TUM

TECHNISCHE UNIVERSITÄT MÜNCHEN
INSTITUT FÜR INFORMATIK

A Comparison between Commercial and Open Source Reference Implementations for the Rugby Process Model

Sajjad Taheritanjani, Stephan Krusche, Bernd Bruegge

TUM-I1630

Technischer Bericht
Technische Universität München
Institut für Informatik

A Comparison between Commercial and Open Source Reference Implementations for the Rugby Process Model

Sajjad Taheritanjani
TU München
Munich, Germany
sajjad.taheri@tum.de

Stephan Krusche
TU München
Munich, Germany
krusche@in.tum.de

Bernd Brügge
TU München
Munich, Germany
bruegge@in.tum.de

***Abstract*: Rugby is a process model for continuous software engineering which allows developers to continuously deliver prototypes and obtain feedback supporting software evolution. There is a reference implementation of Rugby with commercial enterprise tools used in university capstone courses. However, since these tools are expensive, there is a need to study less expensive alternatives which are available on the market to evaluate whether they can be used in Rugby. In this research, we compare a second reference implementation with the existing one, focusing on the core use cases and non-functional requirements of Rugby.**

key words: **Agile, Scrum, Rugby, Coninuous Software Engineering, Issue Tracking, Version Control System, Continuous Integration, Continuous Delivery**

1 INTRODUCTION

Increasingly dynamic environments lead to shorter development cycles [FS14]. Continuous software engineering (CSE) refers to the organizational capability to develop, release, and learn from software in rapid cycles [Bo14] providing the capability for software evolution [HF10]. Rugby is a process models that proposes a development lifecycle for CSE [KAB+14]. It combines elements of Scrum [Sc04] and the Unified Process [JBR+99] with additional workflows to support CSE and software evolution: review management, release management and feedback management [KAB+14]. Developers work in a project-based organization with multiple projects to deliver executable prototypes and to obtain feedback. Bruegge et al. developed a reference implementation of Rugby that is applied in university capstone courses using commercial enterprise tools: JIRA, Bitbucket Server, Bamboo and HockeyApp [BKA15].

However, Rugby is not limited to these commercial tools. Expensive tools can not easily be used by individuals and organizations, e.g. open source projects or young startups. The existing reference implementation also poses challenges, e.g. in terms of usability, because the tools have a high learning curve. Therefore, we investigate Rugby's use cases for its three additional workflows and one Scrum core workflows: issue management. Considering the tight collaboration between these categories in Rugby, we decided to choose the GitHub Issues as issue tracker, GitHub as version control system (VCS) and Travis as continuous integration (CI) server, all from GitHub to ensure their efficient integration. We also chose Jenkins as CI server, because it is open source and used in many projects. Both, GitHub and Travis are cloud-based platforms, that can be used for free in open source projects. Since there is no open source alternative for continuous delivery (CD), we chose Crashlytics, which made all its services free after the acquisition by Twitter. As knowledge management using a Wiki is not included in Rugby's continuous workflow (see section 2), we decided to exclude it from our comparison list.

In this research, we compare the tools in each category with respect to the most important Rugby use cases for developers, managers, users and also with regards to configurability and flexibility. At the end, using GitHub tools, Jenkins and Crashlytics, which all target on the open source projects, we create a second reference implementation to compare and evaluate the differences of how the main use cases are implemented. We also give recommendations which reference implementation can be used.

2 BACKGROUND

2.1 AGILE METHODOLOGIES

Agile software development refers to some of software methodologies which are based on iterative development, where requirements and solutions evolve through collaboration between self-organizing cross-

functional teams. Agile methods or Agile processes generally promote a disciplined project management process that encourages frequent inspection and adaptation, a leadership philosophy that encourages teamwork, self-organization and accountability, a set of engineering best practices intended to allow for rapid delivery of high-quality software, and a business approach that aligns development wit customer needs and company goals. Agile development refers to any development process that is aligned with the concepts of the Agile Manifesto. The Manifesto was developed by a group fourteen leading figures in the software industry, and reflects their experience of what approaches do and do not work for software development.

2.2 SCRUM

Scrum is an Agile method. It is a lightweight process framework for agile development, and the most widely used one. A "process framework" is a particular set of practices that must be followed in order for a process to be consistent with the framework. For example, the Scrum process framework requires the use of development cycles called Sprints. In addition, "Lightweight" simply means that the overhead of the process is kept as small as possible, to maximize the amount of productive time available for getting useful work done [Sc04].

A Scrum process is distinguished from other agile processes by specific concepts and practices, divided into the three categories of Roles, Artifacts, and Time Boxes.

Scrum is most often used to manage complex software and product development, using iterative and incremental practices. It significantly increases productivity and reduces time to benefits relative to classic "waterfall" processes. Scrum processes enable organizations to adjust smoothly to rapidly changing requirements, and produce a product that meets evolving business goals.

An agile Scrum process benefits the organization by helping it to increase the quality of the deliverables, cope better with change (and expect the changes), provide better estimates while spending less time creating them, and be more in control of the project schedule and state.

2.3 CONTINUOUS SOFTWARE ENGINEERING

Continuous software engineering refers to the organizational capability to develop, release and learn from software in very short rapid cycles, typically hours, days or very small numbers of weeks [FS14]. This requires not only agile processes in teams but in the complete research and development organization. Additionally, the technology used in the different development phases, like requirements engineering and system integration, must support the quick development cycles. This includes determining new functionality to build, prioritizing the most important functionality, evolving and refactoring the architecture, developing the functionality, validating it, releasing it to customers and collecting experimental feedback from the customers to inform the next cycle of development. Finally, automatic live experimentation for different system alternatives enables fast gathering of required data for decision-making [FS14].

2.4 Rugby

Rugby is a process model that includes workflows for CD. It allows part timers to work in a project-based organization with multiple projects for the rapid delivery of prototypes and products. Using CD improves the development process in two ways: First, Rugby improves the interaction between developers and customers with a continuous feedback mechanism. Second, Rugby improves the coordination and communication with stakeholders and across multiple teams in project-based organizations with event based releases [KAB+14].

2.4.1 Rugby Environment

Rugby is designed to be used in project-based organizations with multiple projects. A typical project team in Rugby consists of up to eight developers, a team leader and a project leader. The project team is self-organizing, cross-functional and therefore responsible for all aspects of development and delivery of software.

The project leader and the team leader fulfill a role similar to a scrum master while being in a master-apprentice relationship. While the project manager is already experienced, the team leader is an experienced developer. Thus, he is familiar with the infrastructure and the organizational aspects of Rugby. One task of the team leader is to organize the first team meeting and to ensure that the team organizes all following team meetings in a structured way. In the first team meeting, he takes the role of the primary facilitator and introduces the other two important roles in a meeting, the minute taker and the timekeeper. In the following meetings, these roles rotate between the developers so that they also take responsibility in the meeting organization. The job of the team leader is then to make sure, that the developers organize the team meetings appropriately. If e.g. the timekeeper does not interrupt, if the team members discuss too long on an unimportant point of the agenda, the team leader need to interfere and remind the timekeeper about his job. During the project, the team

leaders learn essential management skills by observing the behavior and actions taken by the project leader. Another important task of the team leader is problem solving and the communication of problems to the project leader and the program management. The customer has a similar role as the product owner. Typically, there are different types of customers in software engineering projects. If the customer of a project does not have enough knowledge in the application domain or is not able to make decisions, the project leader helps him. In addition, if the customer is not available due to time reasons or a large physical distance, the project leader takes the role of a proxy customer [KAB+14].

