Daniel Metzner, BSc

Automating the Software Development Life Cycle using GitHub Actions

A practical approach on Catrobat's Share Community Platform

Master's Thesis

to achieve the university degree of

Master of Science

Master's degree programme: Computer Science

submitted to

Graz University of Technology

Supervisor

Univ.-Prof. Dipl-Ing. Dr. Wolfgang Slany

Institute for Softwaretechnology
Head: Univ.-Prof. Dipl-Ing. Dr. Wolfgang Slany

Graz, September 2020

This document is set in Palatino, compiled with pdfLaTeX2e and `Biber`.

The LaTeX template from Karl Voit is based on KOMA script and can be found online: https://github.com/novoid/LaTeX-KOMA-template

Affidavit

I hereby declare that I have authored this thesis independently, that I have not used any other sources/resources than the declared ones, and that I have explicitly indicated all material that has been quoted either literally or by content from the sources used. The text document uploaded to TUGRAZonline is identical to the present master‘s thesis.

08.10.2020 [illegible]

Date Signature

Abstract

Software development involves various challenges and is prone to human errors. Especially in free and open-source software (FOSS), such as Catrobat's Share community platform, contributors often lack the necessary expertise in the project's domain, rules, and principles to develop high-quality software. Automating tasks throughout the software development life cycle and following best practices such as continuous integration can significantly reduce development costs and improve a software's quality. This thesis offers insights into the challenges and applied measures to develop an efficient and maintainable continuous integration system without high acquisition costs using GitHub Actions. Automated software tests that verify every contribution and provide immediate feedback help developers write clean and error-free code. Moreover, from continuous delivery to an automatic code generation and a daily synchronization with a third-party platform, various workflows have been optimized and automated to reduce the complexity and cost throughout the software development life cycle.

Kurzfassung

Softwareentwicklung ist mit den verschiedensten Herausforderungen verbunden und anfällig für menschliche Fehler. Insbesondere bei Free and Open-Source Software (FOSS) wie der Share Community Plattform von Catrobat fehlt den Mitwirkenden häufig das erforderliche Fachwissen in Bezug auf den Bereich, die Regeln und die Prinzipien des Projekts, um qualitativ hochwertige Software zu entwickeln. Durch die Automatisierung von Aufgaben während des gesamten Lebenszyklus der Softwareentwicklung und die Befolgung von empfohlenen Vorgehensweisen wie der kontinuierlichen Integration können die Entwicklungskosten erheblich gesenkt und die Qualität einer Software verbessert werden. Diese Arbeit bietet Einblicke in die Herausforderungen und angewandten Maßnahmen, um mit GitHub Actions ein effizientes und wartbares kontinuierliches Integrationssystem ohne hohe Anschaffungskosten zu entwickeln. Automatisierte Softwaretests, die jeden Beitrag überprüfen und unmittelbare Rückmeldung geben, helfen Entwicklern sauberen und fehlerfreien Code zu schreiben. Darüber hinaus wurden die verschiedensten Arbeitsabläufe wie der Softwareauslieferungsprozess, die automatische Generierung von Code bis hin zur täglichen Synchronisierung mit einer Plattform eines Drittanbieters optimiert und automatisiert, um die Komplexität und Kosten während der Softwareentwicklung zu reduzieren.

Contents

List of Figures

Listings

1 Introduction

Developing and maintaining a software product of high quality is a tedious task. With the always increasing and changing demands of users, a typical software product can be very complicated. More and more features are pushed into projects every day. In addition to that, developers are often under pressure to deliver just in time. Thus, code quality and documentation are repeatedly postponed until it is almost impossible to work with the existing codebase. Hence, a very error-prone and time-consuming code refactoring is necessary to re-implement an existing feature with reduced complexity. Every new update could break an application in unexpected ways. Without proper testing, it can be challenging to develop large applications.

There are seldom any real software products fully developed and maintained by a single person. Efficiently working in a team requires developers to operate in parallel and regularly merge their work. Since developers may change the same code lines differently, merging can be challenging, mainly if applied only in irregular, too long time intervals. Studies even show that large teams of developers compared to small groups often only bring a slight improvement in delivery time, while drastically increasing the development cost (Brooks, 2020).

A counteract to these drawbacks is called continuous integration (CI). CI requires developers to integrate their changes in small patches and short time frames. Besides, CI is all about detecting and preventing issues before the integration. A CI system automatically provides the developers with the necessary feedback about their proposed changes to verify that every contribution achieves a certain quality. Therefore, developers do not need to manually run tests and software metrics whenever they contribute to a project.

This thesis aims to significantly reduce development costs and increase the Catrobat's Share community platform's software quality. Therefore, a CI system is implemented using GitHub Actions to check every contribution to the project thoroughly. Moreover, several workflows throughout the software development life cycle have been automated to reduce the amount of manual work that has to be done by developers repeatedly.

1.1 Catrobat project

People of all ages and social backgrounds should have easy access to education. Consequently, back in 2010, the International Catrobat Association was initiated by the Institute of Software Technology at the Technical University of Graz to provide children and teenagers with free educational resources. Their main focus centers around tools and courses about programming in a playful environment. (Slany, 2020)

The Catrobat project[1] (Figure 1.1) is inspired by the Scratch project[2] developed by the Lifelong Kindergarten Group at the MIT Media Lab. All products associated with the International Catrobat Association follow the philosophy of independent free, open-source software (FOSS). FOSS implies that in addition to the free of charge usage of software, everyone can access, contribute, and modify its source code. All repositories are hosted open to the public on a code sharing and publishing service named GitHub[3].

CATROBAT

Figure 1.1: Catrobat logo.

1. https://www.catrobat.org/ visited on 9 Sep. 2020
2. https://scratch.mit.edu/ visited on 9 Sep. 2020
3. https://github.com/Catrobat visited on 9 Sep. 2020

1.1.1 Catrobat

Similar to Scratch, Catrobat is a visual programming language. Visual programming languages provide users with the possibility to create and modify programs graphically. Instead of using conventional textual languages, this approach simplifies the coding process to a bare minimum. Users do not need to learn any complex syntax. Hence, typing errors can be prevented. Visual programming languages lower the learning curve and increase accessibility to the exciting world of coding for beginners. Users can focus on their creativity and create programs very fast - The perfect baseline to promote computational thinking skills to children, as the *Developer website of Catrobat* (2020) describes it.

1.1.2 Pocket Code

Visual programming languages require a different set of tools for efficient usage compared to traditional approaches. Catrobat's corresponding integrated development environment (IDE) is called Pocket Code. An IDE is a collection of programs to ease the software development process. A small selection from the user interface (UI) can be seen in Figure 1.2.

Projects created in Pocket Code consist of a background and optional objects. By tapping on an object in the project overview, scripts, looks, and sounds can be assigned to the object. The looks and sounds can be created by users or downloaded from a media library. Scripts animate objects and bring life into interactions. Dragging predefined bricks from various categories together forms a script to implement any logic. Their functionality ranges from control bricks to sensor bricks. Already during development, projects can be executed directly in Pocket Code at any time.

While Scratch is developed only for the Web, the Pocket Code apps concentrate on the mobile market. A market with 3.5 billion potential users that is growing every year (Turner, 2020). Besides, many children and teachers in third world countries can not afford a PC, while many have access to smartphones. Pocket Code supports more than 75 different languages and

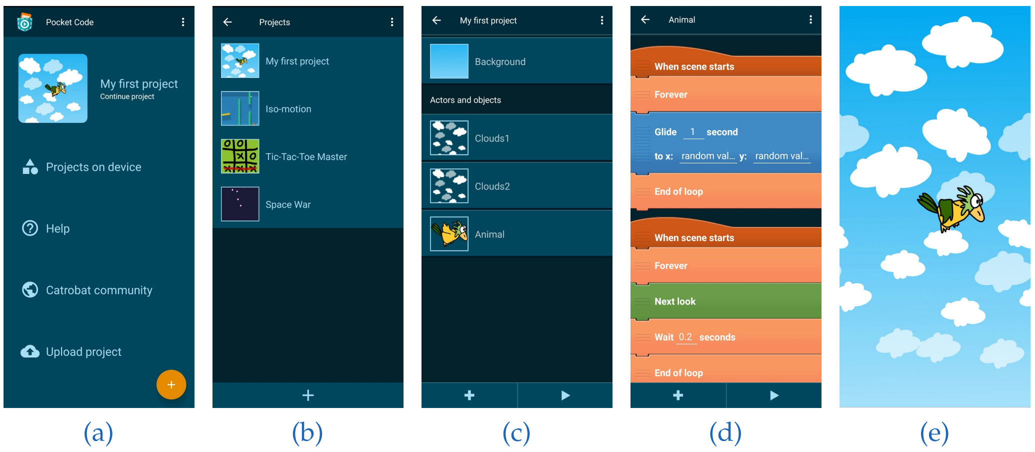


Figure 1.2: Pocket Code Android version 0.9.74: The landing page in (a) provides users with an overview of its functionalities. The projects' overview is given by (b). A Pocket Code project is build using objects (c). Each object can have scripts, looks, and sounds assigned. Scripts are created by connecting various bricks (d). A snapshot of the project during its execution is shown in (e).

already has more than one million users in 180 countries. It is available to everyone and can be download for free on Android[4] and iOS[5].

The Android and iOS versions have been developed by individual teams to utilize the underlying operating system entirely. Due to a different velocity in the development between the teams, it is practically impossible to provide the same features and possibilities in both apps. Additionally, the teams must follow specific guidelines defined by Google and Apple, leading to further differences. For this reason, the Catrobat language version was introduced to determine what features must be implemented to support projects of a specific version. As a result, projects created using any Pocket Code app can be shared among themselves as long as they support the same minimum language version.

Apart from Pocket Code, Catrobat has many sub-projects and services tightly coupled to Pocket Code. For example, a converter from Scratch to

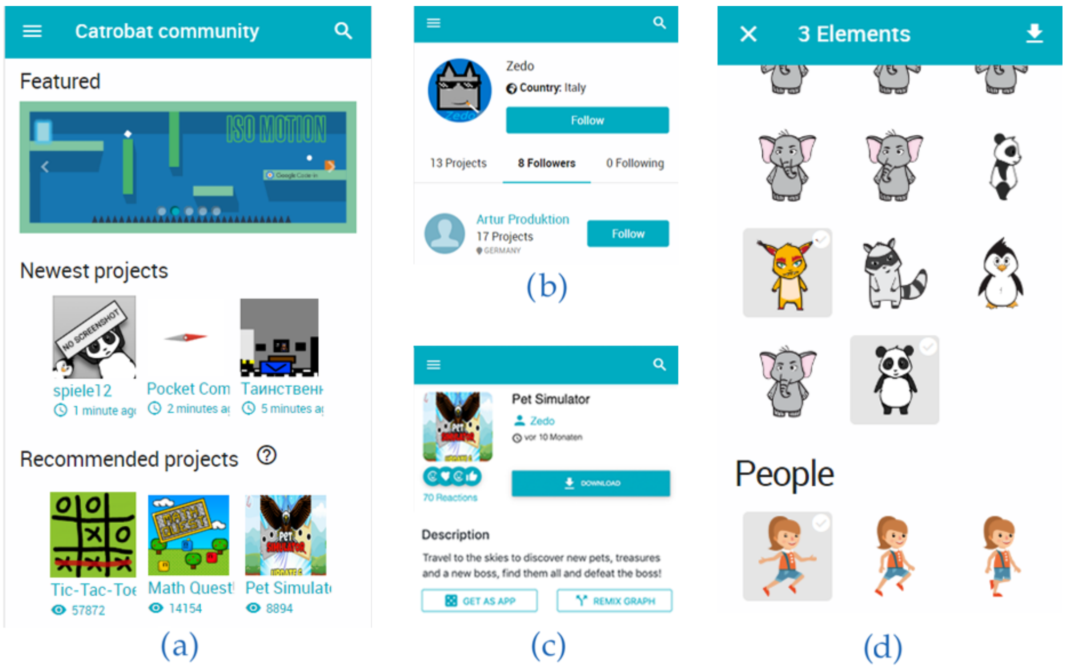
4. https://catrob.at/gp visited on 9 Sep. 2020
5. https://catrob.at/ca visited on 9 Sep. 2020

Catrobat, an image manipulation app, and the Share community platform to publish projects.

1.1.3 Share community platform

The Share community platform is a web application developed under the name of Catroweb and is responsible for hosting dynamic user data, including more than 180 thousand Pocket Code projects. Storing data in a central place enables Catrobat to provide a cloud-like service to its users. Examples of Catroweb's website are given by Figure 1.3.

Projects created in Pocket Code can be uploaded to the platform with a few taps in the app. Once a project is uploaded, it can be shared, downloaded, and remixed by users worldwide. A remix project emerges if an existing project is copied and modified. Additionally, Pocket Code users have access

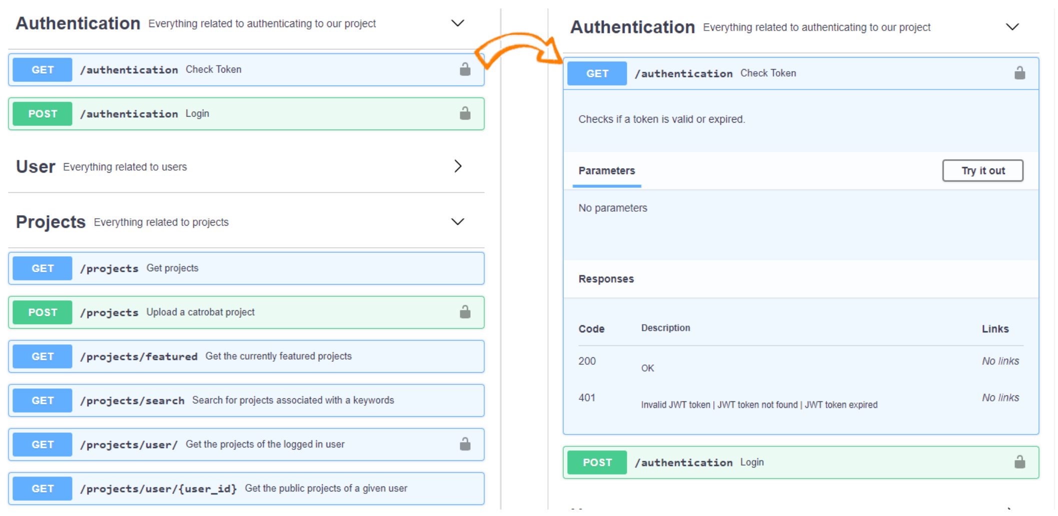


Travel to the skies to discover new pets, treasures and a new boss, find them all and defeat the boss!

Figure 1.3: Share community website: Its landing page can be seen in (a), a profile page in (b), a project page in (c), and the media library in (d).

to hundreds of graphics and sounds in the community platform's media library. Assets from the media library ease and speed up the project creation process. Furthermore, the platform provides users with recommendations, tutorials, statistics, and social features typical of a social network. An integrated building service even enables users to generate an Android Package (APK) from Catrobat projects. APKs can be installed on any Android mobile devices, independent of the Pocket Code app.

The Share community platform's website[6] can be included as a web-view in the apps or accessed using any browser with access to the internet. As an alternative, all data and services can be accessed directly via a RESTful application program interface (API). An API is described as RESTful when it implements the representational state transfer (REST) pattern. REST indicates the design and architecture constraints on how distributed systems can communicate with each other. A section of the publicly available API documentation written with the free and open-source Swagger[7] toolset can be seen in Figure 1.4.


Authentication Everything related to authenticating to our project

GET /authentication Check Token

POST /authentication Login

User Everything related to users

Projects Everything related to projects

GET /projects Get projects

POST /projects Upload a catrobat project

GET /projects/featured Get the currently featured projects

GET /projects/search Search for projects associated with a keywords

GET /projects/user/ Get the projects of the logged in user

GET /projects/user/{user_id} Get the public projects of a given user

Authentication Everything related to authenticating to our project

GET /authentication Check Token

Checks if a token is valid or expired.

Parameters

Try it out

No parameters

Responses

Code	Description	Links
200	OK	No links
401	Invalid JWT token | JWT token not found | JWT token expired	No links

POST /authentication Login

Figure 1.4: Swagger API: Catroweb's API specification is written in Swagger and rendered in a Swagger editor to ease the communication with other teams.

6. https://share.catrob.at visited on 9 Sep. 2020
7. https://swagger.io/ visited on 9 Sep. 2020

Catroweb is developed in an agile test-first environment, relying on the repetition of short iterative development cycles that require developers to write software tests before implementing a new feature. Furthermore, the Share community platform is mainly developed in the back-end and uses a web framework called Symfony to reduce development costs significantly. An indication of the work distribution is given by Figure 1.5. Software tests and test-first development are described in more detail in chapter 3. Chapter 4 provides more details about web development.

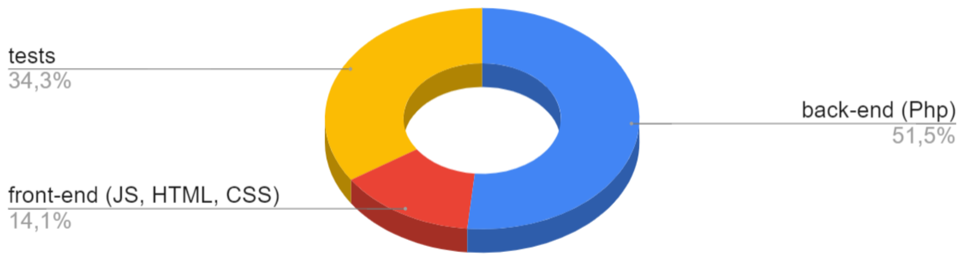


Figure 1.5: Source code analysis: 51.5% of Catroweb's code is written in PHP for the back-end. Only 14.1% is responsible for the front-end, namely JavaScript, CSS, and HTML. The remaining 34.3% empower Catroweb's software tests. Third-party modules, translation files, and configurations files are excluded from these statistics. (*Source code analysis of Catroweb* 2020)

1.2 Motivation

Apart from a handful of employed product owners and volunteers from all around the globe, mainly students of the TU Graz develop the software at Catrobat. Typically, those students only work on the project during their university courses. Contributions are made in irregular intervals by teams with constantly switching members (*Catrobat/Catroweb contributors* 2020). Figure 1.6 shows Catroweb's contribution timeline from GitHub.

During transition phases of old and new team members, expertise can get lost, especially when a team has no active members from the old guard left. Precisely this problem happened to Catrobat's Share community platform during 2018. Due to almost no documentation and the sheer size of the

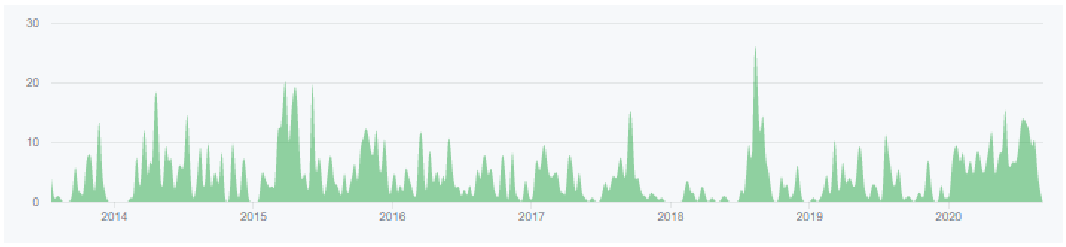


Figure 1.6: Contribution timeline from GitHub: The timeline shows Catroweb's irregular contributions intervals. The X-axis presents the years from 2014 to 2020, while the Y-axis indicates the number of commits to the project. (*Catrobat/Catroweb contributors* 2020)

project, it was unclear to new contributors which feature existed and how to set up and test the project properly. Developers had to invest many resources to maintain and extend the project.

Since then, Catrobat's web team has intensively worked on improving the documentation, test coverage, and code quality. However, without a CI system, every contribution to the project has to be checked by hand. Manual checks cost valuable time and are prone to errors. Using CI, every contribution must fulfill a particular standard since tests run automatically on a server. While a manual check is still necessary to ensure that every new feature is appropriately designed and tested, most redundant monotonous processes that elongate the reviewing process can be automated and checked independently from a human code reviewer.

Apart from automating processes during the review, development costs can be further reduced by automating various additional workflows, from continuous delivery to daily synchronizations with third-party platforms. Automating tasks removes their complexity and saves valuable time for developers. Besides, automated tasks require explicit instructions to work and produce implicit documentation about all project workflows, making them more accessible for future developers.

Therefore, introducing CI/CD combined with several automated workflows during this thesis, is a long-needed step to improve further and retain the Catroweb project's quality while also reducing the overhead for developers and code reviewers.

2 Software quality

The success of software products is directly related to their quality. How customers feel about a software product depends on a high degree on its quality. The better the quality of the software is, the more satisfied the customers are. However, typically customers care about other aspects than product owners or the development team. The software quality depends on the code's structure and how easy it is to maintain for developers. Maintainability is an essential factor in keeping the development going over a long period without exponential development costs. However, since the code itself is not visible to users, it is of no importance to customers. On the other hand, it matters to users that the software has high usability and works as intended and bug-free.

As one can see, quality is a subjective property that is highly conditional based on its receiver and domain. Banking applications, for example, must prioritize their security and reliability at all costs. A gaming application, on the other side, can neglect its security and focus on efficiency. Consequently, there exist various slightly different definitions and notations.

2.1 Defintion

The *ISO 8402-1986 standard* (2020) defines software quality as "the totality of features and characteristics of a product or service that bears its ability to satisfy stated or implied needs."

High-quality software satisfies all requirements and expectations of its customers while being secure, reliable, efficient, and easy to maintain. It is essential to translate the requirements and criteria of software quality into measurable characteristics. In a fast-paced world, needs and expectations continuously change. Dietmar Winkler (2018) realized that without adapting to the changes and the context of a domain, quality measurements only have limited value. Chappell (2020) defines software quality in three distinct but related aspects: functional, structural, and process quality.

2.1.1 Functional software quality

Software products of high functional quality imply that software works in the way it is intended to and meets various criteria to fulfill all expectations and requirements. Excellent performance and clean design with a focus on its usability are high prioritized. The usability defines rules to smooth the human-computer interactions and make them feel as natural as possible. A high usability product should be easy to use, provide feedback at all times, and allow its efficient usage. Functional software quality requires the product to be reliable and defect-free. While it is impossible to produce perfect software, it must be a high priority to prevent critical issues.

2.1.2 Structural software quality

Structural software quality defines how well the code itself is structured and, therefore, is often referred to as the code quality. The consortium for IT software quality (CISQ[1]) measures code quality based on security, reliability, performance, and maintainability.

2.1.3 Process quality

Process quality significantly affects the values received by customers and the development team. The process quality defines how well a project sticks to its delivery times and provided budget (Chappell, 2020). Various tools and methods enable developers to monitor and measure their development process. A high-quality process should be repeatable and continuously produce a reliable output.

1. `https://www.it-cisq.org/standards/` visited on 9 Sep. 2020

2.2 Software quality assurance

Tantawy (2009) describes software quality assurance (SQA) as a planned and systematic approach to improve software quality and ensure that the development adheres to established standards, processes, and procedures. SQA defines various models, such as the total quality management approach or capability maturity model integration.

2.3 Code quality in a learning environment

A team full of experienced developers may produce software of high quality even without a solid foundation. The development of new features is done in the highest possible quality, independently of the existing code. On the contrary, the quality of the existing code profoundly influences junior developers. At Catroweb, the main contributors are students trying to learn and improve, often lacking knowledge in the necessary domain and used technologies. Besides, they are easily overwhelmed by the sheer size of the project.

Observations among new contributors have shown that software quality is highly irrelevant to first-time contributors. Inexperienced developers learn from their environment and usually have trust in the existing codebase. Hence, corrupt coding practices are very likely to be adapted and copied by them. Their only focus aims to produce a somehow working implementation of the specifications.

Keeping a minimum level of quality at all times provides an environment where new team members can learn from the existing code. Contributors can be proud of their team, and the achieved work. Therefore, it is vital to keep the code quality as high as possible, even when this enforces slower development progress. The perfect mediocracy may be hard to find. However, for projects developed by inexperienced developers, it is better to be on the safe side and invest enough resources into the code quality and its assurance.

2.4 Refactoring

Developing a clean code of high quality can be a lingering process taking up many valuable resources. Typically, resources are limited, and whatever industry is concerned, every business attempts to maximize its investment. Up to a certain point, neglecting code quality can reduce development time and costs. Therefore, in the beginning, the output rate of features increases when code quality is not focused. Fixing flaws in the internal implementation, often hidden and not directly visible to users, tend to be postponed to the future until they fall into oblivion. In 2007, even the term ”Later == Never,” known as LeBlanc's law, was coined for this phenomenon (Cooper, 2020).

Nevertheless, code quality implicitly affects the overall software quality. Once a project reaches a specific size and complexity, without a certain minimum level of code quality, there is a high chance that the software is unreliable and challenging to maintain, debug, and extend, which significantly increases the development costs and time. At this point, it is necessary to refactor the existing codebase.

A refactoring can improve the code quality, but it comes with its risks and costs. A code refactoring aims to increase the readability and maintainability of a code section without changing the functionality. However, many new code lines have usually been added to a project since the implementation of problematic code sections. It is no rarity that old flawed code sections profoundly influenced the development of newer features. Therefore, it is typically a lot harder and cost-intensive to fix issues in hindsight.

3 Software testing

Software systems worldwide play an essential role in almost every aspect of our lives. From phones to medical devices and airplanes, software systems are everywhere. High-quality software is vital to provide users with a flawless experience and meet their expectations. However, it is highly unlikely that software is 100 percent bug-free. Nonetheless, at least mission-critical parts must not fail at any time. Already small bugs can result in severe issues. The recent history shows various examples of how extensive and costly the impact of bugs and vulnerabilities in software can be, ranging from crashed space rockets and airplanes to pipeline explosions and billions of stolen money (Harley, 2020).

A method to reduce the risks of software issues and increase its quality is called software testing. Extensive software testing is expensive. However, it is necessary to develop high-quality software products. It is essential to test the software before releasing it to the public and meet customers' expectations. Besides, in the long term, software testing significantly reduces the costs of development. Apart from preventing critical security issues that could damage a whole company's image, software tests significantly reduce maintenance costs.

