POLITECHNIKA ŁÓDZKA
Wydział Elektrotechniki, Elektroniki,
Informatyki i Automatyki

Master thesis

Modern methods of software development based on recommendation platform

Łukasz Chruściel

Nr albumu:

201517

Promotor pracy:

dr hab. inż. Mariusz Zubert, prof. PŁ

Promotor pomocniczy:

mgr inż. Jan Napieralski

Łódź, 2018

Abstract

This master thesis aims to compare modern methods of software development and related tools. The work also aims to present the process of software development based on the iterative-incremental programming techniques and their influence on the final appearance of the application code. The work will describe methodologies such as clean code, *Object Calisthenics*, static code analysis, defensive programming, *SOLID*, software testing, *Behaviour Driven Development*, and continuous integration.

As part of the work, an application was developed that aggregates information about craft beers from its users, it allows expressing their opinions about them and receiving recommendations based on them. The application has been written in *PHP* and makes usage of many libraries released on open source licenses. The application has a *RESTful API* that allows interaction for external users.

The thesis contains six chapters. The first one is an introduction to the software development quality assurance and describes aims of the thesis. The second one presents theoretical basis of the discussed practices and their assumptions. This chapter focuses on explaining their validity and rules, showing the expected effects and difficulties. This section discusses programming aspects such as testing, *SOLID* or clean code. The third chapter introduces shortly to the subject of recommendation algorithms and presents the most popular approaches to this topic. The next section focuses on describing the project requirements, the tools used, and the software development process itself, taking into account the previously described methods. The next chapter presents the effects of usage of described methods, code quality measurements and discusses their results. The last section summarises the consequences of using the described practices.

Keywords: Application testing, clean code, software engineering, application architecture, quality software development

Streszczenie

Celem niniejszej pracy magisterskiej jest porównanie najnowszych praktyk wytwarzania oprogramowania oraz powiązanych z nimi narzędzi. Praca ma na celu również przedstawienie procesu wytwarzania oprogramowania w oparciu o techniki programowania iteracyjno-przyrostowego oraz ich wpływ na ostateczny wygląd kodu aplikacji. W pracy zostaną opisane metodyki takie jak czysty kod, object calisthenics, statyczna analiza kodu, programowanie defensywne, SOLID, testowanie aplikacji, wytwarzanie oprogramowania sterowane zachowaniem czy ciągła integracja oprogramowania.

W ramach pracy została opracowana aplikacja agregująca informacje o piwach rzemieślniczych, pozwalająca na wyrażanie swoich opinii o nich oraz otrzymywania rekomendacji na bazie tych opinii. Do jej napisania została zostały wykorzystane narzędzia takiej jak język *PHP* oraz biblioteki udostępnione na licencjach wolnego oprogramowania. Aplikacja wykorzystuje interface *RESTful API*, który pozwala na interakcję dla użytkowników zewnętrznych.

Praca składa się z 6 rozdziałów. Pierwszy wprowadza do problematyki jakościowego wytwarzania oprogramowania oraz opisuje cele tej pracy. Drugi rozdział przedstawia teoretyczną podstawę omawianych praktyk oraz ich założenia. Rozdział ten skupia się na opisaniu ich zasadności, opisuje ich reguły, przedstawia spodziewane efekty oraz trudności. W tym dziale zostają omówione takie aspekty programowania jak testowanie, *SOLID* czy czystość kodu. Trzeci rozdział opisuje tematykę algorytmów rekomendacji, oraz omawia najpopularniejsze rozwiązania. Następny rozdział skupia się na opisaniu wymagań projektu, użytych narzędzi oraz samego procesu wytwarzania oprogramowania z uwzględnieniem wcześniej opisanych metod. Rozdział piąty przedstawia efekty aplikowania opisywanych metod, pomiary jakości kodu oraz omawia ich rezultaty. Ostatni rozdział podsumowuje konsekwencje wykorzystania opisanych w pracy technik.

Słowa kluczowe: Testowanie oprogramowania, czysty kod, inżynieria oprogramowania, architektura aplikacji, jakościowe wytwarzanie oprogramowania

Contents

Glossary of Terms

- ABV - Alcohol By Volume
- ADR - Action Domain Responder
- API - Application Programming Interface
- AST - Abstract Syntax Tree
- BDD - Behaviour Driven Development
- CC - Cyclomatic complexity
- CCMSI - Covered Code Mutation Score Indicator
- CI - Continuous Integration
- CQRS - Command Query Responsible Separation
- CQS - Command Query Separation
- CRUD - Create, Retrieve, Update, Delete
- DI - Dependency Injection
- DoD - Definition of "Done"
- FIRST - Acronym from five rules: Fast, Isolated/Independent, Repeatable, Self-validating, Timely
- HTTP - HyperText Transfer Protocol
- IDE - Integrated Developer Environment
- JSON - JavaScript Object Notation
- MCC - Mutation Code Coverage
- MSI - Mutation Score Indicator
- MVC - Model View Controller
- NPM - Node Package Manager
- ORM - Object Relational Mapping
- OSS - Open Source Software
- PHP - *PHP*: Hypertext Preprocessor - formerly Personal Home Page.
- PSR - *PHP* Standards Recommendations
- REST - REpresentional State Transfer
- SOAP - Simple Access Object Protocol
- SOLID - Acronym from five rules:
  - Single responsibility principle
  - Open close principle
  - Liskov substitution principle
  - Interface segregation principle
  - Dependency inversion principle
- SpecBDD - Specyfication Behaviour Driven Development
- StoryBDD - Story Behaviour Driven Development
- TDD - Test Driven Development
- UUID - Universally Unique Identifier
- VCS - Version Control System
- VO - Value Object -
- XML - Extensible Markup Language
- YAML - YAML Ain't Markup Language

1. Introduction

Nowadays, software became much more complicated. Thousands of working hours are spent on a most of the applications of websites that people are using every day. From preparation of product backlog, through the development process to the testing and quality assurance which will accept or not proposed changes. The chain is long and complicated. What is more, software does not exist in a vacuum. It has to respond to clients requirements modifications or third-party libraries evolution. In software development, there is only one constant: change. For many years developers try to tackle the complexity of their domain in a way, that will allow for future adjustments. Today problem is still not trivial. Many of the current best practices were described many years ago. The source of ideas behind them could be even dated to 80'. After all these years one can look back to the roots and learn from the pioneers of development instead of reinventing the wheel.

Another fascinating thing is that a lot of the human knowledge is available in just a few seconds for most of the people. Computers and smartphones allow to be up-to-date with newest trends or find way back home when travelling. It is easy to be overloaded with all these data. It is not trivial to find valuable information when thousands of it comes every minute. Whats moreover, advisors reacted to these changes and they almost attacking users with new information. Limited resources of time and money, makes it even harder to choose valuable information among countless possibilities. Which movie to watch? Which dish to try in the restaurant? So many questions require only time to find out the proper answer, but it is always not enough of it. Usually, it is much more convenient just to ask somebody for the recommendation. However what if an application could sense customer needs and recommend something?