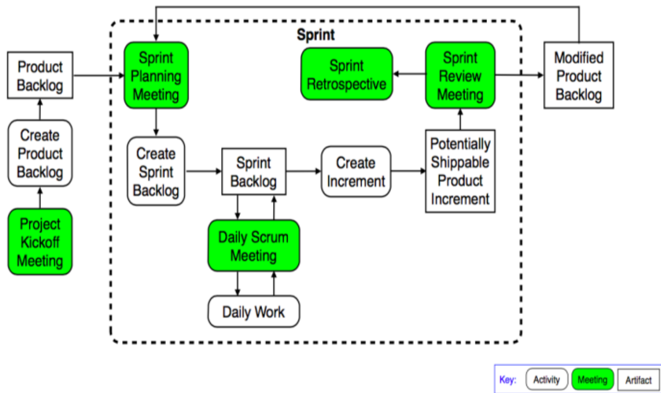

Figure 1: Sprint in Scrum

The eco-system of Rugby is divided into five environments. A developer interacts with the collaboration, development, integration and delivery environment, a user interacts with the collaboration, delivery and target environment. The focus in Rugby is particularly on the collaboration and delivery environments because they bridge the communication gap between developers and users. A user is notified from the delivery environment if a new release is available and can then use the software in his target environment. Feedback of the user is stored in the delivery environment and then forwarded into the collaboration environment, e.g. as feature request. A user can also vote certain features in the collaboration environment.

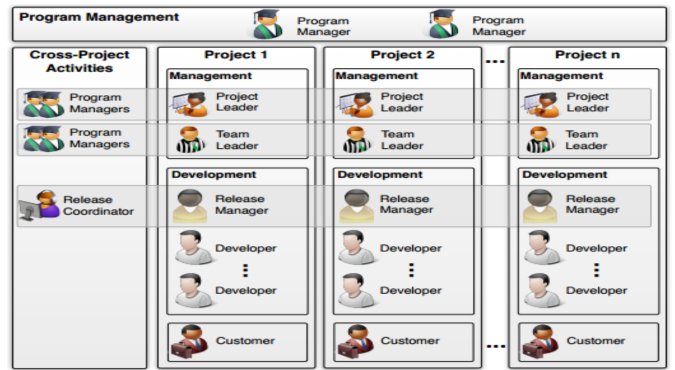

Figure 2: Project-based Organization of Rugby

The figure shows the project-based organization of Rugby. Each development team is represented as a vertical bar, e.g. Project 1. Additionally, multiple cross-project teams are formed in Rugby to further support certain expertise in the development teams. One of these teams is led by the release coordinator, who is responsible for release and feedback management of all projects. Release management includes all activities concerning version control, continuous integration and continuous delivery. The release management team is shown as horizontal box in fig. 3 and consists of one team member of each development team, the release manager [KAB+14].

Cross-project teams meet weekly or biweekly to build up and share knowledge, to synchronize their understanding about tools and workflows and to resolve potential issues. While the cross-project teams are the main resource for team members to gain knowledge on e.g. continuous delivery practices, there should also be other resources like workshops or tutorials to learn the most important aspects about release and feedback management. In the beginning of the project, tailored tutorials show the developers how to use the tools. During the projects, team members reflect over the actual tool usage in retrospective meetings to improve upon common mistakes and to build best practices. With these experiential learning techniques, a culture of continuous improvements and continuous learning within the teams should be established.

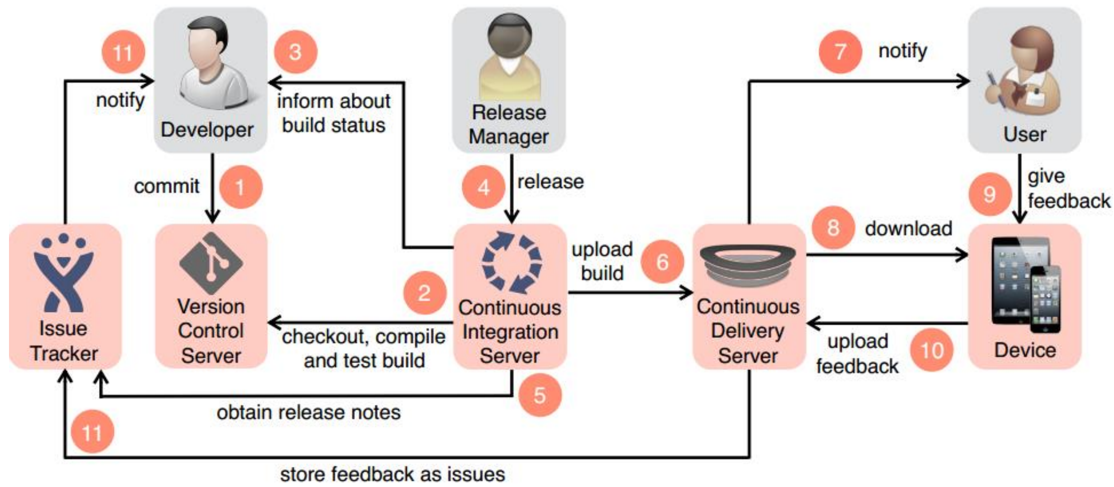

Figure 3: Rugby's continuous workflow with abstract tools and transitions [KAB+14]

2.4.2 RUGBY'S WORK FLOW

Figure 3 shows the integrated continuous workflow together with its tools and transitions. The workflow starts each time a developer commits source code to the version control server, leading to a new build on the continuous integration server. If the build was built successfully and if it passed all test stages, the team can decide to upload it to the delivery server, which then notifies users about a new release. Each release includes release notes, which are collected automatically by the continuous integration server and can be edited in the manual release step if necessary. The user can download the release and recognize easily, which features and bugs were resolved in the release. He can use an embedded mechanism to give feedback in a structured way. This feedback is collected on the delivery server and forwarded to the issue tracker [KA14].

SE and software evolution with one Scrum core workflow together with three additional workflows: (1) issue management needs an issue tracker, (2) review management needs a VCS, (3) release management needs a CI and a CD server, (4) feedback management needs a CD server and an issue tracker [KAB+14].

3 COMPARISON

In this section, first we introduce the tools, both the commercial and open source ones, and their features. Then, we compare the tools in each of the Rugby's important workflows and Scrum core workflow.

3.1 DESCRIBING THE TOOLS AND RESPECTIVE FEATURES

There are four main subsystems which Rugby uses them in its continuous workflow: Issue Tracking, Version Control System, Continuous Integration, and Continuous Delivery. For each of these subsystems, we analyze commercial and open source tools, and extract their workflow and object analysis model diagrams to show similarities and differences among the tools.

3.1.1 Issue Tracking; JIRA vs. GitHub Issues

Issue tracking is a software tool that enables the developers to record and track the status of all issues associated with each configuration object in the project [Pr05]. In this section, we compare two systems, JIRA by Atlassian Tools and GitHub Issues. Both these tools offer project environments, in which the team members can define issues, assign them to team members, track its state and close the issues. Considering their workflows, their functionality is similar:

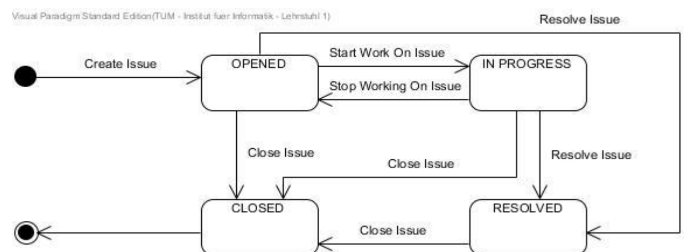

Figure 4: Jira Workflow

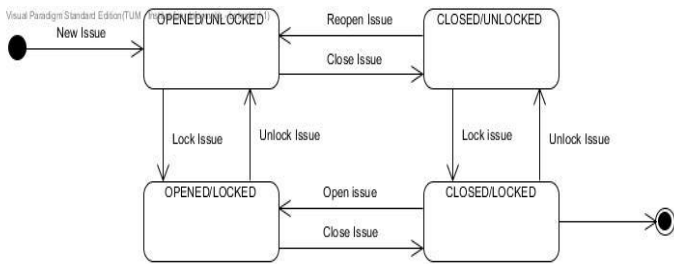

Figure 5: *GitHub Issues* Workflow

However, there are a couple of different features which are available in JIRA and not in GitHub, or GitHub implements them differently.