3.1 Defintion

Software testing is a process to verify and validate a computer program to ensure high software quality products by providing means to reduce errors and cut overall software costs. Software testing assures that software is defect-free and meets all requirements by analyzing the software item (Isha, 2014).

The "Guide for Software Verification and Validation Plans" (1994) defines software testing as the process to detect differences between existing and required conditions. The testing process evaluates the features by checking for bugs, reliability, security, performance, and missing requirements.

In software testing, each independent test is referred to as a test case and describes a unique scenario to evaluate software functionality. A test suite is a collection of test cases, usually focused on a particular software aspect.

3.2 Classification

Software testing is a vast topic in software development, with several countless approaches of subdividing software tests into more specific categories based on various criteria. Figure 3.1 shows a common classification of software testing. However, there also exist different classifications with different granularity.

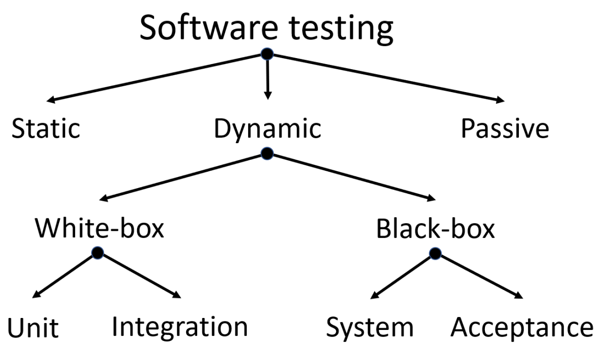


Figure 3.1: Software testing classification: A common approach classifies software testing into static, dynamic, or passive testing.

3.2.1 Manual or automation testing

Manual testing describes the process of verifying and validating by hand that software meets all requirements and is detect-free. Manual testing is a time-intensive task prone to human errors. It can be done from a customer's perspective as well as a developer's perspective.

On the other hand, automation testing uses tools and test scripts to automate the verification and validation processes to reduce testing costs significantly. However, not all aspects of software testing can be easily automated. For example, usability testing depends on real users' reactions that can not be automated.

3.2.2 Static, dynamic or passive testing

Static testing verifies a software product's form and structure by only looking at its source files without executing them. It evaluates the implementation of all possible paths in a program. However, it can not test the behavior of the software. Chapter 8 describes static analysis in more detail.

In opposition to static testing, dynamic testing is responsible for evaluating software behavior while the software is running. No access to the source code is necessary. In return, dynamic analysis is more expensive than static analysis and requires developers to write test cases to cover all test paths. Chapter 9 describes dynamic analysis in more detail.

An additional category describes passive testing, in which no direct interaction between the software and testers exists. Testers only evaluate log files and other monitored traces. (Andrés, Cambronero, and Núñez, 2011)

3.2.3 White-box or black-box testing

A software test is called a white-box test, or structural test, if it is designed to fit the system's internal perspective. White-box tests require access to the software's internal structure and source code. Therefore, white-box testing can be efficiently designed to cover all possible program paths. However, white-box tests require a high technical skill set and must be adapted if the internal implementation changes. (*White-Box Testing* 2020)

On the other hand, black-box testing, or behavioral testing, evaluates software behavior without knowing its inner livings. Black-box tests do not require programming language knowledge and can be independent of the developers. Therefore, black-box testing can avoid bias and expose discrepancies in specifications. However, black-box tests can be challenging to design. There is a high chance that a program path will remain untested or is tested multiple times at once. (*Black-Box Testing* 2020)

A combination of both black and white-box, called gray-box testing, can help develop overall better test cases.

3.2.4 Testing levels

To optimize dynamic software testing and reduce its costs, Cohn (2009) came up with a concept called the testing pyramid (Figure 3.2). The basic concepts of his work state that developers should write tests with different granularity. High-level categories must only consist of a few test cases, while most test cases should be low-level tests. Compared to high-level tests, low-level ones are usually more isolated and less complex, and cheaper to run. Higher-level categories, on the other hand, can be slow and complex and should be executed infrequently. (Fowler, 2020c)

Grouping software tests into levels empowers a systematic approach of applying software testing to a project. Developers and testers can identify

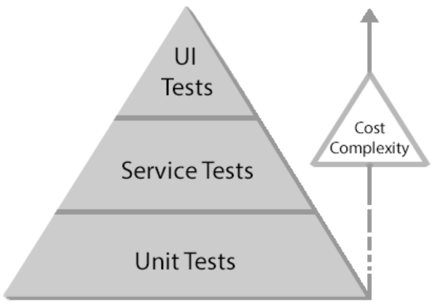


Figure 3.2: Testing pyramid: Cohn (2009) points out that developers should focus on writing many simple unit tests and only a few complex tests to reduce testing costs.

all possible test cases at a particular level quickly since it is strictly defined what a test must and what it must not contain to be part of a particular category. Based on a software test level, it is defined who is responsible for performing the test. Apart from that, the level indicates when the test should be executed during the software development life cycle. (*Levels of Testing in Software Testing* 2020)

The most common approach is to subdivide dynamic software tests into unit, integration, system, and acceptance tests (Figure 3.3). However, various additional testing types and approaches exist that usually can be subordinated to one of those groups. For example, usability testing, regression testing, stress testing, scalability testing, and functional testing are part of system testing.

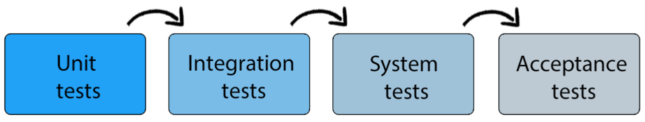


Figure 3.3: Testing levels: Software tests are grouped based on their level of details into unit, integration, system and acceptance tests.

Unit testing

Unit testing defines the process of testing the smallest possible "units" of testable code. Every component is considered a single system and tested isolated from other features to ensure that individual parts are working. A unit test must not depend on any other systems. A unit usually is either a function, method, or class. Unit tests are white-box tests that require access to the source code. They should be done by developers to detect bugs early during development. (Radcliffe, 2020)

Integration testing

Integration testing is done to check the interactions between unit tested components. While unit tests ensure that every module is working independently, integration tests ensure that the components fit together with no defects in their interfaces and produce the expected results. Typically, software testers perform integration testing after unit tests. (Itti Hooda, 2015)

System testing

System testing is part of black-box testing and requires a complete and fully integrated system to run the tests to verify that a software item matches its expected requirements by providing inputs and verifying the output. System tests evaluate overall interactions, usability, performance, reliability, and security. Based on the use case of software and provided tested budget, the applied system testing methods can vary. However, software testers usually perform system testing after the integration testing. (Itti Hooda, 2015)

Acceptance testing

The final layer in software testing, called acceptance testing or end-user testing, is responsible for guaranteeing that all requirements meet the customer's specifications and contracts. While members of the developing software organization perform internal acceptance testing, acceptance testing is usually done externally by end-users and customers. (Itti Hooda, 2015)

3.3 Test-first development

Software tests can be developed independently of new features. However, software testing can only detect bugs and issues if there are sufficient tests to cover all possible program paths and specifications. A best practice to develop software tests is called test-first development. A test-first approach requires developers to write tests before actually implementing a new feature. Figure 3.4 shows an abstraction of the workflow during test-first development.

The most significant benefit of test-first development is that projects are easier to maintain and extended due to high test coverage. Apart from high coverage, a test-first approach forces developers to clearly define the problem and specify the requirements in a detailed fashion before starting to work on a new feature.

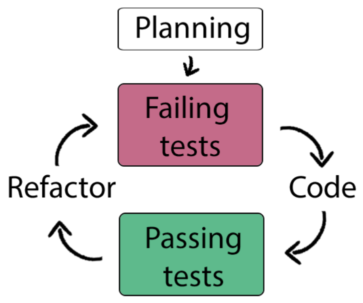


Figure 3.4: Test-first development: Developers write tests before implementing a feature. Code is only developed as long as tests fail.

3.3.1 Test-driven development

As Beck (2002) summarizes test-driven development (TDD), new code lines must only be written if an automated test fails. TDD is a test-first approach relying on short development cycles to write software tests to satisfy developers' needs and reduce maintenance costs. TDD tests are written using regular code and can be easily automated using automation frameworks like PHPUnit (Listing 3.1). Automation frameworks are described in more detail in section 9.1.

```php
public function testNameMustNotContainARudeWord(): void
{
  $user = $this->createMock(User::class)
    ->expects($this->atLeastOnce())
    ->method('getName')
    ->willReturn('rudeword');
  $this->expectException(RudewordInNameException::class);
  $this->validator->validate($user);
}
```

Listing 3.1: PHPUnit test: In TDD, tests are created using regular code.

3.3.2 Behavior-driven development

Behavior-driven development (BDD) evolved from TDD and is also a test-first approach. BDD tests are explicitly designed to satisfy both developers and customers. Customers do not care about the details of implementation. Therefore, BDD checks the actual behavior from the end-user perspective rather than the implementation (Nair, 2020).

BDD features are written in human-readable sentences by introducing an additional layer of abstraction to increase software tests' accessibility to non-developers. Listing 3.2 shows an example of a BDD feature description using Behat, which is a framework for auto testing business expectations. The real logic to test a feature description is hidden in separated files using regular code. An example is given by Listing 3.3. However, not all tests have to be relevant to customers. Therefore, it can be quite useful to use both TDD and BDD in a project to get the most benefits.

```
Scenario: Welcome section
  Given I am on the homepage
  Then I should see the welcome section
```

Listing 3.2: Behat scenario: In BDD, tests are written in a business readable language.

```
/**
  * @Then /^I should see the welcome section$/
  */
public function iShouldSeeTheWelcomeSection(): void
{
  Assert::assertTrue(
    $this->getSession()
      ->getPage()
      ->findById('welcome-section')
      ->isVisible()
    );
}
```

Listing 3.3: Behat context: Logic is hidden and written using regular code.

4 Web development

Software Development refers to all activities dedicated to the process of creating, designing, deploying, and supporting software (IBM, 2020).
The development of web applications, web services or websites, is usually referred to as web development. Web development is tightly connected to the World Wide Web (WWW) and shaped by its client-server architecture. Therefore, web development is generally divided into back-end and front-end. In a nutshell, the front-end is all about the look and feel of a web page. On the other hand, the back-end is responsible for the machinery in the background to make a website work.

4.1 World Wide Web

The Internet is a global network of networks connecting billions of computers that enables users worldwide to access various services. Like emails and file transfers, the WWW, also known as the web, is one of those services provided over the Internet. The WWW is a massively distributed information system that stores information in hypertext documents referred to as web pages.

A website is a collection of web pages grouped and linked together under the same domain name. Each web page has a unique identifier (ID) assigned to it and is connected to other web pages. Websites are hosted on computers, called web servers, that are dedicated to receiving, processing, and responding to requests from clients on the Internet. Clients can navigate the web and access web pages that are served by any web servers connected to the network using a web browser.

4.1.1 History

Back in 1989, there were already millions of computers connected through the fast-developing Internet. However, there was no way to share the information which was distributed between multiple computers easily. A fact that was driving Tim Berners-Lee to invent the WWW. Everything started with his vision proposal, to easily share information between computers, in a document called "Information Management: A Proposal."
By the end of 1990, Tim Berners-Lee already developed the WWW's fundamental technologies, resulting in the first web page served by a web server and the first web browser to visit the page. Nevertheless, at the beginning of the year 1993, there were only 50 web servers online. First, to reach its full potential, it was necessary to release the underlying code for free to the public, resulting in a global boom of the WWW with more than 500 web servers that were already online at the end of the year. (Foundation, 2020)

Today the WWW counts more than 4.5 billion users (Group, 2020). At its first stage, between 1991 and 2004, the WWW was mainly filled with static pages. Few content creators controlled a web page, and rarely changed its content. Over time, web pages became more and more dynamic, and the term web 2.0 became popular. Dynamic web pages allow active participation by embedding user-generated content, like comments, directly on a web page. Building dynamic and responsive web applications require more effort than static pages. However, it provides customers with better experiences.

4.1.2 Fundamental technologies

Modern web applications are crammed with as many features as possible. With the steadily increasing users' expectations, even small websites are built on a large complex codebase, relying on millions of code lines for the underlying architecture. Still, even 30 years later, today's web is built on the same fundamental technologies (Foundation, 2020).

Uniform Resource Identifier

A Uniform Resource Identifier is a string of characters to identify a resource. The most used form, the Uniform Resource Locator (URL), is typically referred to as a web address. URLs contain information about how a resource can be accessed and where it is located in the network. For example, the URL pointing to the Catroweb repository on GitHub is defined as follows.

$$\underbrace{\text{https:}}_{\text{Scheme}}//\underbrace{\text{github.com}}_{\text{Domain name}}/\underbrace{\text{Catrobat/Catroweb}}_{\text{Resource identifier}}$$

The first part of an URL is called the Scheme. The Scheme defines the protocol and methods used to localize and process the resource requested by the remaining URL parts behind the colon. In the Catroweb repository example, the Scheme requires the transmission to use HTTPS, which stands for Hypertext Transfer Protocol Secure. The next section describes this fundamental communication protocol in more detail.

Each device connected to the Internet has a unique Internet Protocol (IP) address. The IP is a commonly used state-less communication protocol to address computers, like web servers, in a network, and send IP-packets. Since IP addresses are hard to remember for humans, domain names can replace a unique IP address assigned to the domain. For example, the IP address behind the domain name 'github.com' is '140.82.114.3'[1]. A domain name system does the conversion between a human-friendly domain name and a computer-friendly IP address in the background. Hence, users only need to type and remember a human-readable name. Moreover, security certificates are only generated for a hostname and not an IP address. Consequently, most browsers provide a warning if IP addresses are used rather than domain names.

The final part of the URL, 'Catrobat/Catroweb,' defines the path to the resource on the web server and is necessary when a specific page of a website is requested. Typically, a website is under one domain but provides access to many different web pages.

1. checked on 4 Sep. 2020

Hypertext Transfer Protocol

”The Hypertext Transfer Protocol (HTTP) is an application-level protocol for distributed, collaborative, hypermedia information systems.” (*RFC2616* 2020)

Every HTTP interaction consists of a request and a response. A client, typically a web browser, requests data and services from a web server by sending a request message to the server. Once the server processes a client's request, the web server sends a response message back to the client who initiated the request (Figure 4.1).

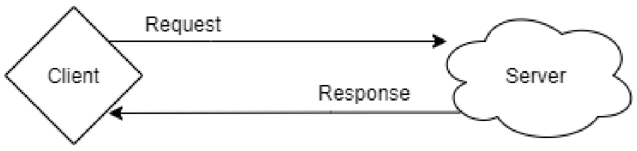


Figure 4.1: Client-server model: A client requests information from a server. A server provides the response to the client.

HTTPS is an extension of HTTP, in which the communication is encrypted to protect users and their data from malicious third parties eavesdropping on sensitive information.

Hypertext Markup Language

The Hypertext Markup Language (HTML) is a markup language to structure the content of a website. A markup language uses tags to define elements within a document in a human-readable format. For example, HTML defines which part of the content represents the title of a web page.

Apart from standards and protocols to ease the transmission process, browsers require specifications to render the content of billions of independent web pages uniformly. HTML is under continuous development to comply with web developers' needs since its release. Therefore, there exist numerous different versions. Since 1994, the World Wide Web Consortium[2]

2. https://www.w3.org/ visited on 9 Sep. 2020

(W3C) has developed specifications and guidelines around the WWW. (W3C, 2020)

A web server can send any content to the client. However, using standards ensures that browsers can understand and interpret the received information. Due to HTML specifications, browsers precisely know how to process the provided information of HTML web pages to present them to a client. However, HTML does only define the structure of a web page and not its style. Therefore, web servers usually also deliver images, Cascading Style Sheets (CSS), and client-side scripts.

Nevertheless, the content could be in any of the countless HTML versions or even entirely different formats. Different browsers and different versions of browsers can support different standards. Only as long as a browser can interpret the response correctly, the information can be rendered.

4.1.3 Back-end

A web server is a computer program that stores, processes, and distributes requested web pages using HTTP. However, the back-end defines more than just the web server itself, namely everything that runs directly on a server, such as server-side scripts and the database. That is also why the development in the back-end is often referred to as the server-side development.

Database

The back-end usually includes a database to store dynamic data, like user names and encrypted passwords. It enables web servers to remember and structure information that needs to be accessed later on. A database and its stored data can be easily accessed, managed, modified, updated, controlled, and organized (Oracle, 2020).

Server-side scripting

Server-side scripts are responsible for the communication between the server, the application, and the database. As the name suggests, server-side scripts are executed directly on a web server. Server-side scripts are responsible for processing and validating requests from clients forwarded by the web server. The resulting response is passed on to the web server to send it back to the client. Server-side scripts enable a web server to create a dynamic response for its clients. For example, the scripts can fill templates with data from the database, such as user names, optimized to any region, language, or time.

There are many different server-side languages available, each with its unique characteristics. Catroweb's server-side scripts, for example, are mainly written in PHP. Originally PHP was the abbreviation for Personal Home Pages. Nowadays, it stands for Hypertext Pre-processor. While PHP is a general-purpose scripting language, it is especially suited for web development where it serves static or dynamic content to clients. Based on *PHP usage statistics for websites* (2020), PHP is by far the most used server-side programming language with a usage of over 75% of all sites on the web. The code written in PHP is wholly processed on the server. Instead of transmitting the source code to the client, an interpreter processes all instructions at runtime on the server before the web server serves the client's response.

4.1.4 Front-end

The front-end, or client-side, defines everything directly visible to a user. A client only interacts with the front-end, empowered by the back-end running in the background. In a best-case scenario, the back-end is invisible to a client. The front-end is responsible for structuring the content delivered to the client using HTML. A web page's content can be styled using CSS and brought to life using client-side scripts to keep a page responsive and vivid.

Cascading Style Sheets

Amongst other things, CSS is responsible for layouts, fonts, colors, and element sizes. The strict separation of concerns between style (CSS) and structure (HTML) has many positive side effects. For example, CSS files enable different presentations for the same content, optimized for different output devices, from small screens on smartphones to large desktop screens. The same style sheet can also be applied to multiple web pages, highly reducing the overhead of defining the same styles repeatedly.

Client-side scripting

In contrast to server-side scripts, a web server does not execute client-side scripts. Instead, they are transmitted to a client, usually embedded in a web page. Client-side scripts are executed directly in a client's web browser, working independently from web servers once a web page is loaded. The most used client-side programming language, also used by the Catroweb project, is called JavaScript.

Client-side scripts, like JavaScript, provide the basis of building fast and responsive web pages. They enable a web page to change its content and design dynamically or even create animations and complex calculations. With client-side scripts, a web page can dynamically reload data in the background. Therefore, it is possible to update only parts of a web page instead of reloading the whole page. Client-side scripts can provide users with immediate feedback about their actions and increase user experience.

Nevertheless, client-side scripts do not perform well in all scenarios. For example, validation can be done on the client-side to give users the first feedback about their inputs. However, validation must be implemented on the server-side to ensure that the validation is not bypassed. Since client-side scripts are executed on the client-side, a client can easily modify the scripts.

4.2 Web framework

While web applications can be very different, they are very similar at their core. Developers could implement every line of code by themselves. However, reusing existing code is improving development speed and is reducing the risks of repeatedly reintroducing bugs for the same problems. Hence, developers can focus on actual business-related requirements and concerns.

A framework does not provide a complete application, but a starting point to any project to speed up its creation and maintenance. There exist many different frameworks for all kinds of domains and languages, from microframeworks to fully-featured ones. A framework has a strong influence on how a project is developed and can determine a project's architecture and used tools. For example, the Share community platform is built with the Symfony[3] framework, designed to do most of the work in the back-end using PHP. Other frameworks may use different languages or are optimized for the work in the front-end. Therefore, choosing an appropriate framework for a project is essential.

Symfony is a high-performance web framework for the development of websites and applications. It is a set of reusable PHP components to enable developers to quickly create flexible and reusable web applications following best practices and standards on top of the Symfony components. Furthermore, the Symfony framework has an active community with over 600.000 developers. (Symfony, 2020)

3. `https://symfony.com/` visited on 9 Sep. 2020

5 Version Control Systems

Software is in constant change. New lines are added to the source code, while old sections get modified or removed. Being able to switch and maintain multiple versions of a project is inevitable. In a typical software project, various developers work on the same source code simultaneously. Specific tools and mechanics have been invented to tackle all kinds of problems while maintaining large software projects to ensure smooth collaboration and software maintenance.

A version control system (VCS) tracks and stores every modification of a file system with additional metadata, like version number, author, notes, and timestamps in a database. Moreover, it provides a complete long-term history of all changes. It is possible to revert any recent changes and rollback to older versions of the software without a hitch in case of mistakes.
Furthermore, a VCS offers countless tools to enable efficient teamwork while developing and maintaining software products, such as branching and merging. Therefore, software development teams of all sizes benefit from a VCS.

”While it is possible to develop software without using any version control, it subjects the project to a huge risk that no professional team would be advised to accept. So the question is not whether to use version control but which version control system to use.” (Atlassian, 2020)

5.1 Git

There exist several different VCSs with their unique pros and cons. One of them is Git, a free and open-source VCS widely used in professional software teams due to its ease of use and efficiency.

In contrast to other VCSs, like Apache Subversion, Git is a distributed system and not centralized. Every developer, who uses Git, owns a local copy of a project, including its whole history of changes. To put it differently, every developer operates on their clone of a project. Hence, most actions can be applied without an active Internet connection. A network connection is only needed when developers push their changes or fetch other developers' changes from a repository hosted on a server.

Git only saves the current state of a project when changes are committed. Therefore, developers are encouraged to regularly commit their work, even if it is still work in progress. The history of a Git project keeps track of every commit. A commit is a snapshot of a project, including additional metadata. The required space is kept at a minimum since Git only stores the differences between two commits while also applying powerful compression algorithms (Scott Chacon, 2020).

Maintaining a detailed project history comes with many benefits. For example, it is possible with specific techniques and commands[1] to find the exact version which introduced a bug. Developers can rollback to any commit at any time by removing all subsequent commits. Commits are only pushed to the server after an explicit push command. When pushing commits to the server, other developers can pull those changes into their local repositories. Once commits are pushed, they should not be deleted anymore because other developers could rely on them. A better alternative for developers is to revert a specific commit without tampering with history. Git can automatically create a new commit inverting all changes of the targeted commit.

Amongst other things, Git supports non-linear development using a branching system. Non-linear development encourages developers to modularize their development process and utilize powerful workflows. For example, multiple branches allow developers to keep a stable version, also referred to as mainline, throughout the development. Hence, developers branch away from the stable version to start working on new features, instead of working on the mainline. By that, multiple different features can be worked on and experimented with, even by various developers, at the same time without getting influenced by the work on other branches. Finished feature

1. `https://git-scm.com/docs/git-bisect` visited on 9 Sep. 2020

branches are then integrated back into the mainline, where the branch once originated. Section 5.3 describes the benefits of workflows in more detail.

Furthermore, Git provides all the necessary tools to merge contributions. From time to time, automatically merging two different branches is impossible. For these so-called merge conflicts, a code merger can manually pick the changes which should be applied. Fortunately, the differences can be explicitly highlighted, which is a huge time saver during reviews.

5.2 GitHub

Git itself is just a command-line tool. GitHub, on the other hand, is a Git repository hosting service to share code and publish services. It is basically like a cloud but optimized for software projects. There are many similar services, such as GitLab and Bitbucket, but with unique strengths and business models. This thesis only focuses on GitHub since all Catrobat teams already use it.

A repository defines the location of all hosted files of a specific project and can be accessed over a unique URL[2]. It can be either public or private. While public repositories are visible to everyone, private repositories can only be seen and found by users explicitly invited by the owner. Private repositories can cost quite some money, depending on the claimed services. Public repositories, on the other hand, can use most services without additional costs. GitHub attaches great importance to supporting open-source projects. Further, a repository can be configured as required. For example, a repository owner can manage access rights, enable branch protection for important branches, or set up project secrets.

GitHub also provides developers and their teams with many features. For example, a web-editor with syntax highlighting, task management tools like a project board and an issue tracker, and a wiki. Somehow it is even a social network for programmers. One of their newest features, called GitHub Actions, will be extensively discussed in section 6.4.2.

2. `https://github.com/Catrobat/Catroweb` visited on 9 Sep. 2020

To ease the review process before merging, developers can create pull requests containing their modifications. A pull request is visible to anyone who has access to the project's repository. While every developer can review the changes, only those with write access can merge them. A pull request is a perfect place to discuss proposed changes and request additional changes if necessary.

5.3 Git workflows

Over the years, powerful Git workflows have been enforced using the VCS Git. Based on the project type and team size, it can alternate which one fits the best.

5.3.1 Single branch workflow

For small projects, a basic workflow where developers push all their changes consecutively on the same branch can be sufficient. A basic single branch workflow is given by Figure 5.1. Primarily when developers work alone on a project, there is often no need to introduce the complexity of more powerful workflows.

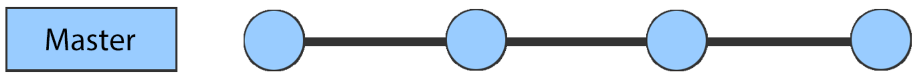


Figure 5.1: Single branch Git workflow: A basic Git workflow is using a single branch, typically called the master branch. In this example, the master branch consists of four consecutive commits. Each commit represents a snapshot of the project.

5.3.2 Feature branch workflow

However, more complicated workflows can have various benefits. For example, they can prevent the integration of bugs and unfinished features into a stable version, highly increasing the quality of life for developers. Figure 5.2 shows a feature branch workflow, which encourages developers to only work on feature branches instead of the master branch. Through multiple branches, developers can work in parallel and do not block each other during development.

A feature branch is created for every new issue and merged back into the master branch when its development is finished. The code written in one feature branch is independent of the code written in a different feature branch. Therefore, developers can regularly commit and push their changes to a feature branch, even if their work is not finished. Possible new errors will not affect other feature branches. The same feature even can be implemented and experimented with in different ways on different branches. Due to its simplicity and efficiency, this workflow is very common in software teams of all sizes.