1.1. Aim of the thesis

This paper will describe state of the art practices in software development, their influence on code development, overhead and profits. It describes an influence of clean code, defensive programming and SOLID paradigm in terms of code complexity and structure. It also takes into account software testing together with Test-Driven Development and Behaviour Driven Development practices. Another part of thesis will summarized software architecture patterns like Onion Architecture or CQRS pattern. This thesis will also present a practical example of usage of these methods as well as tools that can be used to make implementation smoother. It will describe above methods and paradigms on an example of the application with their pros and cons, that one may consider implementing in own project.

Example applications will allow users to create and manage their accounts, add new beers to the database and rate them. Rates will allow to determine recommendation for the users, so they can spend less time to decide what they would like to taste next and the companies should have more happy consumers. Usage of described method provide a base for an application that it prepared for further changes, a well as incremental and lean development. The application will use a graph database to provide customer recommendation based on collaborative filtering pattern.

As a result this thesis should be a comprehensive guide over modern methods of software development that should help other developers to make their minds if they should use some of practices described here or not.

2. Background - Best practices in software engineering

Proper application architecture may have a crucial impact on the whole project. Wrong decisions on the early stage can have tremendous consequences in a future. However, not all choices are equally important. Using YAML (YAML Ain't Markup Language[1]) notation instead of XML (Extensible Markup Language[2]) for service definition is much less important than for example wrong tool or architectural pattern. Overengineering can cause as many issues as using too simple solutions for complicated business problems. Also, every day new requirement can appear, or our customer could understand another fundamental concept of his business which can result in product scope changes. Developers should not force clients to adjust their business to software limitations. They should embrace the change and be servants of their needs.

In this chapter, cutting-edge best practices will be described and the reasoning behind them. All of them are crucial for further project maintenance and are also essential to reduce technical debt. Technical debt is a metaphor introduced by Ward Cunningham to explain need of refactoring to nontechnical product owners[3]. Technical debt can be reasoned by time pressure from the managing side or carelessness, lack of education, poor processes, nonsystematic verification of quality and basic incompetence from the developers side[4]. Of course, it is not possible to foresee all of the usages of our application or needs. Nevertheless, some of the decisions could be postponed in order to make a call when there will be enough of data to decide what suits best.

2.1. Application architecture

The basic concepts are the set of rules that need to be followed despite the language or problem resolved. Described concepts are language agnostic therefore they can be applied to almost every project. The primary aim of this part is to reduce complications of code concepts, increase the code cohesion and make it readable for everybody. These ideas are a general guideline that should be followed but could also be adjusted depending on the current needs. If the rule is too strict for given problem, it is ok just to skip some of the rules. Nevertheless, it should be an exception rather than a practice.

2.1.1. Object-Oriented Programming

Object-Oriented Programming (*OOP*) is probably the most widely used approach. There are a lot of libraries, languages and frameworks that not only follow up this programming paradigm but also force others to use it as well.

With the increasing popularity of Haskel or Erlang and other similar languages, a functional oriented programming paradigm gathers more and more attention. It could be an exciting way of developing in this programming paradigm. On the other hand, *OOP* provides more "ready to use" libraries, which reduce the effort needed, to deliver new functionalities and allows not to "reinvent the wheel".

Programs written in the *OOP* paradigm allows for polymorphism of its classes, logic encapsulation, class inheritance or composition. Polymorphism allows to be agnostic from the concrete implementation and based on abstraction. It is important to stress out that inheritance should not be the primary reuse mechanism. It is a much cleaner approach to use composition instead of inheritance. What is more, such implementation is more straightforward to test and validate[5].

2.1.2. Clean code

Working with the code could be challenging, and there is no point in making it harder. The problem becomes even more significant when the project is maintained by many software engineers over many years. Conflicts start from tabs vs spaces, through the brackets position up to proper class and properties naming. Nevertheless, the same problem can occur, even in single developer projects when the work is spread over many weeks, months or years. Lack of documentation and different code styles could be a nightmare for maintaining.

Many developing teams faced the problem of reduced development velocity during project time. They were able to deliver a lot of new features every month at the early stage of the project. However, after a few releases of the product, adding new features became more challenging and more efforts were needed to fix existing bugs. Then the velocity starts to peak down.

When the code is unclear, even small changes can create new problems. Developing team has to put more and more time to fix existing bugs, and because of that, they do not have time to deliver new values to the business.

Robert C. Martin has described many solutions for the problem described above in the: Clean Code a Hand Book of Agile Software Craftsmanship[6]. The book focuses on the most common issues that could be encountered in code. As described in the book, one of the most challenging parts of code development is naming of the methods, classes or variables. Too many abbreviations or jokes in the code could be misleading and annoying. Names should be short, descriptive and precise. Another problem that is tackled in the book is irrelevant comments as with the proper naming, most of the comments are unnecessary and could only generate more mess. Another important rule is to keep classes small and focused on one goal.

As it was mention at the beginning of this chapter, the problem starts with the simple question: should developers should use tabulators or spaces. This decision does not change anything in the project from outside, so there is no need in fighting about it for many weeks. On the other hand, a messy code would cause significant disruptions in a matter of working with it. There are many ways to make the code a nightmare to read. Too many blank lines, not using them at all, or writing too much code in the single line could be equally dangerous. It seems unimportant, but all those mentioned problems can make a big difference. Especially, if the codebase is not coherent, and different styles are placed all over it.

2.1.3.Object calisthenics

Object calisthenics is a set of pretty harsh rules. It defines set of language agnostic best practices for object-oriented code. Following all of them, all the time is hard. It may be not impossible, but slightly not pragmatic. For example, a rule which suggests that class should have no more than two methods. However, not every object is the service with some logic. Many classes are just simple data structures. The rules for this kind of objects are different. Trying to follow object calisthenics principle can be vital for code quality - a few methods and parameters in services makes it much easier to test and understand.

Another part of object calisthenics defines, that the amount of "if" statements should be minimised and early returns ought to be used instead of "else" or "else if" statements. Next requirement defines that the maximal functional indention(conditions, loops and so on) can be equal to one. These rules not only simplify the classes and make them easier to read but also aimed to reduce to *Cyclomatic Complexity*. *Cyclomatic complexity* is a measurement strategy calculated by computing the number of linearly independent paths[7]. In another words *CC* describes the total amount of paths that are available inside the method or class. It means, that the function with one "if" statement has a *CC* equal to 2. When the *CC* of the method reaches 8, it says, that it should be tested under eight different scenarios to provide full coverage (and still, this does not ensure that code is appropriately tested).

The object calisthenics rules defined by Jeff Bay[8]:

- One level of indentation per method
- Don’t use the ELSE keyword
- Wrap all primitives
- First class collections
- One dot per line
- Don’t abbreviate - naming is one of the hardest things in software development. Abbreviation creates unnecessary mess and confusion. Explicit naming, e.g. "entityManager" instead of "em" is much easier to understand.
- Keep all entities small
- No classes with more than two instance variables
- No getters/setters/properties

These rules are pretty harsh and are not easy to follow. Every principle is just a guideline of how the code should loo. It is not easy to apply them, mainly for the people that are not used to it, although they may noticeably increase the code quality. It is worth trying to support them as much as it is possible even if not all of the rules are followed.