Before starting with JIRA, it is important to know the concept of analysis object model. Analysis object model, or class diagram, captures the concepts that exist in the domain, and their relationships. It illustrates the concepts in the real-world problem domain and only shows essential attributes, relationships and operations. Therefore, using analysis object model, the main concepts and their relationships of JIRA are shown in Figure 6:

Issue (the abstract superclass) is the main concept. Apart from typical attributes like name, summary, difficulty, priority, and so on, when you create an issue, it needs many other considerations as well. An issue can be a *User Story*, *Task*, *Epic*, *Bug* or *Impediment*. It should be assigned to one *Role* (like *Developer*). Each issue can be part of a sprint and belongs to one project (the current project that you are working on) and one backlog (either the Product Backlog or a Sprint Backlog). It can be part of a component and a version. A component is a unit of composition with contractually specified interfaces and explicit context dependencies only. It can be deployed independently and is subject to third-party composition. A version is a way to categorize the unique states of system as it is developed and released. Each issue can have multiple comments from different authors (usually from other developers).

Therefore, JIRA can fully support Rugby in the projects. If we look at GitHub's analysis object model, we can see that some of JIRA's concepts are not available in GitHub or you have to deal with them differently. In GitHub, an *Issue* can have multiple *Label*s, assignees, *Comment*s and *Milestone*s. GitHub does not provide special features to deal with the concept of sprints, backlogs, versions and components. However, we can deal with them by using labels. Unlike JIRA, there are not different types for issues in GitHub, but using labels they can also be categorized using labels.

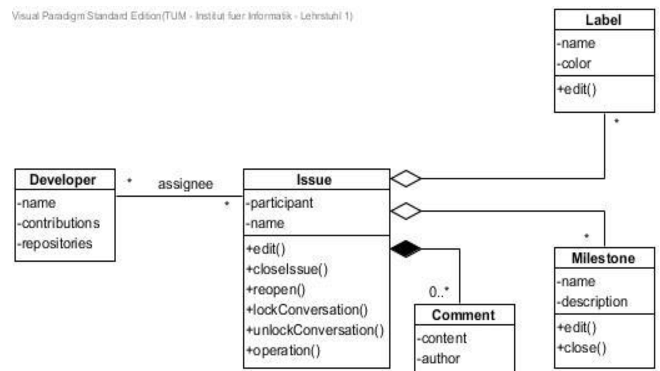

Figure 7: Analysis Object Model describing the main concepts of GitHub Issues

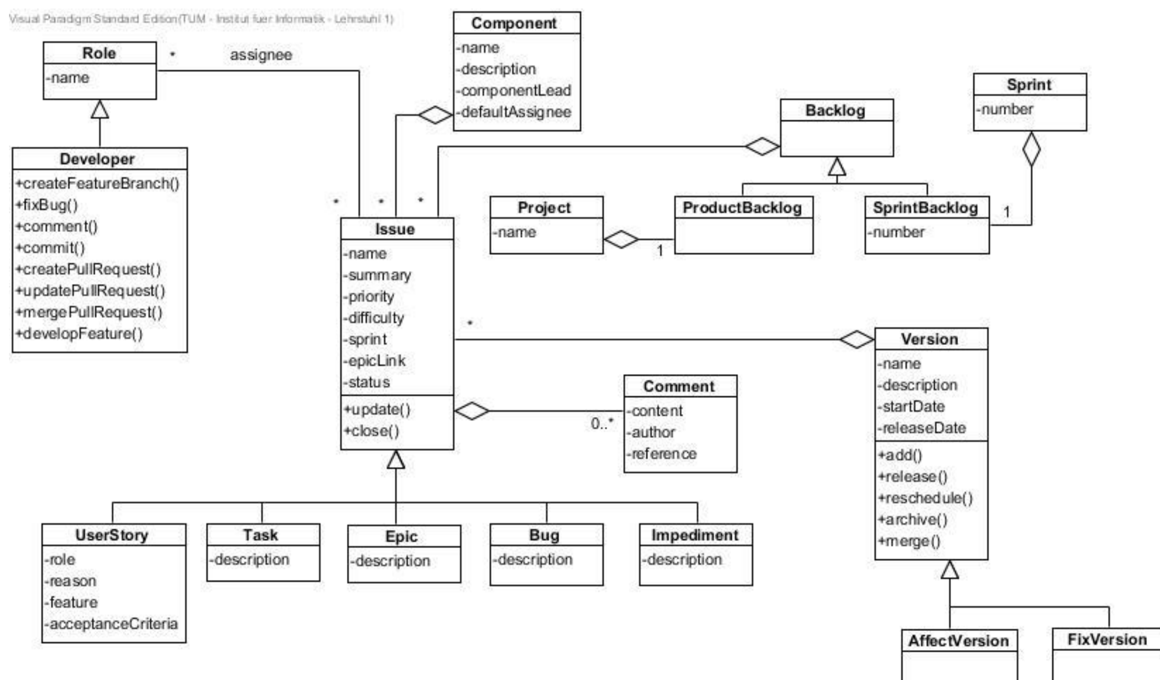

Figure 6: Analysis Object Model describing the main concepts of JIRA

Hence, in the first comparison, JIRA has more special features rather than GitHub. Features like sprint, version, etc. Nevertheless, all of them to some extent can be done differently in GitHub as well using the labels. For instance, we can set a label with “Sprint 2” for an issue which simply means that this label belongs to the second sprint.

3.1.2 Version Control Server; Bitbucket Server vs. GitHub

VCS is a system that keeps track of files and their history and have a model for concurrent access [Ot09]. Two common tools that are used by developers for version control are Bitbucket Server, formerly known as Stash from Atlassian Tools, and GitHub. In this section, we are going to compare them in the matter of features and functionalities that they each have.

The workflow for both Bitbucket and GitHub is the same:

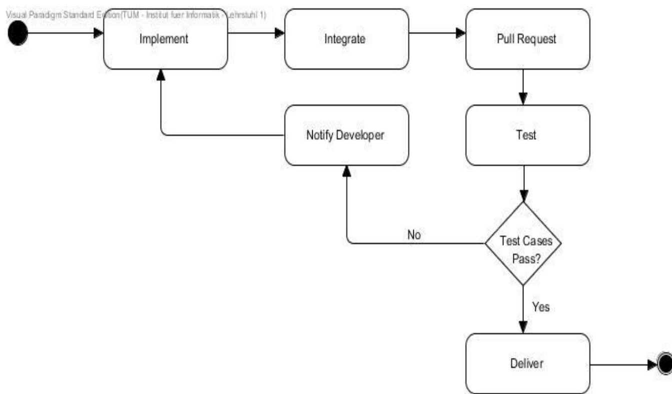

Figure 8: Bitbucket and GitHub Workflows

After commit and merge request, the system checks some test cases on the integrated code. Then it delivers or notifies the developer when the test cases pass or fail respectively. However, there is one important difference between these two tools: when we want to see the changes in the code before merging the new code, before merge request. GitHub retrieves the current system from master branch or Development branch directly after getting the diff request. It is a one-time process and the developer can see the differences between the committed sources with the current one. In a real project, the master branch will notably diverge from any given feature branch. That means other developers will be working on their own branches and merging them in to master continuously. Once master has progressed, a simple git diff command from the feature branch point back to its merge base is no longer adequate to show the real differences between the two branches. We only see the difference between the branch tip and some older version of master.