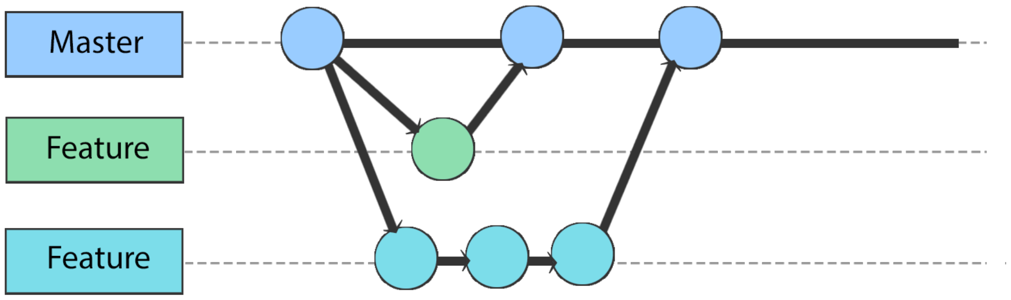


Figure 5.2: Feature branch Git workflow: Developers do not work on the master branch. Every new feature is developed in its separate feature branch. Finished features are merged back into the master branch.

5.3.3 Git-flow workflow

While there are several additional complex workflows to fit large project's requirements with many contributors, Driessen (2020) describes a commonly accepted workflow in large projects. This Git-flow takes advantage of two long-running parallel branches extended by multiple short-lived branches. Its representation can be seen in Figure 5.3.

A stable version of the project is available through a branch typically called the master or main branch. Developers must not directly work on this branch. Therefore, it is essential to protect the master branch and remove developers' write permissions to prevent commits to this branch. This stable version usually represents the state of the deployed production services and must only be updated during a new release or with a hotfix branch if necessary.

Serious problems, like security problems that somehow were integrated into a stable version, can be patched independently from the current development state with a so-called hotfix. A hotfix branch branches off from the

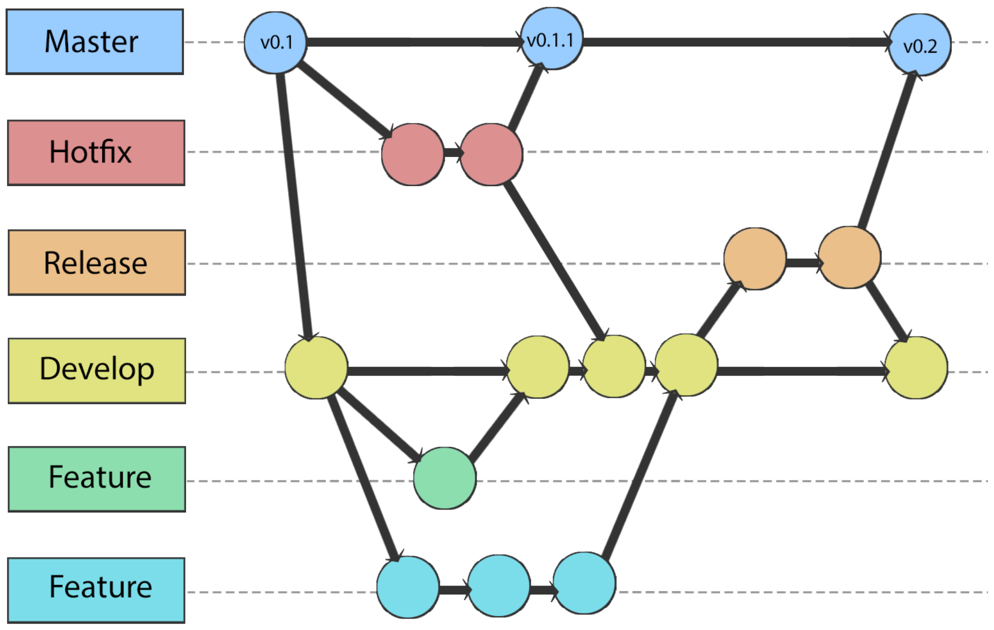


Figure 5.3: Git-flow workflow: This Git workflow utilizes its power of various branches. The master branch represents a stable version of the project at any time. The develop branch reflects the current state of development. With a hotfix branch, severe problems can be directly patched to the master branch. New features are developed in independent feature branches until their development is finished. Finished feature branches can be integrated back into the develop branch. When enough features are implemented, a release branch can be created. After excessive testing, a release branch is merged into the master and develop branches.

master branch, and as soon as the patch is implemented, the hotfix branch can be integrated back into the master branch. A hotfix allows developers to react immediately to severe problems, without the chance of introducing new issues due to new features currently in development.

Furthermore, new features are often still in progress and are not ready to be shipped to the public. Instead, all new features are hosted in the develop branch. As the name indicates, this branch reflects the current state of the development. Still, developers should not directly work on this branch. Instead, every new feature is developed in its unique branch or branches, similar to the described feature branch workflow. A feature branch branches off from the develop branch and is only integrated back into the develop branch when a feature is done. To ensure the quality of the work, developers usually do a code-review before the merging process.

5.3.4 Forking git-flow workflow

Catrobat teams use a forking workflow optimized for open-source projects in extension to the described git-flow workflow. Only a handful of developers, so-called maintainers, need write access to the repository. Technically, one maintainer per repository is sufficient. Controlled rights ensure that a repository structure is kept clean and protect a project from unwanted modifications while still being publicly accessible by anybody. Usually, this public repository is called the official repository.

Every developer must fork the project first instead of directly cloning it to a local machine. Forking creates a personal copy of the repository on the server, which then can be cloned onto a local machine. Having forked repositories ensures that one developer's work does not interfere with the work of others. Developers have no write permission to the official repository. However, they can still create their feature branches or experiment freely in the codebase. When a feature is finished, developers commit and push the changes to their forked repository. As a next step, they can create a pull request from the feature branch in their repository to the develop branch on the public repository.

Finally, a maintainer of the official repository can review this pull request to request changes, decline it, or merge it into the official repository. Developers can pull the changes of newly merged contributions into the official repository, into their forked repositories.

6 CI/CD

In software engineering, CI/CD, short for continuous integration (CI) and continuous delivery and deployment (CD), is a collection of methods, tools, and practices to develop and deliver high-quality software continuously.

Throughout the software development life cycle, CI/CD builds a bridge between development and operation activities to reach the highest possible degree of automation (Figure 6.1). Once successfully introduced and set up in a project, CI/CD highly reduces the development time and costs.

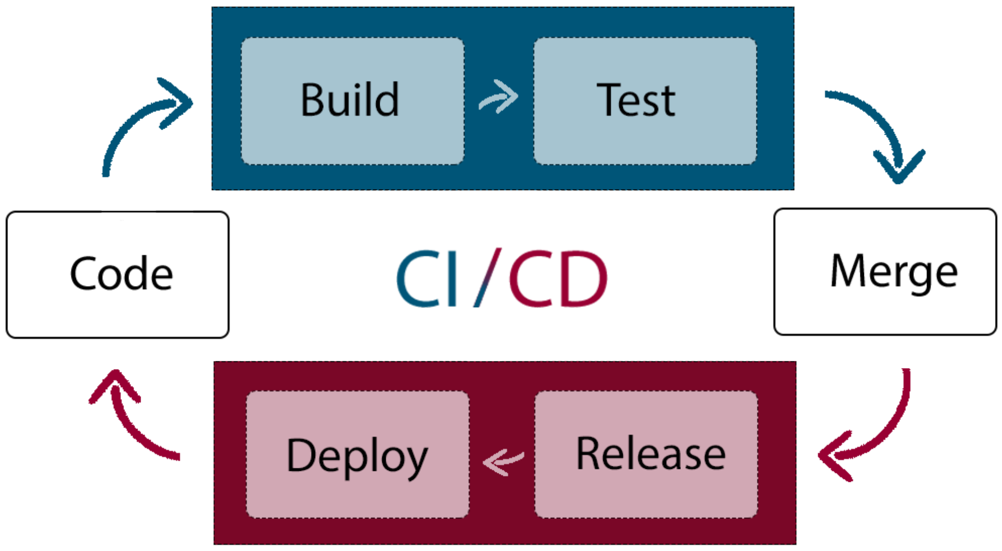


Figure 6.1: CI/CD software development life cycle: CI automates the build and test workflows at every integration step. Subsequently, CD extends the automation to the release and development process.

CI aims to lower the number of integration problems by requiring developers to frequently integrate their changes into a shared repository. A dedicated system and its tools verify every contribution by building and testing the project before its integration. CD adds additional methods to release and deploy new software versions to customers automatically.

The automated CI and CD workflows usually are connected and are referred to as CI/CD-pipeline. This chapter describes the theoretical background to CI/CD and its tools, while chapters 7, 8, 9 describe the steps of implementing a CI pipeline in the Catroweb project using GitHub Actions. Chapter 10 describes the second part of the pipeline responsible for CD. Further, chapter 11 automates various additional workflows.

6.1 Continuous integration

In software projects, developers work independently and simultaneously on different features of the same product. At some point, the components must be reunited to form a single product. CI defines various methods and requirements to reduce software integration costs. The following sections briefly summarize the criteria specified by Fowler (2020a).

6.1.1 Integration

CI requires every part of a project to be stored in a single repository. All developers must have immediate access to all features. Traditionally there exist no rules about the merging process. However, a VCS eases a rational development and integration process and is an essential requirement for every project to apply CI. The main branch is the access point to check-out the most recent version of the project. After developers finish the work on a feature, their feature branch is merged back into the main branch.

Nevertheless, even with a VCS, the integration progress can be a lengthy and error-prone task. The chance of conflicts rises proportionally to the time passed. Every day many different lines of code are added and modified by simultaneously working developers. CI reduces the risk of conflicts during merging. Instead of a few extensive integrations, the merging process is divided into multiple small integrations. Following the spirit of divide and conquer highly reduces the complexity and time expenditure of the integration task. Developers can quickly detect conflicts and issues. Hence, they can react to them early while they are still easy to resolve. As a positive side-effect, developers break their work down into smaller junks, which are more comfortable to debug. Besides, the development process is continuously communicated between developers.

6.1.2 Verification

Another important aspect of CI requires the main branch to be in a healthy state at all times. Every contribution must be verified automatically before its integration. Automating the verification steps is necessary to save valuable resources and prevent human mistakes. A single command should be sufficient to build and launch a system. Building a system can be done with or without testing code. However, working builds do not guarantee that the software is working as intended. Hence, every contribution must be able to test itself.

Software tests do not ensure that software is perfect. However, even imperfect tests can catch many problems when executed properly and regularly. Test suites should be automated to check large parts of the codebase by running a single command. Besides, an automated test must indicate its failure. To increase the informative value of tests, they should be executed in an environment identical to the production environment.
Automated dynamic software tests require developers to develop test cases manually since features without corresponding tests can not be checked for their functionality. Therefore, CI requires a test-first development to force developers to integrate new changes and their tests simultaneously. Code coverage metrics presented in the feedback can assure that developers stick to test-first development.
Furthermore, CI requires test cycles to be kept short. Developers are not willing to wait hours for the tests to be finished. Too long test runs result in infrequent integrations by developers. It is essential to find a balance

between quality assurance and run time. In some cases, it can be useful to execute certain unimportant test cases only after the integration.

6.1.3 Accessibility

Non-developers, like product owners, require easy access to the results of the software development. They rarely find use in the source code itself. Instead, it is preferred to provide an automatically generated report about the quality of the contribution. The feedback result should contain every failing issue and various metrics of all kinds. It should be easily possible to compare the quality of the software before and after an integration.

6.2 Continuous delivery and deployment

CD extends CI with methods and principles to automatically and reliably deliver new software releases to customers in short cycles. A project's delivery process has to be fully automated. Otherwise, too short release cycles waste valuable resources.

A CI/CD server to automate software delivery and deployment removes the necessity of complicated setups and processes. Hence, the required knowledge and time to deploy a new release is significantly reduced. Even non-developers like product owners can quickly release new versions within seconds. A click is enough to initiate the release process. Furthermore, CI/CD requires all CI tests to pass to initiate CD, and therefore a failing CI prevents a new release. Hence, a reliable CI system combined with a high enough code coverage in software tests is essential to apply CD successfully.

Both continuous delivery and deployment are very similar approaches. However, continuous deployment takes automation one step further than continuous delivery. In continuous deployment, a release is initiated without direct human interaction. Every integration is automatically deployed to production. Continuous delivery, on the other side, provides a manual trigger to start the automated workflow.

As a result of using continuous deployment, new releases are shipped daily instead of once every month. There is no need to pause the development for a new release. Continuously pushing small releases to the public provides a higher frequency of user feedback. However, for this approach to work, automated tests must be of the highest possible quality. Only a failing test in the pipeline stops the deployment process. On the other hand, continuous delivery still allows developers to control what should be released and when.

6.3 Benefits and drawbacks

To successfully apply CI/CD's principles to a project, developers must adjust their practices and workflows. Also, a CI/CD pipeline requires dedicated infrastructure and tools to be set up and maintained. In return, CI/CD significantly lowers the risk of software development and its costs. Therefore, in large projects, the upfront investment to use CI/CD is usually worth it. The following key points define the main benefits:

Continuously integrating small patches prevents hasty stressful merging processes of incompatible versions shortly before a new release. Issues can be easier tracked down and isolated in small patches. Besides, the frequent commits build a granular history, clearly communicating the progress in development.

Several metrics measure every contribution's quality and provide extensive feedback, allowing developers to quickly react to issues and conflicts. Thoroughly checking every contribution prevents integration problems and issues from being merged. Therefore, there exists a stable build of the software at all times.

Automated workflows increase accessibility by removing complexity and reduce the overhead to developers. Developers can do more in less time and focus on the creation of new business-relevant features. Furthermore, shorter release cycles result in more frequent feedback from users and allow software teams to react more quickly to marketing conditions.

6.4 Tools

Theoretically, CI/CD can be managed without any additional tools. Nevertheless, manually setting up the CI/CD infrastructure from scratch can be a cumbersome task. Fortunately, there already exists a broad set of automation tools. Several lists on the internet try to rank and describe tools and programs in the field of CI/CD. For example, Stackify[1] alone lists more than 50 tools. However, describing all of them is beyond the scope of this work.

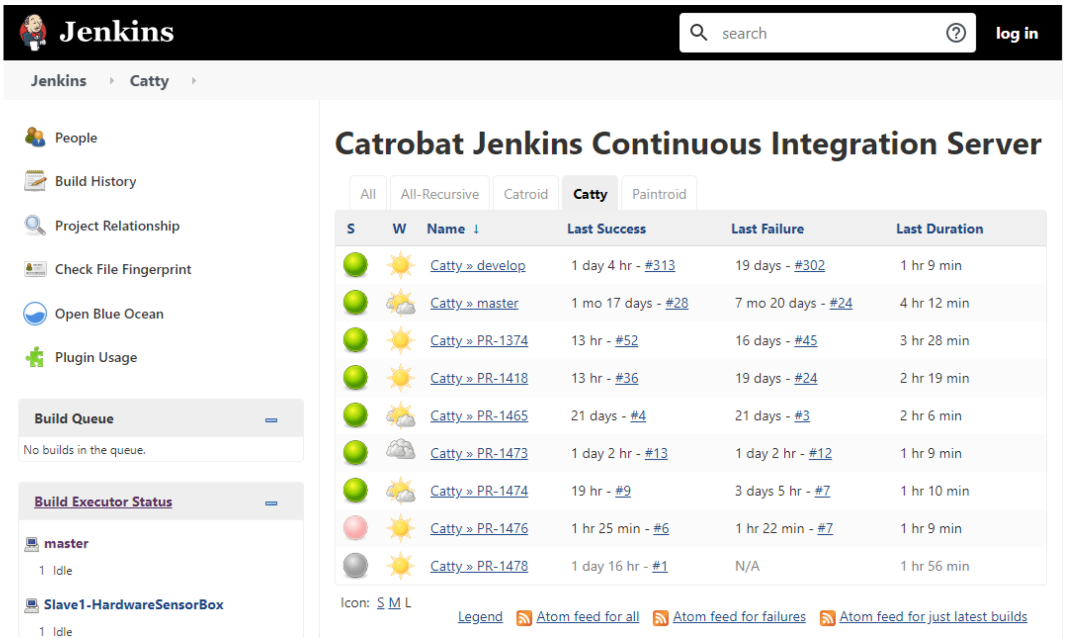
The Android and iOS Catrobat teams have already successfully applied CI/CD using a tool called Jenkins. However, this thesis's practical implementations are built using a new tool to realize CI/CD, namely GitHub Actions.

6.4.1 Jenkins

Jenkins[2] works as a self-hosted open-source CI/CD server with hundreds of plugins to provide a quick and robust chain of build, test, and deployment tools. Stackify (2020) lists Jenkins as the number one open-source project to automate a project. The tool is a self-contained platform-independent Java-based program ready to run out of the box. Developers can build and configure jobs with scripts to automate any process. Additional plugins extend the possibilities of Jenkins without the need to create custom scripts.

A built-in web-interface and RESTfull API provide access to the services. The web-interface contains valuable feedback about running and finished tasks. Besides, developers can adapt configurations and rerun pre-defined tasks through the interface. Catrobat's Jenkins web-interface[3] is publicly available and provides access to detailed feedback for every CI/CD task. Figure[4] 6.2 provides an example of the feedback from Catrobat's iOS development.

1. `https://stackify.com/top-continuous-integration-tools/` visited on 9 Sep. 2020
2. `https://www.jenkins.io/` visited on 9 Sep. 2020
3. `https://jenkins.catrob.at/` visited on 9 Sep. 2020
4. `https://jenkins.catrob.at/view/catty` visited on 9 Sep. 2020



No builds in the queue.

master
1 Idle

Slave1-HardwareSensorBox
1 Idle

Catrobat Jenkins Continuous Integration Server

All | All-Recursive | Catroid | **Catty** | Paintroid

S	W	Name ↓	Last Success	Last Failure	Last Duration
🟢	☀	Catty » develop	1 day 4 hr - #313	19 days - #302	1 hr 9 min
🟢	⛅	Catty » master	1 mo 17 days - #28	7 mo 20 days - #24	4 hr 12 min
🟢	☀	Catty » PR-1374	13 hr - #52	16 days - #45	3 hr 28 min
🟢	☀	Catty » PR-1418	13 hr - #36	19 days - #24	2 hr 19 min
🟢	⛅	Catty » PR-1465	21 days - #4	21 days - #3	2 hr 6 min
🟢	☁	Catty » PR-1473	1 day 2 hr - #13	1 day 2 hr - #12	1 hr 9 min
🟢	⛅	Catty » PR-1474	19 hr - #9	3 days 5 hr - #7	1 hr 10 min
🔴	☀	Catty » PR-1476	1 hr 25 min - #6	1 hr 22 min - #7	1 hr 9 min
⚪	☀	Catty » PR-1478	1 day 16 hr - #1	N/A	1 hr 56 min

Icon: S M L

Legend | Atom feed for all | Atom feed for failures | Atom feed for just latest builds

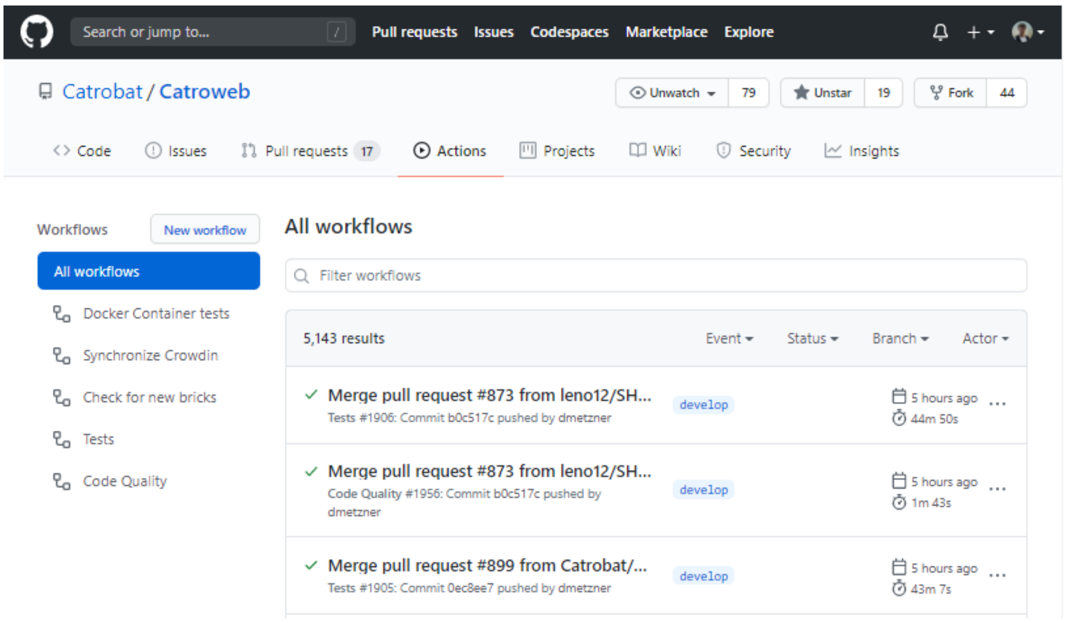
Figure 6.2: Jenkins feedback: The feedback provided by the Jenkins web-interface gives a clear indication about passed and failed CI/CD tasks.

6.4.2 GitHub Actions

GitHub Actions[5] were released by GitHub in 2020 to automate any workflow during the software development life cycle. GitHub Actions are directly integrated into source repositories on GitHub. Figure 6.3 shows the GitHub Actions dashboard[6] of the Catroweb repository. Workflows to build, test, and deploy are directly located in the repository. Even the pipeline feedback is directly integrated into GitHub's web interface, and GitHub Actions are optimized to work with events specific to GitHub.

GitHub Actions are still new on the market. However, the tool and its community are snowballing, especially since it is free to use for open-source projects. On the other hand, each GitHub account only receives a certain amount of free minutes and storage for private repositories. No additional

5. `https://github.com/features/actions` visited on 9 Sep. 2020
6. `https://github.com/Catrobat/Catroweb/actions` visited on 9 Sep. 2020



All workflows

5,143 results

✓ Merge pull request #873 from leno12/SH...	develop	5 hours ago	44m 50s
Tests #1906: Commit b0c517c pushed by dmetzner			
✓ Merge pull request #873 from leno12/SH...	develop	5 hours ago	1m 43s
Code Quality #1956: Commit b0c517c pushed by dmetzner			
✓ Merge pull request #899 from Catrobat/...	develop	5 hours ago	43m 7s
Tests #1905: Commit 0ec8ee7 pushed by dmetzner			

Figure 6.3: GitHub Actions dashboard: GitHub Actions are directly integrated into a repository on GitHub. A dashboard provides access and feedback to all running and finished workflows.

software and self-hosted infrastructure are required. Nevertheless, it is possible to self-host GitHub Actions if necessary.

Workflows can be created from scratch or build using shared actions. A marketplace provides access to thousands of actions. The practical chapters of this thesis provide several examples of how to create and use GitHub Actions.

6.4.3 Comparison between Jenkins and GitHub Actions

While it is possible to use GitHub Actions in a self-hosted environment, the following comparisons between GitHub Actions and Jenkins only refer to the free provided services hosted by GitHub. Apart from that, the following comparisons are tailored to the needs and requirements of the Catroweb project.

Costs

For projects and primarily non-profit open-source projects, it is essential to reduce all costs to a minimum. Paid services like GitHub Actions are usually paid per minute, moving their usage out of the question for projects like Catrobat. Self-hosting a service is usually more challenging but cheaper than monthly payments to a third-party service. Therefore, former Catrobat team members decided to use Jenkins and self-host all CI/CD services.

Self-hosting a service gives a project full control of every minor detail, from explicitly defining the hardware for the tasks to run, up to every little configuration of the services. However, the required infrastructure and knowledge to build and maintain a CI/CD server, even with a tool like Jenkins, still produces noticeable acquisition and development costs. To keep the CI/CD system alive, Catrobat team members regularly have to invest valuable resources.

A turning point in this situation can be GitHubs current politics about open-source. GitHub, which is owned by Microsoft, highly values every contribution to the open-source community. As a result, they offer their services entirely free for open-source projects like Catrobat. Therefore, the costs to maintain and acquire the necessary infrastructure can be reduced to almost zero.

Response time

Catrobat has explicit teams to manage the Jenkins services and the infrastructure. However, developers at Catrobat only have limited time and rarely work full time on the project. In case of errors, it usually takes multiple weeks for the issues to be investigated and fixed. Usually, teams depending on the Jenkins' services have to wait or investigate issues by themselves. Instead, they could better use their time to work on business-relevant stories.

GitHub Actions are not perfect either and have problems as well as only limited availability from time to time (*GitHub Status Incident History* 2020). However, on the other side, GitHub has a dedicated team of developers working around the clock to improve and maintain the services provided by GitHub. Issues usually are resolved within hours. The crucial point is that projects like Catrobat do not have to invest their resources to investigate and fix the issues. On the other side, if the GitHub team does not immediately fix particular issues, the projects using their services can only wait and provide feedback. So there is always a certain amount of risk given by the dependency on a third-party service.

Environment

Out of the box, Jenkins services run all jobs in the same environment. A shared environment between independent jobs can speed up processes but may result in unforeseen issues depending on a particular order of execution. However, Jenkins has plugins to support a strict separation of the environment between different jobs.
On the other hand, GitHub Actions run every job on an independent machine using the desired operating systems, namely Linux, Window, macOS. Moreover, jobs can run inside a Docker container. In return, the caching and sharing possibilities with GitHub Actions are still quite limited.

Both Jenkins and GitHub Actions support parallelism to run numerous tasks simultaneously. Nevertheless, GitHub limits the parallel jobs to 20 machines in their free services. Besides, the maximum number of jobs per workflow is limited to 256, and no workflow can run longer than 72 hours. On the other hand, only the available resources limit a self-hosted service.