2.1.4.Five rules of SOLID principles

The last part of basic concepts focuses on the *SOLID* Principles. The principles have been described by Robert Martin in the "Agile software development: principles, patterns, and practices"[9]. *SOLID* is the acronym of the five rules:

- Single responsibility principle
- Open close principle
- Liskov substitution principle
- Interface segregation principle
- Dependency inversion principle

The main aim of the principles is to embrace the change of code and prepare it for further adjustments in the scope of requirements. It will also make the code more readable and understandable. These are a general set of rules that could be treated as a guideline of how to format the core logic of the application and how it should communicate with each other.

The first rule, *Single responsibility principle* says that: "A class should have one, and only one, reason to change."[9,10]. It means that specialised service should have only one job and therefore just one reason to change. Class with too many responsibilities (more than one) has not correctly decoupled logic what results in lower cohesion and creates unneeded coupling. When one class is responsible for generating the .pdf file and sending emails, intuition can say that something is wrong with this combination. It is also highly probable that both concepts could be reusable in a different part of the system but without any need of knowledge about the second one. When the classes are small and have the only single job to do it is much easier to reuse and maintain them. Poorly decoupled classes are harder to test because more paths are required to cover and probably whole setup and assertion is more complicated.

Open close principle says that: "You should be able to extend a classes behavior, without modifying it."[9,10]. This rule suggests that the class logic should be easily customizable without the need of changing the internal implementation. It means, that class dependencies should be preferably injected in the constructor rather than hardcoded

in the function. Another typical way of applying this principle is to use a composition or decoration pattern.

Liskov substitution principle which is saying that: "Derived classes must be substitutable for their base classes."[9,10]. It describes that when the class depending on the parent class or an interface, it should behave as expected. In this context behaving as expected means that when the implemented method should print the given file, it should at least do this one particular thing. Some of the typical violation of this principle can be spotted when the children class throws *UnnsupportedMethodException* or something similar depending on the language, instead of providing some correct method implementation. The classes should be projected this in such way, that an exception should not be present in the code.

The fourth principle says that: "Make fine grained interfaces that are client specific."[9,10]. Having the interface that contains too many defined method is also considered as a bad practice. Usually, a client uses only a subset of all methods on the interface, and there is no point in giving them access to others.

The last one is the *Dependency inversion principle* which suggests to "Depend on abstractions, not on concretions."[9,10]. This rule aims to make it easier to substitute the constructor parameters or method arguments. It is especially true for strict type hinted languages like *Java* or *C#* and recently even for *PHP*. Then defining arguments for these type languages, one has to specify expected type of class. It can be either some simple type (like *string* or *integer*), concrete class implementation (like *User* or *CustomerEmailService*) or an abstraction of it (like *UserInterface* or *CustomerEmailServiceInterface*). Type-hinting (former name for type declaration in *PHP* - *p*ractice of declaring expected type for method parameters) to the specific implementation will force to use only this particular class. This coupling can be dangerous for the project and reduce flexibility. Depending on the high-level abstraction gives better possibilities in the long term to make changes. Interface only ensure, that given class had some set of behaviour but did force any particular implementation. Essential things are entry data and expected return type. This way, it is much easier to provide an entirely new implementation for given problem. Of course to take a more prominent advantage of this, small, highly

granulated interfaces are required. A class that calculate some value based on an array of objects (for example a price from the lists of order items) can know how to fetch required data from the database. When one day the requirement would change, and data will come from another source (e.g. external API) the whole class will not be usable. Even if the prime concern of it states same - it is still responsible for price calculation. Using abstraction to provide order items will make it much easier to adjust the code in the future. Otherwise, either one has to duplicate the class or rewrite it.

2.1.5.Defensive programming

During driving course, each student is learnt defensive driving. The main idea behind it is that all drivers are just humans and can make a mistake. When developing code, a similar approach can be applied as well. Several best practices allow to be more secure and be prepared for unexpected results of users interactions.

Despite the used language it is crucial to encapsulate all states of classes in code. It is wise to define all classes as final and declare all its properties as private. Reduced amount of public methods and properties to the possible minimum makes it easier to test and not leak application state. Constructors should be the only injection point, and setters should be avoided. Setter will allow internal state mutation of application which can break the application. Only one injection point make it easier to validate the state and throw an exception if provided data is invalid. Value Objects (*VO*) helps to ensure given data is valid. Value Object is a small simple object, like money or a date range, whose equality isn't based on identity[11]. They are immutable and allows to only to fetch its internal, simple value[12], [13].

Form data validation can be taken as an example. Modern frameworks have many libraries that make development much faster. Typical examples of them that are available in almost every language are *ORM* (Object Relational Mapping) mappers, forms and form validation. *ORM* is the technique of bridging the gap between the object model and the relational model, often referred to as O-R mapping or simply ORM. The term comes from the idea of mapping the concepts from one model onto another, with the goal of introducing a mediator to manage the automatic transformation of one to the other[14].

ORMs allow creating database mapping for a class (entity) reasonably smooth without worrying about database implementation or required queries. Often, same entity is passed to the customer view and bind to the form. When the end user changes data to something invalid, validator component should block code execution. Nevertheless, if by mistake, the entity will be persisted to the database, consequences may be severe. Many before-mentioned bugs can be significantly reduced, as a result of *VO*, finalised classes and properties usage. Following practices will allow for state encapsulation and validate data correctness on the level of each class. Interfaces usually provide best place and method of code overriding. Abstract classes or classes can leak state, make classes harder to test and maintain.

Another defensive practice that is worthy to implement in a project is testing everything. Several types of tests provide enough possibilities to choose a more suitable solution for given problem. The project should not have only one type of tests. There is space for each of it. Only when testing, we can ensure that we have achieved what we expected. In the longer term, only automated test suites can ensure, that nothing has been broken[12], [13].

Common problems are most likely appropriately resolved in publicly available libraries. If the problem occurs many times, many developers have an interest in solving similar problems which is a base for robust and battle-tested solutions. These solutions will allow reducing the amount of code to maintain as well as not reinvent the wheel. It is worth mentioning that most of the common problems are usually solved by Open Source Software (*OSS*). *OSS* usually allows to use it free of charge and check the code quality before making something project requirement[12], [13].

Same as with every part of industry standard the rules above are just a guideline. Small projects of short living ones will can well-made without them. Though as projects become bigger and bigger it more relevant to apply them. Developers should not trust in end-user behaviour or input data. They should not trust on other developers level of competence either. Sizable developing teams usually are built from the people with a different technical background, so it is not possible to expect everybody to have an equal skill set. According to "Software Lifetime and its Evolution Process over Generations"1[15]

average software lifetime is ten years. During this period software gains new feature, fixes its bugs or is just kept up-to-date with software environment changes. After all these years many people would be involved in a project. Stern rules are the first line of protection from the common mistakes.

2.1.6. Industry standard recommendation

Among many standards forced by developers, bare minimum should be commonly approved industry standard. As a sake of example in *PHP* community collaborates within *PHP* Framework Interop Group and tries to define standards which will improve code reusable and interchange. Released by them *PHP* Standard Recommendations (*PSR*) are a minimum requirement for many closed and open sourced projects[16].