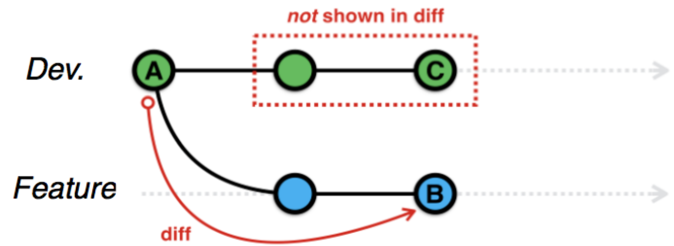

Figure 9: GitHub git diff overview [adopted from Pe15]

Bitbucket shows different versions of the Development branch or master branch to the user when merge conflict happens:

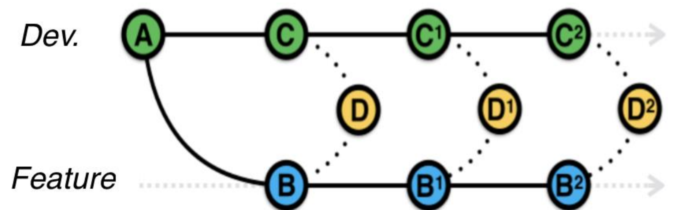

Figure 10: Bitbucket git diff overview [adopted from Pe15]

In fact, each time someone pushes to or merges a branch into master or our feature branch, Bitbucket is potentially going to need to calculate a new merge in order to show us an accurate diff.

The analysis object model for GitHub shows different components that GitHub uses to do version control:

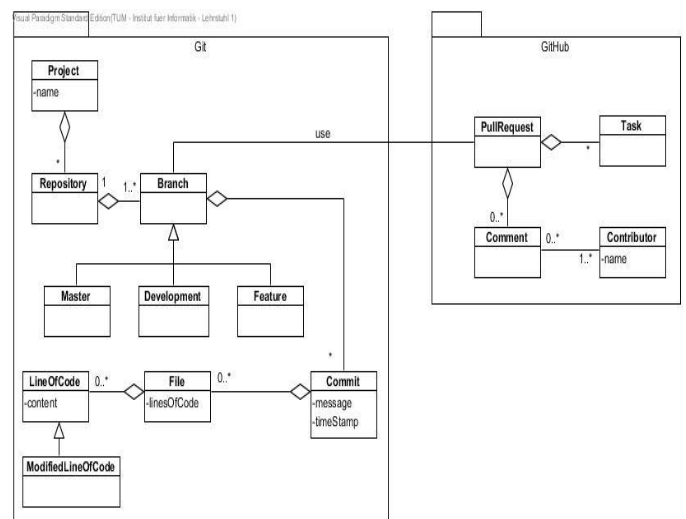

Figure 11: Analysis Object Model describing the main concepts of GitHub

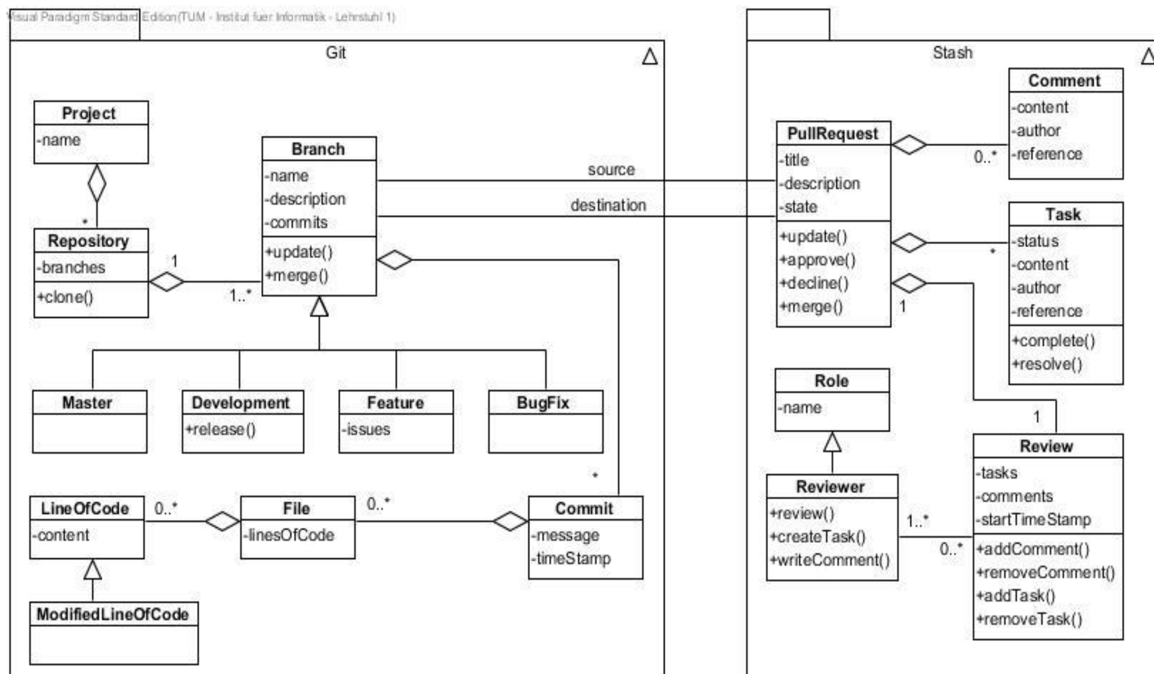

Figure 12: Analysis Object Model describing the main concepts of Bitbucket/Stash

The model consists of two main components, Git and GitHub. Git is the core concept in version control systems and GitHub uses Git to control the versions in the project. GitHub has a *PullRequest* which uses the Git *Branch* for controlling the versions. *PullRequest* consists of many *Task*s and maybe different *Comment*s and each *Comment* has its own *Contributor*, which means the one who wrote this comment.

After the *PullRequest*, GitHub extracts the data like *Branch* type (which can be *Master*, *Development* or *Feature*) and different branch-related data like *Commit*s and the Branch's Repository. A *Commit* has different *File*s and each *File* has many *LineOfCode*. A *LineOfCode* can be a *ModifiedLineOfCode* which should be merged to the *Repository*.

Figure 12 shows Bitbucket analysis object model. We see that exactly like GitHub, Bitbucket uses the Git as the core for controlling the versions in the projects.

The concepts in Bitbucket and GitHub look like the same. *PullRequest* uses *Branch* as source and destination for the merged code. In Bitbucket, each *PullRequest* has one *Review*. *Review* is done for all the merge requests and the aim is to resolve merge conflicts after occurring or even in advance. Each *Review* has one or more *Reviewer* and the *Reviewer* is a *Role*.

Therefore, in the second comparison, we see that both Bitbucket and GitHub are equally the same in the matter of functionalities, but there are two more features in Bitbucket. In Bitbucket, each merge request has to have a review. It also shows us what the resultant merge actually look like during viewing the merge request. It is done by creating a merge commit behind the scenes and showing us the difference between it and the tip of the target branch.

3.1.3 Continuous Integration; Bamboo vs. Travis CI vs. Jenkins

Continuous Integration (CI) is a software development practice where developers integrate their work frequently, which leads to multiple integrations per day. Each integration is verified by an automated build, including test, to detect integration errors as quickly as possible [FF06]. In other words, CI is the process of automatically building and running tests whenever a change is committed.

In this section, we will compare three CI solutions: Bamboo from Atlassian Tools, Travis CI from GitHub and Jenkins. All these three tools are commonly use by developers in different projects. The basic principle behind all of them is the detection of changes in the code repository and triggering a set of Jobs or tasks.

Figure 13 shows Continuous Integration workflow.