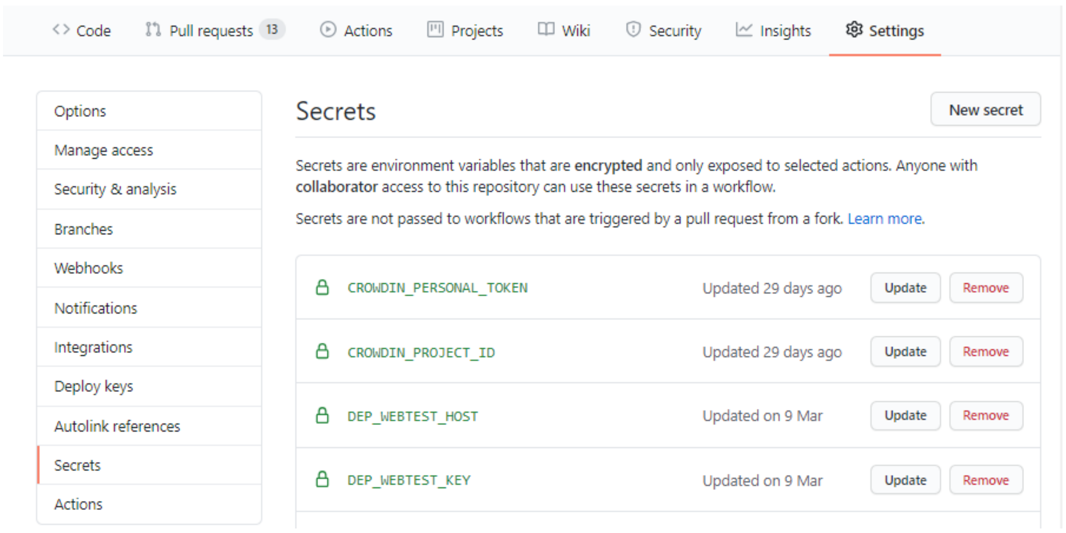
GitHub Actions support multi-platform matrix builds, which allow a job to be created to run parallel in different environments. A Plugin can achieve the same results for Jenkins.

Workflow creation and configuration

Single access to all files and features is an essential requirement by CI and reduces the costs of developing and maintaining any software or service. Besides, it lowers the feeling of inhibition for team members, especially junior developers, to inspect, optimize, or extend any services responsible for CI/CD, even if that is not part of their usual task area. GitHub Actions provide a central point to build and configure any workflow. Every line of code and configuration required to run the GitHub Actions is included directly in a project repository. The Jenkins services at Catrobat, on the other hand, are shared between the project repository requiring the services and an explicit Jenkins repository used by all teams.

Self-hosting a service like Jenkins enables developers to set up any required credentials hidden from the public by directly including them on the CI/CD server. However, the setup process requires a developer to update the private credentials. GitHub, on the other hand, allows project owners to create so-called secrets directly in their web-interface (Figure[7] 6.4). Secrets secure and hide any information, like production credentials, from the public. Only a running workflow can access the information hidden in secrets. Nevertheless, using a plugin, Jenkins can also manage secrets in its web-interface.

Since all GitHub Actions are directly hosted in the project's repository, developers could modify a workflow before its execution. Therefore, GitHub



Secrets

Secrets are environment variables that are **encrypted** and only exposed to selected actions. Anyone with **collaborator** access to this repository can use these secrets in a workflow.

Secrets are not passed to workflows that are triggered by a pull request from a fork. Learn more.

CROWDIN_PERSONAL_TOKEN	Updated 29 days ago	Update Remove
CROWDIN_PROJECT_ID	Updated 29 days ago	Update Remove
DEP_WEBTEST_HOST	Updated on 9 Mar	Update Remove
DEP_WEBTEST_KEY	Updated on 9 Mar	Update Remove

Figure 6.4: GitHub secrets: Production secrets can be hidden from the public in a secure storage integrated on GitHub. A workflow can access the hidden information.

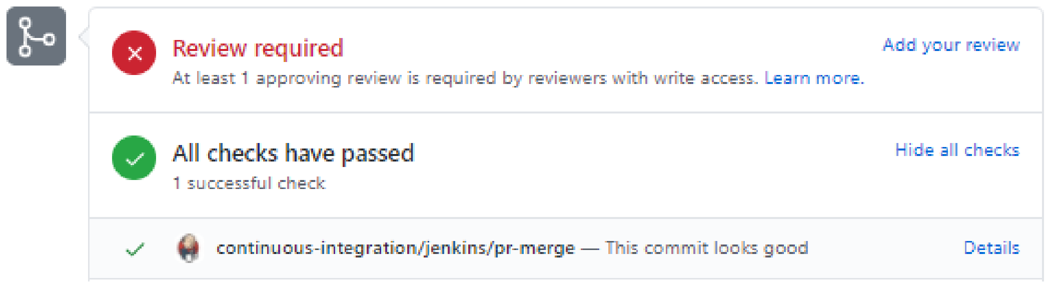
7. `https://github.com/dmetzner/Catroweb/settings/secrets` visited on 9 Sep. 2020

Actions running on forked repositories can not use secrets in the current state. On one side, this is necessary to protect the secrets, but on the other side, it complicates the usage of GitHub Actions in projects using a forking workflow. Moreover, developers could disable CI checks as they please. Only a manual review can prevent this issue.

Feedback

Both tools provide valuable feedback and access to pre-defined actions in their web-interfaces, where jobs can be initiated or rerun. The feedback ranges from an immediate indication of a failed pipeline to execution logs and a detailed report. The main difference between the tools is the location of their interfaces.

Jenkins reports are not directly integrated on GitHub but a website hosted by Catrobat. A plugin is necessary to integrate the Jenkins results into GitHub (Figure[8] 6.5). On the other hand, GitHub fully integrates the GitHub Actions feedback. Issues can be quickly detected and investigated without the need to leave the GitHub repository. Every pull request contains a summary of all checks (Figure[9] 6.6) and provides access to detailed logs (Figure[10] 6.7). Besides, generated content can be downloaded as artifacts.


Review required

At least 1 approving review is required by reviewers with write access. Learn more.

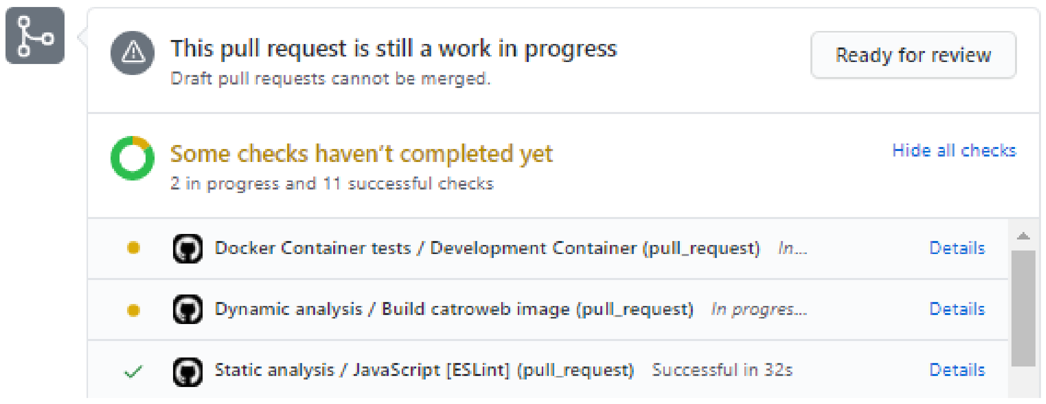
All checks have passed

1 successful check

continuous-integration/jenkins/pr-merge — This commit looks good

Figure 6.5: Jenkins integration on GitHub: Jenkins results can be integrated into GitHub's pull request interface using a plugin. Therefore, CI/CD results are immediately visible.

8. https://github.com/Catrobat/Catroid/pull/3768 visited on 9 Sep. 2020
9. https://github.com/Catrobat/Catroweb/pull/883 visited on 9 Sep. 2020
10. https://github.com/Catrobat/Catroweb/pull/883/checks visited on 9 Sep. 2020

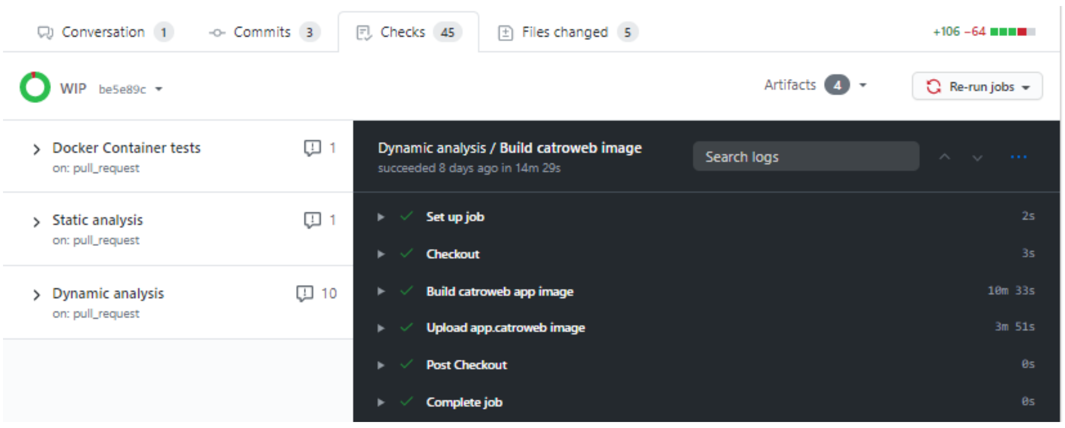


This pull request is still a work in progress
Draft pull requests cannot be merged.

Some checks haven't completed yet
2 in progress and 11 successful checks

Docker Container tests / Development Container (pull_request) In...
Dynamic analysis / Build catroweb image (pull_request) In progres...
Static analysis / JavaScript [ESLint] (pull_request) Successful in 32s

Figure 6.6: GitHub Actions pull request integration: Every pull request summarizes the results of all GitHub Actions in the merge interface of a pull request.



Docker Container tests
on: pull_request 1

Static analysis
on: pull_request 1

Dynamic analysis
on: pull_request 10

Dynamic analysis / Build catroweb image
succeeded 8 days ago in 14m 29s

```
Set up job                   2s
Checkout                     3s
Build catroweb app image     10m 33s
Upload app.catroweb image    3m 51s
Post Checkout                0s
Complete job                 0s
```

Figure 6.7: GitHub Actions pull request feedback: The whole GitHub Actions feedback is fully integrated into the GitHub interface of a pull request, and developers have access to detailed execution logs.

In exchange, the Jenkins web interface contains multiple options to customize the feedback's look and feel. At the same time, GitHub provides no possibilities to tweak the appearance of the integrated GitHub Actions feedback. External third-party services are necessary to reach similar customizability with GitHub Actions as with Jenkins.

Plugins and support

Over the years, countless plugins and tutorials have been written by the Jenkins community. Usually, for every occurring problem, there already exists a solution. On the other hand, GitHub Actions are still very new and have significantly less information on the web. Moreover, even though GitHub provides documentation, the available information is limited and contains little to none practical examples. Nevertheless, the community is proliferating, and more and more content is produced by developers worldwide.

GitHub's counterpart to Jenkins plugins is provided in the form of predefined actions that can be used to build a workflow. Developers worldwide develop new, freely accessible actions every day and provide them in a marketplace[11]. Various hackathons[12] accelerate the creation of new actions to catch up with other CI/CD tools.
In addition to GitHub Actions, projects can include various GitHub Apps to automate their workflows. While most apps are independent of GitHub and developed by some other companies, they are usually free of charge for open-source projects. However, to run successfully, they require various permissions for a project. Providing read and write access to a third-party application leaves a project at a certain risk. Therefore, it is recommended to only use apps verified by GitHub.

Nevertheless, limitations and restrictions in GitHub Actions often still require complicated workarounds to implement specific workflows. For example, projects following the fork-workflow have limited access to secrets and no write permissions on GitHub Actions that run on a pull request. Therefore, many existing actions can not be used without several adaptions first.

11. `https://github.com/marketplace` visited on 9 Sep. 2020
12. `https://githubhackathon.com/` visited on 9 Sep. 2020

Conclusion

"Custom scripts and bespoke systems prevented many of our users from fully realizing the benefits of CI/CD. GitHub Actions gave us a platform to deliver intelligent pipelines with minimal up-front investment and no ongoing maintenance." (Dan Belcher, 2020)

After intensive testing of GitHub Actions, the service was chosen to automate all workflows at Catroweb. The main reasons are the zero acquisition costs and simplicity in creating and maintaining any workflow automation. With GitHub Actions, the Catroweb team is in full control of its CI/CD services and does not depend on other Catrobat teams to accomplish CI/CD. Although some basic features are still missing, and the documentation is pretty sparse, all of Catroweb's requirements have sufficient support. Also, there already exist thousands of actions to automate any workflow. Furthermore, the GitHub team provides a detailed roadmap about the support of future features. Their constant aim to improve their services should allow future developers of the Catroweb team to optimize the workflows implemented in this thesis further.

In case it is necessary to switch to a self-hosted service again in the future, developers do not need to create the automation workflows from scratch. GitHub Actions can be self-hosted.

7 CI - Build automation

Building an executable software product from its source code often requires various steps and different tools. However, automating the build reduces the complexity and needed time to create an executable significantly. A single command is sufficient to initiate an automated build process. Besides, an automated build eliminates variations throughout the building process and provides a foundation for the CI/CD pipeline.
Every contribution must be tested in the same environment. Even minor differences can result in various issues. For example, if developers use and require an updated dependency during development, their code might not work for developers that still use an older version. Therefore, the code working on one machine does not imply it will work on others.
Thorough documentation about all dependencies and their exact version can reduce such issues. However, that requires every developer to stick to the same specifications strictly. On the other hand, automating the build on a CI server removes this hurdle altogether. If one developer updates the environment, the dependencies are automatically updated for everyone resulting in a uniform system and implicit documentation about all requirements.

However, Catroweb's source code is interpreted and must not be compiled beforehand to run. Hence, no build actions are required, and most issues can only be detected by running the Share community platform and its software tests. This problem is described in more detail in section 8.1.
Nevertheless, a specific environment with various installed and configured programs is required for the application to run. To ease the project's setup process, Catroweb uses Docker to build its development and testing environments automatically and uses package managers that support developers to maintain and include additional third-party code into the project. Therefore, the Share community platform's Docker containers must be built and

tested to ensure that all developers can access a working development environment. Furthermore, both Docker and package managers provide the foundation for running static and dynamic analysis checks on the CI server, as described in chapters 8 and 9.

7.1 Package manager

A package, also often referred to as a library or module, is a software component that provides the code to solve a problem. Developers can include and rely on packages for all kinds of use cases. Packages enable developers to share functionality between different projects in a quick but maintainable fashion.

Large projects like Catroweb are built using hundreds of packages and modules. A package manager can be used to ease the integration and segregation process of such third-party components. Package managers automate the processes around installing and removing packages to a project to reduce developers' workload. Composer[1], for example, is the name of the package manager responsible for all PHP packages in the Catroweb project. Npm[2], on the other hand, handles all modules for the front-end.

7.2 Docker

Docker[3] is a software to build, run, and deploy applications using containers that run in isolation. A containerized development approach allows switching between dependencies, without modifying the host system, quickly. Moreover, Docker removes the necessity of expensive manual installation and configuration processes. A single command is sufficient to set up a project on a new machine fully. Besides, all dependencies and configurations

1. `https://getcomposer.org/` visited on 9 Sep. 2020
2. `https://www.npmjs.com/` visited on 9 Sep. 2020
3. `https://www.docker.com/` visited on 9 Sep. 2020

are implicitly documented in a single Dockerfile that must be modified to update the environment. A Dockerfile is a text document that specifies exact instructions to build a Docker image. A Docker image can be seen as a blueprint of an application to build a container. In contrast, a Docker container is a transient and temporary instance of an image to run the application. A Docker container can be stopped and restarted.

Best practices for Docker require each service to be in an independent container. Nevertheless, Docker containers can communicate with each other. Since the Share community platform requires various services to run, namely, a web server, a database, a browser, and the application itself, Catroweb runs every service in a unique container. Therefore, multiple Docker containers must be configured, connected, and started to develop and test the Share community platform using Docker. However, all containers can be started at once with a single command using Docker Compose, a tool optimized to run multi-container Docker applications.
Catroweb's testing environment is slightly different from the development environment. Therefore, the project has two different Docker setups, either for the development or the testing environment. A third one for the production environment is already planned but does not exist yet. The main difference between the settings is the detail of logging and the number of attached tools.

7.3 Automate the process

On every creation and merge of a pull request in the Catroweb repository, a workflow to build and verify Catroweb's Docker containers is initiated using GitHub Actions to ensure that developers have access to a working environment using Docker. In case a Docker container crashes, or the environment and the Share community platform do not harmonize, developers get immediate feedback. Therefore, developers know an integration can not be accepted without further modifications, either in the Dockerfile or the source code.

First, the workflow checks-out the source code. Then the job starts to build and run all containers using Docker Compose. As soon as the containers are up and running, various additional health checks are performed to ensure that their most basic features are working. Therefore, as a next step, the Docker environment is filled with dummy data to provide a livid development environment. Next, the job runs a few shell commands to check the website's availability and response code. Once it is ensured that the Share community platform responds, a few dynamic tests guarantee all automation testing frameworks are set up and working.

Listing 7.1 provides the truncated and simplified code for the GitHub Actions' workflow written using YAML[4]. YAML is a human-readable data serialization language commonly used for configuration files.

```yaml
name: Build
on: [push, pull_request]
jobs:
  container_checks:
    name: Docker container checks
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v2

      - name: Build and start all containers
        run: docker-compose -f compose-file up -d

      - name: Create dummy data
        run: docker exec container creation-command

      - name: Check local share website
        run: |
          install-and-configure-tools
          send-request
          check-web-page-response

      - name: Run a few software tests
        run: docker exec container test-command
```

Listing 7.1: Verify Docker build: Catroweb's Docker containers' essential functionalities are built and verified on every integration.

4. The simplified code is marked red.

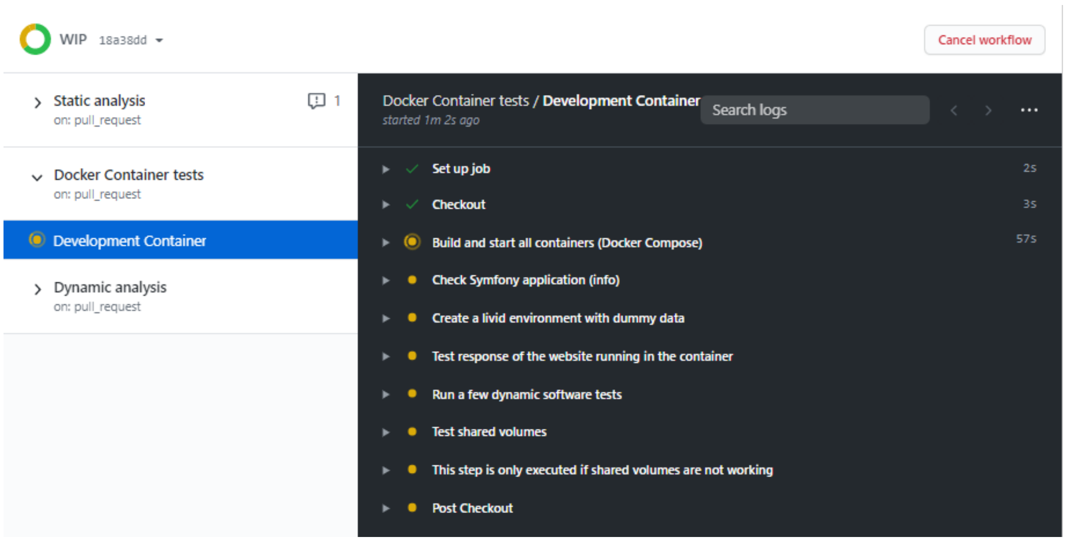
Finally, the functionality of shared volumes is checked. Using shared volumes, developers do not need to rebuild the whole container after every file change. Hence, the build frequency and development costs can be significantly reduced.
A file must be modified and its impact evaluated to verify the functionality of shared volumes. One way to do so requires the GitHub Actions job to write invalid code into a source file. Once a test using the corrupt source file is executed, the test case fails. However, requiring a step to fail without failing the whole pipeline is not possible out of the box using GitHub Actions. Therefore, a little workaround is required (Listing 7.2).

```
- name: Test shared volumes
  id: s1
  continue-on-error: true
  run: |
    echo ::set-output name=status::failure
    echo 'invalid code' > source-file
    echo ::set-output name=status::success
- if: steps.s1.outputs.status == 'success'
  run: |
    exit -1
```

Listing 7.2: Test shared volumes: Testing the impact inside a Docker container after changing a file's content outside of the container is necessary to verify that files are shared between the container and the host. Unfortunately, steps in GitHub Actions can not expect failure out of the box. A second step to verify the previous step's output is required to set up a job that only passes if a step has failed. The step that must fail is set to a state where it can not fail. A second step would stop the job with failure if the first step did not fail.

The workflow's step required to fail must be configured to a state where it continues on errors. At the beginning of the step, a variable for the status must be created and set to failure. Next, the tests are executed, and only if the tests pass, the status variable is set to success. However, in case of an error, the status variable will remain set to failure. In an additional step, the status variable can then be checked. Since an error is expected, the job must only fail in case of no failures. Therefore, a condition checking the status variable is bound to the next step, and only if it is fulfilled, the GitHub Actions job exits with an error code.

The final workflow file to thoroughly check the Share community platform Docker containers is available on GitHub[5]. In case of any issues during the workflow, developers have access to detailed logs of every step directly in the GitHub web-interface. Figure[6] 7.1 shows the execution logs while the GitHub Actions are still running.



WIP 18a38dd

Cancel workflow

Static analysis
on: pull_request

Docker Container tests
on: pull_request

Development Container

Dynamic analysis
on: pull_request

Docker Container tests / **Development Container**
started 1m 2s ago

Search logs

```
✓ Set up job                                                   2s
✓ Checkout                                                     3s
● Build and start all containers (Docker Compose)             57s
● Check Symfony application (info)
● Create a livid environment with dummy data
● Test response of the website running in the container
● Run a few dynamic software tests
● Test shared volumes
● This step is only executed if shared volumes are not working
● Post Checkout
```

Figure 7.1: Docker verification feedback: Catroweb's Docker container test logs are accessible directly in the GitHub interface. The logs are updated immediately while the job is running.

5. https://github.com/Catrobat/Catroweb/blob/develop/.github/workflows/container_tests.yml visited on 9 Sep. 2020
6. https://github.com/Catrobat/Catroweb/pull/883/checks?check_run_id=1091453671 visited on 9 Sep. 2020

8 CI - Static analysis

Static analysis is a method to test and debug software without actually running the program. Methods to statically analyze an application, thoroughly evaluate the source code itself. Static analysis checks cover every possible execution path, which is very cost-intensive to achieve with dynamic tests. Besides, compared to dynamic tests described in chapter 9, static tests find a different set of errors.

In addition to manual code reviews, static analysis tools highly reduce the risk of overlooked errors and vulnerabilities. Automated tools are high-speed and less prone to errors compared to humans. Executing them continuously throughout the development provides developers with a constant stream of feedback, allowing them to react quickly to any found issues. However, static analysis tools can detect false positives, and it is often not possible to strictly follow all rules. A static analysis tool should only be an addition to manual code reviews and not a replacement.

Static analysis can find various static issues such as syntax violations, coding standard violations, typing errors, parameter miss matches, and undefined variables. Moreover, there exist tools that can detect previously unknown vulnerabilities in PHP web applications (Jovanovic, Kruegel, and Kirda, 2006). The type of discovered issues heavily depends on the tool itself. There exist many different tools, optimized for specific languages and domains, to find everything from simple programming errors to a more sophisticated detection of issues and vulnerabilities.

This section describes various measures taken to integrate numerous static analysis tools into a project like Catroweb successfully. On the one hand, to spot and fix issues in the existing code base and, on the other hand, to provide developers with continuous feedback while preventing static code issues from being integrated into the codebase.

8.1 Static analysis of interpreted languages

A significant part of computer programs is written using higher-level programming languages like Java or PHP. Higher-level programming languages provide substantial abstractions about the details necessary to run a program on a computer. Therefore, such languages are easy to read, write, and maintain by human developers. However, to execute programs written in high-level languages, they must first be translated into a lower-level language. Low-level languages are optimized for the underlying hardware and have little or no abstraction.

A compiler is a computer program that translates a programming language into another. It usually translates a high-level programming language into a low-level one. During the conversion, the source code is analyzed and processed. A compiler must know about the complete source code before a program's execution to generate and optimize the code used to run the program. Errors, such as syntax violations, immediately stop the compilation process. Less invasive issues are typically shown as a warning. Hence, compiled languages like Java have an implicit static analysis check included through their compilers. Programs with syntax errors can not be executed and ensure a certain level of reliability.

On the other hand, Catroweb is mainly built with PHP and some JavaScript. Both are interpreted languages, which means that no source code compilation is necessary before running a program. An interpreter translates and executes a program at the same time. With the resulting gained flexibility, interpreters increase the responsiveness during the development process. There is no necessity to wait for the compilation process to finish. It is possible to update a script within seconds without the need to recompile a whole application. In return, the execution is usually slower and less reliable. With an interpreter, the static compiler checks are missing. Many errors found by static analysis remain unnoticed in the codebase and are shown only during run-time.
Several static analysis tools are explicitly integrated into the project to prevent issues detectable by static analysis. The tools and the integration process are described in section8.3.

8.1.1 Type hinting in a dynamic weakly-typed language

Programming languages usually have different types of variables. A variable is a named memory to store information on the computer. The type of a variable determines properties like its size in memory and the possible range of values. For example, an integer stores a whole number, while a float stores a floating-point number. Both types are typically stored using 32 bits in memory. On the other hand, a char can store letters and only needs eight bits to be stored.

Moreover, the type provides the necessary information on how the data in memory can be interpreted. Different types support different operations, and not all types are compatible out of the box. When adding two integers, it makes perfect sense that the result will be an integer, too. However, what about adding an integer with a float or a char? The result heavily depends on the used programming language. Some implicitly convert the result to one of the types. Others may fail with an error.

Statically-typed languages must know the types of a variable at compile-time, while dynamically-typed languages check the types during run-time. Besides, a strongly-typed language prohibits variables from changing their data type during their lifetime. Variables in a weakly-typed language, on the other hand, can change their type.

Java is a strongly-typed language. Therefore, the Java code defined in Listing 8.1 results in an type error. The variable must not change its type from a string to an integer. Also, Java is a statically-typed language, meaning the error already appears during the compilation process. PHP, on the other hand, is a dynamically- and weakly-typed language. Therefore, the PHP code defined in Listing 8.2 results in no type error since the variable can change its type.

```
String x = "2";    //
x = 2;             // error: incompatible types
```

Listing 8.1: Java type error: Assigning values of incompatible types to a variable of a different type in a statically and strongly-typed language like Java will result in a type error during compilation.

```
$x = "2";          // x is a string
$x = 2;            // type changed: x is an integer
```

Listing 8.2: PHP type conversion: Variables can change their type in weakly-typed languages.

