposed and approved Mockup A.1. Issue code quality details are showed in Figure B.2. They are calculated according to the project rules. The label color, that is actually red, is changeable according to the results. Data status (Figure B.3) is represented in similar way as the quality results. Both messages appear on hover over the colored label.
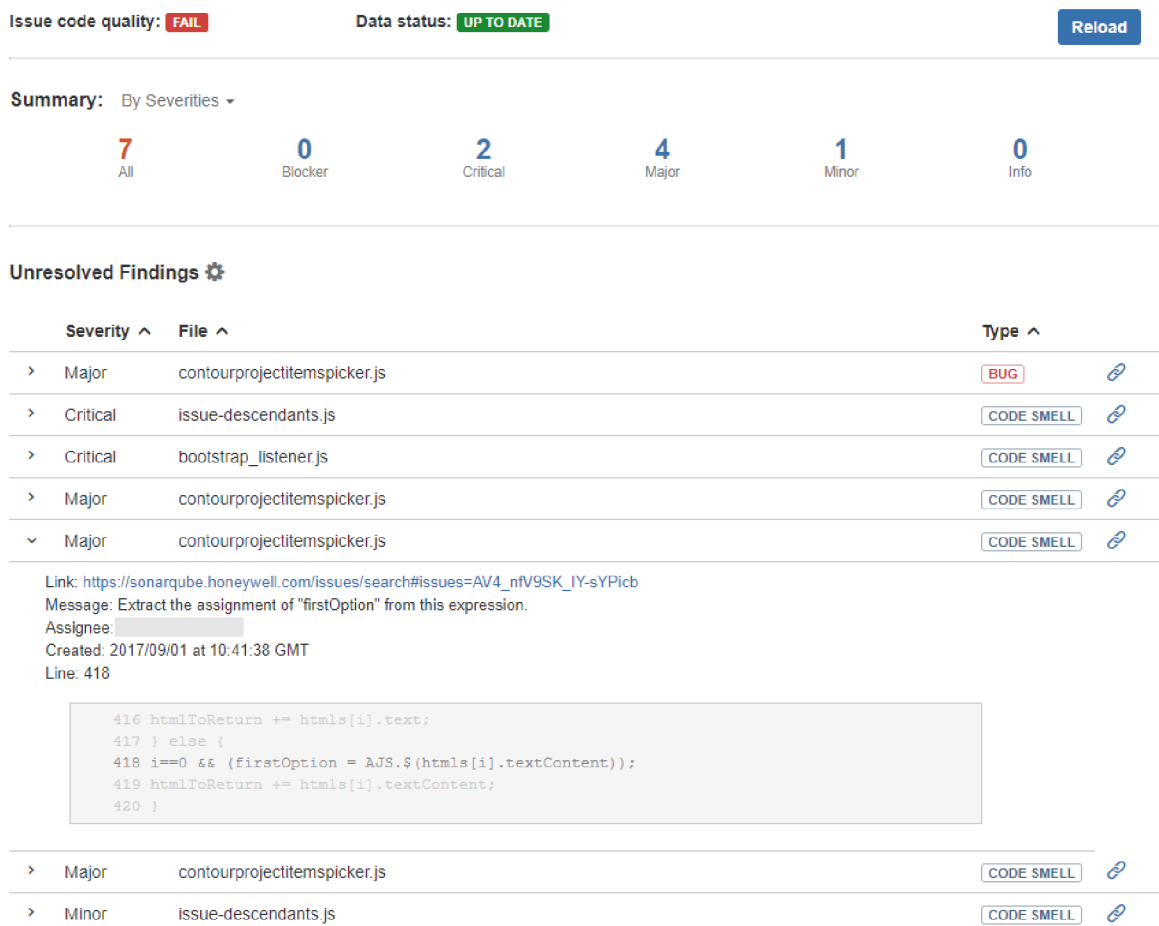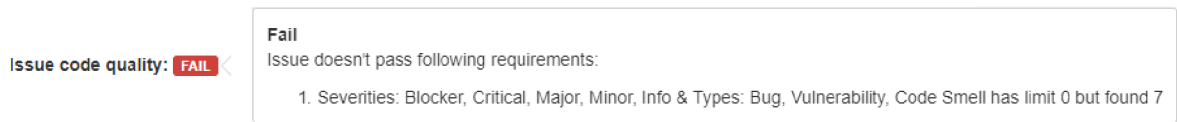

Figure B.1: Final version of the plugin
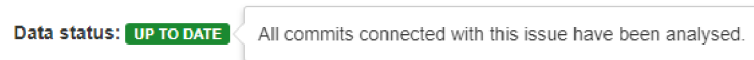
Figure B.2: Quality details.