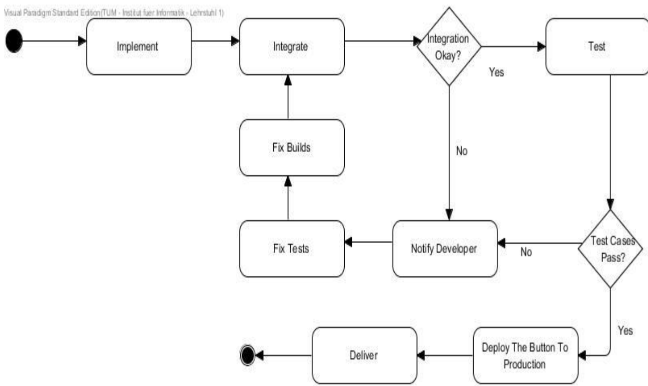

Figure 13: Continuous Integration workflow

After each Merge request, which basically means integration of the new code to the current system, the CI tries to interpolate the code into the system, and if it succeeds to do so, then it tries out different tests which the developer defined to run on the CI. If the system passes all of the tests, then it notifies us that the system is ready and we can deploy it by pushing a button. If the system fails during the integration part or testing, CI notifies us to fix the problems. This is a general workflow for almost all the CIs.

Figure 14 shows different components of Bamboo analysis object model:

The main concept in Bamboo build is *Plan*. A *Plan* defines everything about our continuous integration build process in Bamboo. By default, it has a single *Stage*, but we can use it to group *Job*s into multiple stages. A *Plan* processes a series of one or more *Stage*s that are run sequentially using the same repository. It also specifies how the build is triggered, and the triggering dependencies between the plan and other *Plan*s in the *Project*. Every *Plan* belongs to a *Project*. A *Stage* maps *Job*s to individual steps within a build process of *Plan*. For example, we may have a *Plan* build process that contains an *Integration* step followed by several *Test* steps and a *Delivery* step. We can create separate Bamboo *Stage*s to represent each of these steps. By default, a *Stage* has a single *Job*, but we can use it to group multiple *Job*s. A *Stage* processes its *Job*s in parallel. It must successfully complete all its *Job*s before the next *Stage* in the *Plan* can be processed. A *Job* is a single build unit within a *Plan*. One or more *Job*s can be organized into one or more *Stage*s. The *Job*s in a *Stage* can all be run at the same time, if enough Bamboo agents are available. A *Job* is made up of one or more *Task*s. A Task is a small discrete unit of work, such as source code checkout or running a script.

Figure 15 shows Travis analysis object model.

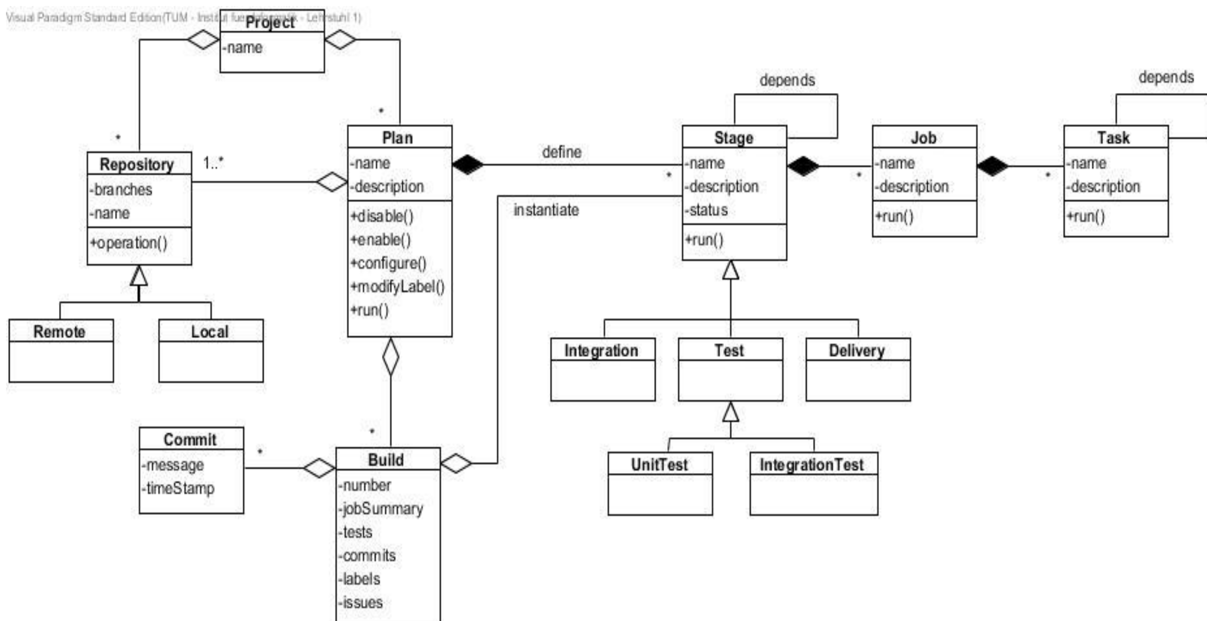

Figure 14: Analysis Object Model describing the main concepts of Bamboo

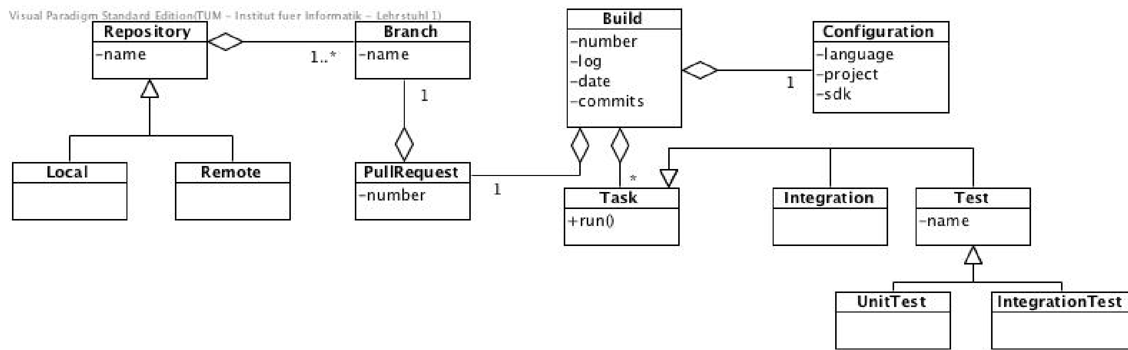

Figure 15: Analysis Object Model describing the main concepts of Travis

When a *PullRequest* is opened, Travis receives a *PullRequest* notification from GitHub. It turns this notification into a *Build* and runs it. Along the way, it updates the commit status of the commits involved, which in turn shows on GitHub as either a warning that the build is still running, which means the *PullRequest* should be merged with caution because the *Build* failed, or that it can be merged safely because the *Build* was successful. Travis builds a *PullRequest* when it's first opened and when commits are added to the *PullRequest* throughout its lifetime. Like Bamboo, it tests the merge between the origin and the upstream *Branch* rather than test the commits from the *Branche*s the *PullRequest* is sent from. Travis needs access to read and write webhooks (an HTTP POST that occurs when something happens), services, and commit statuses. That way that it can create the automated "hooks" it needs to automatically run when we want it to. Travis also uses a YAML file called *.travis.yml* to tell it how to set up a build. Travis most of the time knows what should be done without any need to explicitly define the flow. For example, if there is the build.java file, Travis will understand that it should be compiled, tested, etc. using Java. It inspects our code and acts accordingly. We can switch from different technologies without making any changes to Travis or the configuration file. It has a strong dependency with GIT. In cases when some other version controls system is used, Travis is not a good option. If, on the other hand, we are using GIT, working with Travis is like forgetting that continuous integration even exists. Whenever the code is pushed to the repo Travis will detect it and act depending on changes in the code, including .travis.ylm file. It also removes the need to deal with jobs, configurations and other nuances.