Since PHP is dynamically-typed, possible errors will only occur at run-time. Nevertheless, there are no type errors. In weakly-typed languages, the compiler or interpreter tries to convert the types implicitly. The results can be surprisingly different between various languages. Listing 8.3 and Listing 8.4 show the different result of the same calculation in the weakly-typed languages JavaScript and PHP.

```
3 * "3"                 // = 9
1 + "2" + 1             // = "121"
```

Listing 8.3: JavaScript type conversion: Different mathematical operations between different types in JavaScript may result in different types.

```
3 * "3";                // = 9
1 + "2" + 1;            // = 4
```

Listing 8.4: Another PHP type conversion: PHP converts numbers in string format to integers.

Removing the constraint to check every type can ease the programming process and provide an even higher abstraction. However, it allows developers to develop bad habits and make code more challenging to read and understand. Besides, detecting errors quickly and consistent is a good thing. Imagine that a function is called with a variable of the correct type in 99 percent of all cases. The remaining one percent provides an incompatible type to the function, resulting in an unexpected output. Such issues can be tricky to find and debug.

PHP 7 started to introduce typed properties into the language. In the past, developers had to write annotation comments explicitly to hint static analyzers about the types. The program execution, however, ignores annotation. The validity of annotations only depends on the developers. Typed properties enforce the supported types at run-time. In combination with a stricter typed language, the static analysis tools presented in the next sections can develop a deeper understanding of the source code and provide better analysis reports. Therefore, the code base has been refactored to use typed properties wherever possible. Nevertheless, some features like union types are still missing and can not be type-hinted yet. However, PHP 8 has planned to introduce additional stricter typing features. To further increase the readability and maintainability, refactoring the source code to utilize the new type hints is recommended as soon as PHP 8 is released.

8.2 Coding standard

Efficient teamwork is all about communication. A crucial component of this communication between developers is the code itself. Giving functions and variables meaningful names, while consistently formatting and documenting the code where necessary can significantly increase code readability. Every line of code should be easy to understand, extend, or debug by others because there is a high chance it will be maintained by someone else. A large code base should look just as if it was written by one developer even if hundreds of developers have contributed to the project. In the worst-case scenario, code is tough to read. Like other aspects of development, with a lack of proper communication, the cost of development increases.

Every programming language has its own rules to define the syntax of a program. Some have stricter rules than others, but generally, developers have some freedom to define their instructions. In that sense, code style does not impact the code's performance, nor is it responsible for the code to work. However, on a large codebase with multiple team members, a well-formatted clean and consistent code style can be the key to success. Using a lousy code style is more prone to errors and, all in all, hard to read for developers.

For example, the code in Listing 8.5 is very sloppy formatted compared to Listing 8.6, thus making it challenging to understand this small code snippet right off even though the logic is not complicated. Writing the same functionality in a clean and meaningful way, like in listing 8.6, one can easily understand what the code should do.
Clean code allows developers to understand code faster and without the

need to decipher it character by character. Thoughtful naming gives a clear indication of the intentions and usage of every variable and function.

```
  function   foo(
                                length
  )

{
  let a = 'abcdefghijklmnopqrstuvwxyz0123456789' ;
  let b = '';;;for (
      let
      i=0;i<length;
      ++i
) {
            ri
        = Math.floor(        Math.random   (     )*  a.length);
      b +=
          a.
            charAt(ri)}return b}
    console.log(foo(
      32
))
```

Listing 8.5: Lousy JavaScript code: Bad formatted code is hard to read and understand.

```
function generatePassword(length) {
  let charset = 'abcdefghijklmnopqrstuvwxyz0123456789'
  let password = ''
  for (let i = 0; i < length; ++i) {
    randomIndex = Math.floor(Math.random() * charset.length)
    password += charset.charAt(randomIndex)
  }
  return password
}

console.log(generatePassword(32))
```

Listing 8.6: Clean JavaScript code: Applying the right code style makes code easy to understand and read.

It is hoped that developers do not write code as unreadable as the exaggerated example in Listing 8.5. However, most of the time, developers have their

unique preferences about code style, which will result in inconsistencies all over the place over time. Some use style A, some style B, while others do not care and use none. These inconsistencies increase the complexity, lower the quality, and increase the chance of merging conflicts between multiple developers.
Additionally, these inconsistencies do not go well with the human ego of some developers. Those developers who focus more on a clean code regularly reformat and refactor the code of others. On one side, this might offend the original author of the changed code. On the other side, every minute of refactoring is time lost for features that could benefit real customers.
A team should be a unit working together, and a uniform style is a way to strengthen the alliance between developers. It does not matter what the exact code style is, but it is vital to apply the same style consistently.

8.2.1 Consistent formatting

Since many slightly different code styles are equally good, and developers have unique preferences, best practices may differ for different languages and even projects. Therefore, a developer teams' primary goal should be to come to an agreement about a binding code style to apply and enforce on every contribution. The thing that matters the most is consistency across all files. Typically, code style issues can be fixed automatically. Still, if that is not the case, enforcing an exact code style may result in additional overhead for developers, increasing the development costs even more than a lousy code style.

Unfortunately, at Catroweb, developers in the past did not care a lot about the code style. The documentation only slightly mentioned a few basic coding style rules. Nevertheless, no uniform rule sets have been enforced over many years. The code happened to be a mix up of almost every imaginable coding style.

The first step in the right direction was to define the coding style rules and carve them in stone. Considering that Catroweb is written in multiple languages, it is more efficient to use different configurations optimized for every corresponding language. Thereby, those rules are based on industry standards, with only minor modifications to fit the existing codebase better. Every developer can look up the specifications in the official documentation if necessary.

Even with all those rules specified, without additional tools to automatically test and apply the definitions, the code style would depend entirely on the developers and code reviewers. Therefore, human errors are not avoidable. Besides, reading documentation is a time-consuming process. Luckily, a handful of proven tools developed by the open-source community already exist and are used by thousands of projects free of charge to enforce a specific code style. Instead of writing tools to check the style from scratch, they only need to be included and configured. Typically, a few lines of configuration are enough to overwrite default values to fit the requirements better. Code style checker tools analyze the source code and test every line against defined rules and report all found issues. Some tools even support the automatic fixing of code style issues. Typically, a single command is sufficient to initiate the code style fixing and checking actions. Hence, tools to control the code style are highly reducing the overhead of writing clean and consistent code to a bare minimum. Developers do not have to waste their valuable time on minor code style issues. Besides, the tools provide valuable feedback about what has changed. So if developers care, they can quickly learn to improve their code style with every contribution without the explicit need to read the documentation.

At Catroweb, the project was anything but consistent in its code style. It was necessary to refactor multiple thousand lines of code to match the chosen styles. Luckily the tools have been able to fix most code style errors automatically. The few remaining errors were corrected by hand to provide an error-free baseline for future contributions without existing inconsistencies.

8.3 Static analysis tools

Instead of creating a static analysis tool from scratch, this section describes various open-source software tools that can be used free of charge to analyze any PHP, JavaScript, or CSS files statically. Besides, the Symfony framework already has built-in support to lint configuration and template files to ensure their correct syntax. It is worth mentioning that there exist many more static analyzers. Nevertheless, the following tools were chosen due to their active community and development and their power and ease of use.

8.3.1 Customization

All of the following tools can be customized with a configuration file in the project's root directory to adapt the applied rules to the requirements. Nevertheless, default configurations ensure that the tools already work out of the box. In the Catroweb project, the configurations were chosen to mainly stick to the default values with minor exceptions to fit the codebase's existing style.

8.3.2 Installation

There exist different possibilities to install the tools to analyze the project. Using the package managers, composer, and npm, which already handle all the Share community platform's dependencies, all tools can be easily integrated. However, this approach comes with the drawback of depending on various tools that are not necessary to run the production software. Such artificial dependencies could block dependencies that are required to run the Share community platform. Section 11.4 describes this problematic in more detail. A further possibility is to integrate the tools only as a pre-built executable version, independent of the project dependencies. However, to utilize the benefits of the automatic update process described in section 11.4.1, most of the tools are integrated into the Catroweb project using a package manager. In case there will be breaking dependencies at one point in time, it is necessary to switch to executables.

8.3.3 Analyze PHP files

PHP builds the foundation of the Share community platform. Since there is no perfect single tool to find every kind of issue, combining multiple tools is necessary to detect as many problems as possible.

PHPStan[1] and Psalm[2] are two well-known static analyzers in the PHP community. Although both tools try to achieve similar results, they have unique strengths and can spot slightly different problems. Hence, the combination of both tools detects more issues than just one of them alone. PHPstan and Psalm support the prevention of most type-related runtime errors and check for issues like passing too many arguments to functions, referenced code that does not exist, uninitialized variables, typing errors, and many more. However, they do not check the code style.

PHP-cs-fixer[3] is a code style checker with support to automatically fix code style issues. With thousands of active users and more than 200 contributors, the tool's development seems to be stable for the next years to come. Henceforth, PHP-cs-fixer is responsible to automatically check and fix all coding-standard issues in the Share communities platform's PHP files. The tool is configured to use all rules defined by the Symfony framework with a few overrides to fit the code style of the existing codebase.

Supplementary, a PHP copy-paste detector[4] is introduced to prevent developers from copying the same code repeatedly. Instead, developers should use functions and methods to reuse the same logic to reduce duplicated code and increase maintainability. Extending or modifying the same logic must only be done in one central place.

PHPCodeFixer[5] detects deprecated functionality that will no longer be supported in newer PHP versions. Removing deprecated functionality in a project eases the upgrading process in the future and reduces the risk of failure during an upgrade.

1. `https://phpstan.org/` visited on 9 Sep. 2020
2. `https://psalm.dev/` visited on 9 Sep. 2020
3. `https://github.com/FriendsOfPHP/PHP-CS-Fixer` visited on 9 Sep. 2020
4. `https://github.com/sebastianbergmann/phpcpd` visited on 9 Sep. 2020
5. `https://github.com/wapmorgan/PhpCodeFixer` visited on 9 Sep. 2020

PHPloc[6] provides information about the project's size and complexity. This tool can not fail the CI pipeline but provides various statistics as feedback.

8.3.4 Analyze JavaScript and CSS files

ESLint[7] supports the detection of similar issues like PHPstan and Psalm. Besides, ESLint can detect and fix code style issues. The tool's configuration is set to adapt to the npm standard[8] JavaScript coding style.

For Sass and CSS files, StyleLint[9] was introduced to detect errors and enforce conventions in the style files.

Both standards enforced through the tools are no real web standards but understand the latest syntax with hundreds of rules to catch errors. Besides, they are used by many major companies, like GitHub.

8.3.5 A baseline to optimize initial results

Introducing static analysis into a large legacy codebase can be an overwhelming experience. Integrating the static analysis tools into the CI pipeline does not require much work. However, it can be a very time-consuming process to include them effectively into a project. For example, summing up all PHPstan and Psalm errors in the Catroweb repository showed more than 10 thousand static issues that can not be resolved automatically. Immediately fixing all problems is a highly unrealistic goal, requiring way too many resources.

Adding the checks to CI would result in a failing pipeline over a long period. A failing pipeline contradicts the principles of CI. Further, a pipeline that is regularly failing tends to be ignored by developers, ruining the benefits provided by CI. New errors easily find their way into the project. The same issue applies when configuring workflow steps to a state where they cannot

6. https://github.com/sebastianbergmann/phploc visited on 9 Sep. 2020
7. https://eslint.org/ visited on 9 Sep. 2020
8. https://www.npmjs.com/package/standard visited on 9 Sep. 2020
9. https://stylelint.io/ visited on 9 Sep. 2020

fail and only report bugs. Developers and also code reviewers will ignore the error messages, rendering the static analysis pipeline checks ineffective.

Therefore, as a first step, the tools were set up using a baseline. A baseline empowers test results to filter and highlight only newly introduced errors by remembering the results of a previous run and removing them from the output. Hence, developers can spot their issues instead of overlooking them in an ocean of error messages.

Psalm has already built-in support to create a baseline. PHPStan, on the other hand, is missing a baseline feature. Luckily, it was possible to use a third tool called SARB[10] to create the baseline for both tools. Thanks to a baseline, developers are not flooded with legacy issues on their new contributions. Therefore, to provide a minimum quality level, static analysis can be quickly and still effectively integrated into the pipeline right from the start.

8.3.6 From a baseline to a solid base

Nevertheless, later on, to increase the project's maintainability and eliminate the endless stream of errors and warnings in the project, it was decided to fix as many errors as possible.

Both Psalm and PHPstan have eight different levels. Low levels only highlight critical vulnerabilities. In contrast, stricter levels even show less problematic errors. First, the accuracy was set to the lowest possible level to reduce the sheer amount of errors. Level by level, refactorings of the code base fixed thousands of errors. Only a few issues have been fixed automatically by Psalm, and it was necessary to resolve the majority manually. Luckily many errors had the same pattern. Thus, it was possible to fix them with the tools provided in modern code editors quickly.

Using the highest level of accuracy produces the safest possible code. However, it forces developers to watch every minor detail. Besides, working with third-party libraries can be very challenging. For Catroweb, the final configuration uses a high but not the maximal level of strictness, which

10. `https://github.com/DaveLiddament/sarb` visited on 9 Sep. 2020

seems to be the right balance between spotting those critical errors and not taking up many resources to fix them.

All static analysis tools in the Catroweb project can now be executed without a baseline and still show an error count of zero. There only remain a couple of errors in the project's source code that are not fixable. Those exceptions are explicitly specified in the configuration files and therefore are not reported by the tools anymore. For example, such an exception is necessary when working with the API's auto-generated code, since the OpenApi generator has not been producing 100 percent perfectly clean code yet. The API generation is briefly described in section 11.1.

8.4 Continuous integration of static analysis

Automated static analysis feedback ensures contributions to be well-formatted and free of static errors, requiring a minimum level of quality at all times, with fewer errors in the codebase. Besides, developers get immediate feedback from the CI system, independent of a reviewer. Consistently getting notified by static analysis tools about possible issues, developers learn to write better code. Especially junior developers with little knowledge and experience benefit the most.

Developers could run the tools locally and fix all issues before creating a pull request. Nevertheless, all tools described in the previous sections are integrated into Catroweb's CI system. If developers do not care about their code quality errors or make some mistakes, the automated tests still catch the errors. No time-consuming manual reviews by other developers are necessary to highlight the issues. Developers can quickly react to the problems and update their pull requests before a code reviewer manually checks the code.

Automatically fixing code style issues during the CI workflow would be possible. However, the CI system only validates the contributions style but does not fix it. Hence, developers have to run the commands to fix the problems manually. Implicitly modifying contributions can result in higher overhead and create more problems than it would solve.

8.4.1 Automate the process

Every pull request created on the Catroweb repository triggers a static analysis workflow. In case that developers update their pull requests, the workflow is started again. Besides, to ensure the quality, contributions are tested a second time right after merging the changes. In case a static analyzer detects any issues, the job fails. Developers can click on the failed step definition and look at the logs containing detailed information about all code style issues. For example, a part of the output of a failed PHPStan job can be seen in Listing 8.7.

```
 ----- -------------------------------------------------------
  Line   src/Admin/AllProgramsAdmin.php
 ----- -------------------------------------------------------
   70    PHPDoc tag @var for variable $model_manager
         contains unknown class App\Admin\ModelManger.
 ----- -------------------------------------------------------
  Line   src/Catrobat/Controller/Web/DefaultController.php
 ----- -------------------------------------------------------
   29    Method App\Catrobat\Controller\Web\
         DefaultController::indexAction() has no return
         typehint specified.
   61    Method App\Entity\FeaturedProgram::getUrl() invoked
         with 1 parameter, 0 required.
   63    Parameter #3 $featured of method
         App\Catrobat\Services\ImageRepository::getWebPath()
         expects bool, string given.
```

Listing 8.7: Static analysis feedback (PHPStan): Contributions with static issues result in a failing pipeline. The log of a failed step provides detailed information about all detected issues.

Using GitHub Actions to execute a static analysis tool only requires a few lines of configuration. While static analysis tools require access to the source code, no complicated setups are needed since it is unnecessary to run the Share community platform. The workflow runs every static analysis tool in an independent job. First, a job checks-out the code and sets up the required dependencies to run the tools using Catroweb's package manager (Listing 8.8). In case a tool is not integrated into a package manager, the workflow downloads the latest pre-built executable (Listing 8.9). Finally, the tool can be executed.

```yaml
name: Static analysis
on: [push, pull_request]
job-name:
  name: static-analysis-tool
  runs-on: ubuntu-latest
  steps:
    - uses: actions/checkout@v2
    - run: |
        package-manager install
        run-static-analysis-tool
```

Listing 8.8: Static analysis workflow: Executing a static analysis tool with GitHub Actions does not require a complicated setup, since the tools only analyze the source code and no program execution is required. It is enough to install the dependencies and run the tool on the project files.

```yaml
job-name:
  name: static-analysis-tool
  runs-on: ubuntu-latest
  steps:
    - uses: actions/checkout@v2
    - run: |
        wget url-to-pre-built-executable
        run-static-analysis-tool
```

Listing 8.9: Static analysis workflow job alternative: Tools that are not available through the package manager can be downloaded from the web instead. After the download finishes, the tools can be executed.

8.4.2 Optimizations

The static analysis workflow consists of various independent jobs. However, the jobs themselves are almost all identical to Listing 8.8. Unfortunately, there is no way to reduce the duplicated code in the current state of GitHub Actions. Therefore, the logic must be updated on multiple locations in the same file if the setup process changes. Luckily, GitHub is already working on a solution to this issue, called composites, that enable developers to reuse parts of a workflow. Hence, it is planned to switch to a solution built on composites as soon as possible to increase the workflow files' maintenance.

On the other hand, caching is already possible, with a few minor modifications to the workflow job template (Listing 8.10). Instead of installing all dependencies from scratch, dependencies can be cached with an action explicitly created to improve the performance of repeatedly setting up environments. This action requires developers to specify the files to be cached and a unique key used to store and restore the dependencies. The key is created using the package managers' lock-files since these files change only if a dependency is updated. As a final step, a condition is added to the installation command to execute the installation process only if there was no cache-hit. Therefore, each job tries to restore all dependencies at first. Only if no dependencies have been restored, the package managers' installation process is initiated and cached. In the case of a cache-hit, the workflow finishes within a fraction of time and regularly saves valuable computation time.

```yaml
- uses: actions/cache@v2
  with:
    path: path-to-files-to-store
    key: hash-lock-file
- if: steps.composer-cache.output.cache-hit != 'true'
  run: package-manager install
```

Listing 8.10: Caching: Adding a caching action to store and restore dependencies saves valuable computation time.

The most recent static analysis workflow file, containing additional optimizations and verification checks, is available in Catroweb's GitHub repository[11].

8.5 Code review

Ensuring the quality of every contribution is an essential but cost-intensive goal. A code review is a systematic manual investigation of source code to detect issues and code smell in the source code to enhance the quality.

11. https://github.com/Catrobat/Catroweb/blob/develop/.github/workflows/code_quality.yml visited on 9 Sep. 2020

A so-called code reviewer checks the source code written by another developer during a code review and marks all found issues and problems. The thorough inspection ensures that new contributions meet all requirements and work as intended. Besides, developers can learn from each other through code reviews and strengthen their teamwork. Typically, developers do a code review before integrating new contributions to a project. Nevertheless, it could be done at any time, by any number of reviewers.

A code review is an essential practice to increase the quality of new contributions. However, a manual review is limited by the skills and invested time of the code reviewer and entirely depends on how the developer handles the review. A reviewer must spot all problems. However, this is highly unlikely to manage, even for experienced developers or reviewers in a team with continually switching members. Hence, relying only on manual code reviews is prone to errors.
In open-source projects like Catroweb, every contribution counts. Resources are limited. Therefore, most of the time, minor issues are ignored at code reviews for simplicity to prevent a never-ending back and forth between the reviewer and contributor. If a feature is working as expected, it is usually accepted and merged. It is essential to avoid such abbreviated code reviews whenever possible, to ensure software of high quality.

Nevertheless, mostly students' first contributions are highly error-prone and regularly contain code smells and anti-patterns. Clean and precise code that is easily understandable by other developers is the exception rather than the norm. It is a tedious and time-intensive task for code reviewers to explain and highlight every little detail to new contributors over and over again. Besides, senior developers and code reviewers at Catroweb are just students of higher classes and only have limited time and knowledge. Code reviews that require multiple hours to work up all problems in a pull request are no rarity.

A CI system strives to support code reviewers by automating as many checks as possible. Using CI, every contribution must fulfill a particular standard since tests run automatically on a server for every contribution and automate the most redundant monotonous processes which elongate the reviewing process. Developers get feedback from a CI system almost immediately instead of waiting days for a human code review. However, all

automated tests and checks only have limited value if a manual code review is missing. Therefore, an honest and detailed code review is still essential to improve and maintain the quality of a project continuously and ensure that every new feature is appropriately designed and tested.
Nevertheless, a reviewer can use the gained time to focus on other aspects, like giving tips on more important topics like design patterns instead of checking every line of code for issues that can be auto-detected. Therefore, developers can achieve better reviews in less time.

In addition to automated tests, manual reviews significantly reduce the risk of integrating contributions that break existing features. Therefore, at Catroweb, multiple stages assure the quality of a contribution. After the code of a developer has passed the CI checks, a code review is first done by any developer, followed by a product owner. In case there are any modifications in the front-end, the UX-team is also doing a review of the design.

9 CI - Dynamic analysis

Dynamic analysis defines methods to test and evaluate software while it is running. Compared to static analysis, dynamic analysis does not evaluate every line of source code, but only those parts of the executed code. While static tests verify the source code to prevent defects, dynamic tests validate a program during its execution.
Using dynamic analysis, developers have to think about every possible scenario to test and develop tests for it. Hence, dynamic testing is expensive. In return, dynamic tests are capable of uncovering subtle flaws that are too complicated for static analysis. For example, only dynamic tests can evaluate the behavior of the software. Therefore, dynamic analysis is usually performed after static analysis checks to achieve the best possible results.

Dynamic testing can be classified into two categories, namely, functional and non-functional testing.
Non-functional tests are performed to evaluate and monitor the environment during a program's execution, such as memory and CPU usage. However, non-functional tests are currently only done manually in the Catroweb project and will not be integrated into the CI system.
On the other side, Catroweb heavily uses functional testing throughout the development to ensure that the results of predefined inputs, thrown into the application, match the expected outputs.

Noteworthy, Catroweb's test suites are not strictly separated into particular levels, nor do they follow the testing pyramid's principles. Without strict planning and the missing knowledge in the project over the years, Catroweb's test development has fallen apart. What once started with a thorough plan to test the Share community platform, utilizing BDD and TDD with all their benefits, left the project only with tests that are challenging to maintain. Sooner than later, developers started to mix principles and testing levels, or even completely forgot to test new features. Catroweb's test suites rely on three different test automation frameworks, even while almost all tests are a mix-up between UI and integration tests, with nearly no unit tests.

The following sections describe how the existing test suites in the Catroweb project have been refactored and integrated into a CI system, written using GitHub Actions, to provide a starting point to improve the testing quality and, as a result also the overall software quality.

9.1 Test automation frameworks

Test automation frameworks can significantly reduce software testing's maintenance costs and effort by increasing the speed and efficiency to develop software tests. They define guidelines and provide methods to handle and process test-data. A single command is sufficient to execute whole test suites. Typically, different frameworks are optimized for different domains and languages to utilize all possible benefits.

The Catroweb project is a web application mainly written in PHP and relies on testing tools and frameworks developed for PHP and web development. The majority of tests are written as Behat[1] tests and only a small number is either a PHPUnit[2] or PHPSpec[3] test.

PHPUnit is the leading framework to write unit tests for PHP. With PHPUnit, developers can write any tests without limitations, from complex functional tests to simple unit tests. On the other hand, Behat and PHPSpec explicitly focus on certain types of tests. With Behat being a tool for StoryBDD, only functional tests should be written. In return, they support developers to think about the software's external quality, clarify the domain, and specify all feature requirements. Furthermore, PHPSpec is a SpecBDD tool that forces developers to think about PHP classes' design and behavior first, rather than just writing unit tests. (SymfonyCasts, 2020)

1. `https://docs.behat.org/` visited on 10 Sep. 2020
2. `https://phpunit.de/` visited on 10 Sep. 2020
3. `http://www.phpspec.net/` visited on 10 Sep. 2020

”StoryBDD and SpecBDD used together are an effective way to achieve customer-focused software.” (*Introduction to SpecBDD and StoryBDD* 2020)

While each framework could provide its benefits, they are currently not effectively utilized in the Catroweb project. The Share community platform is mainly developed by students who miss the knowledge and time to learn three test automation frameworks at once because, at the same time, they also have to learn various aspects of web development. Without proper knowledge about their differences and how to efficiently create software tests with them, a tool may be misused, and hence, the usage differs from the intended use case. The past years of the Catroweb project have shown that many students stick to one framework to rule them all. For example, many failing PHPSpec and PHPUnit tests have not been fixed or extended. Developers have only written Behat tests for every possible use case, slowly drifting away from its StoryBDD principles.

As long as developers do not utilize each testing framework's benefits, developers only enjoy the drawbacks of maintaining three different frameworks. Therefore, the least used framework, PHPSpec, is removed from the project to reduce the complexity of writing and maintaining software tests to counteract this issue. All existing PHPSpec tests have been refactored to PHPUnit tests. Having only two different frameworks to learn should be less daunting, and a clear assigned purpose for each framework should highlight the difference between unit and functional tests. Summarized, this should facilitate inexperienced developers the entry into the world of testing. In the future, developers should write the majority of all tests as unit tests using PHPUnit and a few functional tests with Behat to reduce testing costs. A more granular separation of testing types and levels does not seem to be feasible at the moment.

9.2 Continuous integration of dynamic analysis

Automated dynamic analysis feedback in the Catroweb project aims to immediately detect issues and broken features due to integrating incompatible new contributions. Therefore, before, but also after every integration, a pull request is tested to match the expected outcome of various predefined test cases. In return, developers get immediate feedback if a new feature changes or breaks the existing feature specifications. However, all features must be sufficiently tested to ensure that the tests can detect problems automatically. Hence, to motivate developers to write enough tests, code coverage metrics are added to the feedback to indicate insufficiently tested features.