2.1.7. Onion Architecture

The Onion Architecture pattern[17] describes the code structure build on the top of layers. Each layer can use its services and interact with everything below it, although it cannot call any layer above. The main benefit of its usage is control over the coupling. Only highest layers can be coupled with frameworks or external services. The core logic is independent of any other library and can be quickly applicable in different context. The pattern itself embraces the *Dependency inversion principle*. As presented on *Fig. 1* all coupling could be done towards the application core and if any external communication has to be done throughout interfaces, which gives the possibility to change an implementation later. Applying this approach allows to postpone some critical decision and choose the tools in the later phase of the project. In the meantime, the domain knowledge among the developers would be much higher which reduce the cost of maintenance and probably will result in much better decisions.

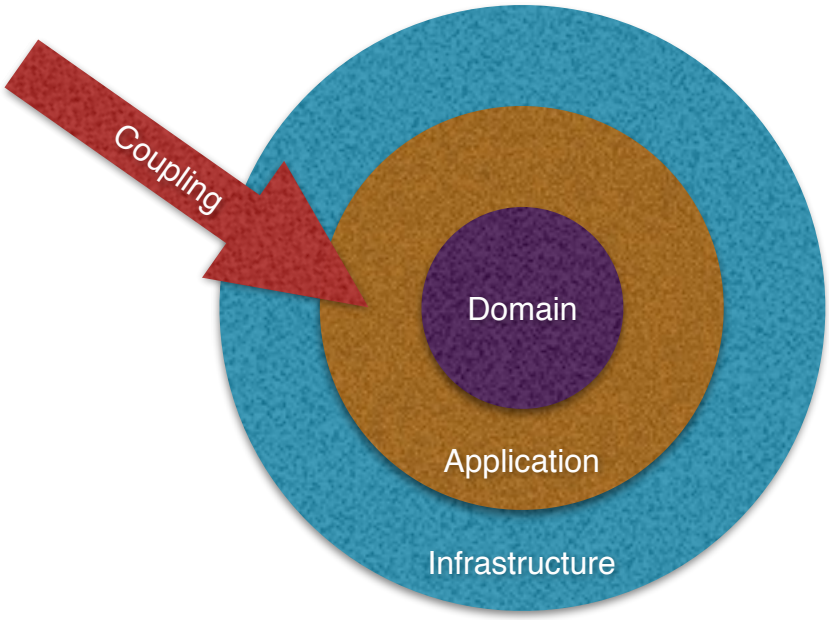


Figure 1. An onion architecture structure (source own)

a) Domain

The most constant and vital parts of an application are placed in the Domain. It is the heart of the software. Every critical business logic should be encapsulated here. Because each layer is allowed to depend on themselves and layers inside it, the domain layer can be only coupled to itself. This segment does not know anything about its storage or the way how the end user will interact with it. It is focused on business values only. It should resolve crucial problems that software aims to tackle and give an application a competitive advantage. Because the layer is coupled only to itself the logic written here can be applied from different perspectives what allows for more comfortable change of outer layers if needed.

b) Application

Application focus on logic orchestration, provide some services and make it much easier to interact with an app. Application together with a domain creates a valuable product which can deliver value to the business. Most of the domain class calls happen here, and this layer is responsible for covering all actions that represent given intention. Despite the way, how the user will use an app, all the logic is done here. It is not the most

crucial one (as this one should be placed in a domain), but it makes an application easily reusable.

c) Infrastructure

An infrastructure layer is the most uncertain one. This layer is responsible for the framework integration, third-party libraries usage or external *API*'s handling. In this segment, the most significant amount of code can change over the years. It is notable risky to couple a business logic into the libraries that are not controlled by the team. If the core logic will be coupled to some external resource, then the whole app now depends on the business behind that resource. Onion application pushes developers to put coupling logic inside an infrastructure, to reduce the cost of changing it later. The fundamental business logic is placed in the inner layers, so adjusting to different external needs is rather cheap. This separation allows being responsible for marketing needs. If the user interacts with an app throughout the website, it is pretty straightforward. If *API* endpoints are needed, then only new adapters are required. If the app exceeds an expected interest, just new adapters are introduced which allows for higher traffic data manipulation.

d) Tests

The most external layer could be a test layer which tests the full integration itself. On this level, developers try to mock real user behaviour (which usually involves automated clicking on a headless browser or clicking on application UI) and assert the current state on the returned views. Such tests allow checking the whole intention flow throughout the system. They will ensure that not only code works as expected but third-party systems are correctly configured.

2.1.8. Command-Query Separation and Command Query Responsibility Segregation

Command-query separation (*CQS*) was firstly described by Bertrand Meyer in the Object-oriented software construction[18]. It states that there could be only two kinds of operations done inside the system: *Commands* which can change the internal state of an application, although they will never return any value and the *Queries* that can fetch the

data without changing the state. A typical example of the function that broke this pattern is pop action on a stack. This operation will remove the first element from the stack and return it to the user. Usage of this function can be handy but have to be done with caution. Separation of the methods that read and write from the model allows to read the data with the confidence, and one will never break the internal state of a system just by reading it. This separation also embraces the Single Responsibility Principle. An important factor of *CQS* is that it operates on the single model, so there is always *Single Source of True* for the data.[19]

If the services that are responsible for write and read actions will be split, then the Command Query Responsibility Segregation[20] (*CQRS*) paradigm is used. The main benefit of this pattern is that having services with different responsibilities. Despite the fact that they can operate on the same data, it is much easier to multiply they instances independently. Commonly, together with service separation, the data model is also separated. It opens a door for different data storages for each of model, and the models could be optimised for their needs. It allows for faster data retrieval, and it will be easier to build a model view on a top of it.

When using this distinction developer can decide that there is no need in waiting for command execution and process the logic further, when the command is put queued and executed independently from the action that triggered it. It can be done synchronously or asynchronously, which allows for much easier scaling up.

2.2. Testing

Engineers have to measure the effectiveness of work and check if their hypothesis is correct. One of the possible ways of doing it is to test what was developed manually to receive instant feedback about current part of the system. However with time, when more and more features are added to the system, it becomes harder and harder to check, if all system works as expected. The software is complicated and has tons of dependencies. Each new change or small refactor could have an impact in some other part of the system even if they were not expected. Services could be injected in several places of application. It can happen that refactoring one service and checking if it works in one context will not show

the problem in another one. As a result, one has to check the whole system each time some change is made, but it is costly.

2.2.1. Types of tests

Another common focus for engineers is automation of repetitive work. If the testing requires the same amount of work each time, it is a right place for automatization. There are several types of tests.

a) Unit tests

Unit tests work on the lowest level and test atomic part of class - methods. These tests are done in isolation from other parts of a system and even from the class itself. External class dependents are mocked to simulate its behaviour. It is expected that class s will return the same result for given set of entry data each time test is executed.

b) Integration tests

Integration tests work with the configured set of classes, which should communicate with each other. When doing integration tests not all of the related services are mocked. It is crucial to check behaviour and communication between classes at the same time. Otherwise, creating wrong mock can hide some bugs, what can be costly if it occurs in a later phase of project lifetime.

c) End-to-end tests

On the top of it, there are smoke tests, the end to end tests or UI tests. They involve some way of interaction with the whole app, as they are mainly mocking end-user interaction. These tests are used to check the application from entrance though out database connection up to application response. Depending on use case they are executed on working application or some headless browser to increase execution time. Usually, all of them are part of acceptance tests, tests that are agreed that if passed are proof that business requirement has been fulfilled.