Figure 16 shows Jenkins analysis object model. Apart from project structure and build plan, both Jenkins and Bamboo work in the same way. With Jenkins we start by creating Build Project, which is called Build Job. Drawing an analogy to Bamboo, Jenkins has a build plan with single stage containing single job and list of tasks (Build Steps and Post-build Actions are nothing but tasks). There are no stages and no way to run anything in parallel.

Jenkins is easy to extend, powerful and free. Its main advantage is in the number of plugins and community support. One can hardly imagine a need that is not already covered by one or more plugins. Jenkins can be extended easily. As a downside, such architecture based on plugins comes at a cost of stability. Plugins are of different quality and it is not uncommon for an update to break existing jobs or to provoke unexpected behavior of the system. If one is looking for robust and flexible solution without any cost, Jenkins is the best choice.

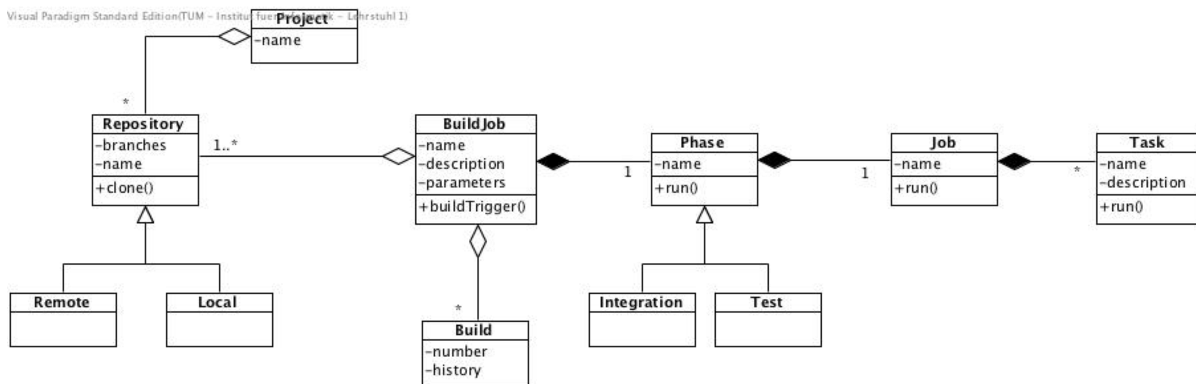

Figure 16: Analysis Object Model describing the main concepts of Jenkins

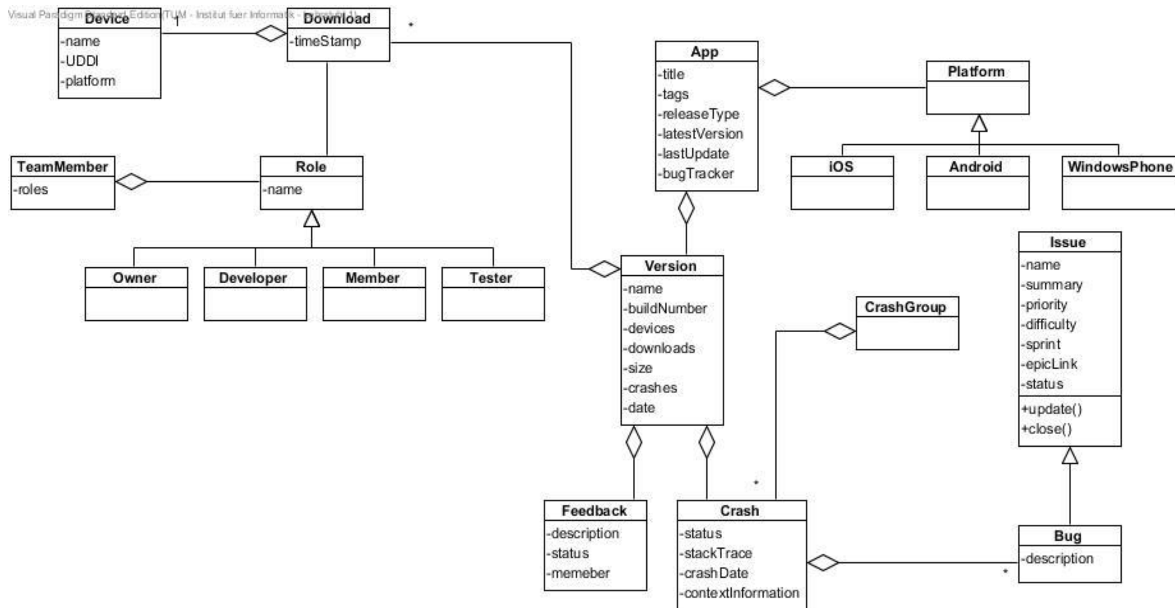

Figure 17: Analysis Object Model describing the main concepts of HockeyApp

3.1.4 Continuous Delivery; HockeyApp vs. Crashlytics

Continuous Delivery (CD) is a software engineering approach in which teams keep producing software in short cycles and ensure that the software can be reliably released at any time [Ch15]. Rugby uses an enterprise app store as CD server, a web portal through which end users can access, download, install approved software applications, send feedback about the application and report crashes. To report the crashes, Enterprise AppStore should have two components: a reporting library and a server-side collector. The role of the reporting library is to prepare the details about a crash and the role of the server-side component is to collect the crash data and present it in a meaningful way. In this section, we will compare three enterprise app stores: HockeyApp and Crashlytics (named fabric after the acquisition from Twitter).

Figure 17 shows HockeyApp main components. It supports the management and recruitment of testers, the distribution of apps and the collection of crash reports. It also supports apps on iOS, Android, Mac OS X, and Windows Phone as well as custom apps. Crash reports are working on all those platforms. Beta distribution is functional on iOS, Android, Windows Phone, and Mac OS X; for custom apps, we can only notify testers via email, but there is no in-app update functionality. On iOS and Mac OS X, the SDK leverages the PLCrashReporter framework by Plausible Labs. PLCrashReporter is open source and creates full standard crash logs with all threads. After the user has sent the crash log, the HockeyApp server collects all crash information and automatically symbolicates all threads to provide class names, method names and even line numbers. To achieve this, developers need to upload the dSYM package for each app version for iOS and Mac OS X apps. It is not necessary to upload the app binary. HockeyApp groups the crash reports on all platforms by similarities, so developers always see the critical parts quickly and easily.

Crashlytics is a free service offered by Twitter that collects our crashes and various other bits of information. It is easy to setup and install and it instantly starts providing value as soon as we install it. It is completely free with unlimited apps, unlimited users, unlimited crashes, and unlimited keys.

Crashlytics has a proficient symbolication, the ability to browse logs and their usability on the server are good and attentive. There is also a wait list, which means we may have to wait to get notified. However, the wait time at the moment does not appear to be very long and we are often notified within minutes.

Figure 18 shows Crashlytics object analysis model and we can see that to some extent, it looks like HockeyApp. They both save the crashes and bugs as issues within the version of the application. They also can connect to an Issue Tracker and make use of the issue types for further development and maintenance.