Like static analysis, all dynamic checks could be run locally by developers before creating a pull request. However, the tests require multiple hours to finish and a fully configured environment to run the testing frameworks. Apart from that, flaky tests, as well as caching issues, can distort the real results. With automated dynamic analysis in the CI system, code reviewers do not need to trust developers to run the tests or run them by themselves, significantly reducing the required time of a code review. Looking at Catroweb's history, it can be seen that developers tend to forget to run all tests, especially on small patches. However, in a complex system with many dependencies, this regularly resulted in broken features that required an immediate hotfix later on.

9.2.1 Automate the process

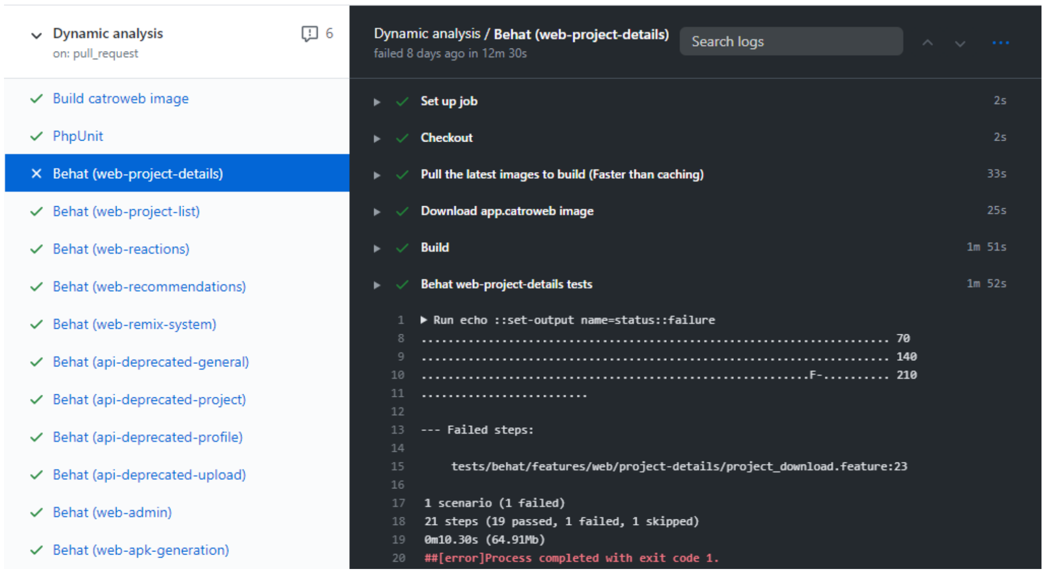
Every creation, update, and merge of a pull request in the Catroweb repository triggers a dynamic software testing workflow. While previous static analysis checks only required access to the source code and a few installed dependencies, Catroweb's dynamic analysis checks require a running application in a fully set up environment. Installing and configuring various programs during the workflow would be possible but challenging to maintain. However, using Docker, setting up the environment for software testing is significantly simplified.

After checking out the source code, the workflow uses Docker Compose to start all of Catroweb's Docker containers. Docker Compose automatically downloads and builds all missing images first. Once the containers are up and running, the tests are executed in the Share community platform Docker container (Listing 9.1).
If a test case fails, also the workflow step fails. However, like the static analysis workflow jobs, all tests finish even after failures and do not stop

testing in the middle of the process. As a result, the CI system provides developers with full feedback at all times. Every failed job provides logs that contain useful details on which tests failed (Figure[4] 9.1).

```yaml
name: Dynamic analysis
on: [push, pull_request]
jobs:
  job-id:
    name: name-of-the-testing-framework
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v2
      - run: docker-compose -f compose-file up -d
      - run: docker exec container test-framework
```

Listing 9.1: Dynamic analysis workflow: Dynamic analysis requires a program to be running. Using Docker, the environment setup process is simplified to a single command. The test automation frameworks are then executed in the running container.



Dynamic analysis / Behat (web-project-details)
failed 8 days ago in 12m 30s

- ✓ Build catroweb image
- ✓ PhpUnit
- ✗ Behat (web-project-details)
- ✓ Behat (web-project-list)
- ✓ Behat (web-reactions)
- ✓ Behat (web-recommendations)
- ✓ Behat (web-remix-system)
- ✓ Behat (api-deprecated-general)
- ✓ Behat (api-deprecated-project)
- ✓ Behat (api-deprecated-profile)
- ✓ Behat (api-deprecated-upload)
- ✓ Behat (web-admin)
- ✓ Behat (web-apk-generation)

```
▸ ✓ Set up job                                              2s
▸ ✓ Checkout                                                2s
▸ ✓ Pull the latest images to build (Faster than caching)  33s
▸ ✓ Download app.catroweb image                            25s
▸ ✓ Build                                               1m 51s
▸ ✓ Behat web-project-details tests                     1m 52s

▸ Run echo ::set-output name=status::failure
...................................................................... 70
...................................................................... 140
...........................................................F-.......... 210
.........................

--- Failed steps:

    tests/behat/features/web/project-details/project_download.feature:23

 1 scenario (1 failed)
 21 steps (19 passed, 1 failed, 1 skipped)
 0m10.30s (64.91Mb)
 ##[error]Process completed with exit code 1.
```

Figure 9.1: Failing test suite feedback (Behat): The GitHub Actions interface provides an immediate indication of failed jobs and provides detailed information about each failing test case.

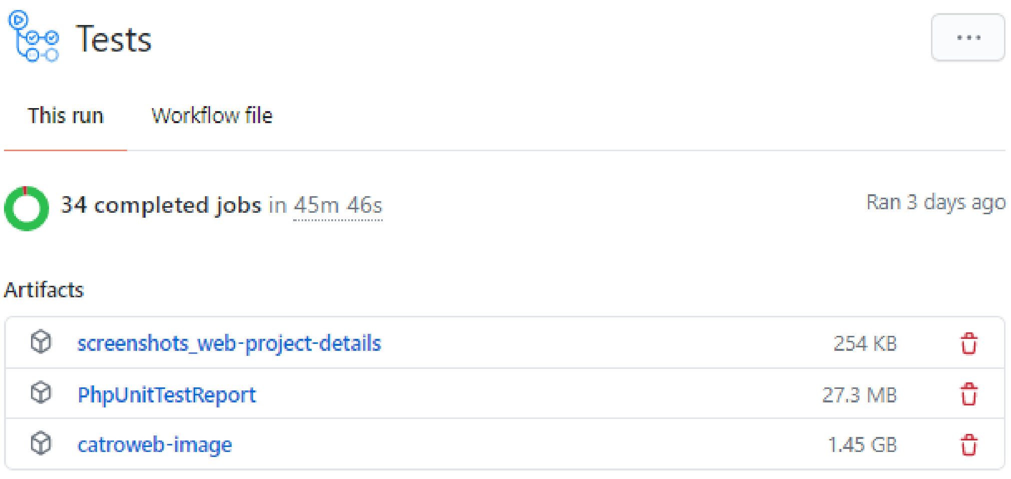
4. `https://github.com/Catrobat/Catroweb/runs/1036079860` visited on 9 Sep. 2020

The logs are immediately available and do not require the whole workflow to finish. Therefore, developers can already fix the first issues while the remaining tests finish.

Still, several optimizations described in the following sections are necessary to increase the workflow's usability, reliability, and performance. The final resulting workflow file is available on GitHub[5].

9.2.2 Artifacts

GitHub Actions can upload any files as artifacts, which is crucial to share content between workflows and jobs. Besides, an uploaded artifact can also be downloaded directly from the GitHub Actions interface on GitHub. Therefore, generated content like feedback reports or debugging information like screenshots of the browser of failed scenarios can be easily accessed and downloaded from GitHub (Figure[6] 9.2).



34 completed jobs in 45m 46s

Ran 3 days ago

Artifacts

screenshots_web-project-details	254 KB
PhpUnitTestReport	27.3 MB
catroweb-image	1.45 GB

Figure 9.2: Download artifacts: Uploaded artifacts can be used in other jobs or downloaded from the GitHub interface.

5. `https://github.com/Catrobat/Catroweb/blob/develop/.github/workflows/tests.yml` visited on 9 Sep. 2020
6. `https://github.com/Catrobat/Catroweb/actions/runs/226844591` visited on 30 Aug. 2020

The action to upload an artifact requires only two configuration parameters: a name for the artifact and the file's path to upload. However, in case the artifacts should be uploaded after a failed step, it is crucial to binding the step to a condition that ensures that the step is always executed even if the job failed (Listing 9.2). The associated action to download an artifact is almost identical to use.

```yaml
- uses: actions/upload-artifact@v2
  if: always()
  with:
    name: name-of-artifact
    path: path-to-file(s)
```

Listing 9.2: Upload artifacts: Files can be uploaded during GitHub Actions as an artifact. The uploaded artifacts can be shared between various jobs in a workflow.

9.2.3 Parallel testing

Catroweb has written many of its tests in Behat. To be specific, currently, there exist approximately 15.000 steps that form 900 test cases. Unfortunately, Behat tests do not support the parallel running of multiple tests out of the box. Trying third-party packages to do so only results in various issues. Most Behat tests in the Catroweb project require a particular setup of the environment and database. Usually, the database is filled in the first steps of a scenario. Running multiple test cases at the same time results in numerous race conditions and invalid data fixtures. Preventing those issues would require a very time-consuming refactoring of all test suites.

However, GitHub Actions have support to run multiple parallel jobs at once. Splitting the Behat tests into numerous test suites with a few configuration lines is sufficient to achieve parallelism. There is no need for a costly refactoring and modification of the existing tests and framework. With a matrix build in GitHub Actions, a workflow spawns a new job for every test suite (Listing 9.3). Each of these jobs runs independently from others. While this does not reduce the overall computation costs, it significantly reduces the time until the workflow is finished. The distributed test suites finish their test runs in under half an hour instead of two.

```yaml
tests_behat:
  name: Behat
  needs: build
  runs-on: ubuntu-latest
  strategy:
    fail-fast: false
    matrix:
      testSuite:
        - admin
        - api
        - projects
        - user
        - ...
  steps:
    - uses: actions/checkout@v2
    - run: docker-compose -f compose-file up -d
    - name: Behat ${{ matrix.testSuite }} tests
      run: docker exec container path-to-behat --suite ${{
         matrix.testSuite }}
```

Listing 9.3: Matrix build: GitHub Actions support a matrix build to spawn several similar jobs in a workflow. Running every Behat test suite parallel in its independent job reduced the workflow's required time to finish.

Since the tests still run quite long, developers should get full feedback at all times. Providing developers with only one issue per run is not very efficient. Therefore, it is vital to turn off the fail-fast workflow strategy to ensure that all tests get a chance to finish, even if others fail.

9.2.4 Caching

Starting Catroweb's Docker containers, especially building the Share community platform image, requires much time since it is built numerous times throughout various jobs. Unfortunately, GitHub Actions yet do not support layered caching for Docker Compose. Therefore, each job requires around 15 minutes to build the image. However, it is possible to significantly reduce the building time by repeatedly reusing the same image throughout the workflow file. Therefore, the build is performed in its independent job to accomplish this task. Once the image is built, it is saved and uploaded to

GitHub as an artifact (Listing 9.4). The artifact is then downloaded and used in other jobs (Listing 9.5). Instead of building the same image several times, the jobs depending on it can wait for the build job to finish and load the image into Docker before initiating the Docker Compose process to start all containers. Hence, the overall build time is significantly reduced.

```
build:
  name: Build image
  runs-on: ubuntu-latest
  steps:
    - uses: actions/checkout@v2
    - run: |
        docker-compose -f compose-file build image
        docker save image > image-name.tar
    - uses: actions/upload-artifact@v2
      with:
        name: image-name
        path: docker-path/image-name.tar
```

Listing 9.4: Save Docker image: After building an image, Docker can save and compress the image to store it in the filesystem. The saved image then can be uploaded and shared as an artifact.

```
<job-id:
  needs: build
  name: name-of-the-testing-framework (test-suite)
  runs-on: ubuntu-latest
  steps:
    - uses: actions/checkout@v2
    - uses: actions/download-artifact@v2
      with:
        name: image-name
        path: docker-path
    - run: |
        docker load < docker-path/image-name.tar
        docker-compose -f compose-file up -d
    - run: docker exec container path-to-testing-framework
```

Listing 9.5: Load Docker image: Instead of building a Docker image from scratch in every job, it can be downloaded as an artifact. Docker can load a saved image in a fraction of the time it would require to build it.

9.2.5 Flaky tests

Software tests should be reliable, scalable, and reusable. However, not all code in software is deterministic. Random aspects like the timing can have a significant impact on the result of an action. Therefore, software testers try to remove as many randomnesses as possible from a test-case and produce zero entropy environments.

A flaky test is a non-deterministic test that sometimes fails, but some times not. Especially UI tests are very prone to fail with regard to concurrency and timing issues. Flaky tests are even worse than no tests since they reduce the trust in the automated test system. In the worst-case scenario, developers accept a failing test thinking it is a flaky test, while the feature is broken. Besides, developers want immediate feedback about failed scenarios without manually verifying the results because of false negatives. However, from time to time, flaky tests are usually integrated into a project. Even large companies like Google point out that 1.5% of all their tests are continually reporting a flaky result. (Micco, 2020)

Unfortunately, half of Catroweb's Behat tests are fragile UI tests that control a browser where the result depends on the exact timing. Approximately 450 out of 900 test cases are exposed to the danger of being flaky. Running those 450 tests 100 times results in an average of 73 failed test cases per run, leaving the test suites with a rate of false negatives of over 15 percent.

Therefore, the Behat scenarios have been refactored to wait for a web page and its content to be loaded explicitly before starting the validation process to reduce the flakiness (Listing 9.6).

```
Scenario: Welcome section
  Given I am on the homepage
  And I wait for the page to be loaded
  Then I should see the welcome section
  And I wait for the sign-in button to be visible
```

Listing 9.6: BDD feature description: A statement to wait for the page to be loaded is added to every single test and page transition. This approach is significantly cheaper than refactoring every unique method to wait for a particular element to prevent flaky tests. However, some elements may still require polling.

Furthermore, dynamic elements are checked several times to ensure that a test did not fail because of bad timing until they run into a timeout after a few seconds. The performance drop in overall speed was only minimal and can be neglected. However, with this modification, a test run only contains, on average, five tests out of 450 with flaky results. Therefore, the flakiness of the tests has been reduced to around one percent. To further reduce the flakiness, specific measures optimized for the particular test case must be applied. Still, even with all the modifications, flaky tests cannot be 100 percent prevented. Besides, the provided measures are only successful if developers keep to utilize them while developing new tests.

Manually rerunning test suites multiple times to ensure that a test case did not just fail because of bad timing is very cost-intensive. Instead, the workflow is modified to rerun a failed test case three times automatically. If a test case still fails, there is a high chance the issue is real and was not caused by bad timing. Luckily, Behat has already built-in support to rerun only failed scenarios. However, the GitHub Actions job must only fail if there is still a failing test after all reruns. Therefore, it is essential to explicitly configure each of the previous test run steps so that they must not fail by setting them to continue on error. Only the last rerun must not continue on an error but fail.
However, if no test case fails, the rerun command reruns all test cases again. Therefore, it is important to execute the rerun steps only if necessary. In their current state, steps in GitHub Actions can not access the result of a previous step. However, a variable can be explicitly set to failure at the beginning of a step. Next, the tests are executed, and if a test fails, the step stops before executing the remaining command to set the variable to success. It is essential to provide each step with a unique ID to access the variable from another step. By binding a condition to an additional step based on the previous steps' variables, the workflow job can decide if it should run a step or not. (Listing 9.7)

Unfortunately, this approach is limited. Since only the last test run can fail, issues in the first run may be missed. For example, tests with undefined or pending step definitions are not included in the tests that must be rerun. Apart from that, severe issues can crash the browser in which the tests run. If something like that happens, it is not useful to rerun the tests. Therefore, the first test run is again modified to prevent those issues. The output is logged into a file, and additional steps parse the output to ensure that the log does not contain any of the breaking issues. In case they detect one of those issues, the workflow fails immediately without rerunning the tests. (Listing 9.8)

```yaml
- name: Test run
  id: test-run
  continue-on-error: true
  run: |
    echo ::set-output name=status::failure
    docker exec container path-to-framework
    echo ::set-output name=status::success

- name: 1. rerun
  if: steps.test-run.outputs.status != 'success'
  id: test-rerun-1
  continue-on-error: true
  run: |
    echo ::set-output name=status::failure
    docker exec container path-to-framework --rerun
    echo ::set-output name=status::success

- name: 2. rerun # (Similar to 1. rerun)
  if: steps.test-run.outputs.status != 'success' && steps.
     test-rerun-1.outputs.status != 'success'
  id: test-rerun-2
  continue-on-error: true
  run: |
    echo ::set-output name=status::failure
    docker exec container path-to-framework --rerun
    echo ::set-output name=status::success

- name: 3. rerun
  if: steps.test-run.outputs.status != 'success' && steps.
     test-rerun-1.outputs.status != 'success' && steps.test
     -rerun-2.outputs.status != 'success'
  run: |
    docker exec container path-to-framework --rerun
```

Listing 9.7: Rerun failed Behat tests: All failing Behat scenarios must be rerun three times. Only the last iteration should decide if a test fails. This approach reduces the risk of a flaky test result in the final feedback.

```
- run: |
    cat path-to-log-file;
    if grep -q "error-ID" < path-to-log-file; then
      false;
    fi
```

Listing 9.8: Filter output for critical issues: Since the first test run can not fail, the output must be thoroughly checked for errors that should immediately fail a job instead of rerunning the test suite.

9.2.6 Code coverage

Code coverage is a collection of metrics computed by various tools during the execution of test suites to describe the degree of tested source code lines to assess software tests' quality. Usually, many test automation frameworks support code coverage metrics out of the box to automatically generate code coverage reports during a test suite's execution.

There is a high chance that software with high code coverage is better tested than software with low coverage. However, code coverage can only hint at the quality of software tests without guaranteeing it. Test quality can only be assured by including various additional metrics that have to be evaluated manually by static reviews, such as efficiency and effort (Swati Seela, 2020). Without quality testing, as Fowler (2020b) points out, high code coverage numbers can be easily accomplished. Nevertheless, software teams should strive for a code coverage between 80 and 90 percent.

PHPUnit has already built-in support to measure the code coverage. However, for Behat, code coverage is not supported. Kudryashov (2020), a maintainer of Behat, points out that code coverage is nonsense for storyBDD, especially in Behat, since feature descriptions are not tests about the code itself but the behavior. Nevertheless, in many projects, such as Catroweb, developers do not draw such a strict distinction between the description in feature files and the hidden logic in context files (Wright, 2020). Fortunately, there are plugins for Behat to support code coverage, requiring only a minimum of configurations.

Providing developers with continuous feedback about the coverage metrics helps them ensure that enough tests have been written. In the current state of Catroweb, the code coverage is still meager, and developers at Catroweb should focus on increasing it. Therefore, the test automation frameworks have been configured to generate code coverage reports during their execution. However, since the tests are split into numerous independent test suites, it is essential to combine the reports first, rather than uploading countless independent coverage reports as artifacts. Therefore, a tool, free to use for open-source projects, called Codecov, is added to the project to accomplish this task.

Codecov

Codecov[7] can be configured through a single file in a project's repository. After initializing Codecov in a project, the official Codecov action (Listing 9.9) is used to upload every report created by the parallel and independently running test suites of this workflow. Codecov automatically merges all coverage reports of the same pull request and provides developers with a single feedback report.

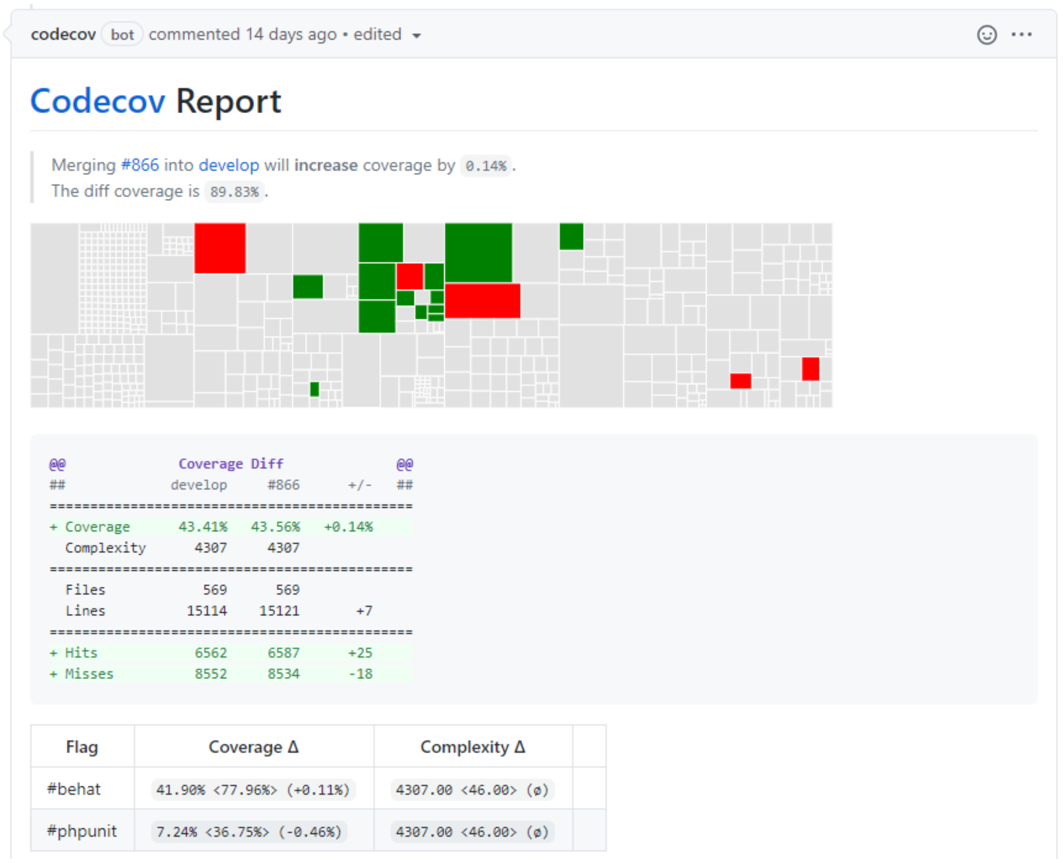
```
- uses: codecov/codecov-action@v1
  with:
    file: path-to-code-coverage-file
```

Listing 9.9: The official Codecov action is used to upload code coverage reports to Codecov.

Codecov is integrated seamlessly into GitHub. Once the report is fully assembled, a summary of the code coverage metrics is added as a comment to a pull request, giving a clear indication about the increase or decrease of code coverage due to this new contribution (Figure[8] 9.3). A more detailed report with access to inspect every line of code is not included in GitHub but on the Codecov web-interface. Codecovs web-interface also provides various additional charts to analyze the code coverage metrics further.

7. https://codecov.io/ visited on 9 Sep. 2020
8. https://github.com/Catrobat/Catroweb/pull/883 visited on 9 Sep. 2020

Furthermore, this tool can be configured to fail the CI pipeline if the code coverage falls below a certain threshold. This threshold can be absolute or relative to the changed files of a pull request. Since Catroweb's current code coverage is still under 50%, using the relative coverage measurements seems more suitable. The relative code coverage is only concerned about files that are modified in the pull request.


codecov bot commented 14 days ago • edited

Codecov Report

Merging #866 into develop will **increase** coverage by `0.14%`.
The diff coverage is `89.83%`.

```
@@            Coverage Diff            @@
##           develop     #866   +/-   ##
=========================================
+ Coverage    43.41%   43.56%   +0.14%
  Complexity    4307     4307
=========================================
  Files          569      569
  Lines        15114    15121       +7
=========================================
+ Hits          6562     6587      +25
+ Misses        8552     8534      -18
```

Flag	Coverage Δ	Complexity Δ	
#behat	`41.90% <77.96%> (+0.11%)`	`4307.00 <46.00> (ø)`	
#phpunit	`7.24% <36.75%> (-0.46%)`	`4307.00 <46.00> (ø)`	

Figure 9.3: Codecov GitHub comment: The code coverage reports are uploaded and merged through GitHub Actions using Codecov. Then, Codecov comments a summary to the pull request.

10 Continuous delivery

Apart from a reliable CI system, it is crucial to automate the release process required by CD to establish CI/CD in a project. Short release cycles require a well-tested codebase to prevent releasing critical bugs and issues to the public. While the code coverage of the Share community platform could be higher, the CI system implemented in this thesis should provide a strong foundation to use continuous delivery in production at Catroweb.

On the other hand, applying continuous deployment in the project's current state would be of high risk. Without any human interaction, it is necessary to improve the tests' quality and coverage first. Besides, it would be required to rework the currently used workflow to use continuous deployment. In contrast, Catroweb's workflow perfectly fits a development using continuous delivery. However, continuous deployment is already used for deployments in the test environment.

This chapter describes test deployments of pull requests during development, followed by an automated release to production.

10.1 Continuous test deployments

It is essential to test new features before releasing them to production. To manually test a contribution, it is necessary to set up the project and checkout the pull request. Without knowledge of the project and the deployment process, it can be challenging to accomplish this task. Automatically deploying the changes to a public server improves the accessibility and removes the hurdle for non-developers. Therefore, new features can be accessed using only a browser without a complex local environment to host the project.