2.2.2. Test pyramid and tests economy

It is quite clear that the last type of tests provides the best value. They ensure that some defined data can be repetitively passed through the whole app and returned response stays same. The disadvantage of these tests is the cost of execution. High-level tests are costly in CPU power and time, as their need to load and run the whole application each time tests are executed. Furthermore, as they should be repetitive and independent, the necessary data should be set up each time. Such a setup requires much time. On the contrary, unit tests work on each class level, so almost no configuration is needed. Some result is expected for given data, with the mocked responses from external services from time to time. Mocking is crucial for unit tests, as they should not interact with any other class. The number of tests required for each test should be present in the code base is described by the test pyramid. The test pyramid (*Fig. 2*) is a way of thinking about different kinds of automated tests that should be used to create a balanced portfolio[21].

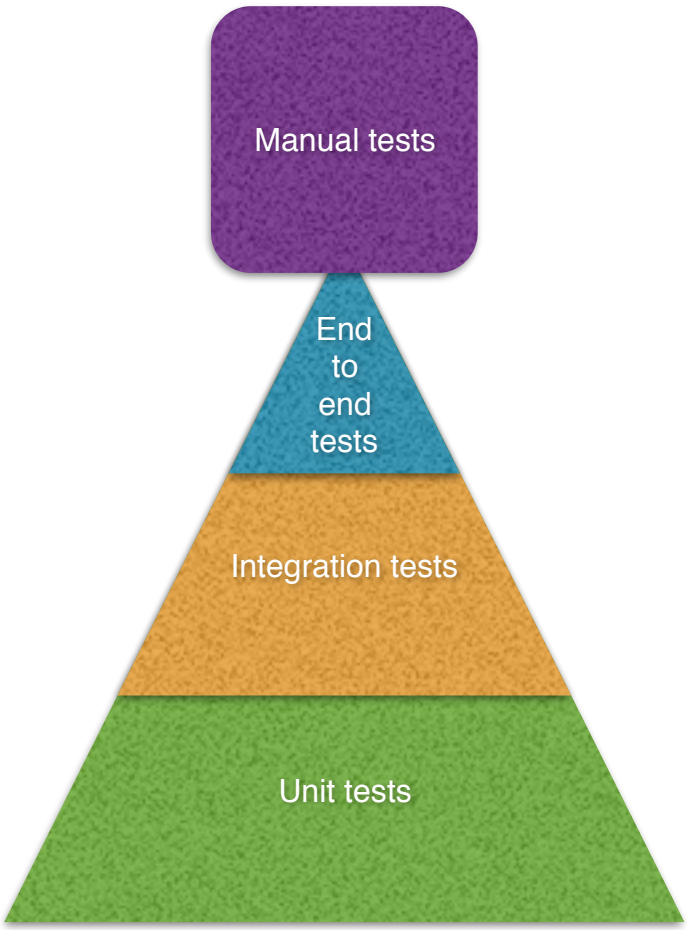


Figure 2. Test pyramid (source own)

Because unit tests are the cheapest and the fastest part of test suite they should build the base for test pipeline. This way almost instant feedback can be provided to the working developer. In a few seconds, hundreds of classes can be confirmed that they produce expected results. Integration tests are more time consumable, but provide more valuable feedback, as they check if the set of classes and methods work correctly. On the top of the pyramid, there are end-to-end tests. They are required for the final assertion, but having too many of them will significantly increase the time of tests execution. Tests run should be a development routine. Too long execution time can create the bottlenecks or discourage developers to skip testing step. That is why it is crucial to provide just-in-time results. With the proper test suites, the developer can just run unit tests or some subset of them to check if his hypothesis is correct and rest of the system works as expected. Then to some external *CI server* will execute rest of tests and that can take minutes or even hours depending on the project size and tests optimisation.

2.2.3. Test automatization

Automated tests suites are just one of the tools in the software engineers toolbox. At the end nobody is perfect, and each time, at the end of the process manual testing should be executed as well. The edge case can be missed, some button forgotten or just specification can be misunderstood. When all tests suite passes, it is just a hint if everything works as expected. It is common to say that tests are green if they pass. Otherwise, we can say, that build is red.

Adding new features is a weak example of test usefulness. Tests are much more valuable when the specification changes or some parts of a system is refactored. With well-written tests, when the changes in the code are done, if the tests are still green one can be quite sure that everything still works as expected.

2.2.4.Five rules of good tests

Not only the production code should be clean and shiny. The test code also should follow some rules to be considered as clean. Despite the regular terms of readable code as

a few conditions or coherent code styling, there is one more acronym to follow. The test that follows *FIRST* principles[22] should be:

- Fast - to give the feedback as soon as possible and encourage a developer to run it frequently.
- Independent - the test should not create an environment for the next test execution. Otherwise, changing the order of them will give false positive feedback that the code is broken or fail in the previous test can cascade the error further what makes it harder to spot the original problem.
- Repeatable - the test should provide the same result each time it is run despite the environment where it is run. It should not matter if it is staging server or local laptop, the result should be same.
- Self-Validating - test should never return the table of results that have to be studied to define if the tests passed or not explicitly. Each test should return just boolean value if the feature is working or broken.
- Timely - tests should be written before the production code to enforce best practices and to ensure that code is testable.

2.2.5. Test Driven Development

As tests become a crucial part of software development, a Test Driven Development (*TDD*) approach appeared. Test Driven Development tries to improve application development by embracing writing tests first. This technic should reveal at the early stage problems with code coupling or not clear class responsibilities. The clue of this paradigm focus on writing tests before writing a production code which should result in failing build, as the asserted function does not exist or doesn't cover given edge-case. The next step is to write the most straightforward code to fulfil tests assumptions and make the build green. Nevertheless, the crucial phase starts with the green build. Now, the code should be refactored to remove duplication and follow other architectural paradigms. This proces has been presented on *Fig. 3*.