Figure B.3: Data status details

Finding filters are completely hidden as default. Whole section was replaced with the configuration icon. When a user clicks on this icon, it shows filter options (Figure B.4). Selection activates the new icon that removes selection and displays all findings again.
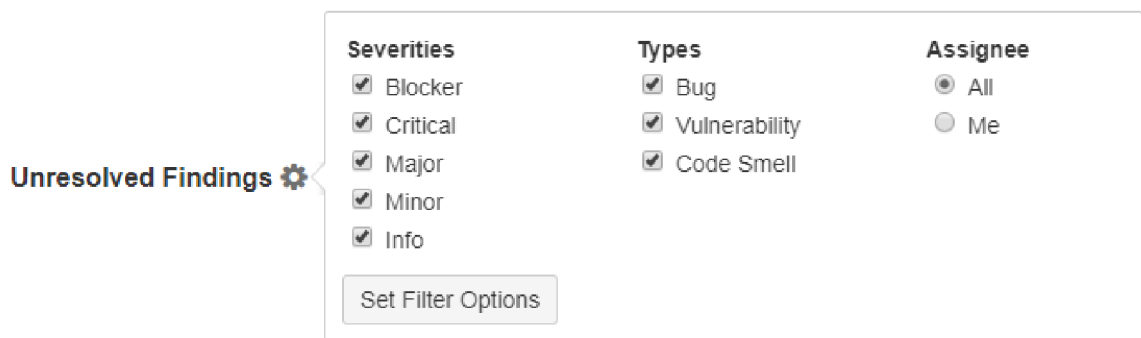

Figure B.4: Filtering options

The section that contains summary by severities or type is set to "Severities" as default. Switching the view to "Types" displays statistic that is shown in Figure B.5.
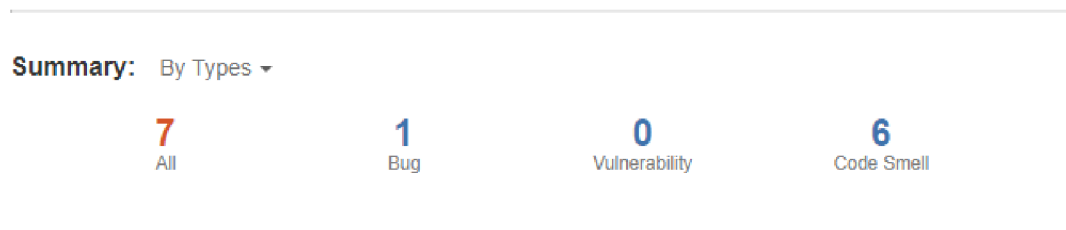

Figure B.5: Statistic by types of findings

Appendix C

Installation

This appendix section contains instructions for the plugin installation.

Prerequisites

The SonarQube plugin requires following tools to be able to do analyses appropriately:

- Jira version 7.3,

- SonarQube version 5.6 or higher,

- Unity,

- Bitbucket version 4.14 or higher,

- SonarQube integration plugin JAR file.

Installation Steps

- Go to the Jira in the browser.

- Log in as a Jira administrator.

- Navigate to the "Manage add-ons" section.

- Click on the "Upload add-on" link.

- Click on the "Choose File" button, select the SonarQube plugin JAR file, and upload the file.

Appendix D

CD Content

The attached CD contains following content:

- The text of this thesis in PDF `BT_xjanco06`,

- Compiled plugin as the JAR file `sonarqube-integration-plugin`,

- `README` file,

- `plugin` folder contains plugin implementation and unit tests,

- `reports` folder contains test reports,

- `doc` folder contains source code documentation,

- `text` folder contains LaTeX source files.