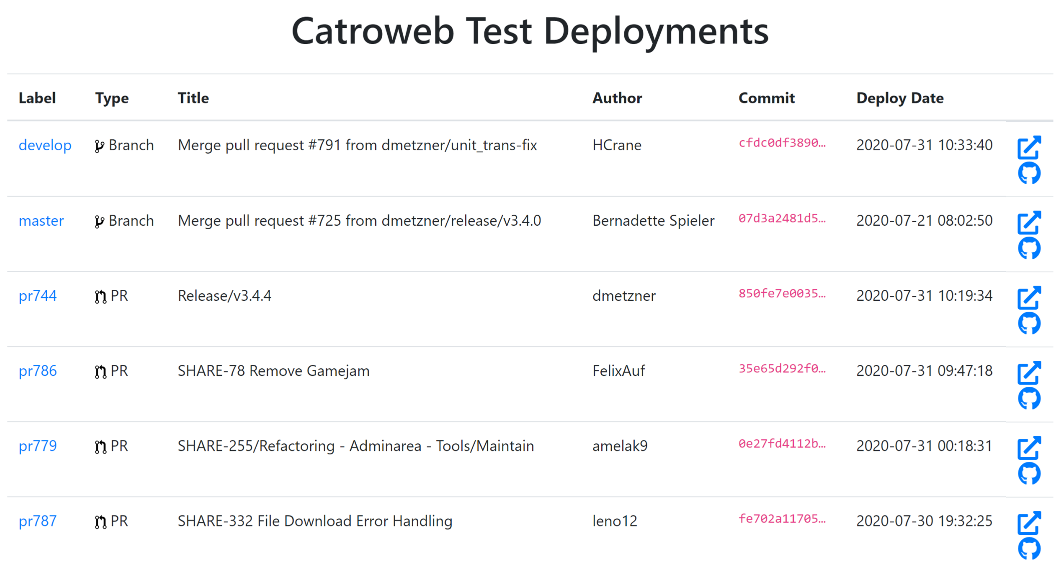
However, in the past, this deployment process was done manually by developers and required valuable time. Additionally, it was only possible to deploy one pull request per server at once. With limited available servers, developers were in the constant need to switch the deployed versions.

10.1.1 Automate the process

At the same time, when this thesis started, a member of the Catroweb team developed a program to deploy every open pull request to the Catroweb repository automatically. This auto-deploy program utilizes a wild card domain to deploy multiple pull requests on the same server at once, which reduces the needed number of servers in use by the Catroweb team. A scheduled task on a test server runs a python script to check for new pull requests on GitHub every few minutes. The script deploys a new pull request to the test server shortly after its creation on GitHub. In case a pull request is updated, the system redeploys the pull request. After merging or closing a pull request, the auto-deployer removes all the pull request data from the test server. Figurer[1] 10.1 shows the web-interface of the auto-deploy program that allows users to navigate between the different deployed versions. The script is available as an open-source project on GitHub[2].

In projects that apply CD, a passing pipeline is usually the baseline to initiate the deployment process. However, this script works independently of the CI system. Even a failing pipeline deploys a pull request. The checks are ignored by intention to enable a pull request to be manually tested while it is still work in progress. Especially in a test-first development, unfinished features have new tests that fail during the development. This approach does not fit the production environment but works well during development. Hence, non-developers, such as the usability team and product owners, can react quickly if a feature development went wrong.

1. `https://web-test-1.catrob.at/` visited on 31 Jul 2020
2. `https://github.com/Catrobat/Catroweb-AutoDeploy` visited on 9 Sep. 2020



Catroweb Test Deployments

Label	Type	Title	Author	Commit	Deploy Date	
develop	Branch	Merge pull request #791 from dmetzner/unit_trans-fix	HCrane	cfdc0df3890…	2020-07-31 10:33:40	
master	Branch	Merge pull request #725 from dmetzner/release/v3.4.0	Bernadette Spieler	07d3a2481d5…	2020-07-21 08:02:50	
pr744	PR	Release/v3.4.4	dmetzner	850fe7e0035…	2020-07-31 10:19:34	
pr786	PR	SHARE-78 Remove Gamejam	FelixAuf	35e65d292f0…	2020-07-31 09:47:18	
pr779	PR	SHARE-255/Refactoring - Adminarea - Tools/Maintain	amelak9	0e27fd4112b…	2020-07-31 00:18:31	
pr787	PR	SHARE-332 File Download Error Handling	leno12	fe702a11705…	2020-07-30 19:32:25	

Figure 10.1: Catroweb's test deployments: Catroweb's auto-deploy system provides an interface to navigate between pull requests from GitHub that have been automatically deployed onto a test server by a scheduled script which runs on a test server.

10.1.2 Challenges

Implementing GitHub Actions to initiate test deployments is impossible using the pull request event as a trigger. Several secrets are necessary to configure the deployment process. Due to the git fork workflow used by the web team, GitHub Actions triggered by forked repositories have no access to the secrets defined in the official repository. However, a similar implementation via GitHub Actions using scheduled events would be possible. Still, no such workflow has been implemented during this thesis since the existing approach works perfectly fine and does not require many resources to maintain. Implementing scheduled GitHub Actions that can be used to access secrets without the security risks of sharing secrets is described in chapter 11.

10.2 Continuous release deployment

As already mentioned in section 5.3, there exists a master branch in the Catroweb repository that should represent a direct reflection of the currently deployed Share community platform. Hence, the master branch can also be referred to as the production branch. However, in its current state, deploying a new release is possible without the need to integrate the changes into this branch or vice versa. There exists nothing to enforce this one-to-one relationship. When comparing the latest versions of the production branch on GitHub with the versions deployed to the server, some differences can be seen. Apart from that, manually deploying a release still requires a developer with access to all credentials and a fully set up development environment. The need for a fully configured environment makes it complicated for non-developers like product owners to quickly deploy a new release. This section describes a workflow that automatically deploys the newest release to the production server to meet the technical requirements of using CD in the Catroweb project.

10.2.1 Automate the process

The workflow described in this section automatically deploys a new version as soon as the production branch changes. Consequently, developers do not need to care about releasing the latest version anymore. As soon as a new release is merged into the production branch, the delivery process is initiated. A merge into other branches, such as the develop branch, does not trigger the process because other branches usually contain new features that may not be ready to be released yet. Therefore, as a first step, it is critical to define that the job must only react to a push into the production branch (Listing 10.1).

```yaml
on:
  push:
    branches: [ master ]
```

Listing 10.1: The workflow is only initiated on a push to the master branch.

Next, the workflow checks-out the source code and installs a program called OpenConnect. Using OpenConnect, a virtual private network (VPN) connection to the server's location in the network of TU Graz is established, which is necessary because the server is only reachable using SSH within the university network for security reasons. SSH, short for Secure Shell, is a cryptographic network protocol to secure connections in an insecure network, like the internet. The OpenConnect VPN connection is configured with credentials piped into the process as parameters. (Listing 10.2)

```
- run: |
    sudo apt-get --yes --force-yes install openconnect
    printf "${{ secrets.VPN_PASSWD }}" | sudo openconnect
        vpn.tugraz.at --user=${{ secrets.VPN_USER }} &
        disown
```

Listing 10.2: Establish a VPN connection in GitHub Actions using secrets.

It is important to note that the credentials are stored as secrets on GitHub instead of directly including them in the workflow file. For simplicity, the workflow reuses Catroweb's existing deployment script from here on. However, setting up the environment requires even more credentials. Most required credentials, hidden in the secrets' storage, must be written into the local environment files. (Listing 10.3)

```
- run: printf "key=${{ secrets.key> }}\n" >> .env.local
```

Listing 10.3: Set up private credentials using secrets: Various secrets must be piped into the configuration files.

Catroweb's deployment script is built within the Symfony framework and its packages. Therefore, Docker is used to set up the necessary environment. Next, inside the Docker container, SSH must be installed and configured with a private SSH key. The corresponding public key is registered on the server. Setting up the server must be done once manually and independently of this workflow. Finally, the script initiates the deployment. The script is responsible for updating the server's code, clearing the cache, installing new assets and dependencies, migrating the database, and restarting all services. (Listing 10.4)

Although this workflow's complexity compared to previously defined workflows, less than 50 lines are sufficient to create the workflow. The code is available in the Catroweb GitHub repository[3].

```
- run: docker-compose -f compose-file up -d
- run: docker exec container-name install-ssh
- run: docker exec container-name configure-ssh
- run: docker exec container-name deploy-command
```

Listing 10.4: Initiate deployment from inside a Docker container: The deployment is done inside a Docker container. Therefore, SSH must be installed and configured before initiating the deployment script.

10.2.2 Challenges

With the need for five different secrets, it is essential to keep them up-to-date for the GitHub Actions to work flawlessly. An invalid or missing secret results in a failing deployment process. Some secrets must be updated from time to time. For example, the university credentials are only valid for a maximum of 450 days. In case of a failing deployment process, it is necessary to update the secrets and rerun the workflow.

During the development, from time to time, not only the code changes but also its environment. Environmental changes, like a new PHP version, must still be done manually. Therefore, before starting the deployment, a developer must connect to the server and upgrade all necessary dependencies. To prevent this manual interaction, the deployment process at Catroweb requires a fundamental change. Instead of a native setup of the environment, Docker containers will be responsible for hosting the production services, similar to the testing and development environment. Instead of only deploying a Docker image, Kubernetes[4] will be used to cluster multiple containers. The transition to Kubernetes is not part of this thesis. However, it is already a work in progress by the Catrobat team.

3. https://github.com/Catrobat/Catroweb-API/blob/develop/.github/workflows/deploy.yml visited on 9 Sep 2020
4. https://kubernetes.io/ visited on 10 Sep. 2020

11 Workflow automation

Apart from CI/CD, several other software development tasks are applicable to be automated. Automating workflows removes the risk of human errors and highly reduces their complexity, even allowing non-developers to take care of those tasks. Less manual work drastically reduces the workload on developers and hence reduces the development costs.

The workflows to automate can be entirely independent of the code in the project. From sending out notifications or creating meeting notes once a week, almost everything is possible. The GitHub Actions marketplace provides access to all kinds of pre-defined workflows. In case no appropriate GitHub Actions exist, writing one from scratch has almost no limitations. However, this chapter describes how workflows can be automated using GitHub Actions in the Catroweb project.

11.1 API code generation

Until 2020 unstructured routes without any standards nor documentation were the weak fundament of Catroweb's API. Other teams could not use the API effectively. Every small detail had to be reverse-engineered by manually searching through the Share community platform's codebase. Besides, the web team communicated no changes to other teams, which resulted in problems too often. Therefore, based on an explicitly defined REST OpenAPI specification, the development of a new API system has started during the last year. The OpenAPI[1] specification is a broadly adopted standard in the industry for describing modern APIs. This approach defines

1. `https://www.openapis.org/` visited on 9 Sep. 2020

Catroweb's API specifications in a single Swagger file[2]. The specifications describe in a detailed fashion how to construct a request and what responses to expect, providing a clear contract between the web team and everyone using the API. Any Swagger editor can render the specifications to create a pleasant and precise representation of the API.

Catroweb uses an OpenAPI code generator[3], with configurations, enabled to support PHP and Symfony, to generate a significant part of the code necessary to implement the described API. Catroweb uses a repository on GitHub to host the auto-generated code as a package. Therefore, a dependency manager like Composer can include the package into the Catroweb project. Developers only need to implement the remaining missing logic, like database queries, to satisfy the API requirements. Such an automated process highly reduces the work that must be done by the developers. Moreover, generated routing configurations, interfaces, models, and controllers ensure the consistency between the actual implementation and the specifications.

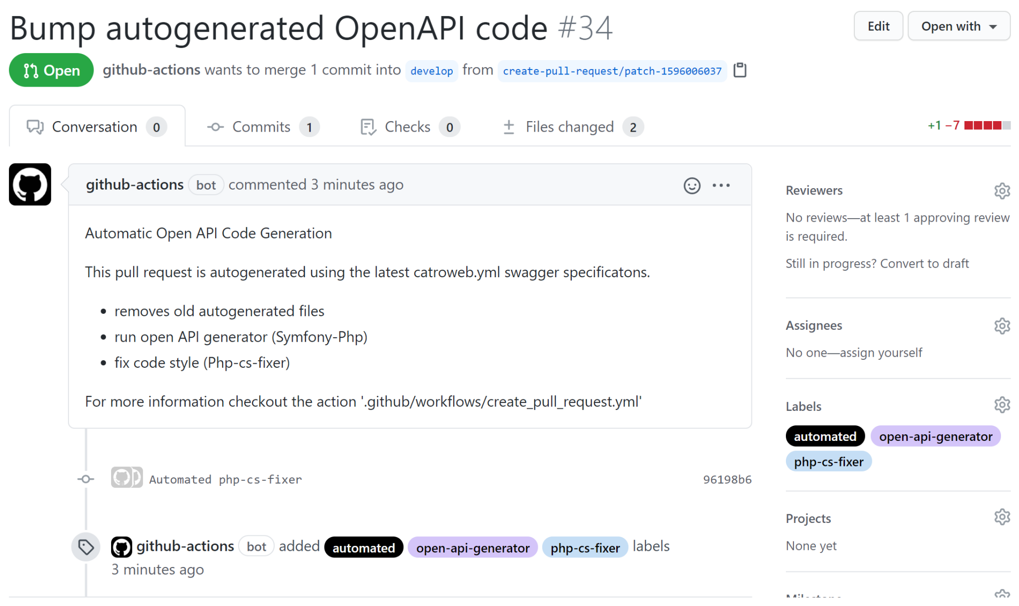
11.1.1 Automate the process

The API code itself is auto-generated, but the process of generating the code still requires manual work. The code must be up-to-date with the specifications at all times. This section implements a workflow using GitHub Actions to automate the code generation process to reduce developers' workload.

Whenever the specification is updated, developers must set up the generator, delete the old files, re-generate the code, fix its style, and push it to the repository. From now on, the development team only has to click a button on GitHub to trigger the generation whenever the content of the Swagger file changes. The generation process using this workflow is fast and does not need any local setup. Therefore, it can be done even by non-developers, like product managers. As a result, the workflow creates a pull request during its execution containing all updates (Figure[4] 11.1).

2. `https://github.com/Catrobat/Catroweb-API/blob/develop/catroweb.yaml` visited on 9 Sep. 2020
3. `https://github.com/OpenAPITools/openapi-generator` visited on 9 Sep. 2020
4. `https://github.com/Catrobat/Catroweb-API/pull/34` visited on 29 Jul. 2020

After being manually triggered, the workflow checks-out the code and removes all old auto-generated files. Removing all files first prevents code that is not used anymore from living in the package as dead code. After the deletion, the workflow pulls the latest OpenAPI generator Docker image from the network. As soon as the download ends, the generator is configured and executed to produce a Symfony compatible code. In the post-processing, the code style is fixed with the PHP-cs-fixer to match Catroweb's project style. Finally, the new files are automatically committed and pushed to the repository. In case of changes, the workflow creates a pull request into the develop branch. Instead of implementing the whole interaction with the GitHub API from scratch, the workflow utilizes an action[5] already written by other developers to create and update pull requests using GitHub Actions. Using this action reduces the needed code for this step to a few lines of configuration.


Bump autogenerated OpenAPI code #34

Open — github-actions wants to merge 1 commit into `develop` from `create-pull-request/patch-1596006037`

Conversation 0 · Commits 1 · Checks 0 · Files changed 2 · +1 −7

github-actions bot commented 3 minutes ago

Automatic Open API Code Generation

This pull request is autogenerated using the latest catroweb.yml swagger specificatons.

- removes old autogenerated files
- run open API generator (Symfony-Php)
- fix code style (Php-cs-fixer)

For more information checkout the action '.github/workflows/create_pull_request.yml'

Automated php-cs-fixer 96198b6

github-actions bot added automated open-api-generator php-cs-fixer labels 3 minutes ago

Reviewers: No reviews—at least 1 approving review is required. Still in progress? Convert to draft

Assignees: No one—assign yourself

Labels: automated, open-api-generator, php-cs-fixer

Projects: None yet

Figure 11.1: Code generation pull request: The code generation workflow creates a pull request during its execution. The pull request contains all automatic generated code updates in one commit with additional information in the description. The author of the commit is a GitHub bot.

5. https://github.com/peter-evans/create-pull-request visited on 9 Sep. 2020

Less than 30 lines of code define the resulting workflow file. A short version can be seen in Listing 11.1 while the complete and fully configured workflow is available on GitHub[6].

```yaml
name: Code Generation Pull Request
on:
  workflow_dispatch:
jobs:
  createPullRequest:
    name: Code Generation Pull Request
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v2
      - run: |
          remove-old-files
          docker run openapi-generator-with-config
      - uses: docker://oskarstark/php-cs-fixer-ga:latest
      - uses: peter-evans/create-pull-request@v2
```

Listing 11.1: Code generation workflow: After removing all files, new code is generated using Docker. Once the code style is fixed using PHP-CS-Fixer, a shared action is used to create a pull request.

11.1.2 Challenges

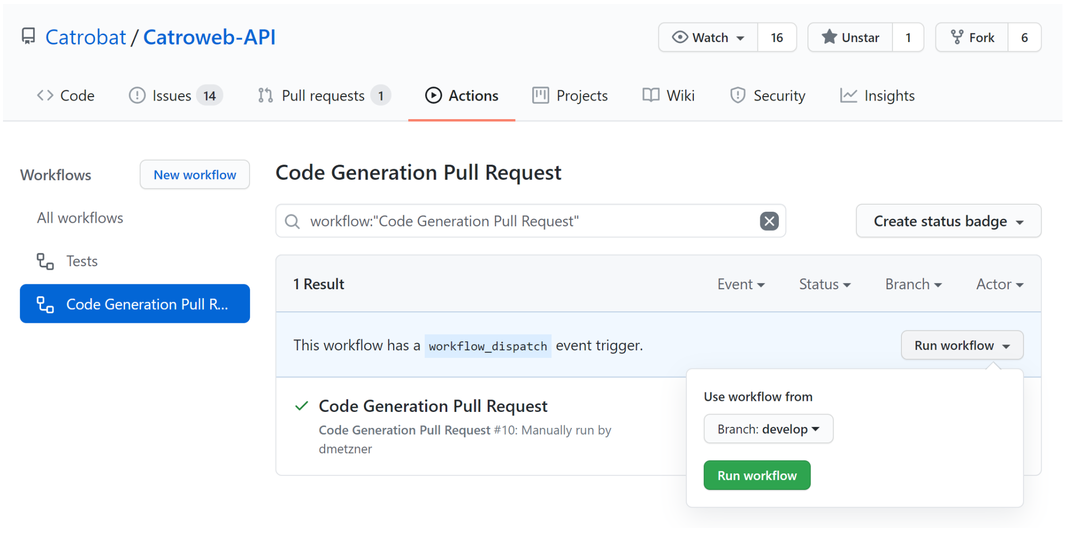
Automatically merging is no option with the workflow used by Catroweb. Before a merge, the new changes must be reviewed by at least two different developers. Therefore, pull requests have been chosen to integrate the updated code into the repository while keeping the four-eyes principle to maintain quality. Besides, it is impossible to merge into the develop branch without a pull request, since the develop branch is protected.

Different types of events to initiate the workflow have their unique benefits and drawbacks. A manual trigger has proven to be the most efficient one to use for this automatic code generation. Figure 11.2 shows the interface of this trigger directly integrated on GitHub. Using an event triggered by a pull request's creation has the same integration overhead as if developers would

6. `https://github.com/Catrobat/Catroweb-API/blob/develop/.github/workflows/create_pull_request.yml` visited on 9 Sep. 2020

create the code themselves. Pull requests would have changes in many different files. However, having only one modified file instead of many files reduces the risk of merge conflicts between multiple pull requests. Resolving fewer merge conflicts reduces development costs.

Using a trigger on the merge of a pull request prevents merge conflicts in auto-generated files. Nevertheless, a trigger like this is not perfect either. Typically, there exist multiple open pull requests to change the specification at once. There is a chance that developers merge pull requests in a short interval from time to time. With GitHub Actions in their current state, it is impossible to prevent multiple parallel executions of a workflow. Hence, race conditions might appear while working with a single branch for the code generation pull request. The only way to bypass this risk is to use a new branch accompanied by a new pull request for every new run. Since only the latest pull request contains the most recent changes, this wastes the CI system's computation time by generating pull requests with outdated content. All outdated pull requests must be closed and deleted manually, leaving developers at risk to remove the wrong pull request and merge an obsolete version. A human error like this might break a new release.



Code Generation Pull Request

workflow:"Code Generation Pull Request"

1 Result

This workflow has a `workflow_dispatch` event trigger.

✓ Code Generation Pull Request
Code Generation Pull Request #10: Manually run by dmetzner

Figure 11.2: Manual workflow trigger: A click on the manual trigger button initiates the code generation workflow. It is possible to configure the workflow's execution branch in hindsight.

A workflow utilizing a scheduled job at regular intervals prevents the parallel execution. However, it is still essential to use large enough time frames to avoid overlaps. Due to the mentioned restrictions, the manual trigger provides various benefits. The manual workflow trigger must be used once after integrating the specification changes before releasing a new API version. This approach prevents code generation spam in the repository due to outdated pull requests. It also keeps the git history shorter and saves valuable computation time of the CI/CD server.

11.2 Synchronize Crowdin translations

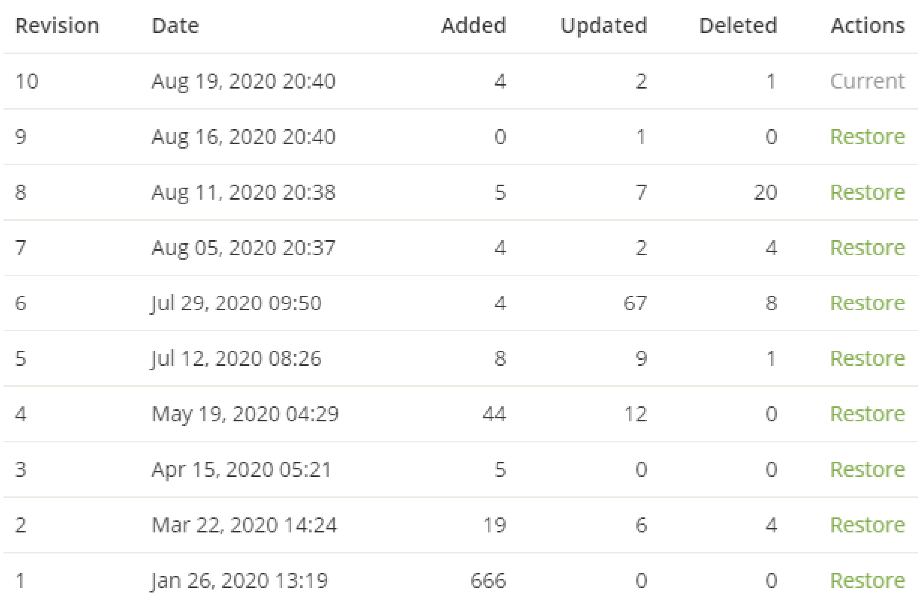
The Catrobat project, including its Share community platform, attaches great importance to reaching users worldwide. An essential aspect of achieving this is the internationalization and localization of all its services. While automatic translation continuously improves, it still lacks accuracy based on the context (Voita, Sennrich, and Titov, 2019). Therefore, to effectively manage all of the multilingual content, Catrobat uses Crowdin and manual translations.

Crowdin is a platform where projects, like those from Catrobat, regularly upload their translation source files. Catroweb provides its strings, which build the base for all other languages exclusively in English. Every translation pair consists of an ID, and the text requiring translations. At Crowdin, volunteers worldwide are allowed to translate the uploaded strings into any language. For every different language and even dialect, the translations can then be downloaded from Crowdin and included in the project. Thanks to Crowdin's service, the Share community platform can provide its users content in more than 100 languages and dialects. In case a translation is missing, the English base is available as a fallback alternative at all times.

11.2.1 Automate the process

New contributions frequently introduce new strings to the project. However, developers must upload them manually to the platform, and as soon as new translations are available at Crowdin, they must be downloaded and pushed into the Catroweb project. The upload and download process requires authentication, access to the correct configurations, and a developer's precious time. Even if this synchronization process only takes up a few minutes, it sums up high amounts of wasted resources over a more extended period. Therefore, the synchronization process was done at a maximum once a month to prevent the manual overhead (Figure[7] 11.3).

Consequently, newly released features are typically missing their translations for the first few weeks. However, by running this process daily, there is plenty of time between creating a new string and its release. Therefore, this continuous updates highly increase the chances of already providing fully localized and translated features on the day of release.



Revision	Date	Added	Updated	Deleted	Actions
10	Aug 19, 2020 20:40	4	2	1	Current
9	Aug 16, 2020 20:40	0	1	0	Restore
8	Aug 11, 2020 20:38	5	7	20	Restore
7	Aug 05, 2020 20:37	4	2	4	Restore
6	Jul 29, 2020 09:50	4	67	8	Restore
5	Jul 12, 2020 08:26	8	9	1	Restore
4	May 19, 2020 04:29	44	12	0	Restore
3	Apr 15, 2020 05:21	5	0	0	Restore
2	Mar 22, 2020 14:24	19	6	4	Restore
1	Jan 26, 2020 13:19	666	0	0	Restore

Figure 11.3: Translation file revisions: The history of Catroweb's translation source file (catroweb.en.yml) shows the infrequent uploads to the Crowdin platform done by the Catroweb team in the past. Since the automated synchronization was introduced in July, the file is updated more frequently with smaller patches.

7. `https://crowdin.com/project/catrobat/settings#files` visited on 20 Aug. 2020

The introduced synchronization workflow (Listing 11.2) process runs every 24 hours and can be triggered manually if necessary. Whenever the project contains new strings, they are uploaded to Crowdin[8]. If there exist new translations, the workflow automatically creates a pull request containing all updates from Crowdin to Catroweb's repository. An already open pull request updates itself with the latest changes every day.

```yaml
name: Synchronize Crowdin
on:
  workflow_dispatch:
  schedule:
    - cron: '0 0 * * *'
jobs:
  synchronize-with-crowdin:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v2
      - uses: crowdin/github-action@1.0.9
        with:
          upload_translations: true
          download_translations: true
          config: 'crowdin.yml'
        env:
          key: ${{ secrets.key }}
```

Listing 11.2: Crowdin synchronization workflow: Translation resources are synchronized daily with Crowdin. The workflow uses an official action from Crowdin. GitHub secrets hide the credentials from the public.

The steps to implement the mentioned procedure are all handled by an official action developed by Crowdin. In the past, the Crowdin configuration file was never publicly available. Now, these configurations are directly located in the project. However, it is essential not to commit any credentials to the project, like the secret project ID and token. Hence, the configuration file only contains placeholders for those values, and GitHub secrets set up those private credentials during the process.

The workflow is triggered automatically every day at midnight, using the scheduled event of GitHub Actions. The Crowdin project ID and token are stored in GitHub's secure storage for secrets and provided to the workflow

8. https://crowdin.com/project/catrobat visited on 9 Sep. 2020

only as environment variables. The settings enable the upload and download process and define the path to the configuration file. The Crowdin action works out of the box, and further tweakings are not necessary. The workflow file defines the synchronization process with less than 30 lines and is available on GitHub[9].