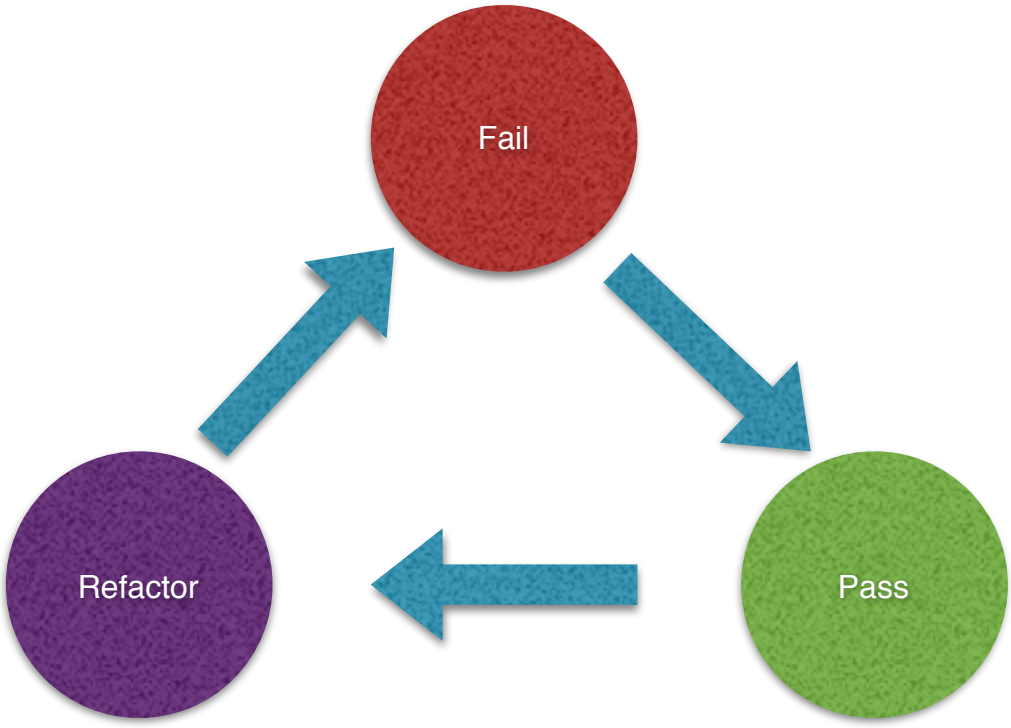


Figure 3. Test-Driven Development cycle (source own)

The main benefit of this approach is to generate short feedback loop and inform developers about the status of an application in short time. Reducing the time gap between developing the code and ensuring that works allow for faster fixes to the codebase. This short time gap can significantly improve the speed of development. It is worth mentioning that even if writing test requires some additional effort, it can drastically reduce to cost of fixing bugs. During the development process, each stage increase the cost of the fix. If the problem will be spotted out in the early stage of development, it is rather easy to repair. If the problem appears in the production environment, it can be costly in time and money to properly fix it. It also forces a developer to rethink class behaviour before its implemented. Shortcuts or dirty solutions are reduced due to the problems with testing them. Moreover, tests are also live documentation of code. Tests are all about the usage of the app. They are example usage which can help to understand at least how the class should behave according to the developer. This information can be invaluable for the newcomers. Missing documentation is a typical problem for many software, as developers usually prefer to solve the issues and write yet another line of code instead of documenting their work. *TDD* requires writing the test before the real code and then write only enough of it to make the test build green. It helps to keep the codebase clean and shiny. One should only write

enough code to pass tests and then just improve it. No additional functionalities should be added during that loop.

Of course, this approach has its downsides. It is apparent that it will require more effort and time to produce a feature. The time gap is the biggest at the early stage of the project especially. In the later phases of the project, it appears that tests help to keep a feature velocity on the similar level when the projects which lack the test increase the time required to deliver requested work. Another problem can lay on the developers. Writing tests first requires knowing the problem domain and expected returned values before starting writing the code. It can be not that simple when the application becomes much more complicated than simple calculators. What is more, it requires some practices what is not that simple to gather. Lack of knowledge and good habits results in the weak adoption of this approach among software houses. Developers should be aware, that "the only way to go fast is to go well".[23]

2.2.6. Behaviour Driven Development

Behaviour Driven Development (*BDD*) emerges from *TDD* and is partially its evolution. It tries to embrace the communication inside development team and between developers and business. The *BDD* process starts with defining the scenario which will present the background and actions need to achieve a particular goal. The scenario should be written by developer and somebody responsible for the whole app together. This way, a developer will know exactly how the app is expected to behave and why such a feature is essential. It gives greater freedom to developer team and provides an opportunity to suggest the best solution from the technical perspective. *BDD* also defines that all people involved in the project should use a standard, ubiquitous language. It means that terms used in project brief and description should be understandable for each team member. They should also be focused on the problem domain and doesn't contain many technical words which are not related to it. Such a solution helps to focus on the main problems instead of teaching non-technical stuff how the software works internally.[24] The most significant benefit of using this methodology is to look at what should be done to the developers so they can deliver proper business value. Fails of many projects are caused by a misunderstanding of the project goals. *BDD* aims to resolve this problem.

2.2.7.Story Behaviour Driven Development

The core part of *StoryBDD* is scenarios. Scenarios should be written in predefined structure with the *Gherkin* syntax[25]. Each feature description starts with the name, reason and the expected business value as presented in *Listing 1*.

```
Feature: Account registration
In order to make future purchases with ease
As a Visitor
I need to be able to create an account in the store
```

Listing 1. Header of a Sylius feature description[26]

The feature presented above is the example of usage. It is a part of Sylius e-commerce platform, which is an open source e-commerce framework. It gives a brief insight that given scenario will focus on account registrations and some key facts about it:

- Main benefit: "in order to make future purchases with ease"
- An actor: "as a Visitor"
- What should be achieved: "I need to be able to create an account in the store"

The feature value description is followed by the list of scenarios which describes real examples how it should be achieved. Each scenario follows the same pattern[19]:

- Describe an initial state
- Describe the required actions
- Describe expected outcome

Each description is done in steps which should start with some keywords. The *Table 1* contains summarised description of each step according to the *Gherkin* reference[19].

Keyword	Description
Given	Describes context for the feature
When	Describes action that should be executed
Then	Describes expected outcome
And/But	Helpers which allows writing more natural scenarios.

Table 1. Gherkin key words reference (source own)

It is essential to describe the feature in the most readable for the business way as it is possible. The scenarios should not contain any *UI* or technical details. The main aim of these descriptions is to create a communication bridge between the development team and the product owner. These scenarios are the primary reference to check if the application fulfils what has been requested. It ensures application owner that it will work as was planned and the developers that if the acceptance tests are passed customer cannot complain that something else was expected. It is an insurance policy for both sides.

```
Background:
    Given the store operates on a single channel in "United States"

Scenario: Receiving a welcoming email after registration
    When I register with email "ghastly@bespoke.com" and password "su-
itsarelife"
    Then I should be notified that new account has been successfully
created
    And a welcoming email should have been sent to "ghastly@bespoke.-
com"
```

Listing 2. Background of feature file[26]

In the *Listing 2*, the expected result is to send an email to the newly registered user, and described it by two sentences started with *Then* and *And*. This action should take place just after a registration, what is described in the *When* sentence. The whole example takes place in the store that operates on some single channel in United States. It is worth mentioning that all scenarios do not contain any technical details. It is not known if it should be interpreted on the website, some desktop app or with *API* request. Implementation details are left to the developers. Also, some concepts tackled in the scenario are not apparent if one is not familiar with the domain. *BDD* embrace the ubiquitous language scenarios to provide some value for everybody involved in the platform, not for everybody everywhere.