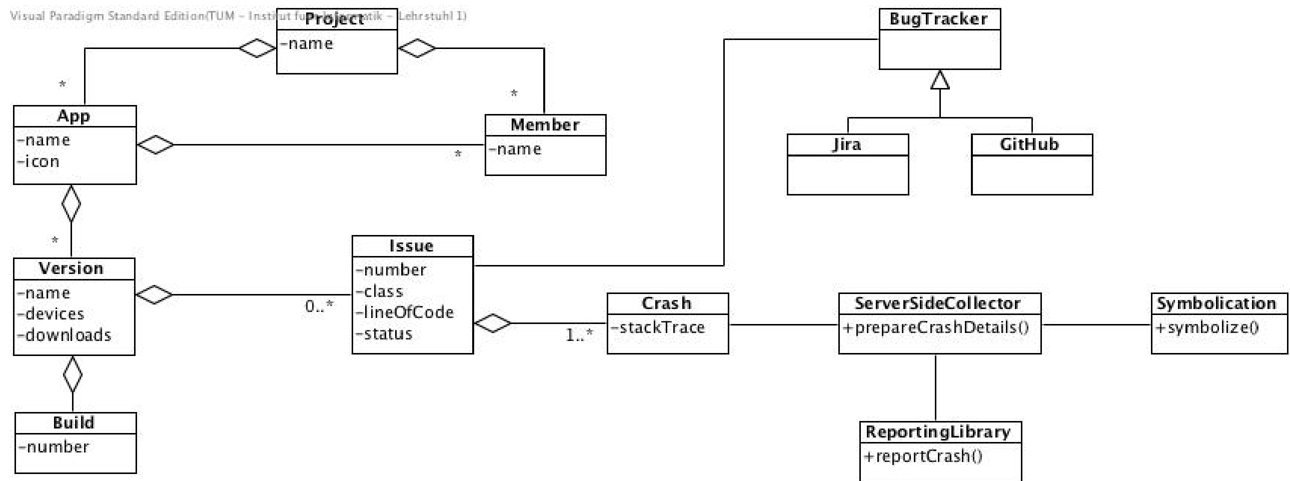

Figure 18: Analysis Object Model describing the main concepts of Crashlytics

3.2 Comparison between Different Tools in each Workflow

For each of Issue Management, Review Management, Release Management and Feedback Management workflows, we select alternatives and compare them with the corresponding commercial tools in the existing reference implementation with respect to the most important core use cases and non-functional requirements in each workflow. At the end of each section, there is a comparison table which shows scores for each feature of the tool (combination). Scores range from 1 (very poor quality or support) to 5 (excellent quality or support). In case a tool does not support a feature, a dash sign is shown (-).

3.2.1 Issue Management Workflow

Issue tracking is a software tool that enables the developers to record and track the status of all issues associated with each configuration object in the project [Pr05]. In this section, we compare JIRA by Atlassian and GitHub Issues for issue management.

JIRA has a customizable workflow for issues which makes it easy to tailor based on different needs in each project. The issue workflow consists of statuses and transitions that an issue goes through during its lifecycle modeled as a finite state automaton. The workflow can be edited or a new one can be created from scratch. JIRA supports sprint planning and sprint reviewing as defined in Scrum.

On the other hand, GitHub Issues, does not have customizable workflows for issues. However, a list of milestones can be configured with a corresponding due date. A milestone shows the last edit date, percentage of completed tasks, the number of open and closed issues and merge requests, which can give a general overview of the project progress. With some configurations, GitHub Issues can support agile project management. Using milestones and labels, it is possible to simulate sprint backlogs and versions which are not available by default [Ke15].

Therefore, the main advantages of JIRA in comparison with GitHub Issues are its customizable workflows for issues and support for variety of issue types as well as different reports and charts. GitHub Issues main advantage is its usability which is very simple and easy to work in comparison with JIRA.

Relevant features for Rugby	**JIRA**	**GitHub**
Customizable issue states and transitions	5	1
Taskboard support	5	3
Versions support	5	4 (using Labels)
Backlogs support	5	3 (using Milestones)
Sprint planning/review support	5	3
Usability	3	5

Table 1: Issue management workflow comparison

3.2.2 Review Management Workflow

VCS is a system that keeps track of files and their history and have a model for concurrent access [Ot09]. In this section, we compare Bitbucket Server, formerly

known as Stash, by Atlassian and GitHub for review management.

Both Bitbucket and GitHub use Git as distributed VCS to control the source code versions. Features like merge requests (a.k.a. pull request), branch management and merge management are the most important functions [CS14]. Therefore, Bitbucket and GitHub function similarly in their core features. Bitbucket only supports sharing code files as snippets (known as Gists) with additional plugins. Its web interface lacks inline editing unless additional plugins are used.

In contrast to GitHub, Bitbucket's web interface does not show tags. However, this does not affect its functionalities, since it supports all the GitHub's features without needing to use labels. In terms of usability, GitHub is slightly better than Bitbucket. It is simple and everything is easy to find in the menus. One important difference between Bitbucket and GitHub is that they use different mechanism to show compare view (diff) [Pe15]. Although Bitbucket's approach is more complex and has an overhead, it provides more accurate and useful merge request diff.

Relevant features for Rugby	**Bitbucket Server**	**GitHub**
Merge requests	5	5
Inline code commenting	5	4
Web inline editing	4 (using plugins)	5
Sharing code files and snippets support	4 (using plugins)	5
Commit comments	5	5
Accurate compare views	5	4
Branch/merge management	5	5
Collaborative code review	5	5
Usability	4	5

Table 2: Review management workflow comparison

3.2.3 Release Management Workflow

CI is a software development practice where developers integrate their work frequently, which leads to multiple integrations per day. Each integration is verified by an automated build, including test, to detect integration errors as quickly as possible [FF06]. In other words, CI is the process of automatically building and running tests whenever a change is committed. On the other hand, CD is a software engineering approach in which teams keep producing software in short cycles and ensure that the software can be reliably released at any time [Ch15]. Rugby uses an enterprise app store as CD server, a web portal through which end users can access, download, install approved software applications, send feedback about the application and report crashes. In this section, we compare three combinations of CI and CD, Bamboo and HockeyApp, Travis and Crashlytics, and Jenkins and Crashlytics.

Travis is a maintenance free CI server, since it is a hosted service and all the installations and updates are performed server side. Although Jenkins can be a hosted service as well, it is usually use on premise, needing administration effort for installations and updates. Bamboo can be used as cloud service or as on premise solution. Jenkins does not use a database for storage which makes it flexible and portable. Only by copying the configuration, Jenkins jobs can be easily migrated across multiple instances which is not possible in Bamboo. Therefore, maintainability in Jenkins is easier.

Additional build agents are needed to scale CI servers horizontally. Bamboo licenses are priced per agents that need to be installed and configured. In Jenkins, unlimited amount of build agents is available and configuration of build agents is very easy. The build agents are configured via Jenkins UI and all of the tools can be installed automatically. Therefore, Jenkins is easier to scale than Bamboo.

A deployment project in Bamboo is a container for holding the software project which is deployed. Besides, plan branches represent a build for a branch in the version control system. When the plan branch build succeeds, it can be automatically or manually merged back into master. Bamboo deployments allow a plan branch to be deployed to a test environment and the feature source code can be tested and evaluated in a real server environment before the code is merged back to master. In Jenkins, there are available plugins to support deployment and branches as well, while in Travis they are supported by Travis' configuration file. Concluding, all tools support deployment and branches. However, in Bamboo, when build jobs call deployments, it is not possible to go back to the build job to perform some post-deployment tasks. In addition, inability to provide input parameters, especially for deployment jobs, is a problem in Bamboo as well.

Each release includes release notes, which describe features and resolved bugs. The release notes are collected by the CI server and can be edited in the manual release step if necessary. Non of the CI solutions can create release notes automatically.

Code testing is the process of automatically building and running tests whenever a change is committed. Code

integration is the process of automatically integrating the code after a committed change. All three CI tools support testing and integration.

It is not possible to get crash logs in Travis, without setting up scripts to upload them to a third-party service after completion of a build. In Bamboo and Jenkins, there are a lot of useful information regarding crash logs in test output display. Bamboo has an easy to use UI, Travis' UI is even more simple. However, Jenkins' UI looks out of date. Both HockeyApp and Crashlytics, notify the users about new releases and support feedback notification as well. They enable the CI server to automatically upload a new build and support the download of releases, new versions, push notifications and publication configurations.