Nevertheless, to prevent unnecessary runs of scheduled GitHub Actions in forked repositories, a condition is added to the job to ensure that it will only be executed in the official repository (Listing 11.3).

```
synchronize-with-crowdin:
  if: github.repository == 'Catrobat/Catroweb'
  runs-on: ubuntu-latest
```

Listing 11.3: Job condition: A job can have a condition on which the workflow decides if a job should be executed or not.

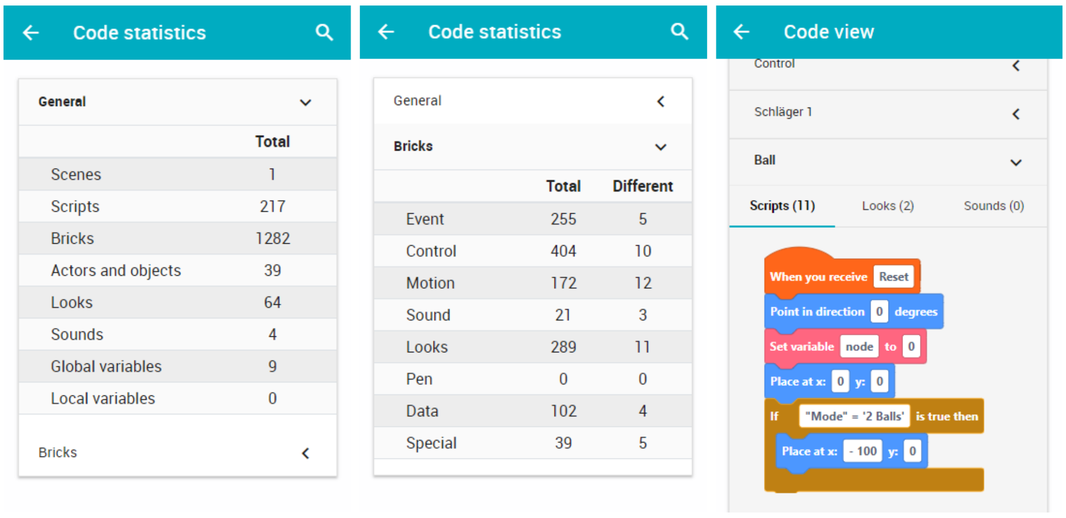
11.2.2 Challenges

Bearing in mind the need for secrets and the git fork workflow used by Catroweb, it is impossible to trigger this workflow on a pull request. Contributions from a forked repository have no access rights to the secrets defined in the central repository. It would also not make any sense to prepare translations for strings before they get integrated into the project's mainline. There is a chance they will be changed during the review process.

Initiating the workflow on a merge event would be possible. However, it was decided to run the task on schedule. This approach provides the benefit of being independent of pull requests. After all, the download process of new translations has nothing to do with new project contributions. Therefore, synchronization is done once a day, which should be often enough for all use cases. Furthermore, it prevents possible issues during multiple parallel executions of the same workflow. Additionally, there is a manual trigger available to initiate the workflow in case of problems.

9. https://github.com/Catrobat/Catroweb/blob/develop/.github/workflows/sync_crowdin.yml visited on 9 Sep. 2020

11.3 Checking for new bricks



Code statistics

General

	Total
Scenes	1
Scripts	217
Bricks	1282
Actors and objects	39
Looks	64
Sounds	4
Global variables	9
Local variables	0

Bricks

Code statistics

General

Bricks

	Total	Different
Event	255	5
Control	404	10
Motion	172	12
Sound	21	3
Looks	289	11
Pen	0	0
Data	102	4
Special	39	5

Code view

Control

Schläger 1

Ball

Scripts (11) Looks (2) Sounds (0)

```
When you receive Reset
Point in direction 0 degrees
Set variable node to 0
Place at x: 0 y: 0
If "Mode" = '2 Balls' is true then
    Place at x: - 100 y: 0
```

Figure 11.4: User project's code statistics: Users of the Share community platform have access to a project's code view and its statistics on a project's detail page. The Catblocks team is responsible for the code view. The web team only remains to maintain an old version of it.

The Share community platform hosts projects from users all around the world. Projects uploaded to the platform are analyzed to fill the detail page of a project. Amongst other things, the provided content includes information about the objects and bricks used to create the project. Figure 11.4 shows the code statistics[10] and code view[11] of a project. To parse the information and calculate the correct numbers, Catroweb must know about all existing bricks. However, the Catroid team, responsible for the Android implementation of Pocket Code, regularly implements new bricks and releases them to the public. Therefore, the web team should already register those bricks in the Share community platform before their public release. Else, the bricks fall into an unknown category, and correctly rendering the bricks is impossible.

10. https://share.catrob.at/app/project/32577/code_statistics visited on 9 Sep. 2020
11. https://share.catrob.at/app/project/32577/code_view visited on 9 Sep. 2020

In the past, the Catroid team had to communicate new bricks to the Catroweb team so that the web-team could set up the new bricks for the Share community platform. However, the communication was missing from time to time, and Catroweb was not supporting all existing bricks. Hence, to check which bricks are exactly missing, it is necessary to manually compare the files in the Catroid and the Catroweb repository.

11.3.1 Automate the process

A scheduled workflow using GitHub Actions compares the bricks registered in Catroweb with all bricks from Catroid once a day. Missing bricks in the Share community platform fail the workflow. The failing step contains detailed information explaining developers the steps needed to implement a missing brick. The automatization of the brick comparison prevents that information gets lost in team overlapping communication. The workflow reminds the web team daily until the problem is solved.

A PHP script[12] downloads all files containing information about the implemented bricks from the Catroid repository. The files are from the develop branch, and therefore, the Catroweb team can be notified even before the Android team releases the bricks to the public. Once the download finishes, the script parses the files for information about the implemented bricks. A similar process parses the file responsible for registering bricks in the Share community platform. In the end, the program compares all gained information from both repositories. If there are differences, the workflow fails and displays information on how to proceed.

The GitHub Actions specific code requires a minimal set up of a few lines to execute the script. The workflow must install PHP and set up the permissions to execute the script. However, from there on, the only remaining thing to do is to run the script. The workflow is provided in Listing 11.4 and is available on GitHub[13].

12. `https://github.com/Catrobat/Catroweb/blob/develop/bin/checkCatroidRepositoryForNewBricks` visited on 9 Sep. 2020
13. `https://github.com/Catrobat/Catroweb/blob/develop/.github/workflows/check_for_new_bricks.yml` visited on 9 Sep. 2020

```yaml
name: Check for new bricks
on:
  schedule:
    - cron:  '0 0 * * *'
  workflow_dispatch:
jobs:
  check_bricks:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v2
      - run: install-php
      - run: check-bricks-command
```

Listing 11.4: The Pocket Code apps are checked for new bricks once a day.

11.3.2 Challenges

Automating the checks for new bricks requires several assumptions about the project structures and concrete implementation details. The URLs to the required files have to be constant and can not change. Also, the parser depends on specific patterns in the files. Since this workflow is scheduled independently from changes in the Catroweb team, and especially the Catroid team, breaking changes could be integrated into one of the repositories. If that is the case, it is necessary to adapt the workflow in retrospect. Until a developer fixes the script, the workflow will not provide any useful information and fail.
However, it would be of no advantage to trigger the workflow on every pull request. In almost all cases, a new contribution has nothing to do with the failure of this check. Typically, this workflow will only fail after the Android team includes a new brick. Therefore, Catroweb contributions should be entirely independent of this check. If this workflow fails, it should not concern a pull request and its developer.

Initiating the process after the merge of a pull request is possible but not necessary. Developers usually merge in irregular intervals. Therefore, it is better to keep a fixed schedule to check for updates. In case additional checks are required, it is possible to dispatch the workflow manually with a click on a button.

11.4 Dependency management

Instead of building every little detail in a project from scratch, developers often rely on frameworks and include certain functionality with third-party libraries. Using a library prevents developers from writing the same code repeatedly and allows them to share functionality between different projects.

In the Catroweb project, package managers handle all the included third-party code. While it would be possible to manually integrate all libraries, it can be challenging, especially managing all the additional interrelated dependencies. A developer can require specific versions of libraries to include the project. The package manager implicitly installs all other needed dependencies needed for the library to work.

However, having too many dependencies on libraries has unique risks and problems. For example, it can be necessary to use a specific version of package A to use package B with an outdated version. Consequently, the usage of a library can prohibit the usage of another. From time to time, libraries even get abandoned and are no longer maintained and updated. A single dependency can block all other dependencies from upgrading, which leaves the project at risk of security breaches. The necessary refactoring can be cost-intensive or require the development team to find an alternative or entirely remove the functionality. Apart from resolved security issues and bugs, newer versions typically have improved performance and functionality. Therefore, keeping dependencies up-to-date is essential to a project.

Nevertheless, the only possibility to entirely avoid such a technical debt is to stop using third-party libraries. Instead of building a project on a framework, or include certain functionality, projects must write every little detail from scratch. In reality, software projects do not have the resources to accomplish this task. In its current state, the Share community platform has way more than 100 dependencies (*Catrobat/Catroweb dependencies* 2020). Instead, teams must aim to find the right balance and evaluate a library before integrating the new dependencies to the project. Google even has its dedicated Fixit days and teams to measure and pay down the debts continuously (Morgenthaler et al., 2012).

To upgrade dependencies, developers have to manually increase the version number of a package and run the update process. Checking all libraries for their newest version and getting hold of the release notes requires valuable time. Far too often, projects are not updating their dependencies until months after their integration. Regularly updating the dependencies usually requires only small patches to keep the project up to date. Instead, there typically exist many updates after an extended period, and it is necessary to apply a large patch, highly increasing the upgrade's complexity. Based on the principles of CI, it is essential to enforce continuous updates. With limited project maintainers, the only possibility of achieving continuous updates is to automate the process and almost leave no work for the developers.

11.4.1 Dependabot

GitHub has an integrated tool, named Dependatbot[14] to automate dependency updates. The tool is free of charge and can be configured with a configuration file directly in a project's repository. Dependabot analyses dependency manifest files in a GitHub repository to filter insecure and outdated requirements. In the event of outdated dependencies, Dependabot opens an individual pull request for each available update (Figure[15] 11.5).

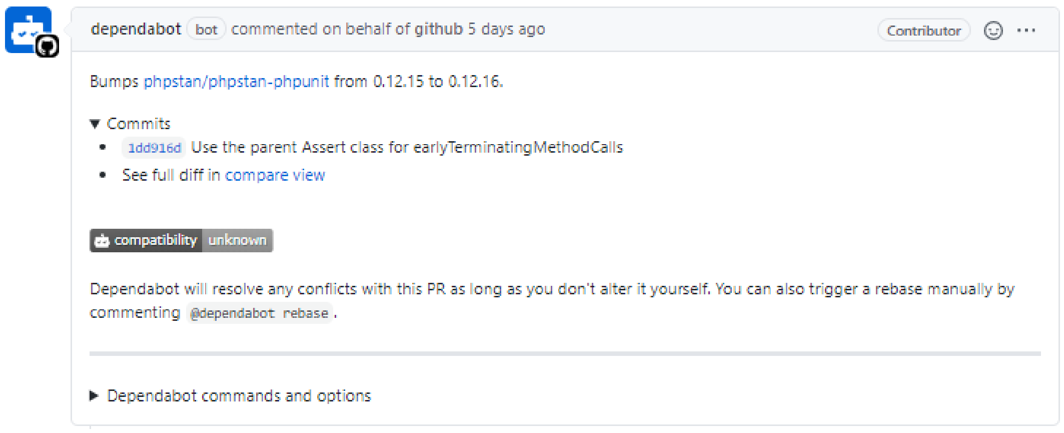
Explicitly upgrading dependencies has multiple benefits. For example, it eases debugging in case of an error, since developers can test each upgrade independently. Besides, Dependabot appends the release notes of the new dependency version to the commit message of the pull request. Therefore, code reviewers can quickly check the changelog and the tests provided by the CI system and confidently merge the updates within seconds. Without automated tests, the overhead of manually testing and checking every update would be impossible to carry out.

Dependabot reacts to specific comments of a pull request and interprets them as commands. In case a dependency must not be updated for any

14. https://dependabot.com/ visited on 9 Sep. 2020
15. https://github.com/Catrobat/Catroweb/pull/807 visited on 11 Aug 2020

reason, a command is enough to close the pull request and prevent Dependatbot from re-opening it again in the future.

New updates in dependencies regularly contain breaking changes. From changed to removed functionality, in any case, it may require adaptions throughout the codebase. Dependabot does not search and fix such issues. Instead, developers have to fix them manually. Luckily, in most cases, this can be done within a few minutes, mainly since the commit message already contains the exact changelog. After fixing the dependency issues, developers push the changes to the pull request. Dependabot will not overwrite any manual modifications.



dependabot bot commented on behalf of github 5 days ago

Contributor

Bumps phpstan/phpstan-phpunit from 0.12.15 to 0.12.16.

▼ Commits

- `1dd916d` Use the parent Assert class for earlyTerminatingMethodCalls
- See full diff in compare view

compatibility unknown

Dependabot will resolve any conflicts with this PR as long as you don't alter it yourself. You can also trigger a rebase manually by commenting `@dependabot rebase`.

▶ Dependabot commands and options

Figure 11.5: Bump dependency pull request: Dependabot creates an individual pull request to update an outdated dependency. The pull request's commit message contains a changelog of the new version. The CI system runs its tests like for every other pull request.

12 Conclusion and future work

CI/CD describes a philosophy and best practices to deliver software products more frequently and reliably. This work's focus has been to develop a CI/CD system and various additional automated workflows for Catrobat's Share community platform Catroweb using GitHub Actions.

As a result of this thesis, the Share community platform's software quality has increased in maintainability and reliability. Apart from enforcing exact code styles, every project's contribution must pass various automated static and dynamic analysis checks before its integration into the official codebase. Hence, combined with a well-tested codebase, the risk of unintentionally breaking existing features is remarkably low. Furthermore, Dependabot resolves security issues in third-party code quickly, while static analysis checks detect deprecated and unsafe methods. Therefore, the project's risk of security breaches is reduced.
On the other hand, the project's efficiency and usability only implicitly benefit from the CI system. The CI system currently contains no checks that explicitly target the design or performance of the software. However, developers save vast amounts of resources due to the automation of all kinds of workflows handled by the implemented system - Resources can now be better distributed throughout the project. Besides, the automated workflows reduce the complexity and implicitly create documentation for every task. Consequently, even non-developers can carry out the tasks, such as starting a new release.

However, Vasilescu et al. (2014) points out that many development teams introduce CI without using it. In case developers and code reviewers ignore its feedback and principles, the implemented system's actual benefits drop to zero. It is crucial that code reviewers manually ensure the quality of new features and their tests. A pull request has to pass all CI tests before being merged. Besides, the system does not cover every aspect of a useful review. Additional feedback is still required do be done manually. Hence, qualitative code reviews are still necessary. Moreover, new contributors must be taught the principles of CI/CD.

Furthermore, the costs of the introduced GitHub Actions are close to zero. The services are free to use for open-source projects and do not require any self-hosted hardware. While there are limitations, they are small enough and do not impact the Catroweb project's development. Besides, the GitHub Actions defined in this thesis do not require many resources to maintain and can be modified directly by the Catroweb team. There is no need for additional teams to handle any infrastructure nor CI/CD system. All GitHub Actions are developed directly in the Catroweb repository using a common syntax, and the whole system is built with only a few hundred lines of code. Besides, Dependabot will automatically upgrade any dependencies in the workflows.

12.1 Future work

The implemented system is fully working and already covers many aspects. Still, there is room for improvements and optimizations[1].

On the one hand, GitHub Actions are pretty new, and GitHub releases new improvements continuously. The systems maintainability and efficiency will highly benefit once further optimizations can be used.

On the other hand, the CI system could be improved by adding additional software tests, such as performance tests. However, Catroweb should first aim for higher code coverage and utilize the testing pyramid principles in the existing test suites.

Moreover, various additional workflows could be automated over time to reduce development costs further. The GitHub marketplace is filled with creative ideas to automate the software development life cycle.

1. All described GitHub Actions remain accessible unaltered in a protected GitHub repository (`https://github.com/dmetzner/AutomatingSDLC`).

Appendix

Acronyms and Abbreviations

APK	Android Package
BDD	Behavior-driven development
CI	Continuous integration
CD	Continuous delivery and deployment
CSS	Cascading Style Sheets
FOSS	Free and open-source software
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
ID	Identification/Identity/Identifier
IDE	Integrated development environment
IP	Internet Protocol
PHP	Hypertext Preprocessor
REST	Representational state transfer
SSH	Secure Shell
SQA	Software quality assurance
TDD	Test-driven development
UI	User interface
URL	Uniform Resource Locator
VCS	Version control system
VPN	Virtual private network
WWW	World Wide Web

Bibliography

Andrés, César, M. Emilia Cambronero, and Manuel Núñez (2011). “Passive Testing of Web Services.” In: *Web Services and Formal Methods*. Ed. by Mario Bravetti and Tevfik Bultan. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 56–70. ISBN: 978-3-642-19589-1 (cit. on p. 15).

Atlassian (Apr. 2020). *What is version Control*. URL: https://www.atlassian.com/git/tutorials/what-is-version-control (cit. on p. 29).

Beck, Kent (2002). *Test Driven Development. By Example*. Addison-Wesley Longman, Amsterdam. ISBN: 0321146530 (cit. on p. 19).

Black-Box Testing (Sept. 2020). URL: https://softwaretestingfundamentals.com/black-box-testing/ (visited on 01/09/2020) (cit. on p. 16).

Brooks, Fred (Mar. 2020). *More People – More Bugs*. URL: https://www.projectmanagement.com/articles/227526/More-People--More-Bugs (visited on 04/03/2020) (cit. on p. 1).

Catrobat/Catroweb contributors (Sept. 2020). URL: https://github.com/Catrobat/Catroweb/graphs/contributors (visited on 04/09/2020) (cit. on pp. 7, 8).

Catrobat/Catroweb dependencies (Aug. 2020). URL: https://github.com/Catrobat/Catroweb/network/dependencies (cit. on p. 111).

Chappell, David (Aug. 2020). *The three aspects of software quality: functional, structural, and process*. URL: http://www.chappellassoc.com/writing/white_papers/The_Three_Aspects_of_Software_Quality_v1.0-Chappell.pdf (visited on 08/08/2020) (cit. on p. 10).

Cohn, Mike (2009). *Succeeding with Agile: Software Development Using Scrum*. 1st. Addison-Wesley Professional. ISBN: 0321579364 (cit. on p. 16).

Cooper, Ryan (July 2020). *Why You Won't Fix It Later*. URL: http://on-agile.blogspot.com/2007/04/why-you-wont-fix-it-later.html (cit. on p. 12).

Dan Belcher, mabl Co-Founder (Apr. 2020). *Automate your workflow from idea to production - What our community is saying*. URL: https://github.com/features/actions (cit. on p. 51).

Developer website of Catrobat (Apr. 2020). URL: https://developer.catrobat.org/ (visited on 09/04/2020) (cit. on p. 3).

Dietmar Winkler Stefan Biffl, Johannes Bergsmann (2018). *Software Quality: Methods and Tools for Better Software and Systems: 10th International Conference, SWQD 2018, Vienna, Austria, January 16–19, 2018, Proceedings*. ebook. Springer International Publishing (cit. on p. 10).

Driessen, Vincent (June 2020). *A successful Git branching model*. URL: https://tex.stackexchange.com/questions/109940/citing-author-or-year-only-without-natbib (cit. on p. 33).

Foundation, World Wide Web (Apr. 2020). *History of the Web*. URL: https://webfoundation.org/about/vision/history-of-the-web/ (visited on 06/05/2020) (cit. on p. 22).

Fowler, Martin (Apr. 2020a). *Continuous Integration*. URL: https://martinfowler.com/articles/continuousIntegration.html (visited on 05/04/2020) (cit. on p. 38).

Fowler, Martin (Aug. 2020b). *TestCoverage*. URL: https://martinfowler.com/bliki/TestCoverage.html (cit. on p. 89).

Fowler, Martin (Aug. 2020c). *The Test Pyramid*. URL: https://martinfowler.com/articles/practical-test-pyramid.html (cit. on p. 16).

GitHub Status Incident History (Sept. 2020). URL: https://www.githubstatus.com/history (visited on 08/09/2020) (cit. on p. 45).

Group, Miniwatts Marketing (Apr. 2020). *Internet growth statistics*. URL: https://www.internetworldstats.com/emarketing.htm (visited on 06/05/2020) (cit. on p. 22).

"Guide for Software Verification and Validation Plans" (1994). In: *IEEE Std 1059-1993*, pp. 1–87 (cit. on p. 14).

Harley, Nick (Aug. 2020). *11 of the most costly software errors in history*. URL: https://raygun.com/blog/costly-software-errors-history/ (visited on 10/08/2020) (cit. on p. 13).

IBM (June 2020). *Software Development*. URL: https://researcher.watson.ibm.com/researcher/view_group.php?id=5227 (cit. on p. 21).

Introduction to SpecBDD and StoryBDD (Aug. 2020). URL: http://www.phpspec.net/en/stable/manual/introduction.html (cit. on p. 79).

Isha, Sunita Sangwan (2014). "Software Testing Techniques and Strategies." In: *Isha Int. Journal of Engineering Research and Applications* (cit. on p. 13).

ISO 8402-1986 standard (Aug. 2020). URL: `https://www.iso.org/standard/15570.html` (visited on 07/08/2020) (cit. on p. 9).

Itti Hooda, Rajender Singh Chhillar (2015). "Software Test Process, Testing Types and Techniques." In: *IEEE Std 1059-1993* International Journal of Computer Applications (0975 – 8887) (cit. on p. 18).

Jovanovic, N, C Kruegel, and E Kirda (2006). "Pixy: a static analysis tool for detecting Web application vulnerabilities." eng. In: IEEE, 6 pp.–263. ISBN: 0769525741 (cit. on p. 59).

Kudryashov, Konstantin (Aug. 2020). *Does Behat generate Code Coverage?* URL: `https://github.com/Behat/Behat/issues/92#issuecomment-3719726` (cit. on p. 89).

Levels of Testing in Software Testing (Aug. 2020). URL: `https://www.guru99.com/levels-of-testing.html` (cit. on p. 17).

Micco, John (Aug. 2020). *Flaky Tests at Google and How We Mitigate Them*. URL: `https://testing.googleblog.com/2016/05/flaky-tests-at-google-and-how-we.html` (cit. on p. 86).

Morgenthaler, J. D. et al. (2012). "Searching for build debt: Experiences managing technical debt at Google." eng. In: *2012 Third International Workshop on Managing Technical Debt (MTD)*, pp. 1–6 (cit. on p. 111).

Nair, Jithin (July 2020). *TDD vs BDD – What's the Difference Between TDD and BDD?* URL: `https://blog.testlodge.com/tdd-vs-bdd/#:~:text=In%20TDD%20(Test%20Driven%20Development,from%20the%20end%20users%20perspective.` (cit. on p. 20).

Oracle (Sept. 2020). *Database*. URL: `https://www.oracle.com/database/what-is-database.html` (visited on 03/09/2020) (cit. on p. 25).

PHP usage statistics for websites (June 2020). URL: `https://w3techs.com/technologies/details/pl-php` (visited on 04/06/2020) (cit. on p. 26).

Radcliffe, Matthew (Aug. 2020). *Unit Test Your Code With PHPUnit*. URL: `https://softpixel.com/~mradcliffe/files/columbusphp-unit-testing.pdf` (cit. on p. 17).

RFC2616 (Sept. 2020). URL: `https://tools.ietf.org/html/rfc2616` (visited on 04/09/2020) (cit. on p. 24).

Scott Chacon, Ben Straub (2020). *Pro Git book* (cit. on p. 30).

Slany, Wolfgang (June 2020). *Founder of Catrobat - Wolfgang Slany*. URL: https://www.linkedin.com/in/wolfgangslany/?originalSubdomain=at (cit. on p. 2).

Source code analysis of Catroweb (Aug. 2020). URL: https://github.com/Catrobat/Catroweb (cit. on p. 7).

Stackify (Apr. 2020). *Top Continuous Integration Tools*. URL: https://stackify.com/top-continuous-integration-tools/ (cit. on p. 42).

Swati Seela, Ryan Yackel (Aug. 2020). *64 Essential Testing Metrics for Measuring Quality Assurance Success*. URL: https://www.tricentis.com/blog/64-essential-testing-metrics-for-measuring-quality-assurance-success/ (cit. on p. 89).

Symfony (Sept. 2020). *What is Symfony*. URL: https://symfony.com/what-is-symfony (visited on 03/09/2020) (cit. on p. 28).

SymfonyCasts (Aug. 2020). *phpspec? PHPUnit? BDD? TDD? Buzzwords?* URL: https://symfonycasts.com/screencast/phpspec/unit-integration-functional (cit. on p. 78).

Tantawy, Alshaimaa (Sept. 2009). "Software Quality Assurance Models: A Comparative Study." PhD thesis (cit. on p. 11).

Turner, Ash (Apr. 2020). *How many smartphones are in the world?* URL: https://www.bankmycell.com/blog/how-many-phones-are-in-the-world (visited on 09/04/2020) (cit. on p. 3).

Vasilescu, Bogdan et al. (Dec. 2014). "Continuous Integration in a Social-Coding World: Empirical Evidence from GitHub." In: DOI: 10.1109/ICSME.2014.62 (cit. on p. 115).

Voita, Elena, Rico Sennrich, and Ivan Titov (May 2019). *When a Good Translation is Wrong in Context: Context-Aware Machine Translation Improves on Deixis, Ellipsis, and Lexical Cohesion* (cit. on p. 104).

W3C (Aug. 2020). *The history of the Web*. URL: https://www.w3.org/wiki/The_history_of_the_Web (cit. on p. 25).

White-Box Testing (Sept. 2020). URL: https://softwaretestingfundamentals.com/white-box-testing/ (visited on 01/09/2020) (cit. on p. 15).

Wright, Doug (Aug. 2020). *behat-code-coverage*. URL: https://github.com/dvdoug/behat-code-coverage/blob/master/README.md (cit. on p. 89).