The side-effect of this methodology is that these scenarios could be turned into acceptance tests. It is also possible to interpret them on different levels of an application. Such a test suite gives extreme confidence that a proper business value has been delivered to the client. When the scenarios could be interpreted as user actions, then they can be connected with some continues integration pipeline. Thus, each time new features are added, the whole platform is checked that nothing else has been broken. It makes the work more straightforward, and developers can focus on delivering new values to the business which can grow without any maintenance problems.

2.2.8.Specification Behaviour Driven Development

When the scenario is written and accepted, developer proceeds to implementation. As a result that *BDD* is adopting an outside-in approach to software development, the developer starts implementation at the outermost layer of an application. Each service required to fulfil scenario requirements, one by one, is described by unit test, then implemented with enough code to meet preconditions. If the tests start passing on a unit level, then functional or end-to-end test can be run. If they pass, a tested code can be refactored or new feature implemented. Otherwise, the code should be fixed or new unit test written to finish user story.

The fundamental part of *SpecBDD* is that it takes effect only on the level of each class. Each class is a black box from the implementation perspective. What is more interaction with other services is expressed by readable methods like *willReturn* or *shouldBeCalled*. Test code can be read almost by anybody. It can be useful to ask no technical people or developers with a different background is that ok.

An example below shows test of the *Email* value object creation. The first stage is to define value object requirements, as was presented on *Listing 3*.

```
final class EmailSpec extends ObjectBehavior
{
    function let(): void
    {
        $this->beConstructedWith('email@example.pl');
    }

    function it_is_an_email(): void
    {
        $this->value()->shouldReturn('email@example');
    }

    function it_throws_an_exception_if_an_email_is_invalid(): void
    {
        $this->shouldThrow(InvalidEmailException::class)-
>during('__construct', ['notemail']);
    }
}
```

Listing 3. Example spec file

The *Listing 3* contains a lot of programming language related noise. The code will be understood by developers, but it can be not clear for non-technical people. Here is a simplified description for the sake of example:

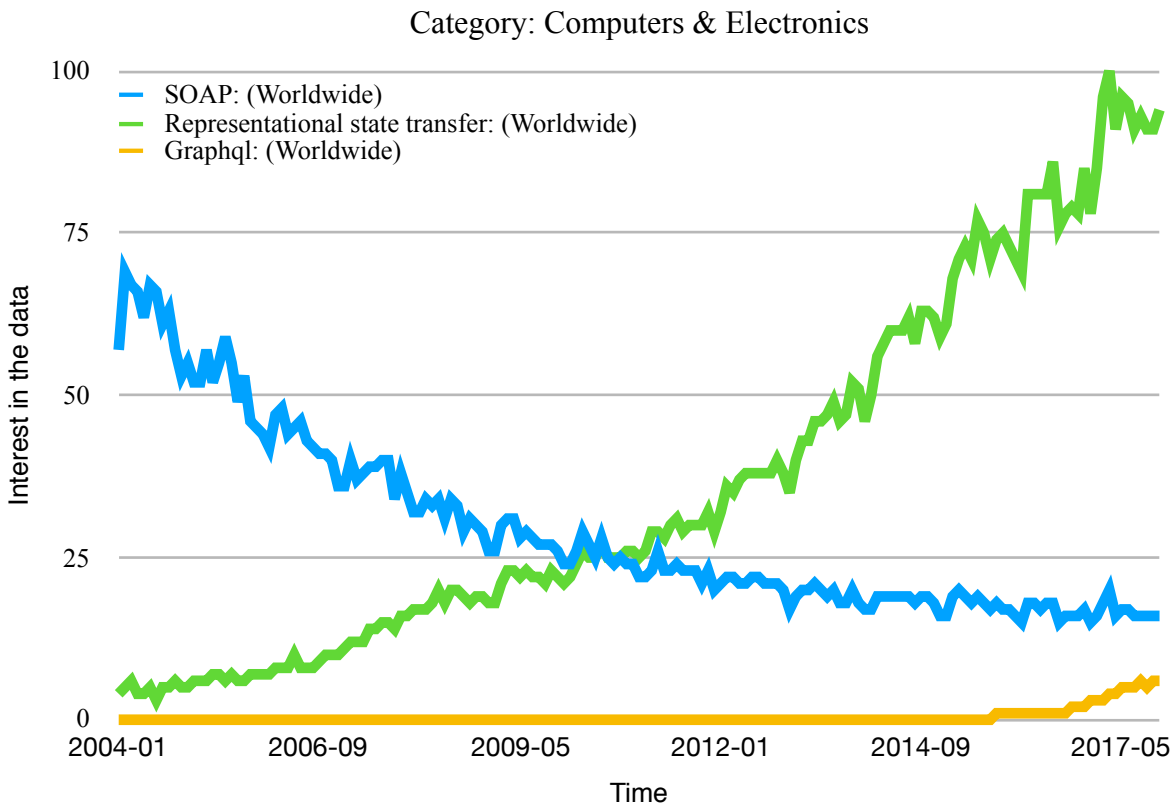
```
Email

    let this be constructed with email@example.com

    it is an email
        this value should return email@example.com

    it throws an exception if an email is invalid
        this should throw exception during construct notemail
```

Listing 4. Simplified example spec file

The reduced class definition is easy to understand by everybody regardless of their background. It is obvious what are an expectation and how class should behave. It is a good background for discussion if the behaviour is correct and documentation for future. Thanks to this format it is possible to interpret it as a test to protect code from regression. Once description is defined, a developer can implement required features. Of course, the code should only make test pass. After that, some refactoring should be done to fit projects *definition of "Done"*. *Definition of "Done"* is a definition of required criteria which have to be fulfilled in order to define some work as done[27]. If succeeded, then other specification can be written, or another part can be added to current one.

2.3. API communication

Over the years customer needs significantly increased which results in much more advanced and complicated platforms that serve them. It became not efficient to develop everything from scratch and put all required features into one core application. The codebase would become unmaintainable because of its size and probably ineffectiveness of creating a custom solution. The software tries to resolve some specific problem, and it is not wise to *reinvent the wheel*. As a result, companies try to focus on critical parts of the software that gives them a competitive advantage over others and purchase other parts of needed software. It decrees the amount of maintained code and also splits the responsibility of some parts of the system. Generic buyable systems have to be integrated with many custom solutions. Many of them are not tightly coupled to the primary system just providing required functionality over a Hypertext Transfer Protocol (*HTTP*) connection. *HTTP* protocol is one the most popular communication standards over the network, but it also needs some rules to improve the way, how external systems share their data. Three communication protocols are worthy mention at the moment.



Plot 1. Number of searches of SOAP, Representational state transfer and GraphQL terms in the Google Search since 2004 in Computers & Electronics related topic[28]

Data shown in the *Plot 1* above represents a number of searches in the Google Search since 2004. The data has been filtrated to aggregate only searches related to Computers & Electronics topic. Otherwise, the data could be misrepresented as some of the terms of the plot have a significant number of searches outside computer science field. Numbers on the y-axis represents interest in the data. It means that they represent search interest relative to the highest point on the chart for the given region and time. A value of 100 is the peak popularity of the term. A value of 50 means that the term is half as popular. A score of 0 means there was not enough data for this term.