Relevant features for Rugby	**Bamboo + HockeyApp**	**Travis + Crashlytics**	**Jenkins + Crashlytics**
Build plan configuration	5	4	4
New commit/branch detection	5	4	4
Release notes creation	-	-	-
Testing	5	5	5
Integration	5	5	5
Build status notification	5	2	5
Deploy build to CD server	4	5	5
Feedback/new release notification	5	5	5
App version automatic upload	5	5	5
Release download	5	4	4
CI scalability	3	-	5
CI maintainability	3	5	4
CI usability	4	5	2
App version download usability for the user	5	4	4
Support of multiple mobile platforms	5	4 (iOS, Android)	4 (iOS, Android)

Table 3: Release management workflow comparison

3.2.4 Feedback Management Workflow

In Section 3.3, we mentioned that Rugby uses an enterprise app store as CD server. In this section we compare the combinations of HockeyApp and JIRA, and Crashlytics and GitHub Issues. Both Crashlytics and HockeyApp store crash reports and errors as issues. However, since GitHub Issues does not support multiple issue types, it is not possible to convert issues to appropriate work items like bugs or improvements.

In Crashlytics, different user groups can be defined and different users can be added to them. HockeyApp supports five user roles: owner, manager, developer, member and tester. HockeyApp supports more feedback management features in comparison with Crashlytics; while the user can reply to the developer's questions and his feedback records usage context, developer can pull them and reply to them.

Therefore, the main differences between HockeyApp and Crashlytics are the multiple mobile platforms support, user device registration and developer device management. While HockeyApp supports different platforms, Crashlytics is only limited to iOS and Android. Although both of these tools have a good usability, HockeyApp is more simple and easier to use. While users can register their iOS and Android devices in HockeyApp, they can do so only for the iOS devices in Crashlytics. HockeyApp also supports developer device management which is not supported in Crashlytics.

Crashlytics takes into account that often a crash occurs and assigns it an impact level. It notifies when a specific crash is more critical than another one. As a particular crash is reported more and more, Crashlytics tracks that information and calls out the crashes that should be dealt with next. On the other hand, HockeyApp provides more depth and accuracy of crash logs. However, it does require more setup compared to Crashlytics.

Relevant features for Rugby	**HockeyApp + JIRA**	**Crashlytics + GitHub Issues**
User device registration	4 (iOS and Android)	3 (iOS)
User feedback upload	5	5
User reply to developer question	5	5
User attach media	5	5
Developer device management	5	3
Developer feedback pull	5	5
Usage context record	5	5
Feedbacks and analytics for developer	5	5
Store crash reports and errors as issues	5	5
Issue conversion to appropriate work	5	1

item (issue)		

Table 4: Feedback Management workflow comparison

4 Conclusion

In this section, we create a second reference implementation with tools that can be used for free in open source projects, using GitHub tools, Jenkins and Crashlytics, and compare it with the commercial reference implementation. The second reference uses GitHub Issues as Issue Tracking, GitHub as VCS, and Crashlytics as CD server. For CI, there are two options to choose: Travis and Jenkins. Both of these CI servers are candidates to become the CI solution. Cloud vs On Premise of the code repository is the most important factor in choosing, followed by project type. If the project is open source, Travis would be the better option, because there is no setup time and fee. If it is a company project with privacy concerns, Jenkins is a better option, because the setup time is minimal and maintaining the server is rather simple.

Due to the fact that JIRA, Bitbucket and Bamboo are all Atlassian tools, they can integrate seamlessly using minimal effort and have a high overall usability as well, while the learning curve might be high for new users. On the other hand, GitHub Issues, GitHub and Travis are from GitHub and they also integrate with each other. GitHub tools are simpler and have fewer features than Atlassian tools, so they are easier for new users and can be used for free in open source projects. In addition, maintenance is not an issue while working with GitHub tools, because they are all hosted in the cloud.

Considering the price plan of different tools, the proposed reference implementation using open source tools can be used for free, especially for small startups that work on public repositories. However, if private repositories and concurrent jobs in CI are necessary, the price varies and there is not too much difference between the two reference implementations in the term of price. Using the commercial reference implementation for middle sized companies, with around 100 persons, can be expensive. For such middle sized companies, the open source reference implementation cost is about half of the commercial one.

Therefore, both the existing commercial reference implementation, using JIRA, Bitbucket, Bamboo and HockeyApp, and the proposed open source implementation, using GitHub Issues, GitHub, Travis/Jenkins and Crashlytics, cover most of Rugby's important use cases. The commercial solution is a better choice when security and privacy are important and repositories should be private. On the other hand, when the project can be public, the open source solution should be preferred. Rugby can be implemented with different tools, either for commercial projects or open source projects. However, there is always a tradeoff between the tools quality and their price. It remains future work to compare other tools for their use in Rugby projects.

5 References

[FS14] B. Fitzgerald, K.J. Stol: "Continuous software engineering and beyond: trends and challenges." Proceedings of the 1st International Workshop on Rapid Continuous Software Engineering. ACM, 2014.

[Bo14] J. Bosch: "Continuous software engineering: An introduction," in Continuous Software Engineering. Springer, 2014; Pages 3–13.

[Sc04] K. Schwaber: Agile project management with Scrum. Microsoft Press, 2004; Chapter 4.

[JBR+99] I. Jacobson, G. Booch, J. Rumbaugh, J. Rumbaugh, G. Booch: The unified software development process. Addison-wesley, 1999; Page 92.

[HF10] J. Humble, D. Farley: Continuous delivery: reliable software releases through build, test, and deployment automation. Pearson Education, 2010; Pages 24-29.

[KAB+14] S. Krusche, L. Alperowitz, B. Bruegge, M. O. Wagner, Rugby: An Agile Process Model Based on Continuous Delivery, Proceedings of the 1st International Workshop on Rapid Continuous Software Engineering. ACM, 2014.

[KKP+15] S. Klepper, S. Krusche, S. Peters, B. Bruegge, L. Alperowitz: Introducing Continuous Delivery of Mobile Apps in a Corporate Environment: A Case Study, 2015.

[BKA15] B. Bruegge, S. Krusche, L. Alperowitz: Software Engineering Project Courses with Industrial Clients, ACM Transactions on Computing Education, 2015.

[Pr05] R.S. Pressman: Software engineering: a practitioner's approach (7th Edition). Palgrave Macmillan, 2010; Page 595.

[KA14] S. Krusche, L. Alperowitz: Introduction of Continuous Delivery in Multi-Customer Project Courses, Proceedings of the 36th International Conference on Software Engineering, 2014; Pages 2-3.

[Ke15] H. Kellaway: Using Github for Lightweight Software Project Management, 2015. Retrieved 06-October-2015 from http://harlankellaway.com/blog/2015/04/02/using-github-issues-for-software-project-management/

[Ot09] S. Otte.: Version Control Systems. Computer Systems and Telematics, 2009 Institute of Computer Science, Freie Universität, Berlin, Germany.

[CS14] S. Chacon, B. Straub: Pro Git (2nd Edition). Apress, 2014; Pages 89-98.

[Pe15] T. Petterson: A better pull request, 2015. Retrieved 13-September-2015 from https://developer.atlassian.com/blog/2015/01/a-better-pull-request/

[FF06] M. Fowler, M. Foemmel: Continuous integration. Thought-Works) http://www.thoughtworks.com/ContinuousIntegration.Pdf, 2006.

[Ch15] L. Chen: Continuous Delivery: Huge Benefits, but Challenges Too. Software, IEEE, 2015.