2.3.1.Simple Object Access Protocol

SOAP is the oldest one of the mentioned three. *SOAP* has been developed in the late 90s and is crafted to address the problem of large industries and aims to help with the needs of the enterprise market. The basic concept assumes that the data is accessible via descriptive methods like *getProduct*. It follows an enterprise approach and works well with any communication protocol. Unfortunately, it is pretty hard to implement, and it uses only

XML notation. It has noticeably high data overhead and is uncommon among web and mobile developers. It was the main technology during the first decade of 2000's, but with the increased interest in *RESTful API*'s it lost a leadership position around the 2010s'. It is mostly used by financial and sales related companies which don't need to have easy accessible *API*'s.

2.3.2. Representational State Transfer

REpresentational State Transfer (*REST*) is a lightweight protocol it embraces the philosophy of an open internet. It is easy to use and pretty straightforward to use. It represents data with the nested nouns to describe the connection between resources. It does not enforce any specific message format so either *XML* or *JSON* could be used, although *JSON* is the most popular choice. It is widely used by social media, web chats or mobile services. It is the typical language of the Web nowadays. Not that popular at the early stage, but it is the primary communication protocol since 2010.

2.3.3. GraphQL

Newest among mentioned communication protocols, much younger then competitors. It tries to resolve the problem of defining the outputs by servers rather than clients. As each client is different, usually it requires modified set of data. *GraphQL* exposes a single entry point for the whole application and available data, and what would be delivered is up to the client. There is a long way before it will become a mainstream syntax but as it is used by big software players like *Facebook* or *GitHub* one day it will.

2.4. Data storage

Most of the applications cannot work correctly without some data storage. Depending on needs it could be some memory storage, files or some database. This information allows keeping track of users and their personalised profiles. Data could be different and depend on its form different approaches should be applied. Storing data just in the *RAM* will erase it with the time. The files are hard to maintain and gives some overhead over reading and writing operations. The most common solution is some database, which offers two kinds of approaches: *SQL* and *NoSQL* implementations.

2.4.1. SQL databases

Relational databases represent an *SQL* approach to data storage. In relational databases data is normalised to the simple values and stored in the tables. Relations between the objects are expressed by the reference to the object identifiers. Building algorithms ensure fundamental data correctness and existence of both sides of the relationship. On the other hand, this storage style does not allow to keep any information on a relation itself. To express some relation value, for example, a custom object with two relationships is needed. Nevertheless, this kind of database is perfect storage for the most of data, as it is a clean way of representation. It also allows for data type and relation validation, and it is straightforward to use. A perfect solution for application which primary concern is to get some data, update the record and persist it back to database (*CRUD* applications) or data integrity is essential.

2.4.2. NoSQL databases

On the contrary, there are *NoSQL* databases. The main difference is that the data is stored in *JSON*-like field-value pair documents. It gives a significant flexibility advantage over *SQL* database implementation because there are no constraints on data structure. On the other hand, the most significant benefit can become a disadvantage. It is much harder to keep data consistency or represent many-to-many relation between objects. Lack of transaction support is also a disadvantage of *NoSQL* like databases. An atomic update is done on each document separately, so to keep synchronisation between two documents some additional code on client side is required. It is perfect for frequently changing data structure, keeping geological data or graph-related problems.

2.5. Development process

Every engineer aims to produce a reliable solution with the most suitable and best available tools. In the engineers work not only the result matters yet its quality, difficulty in maintaining and the measurement of project goal fulfilment. A proper set of tools can help a lot in order to keep the project in good shape and with the high quality.

2.5.1. Version control system

Software development is a process that goes in months or years. During that time, requirements can change, or new features are added. In order to keep pace with them, many developers have to collaborate to fulfil the business expectation. When the code change frequently and many people add their changes to the same code repository it is not hard to make a mistake or break somebody's else work. Version control systems (*VCS*) aims to solve this problem and allow for smooth collaboration over big developing teams. During the years many approaches appeared on the market. *SVN* was the first significant improvement. It allows to keep the code in centralised place, and everybody could synchronise its code with the central catalogue and fix conflicts if necessary. Still, a drawback of this solution was that only connected current state of code could be saved when one was connected to the internet. Also, it did not support natively many branches of the same code in the same repository.

A few years later *Git* appeared. *Git* is a distributed version control system, which allows keeping track of code changes during the project lifetime. As every version control system(*VCS*), *Git* provides a way to record changes to the set of files and recall to the specific version later. It will also keep saving an author of change, so it is much easier to find out who and why made such a decision. Bringing back to the last working state is effortless if some error will occur. It is also straightforward to compare changes over time and find out the source of the problem. Version controls system could be local, centralised or distributed. While a local type is pretty handy for the developer, it does not allow to collaborate with the team. This problem may be solved by using a centralised *VCS*. It enables to share the results of the work of each team members and gives fine-grained control over who does what. The critical disadvantage of such a solution is a single point of failure. If the primary repository will fail, the collaboration is blocked, and without a proper backup, the history of changes can be lost. Hence the distributed *VCS* comes to the game, where each instance is a direct mirror of the central repository. If any of them will fail, it is pretty easy to recover them. *Git* is slightly different from other *VCSs* because instead of storing the change that happens, it saves a snapshot of the whole system in the

given point of time. If the file does not change, then it just copied the reference to the previously saved state. It makes it lightweight and efficient.

2.5.2. Version control system hosting

GitHub is a web-based *Git VCS* hosting service. It not only allows to store the code outside the local computer which can be a code backup but also allows for code review and outside integration. The code review process could be useful even in one-men-army projects. Pull Requests - proposal of new code changes to the central repository can be blocked if some set of requirements will not be fulfilled. Code conducted in the one view with the direct comparison between the previous and the current version can be useful to spot the problems in the code or style violations. Next important feature of *GitHub* is that its ready to use integration with other systems such as deployment platforms or continuous integration servers.

2.5.3. Continuous Integration

To ensure proper code quality and avoid code regression project needs tests. However, these tests have to be run over and over with each new change proposed to be a part of the central code repository. It is inefficient to run them locally or just expect the team members to run whole tests suite each time they propose code change. Tests could take even hours to complete entire test pipeline, and people can tend to avoid running them. Some external check should be a part of developing process. The best solution implies running a status check each time a new change was proposed and block merge possibility if the check fails. Another test should be done after merging to ensure that nothing has been broken in the meantime. *Continuous integration* servers could be a potent tool in order to keep code in good shape for a long time. It may not only check if the tests pass but also if the vendor libraries are corrected installed, the code has expected quality or statically analyse it to give some valuable feedback to the developer with the possible code improvements.

2.5.4.Static code analysis

Static code analysis is the analysis of the computer software without actually running programs. In most cases, it is performed on some version of the source code done by the automated tool. It is a complementary technique to the conventional dynamic testing to obtain additional insurance on software. Not many standards are defined, and the impact of this method is hard to measure. However, according to the "Industrial perspective to the static analysis"[29] the static analysis is recommended for almost all critical software, but the less in-depth study could be used with nearly any type of software. Lack of defined standards requires additional work in identifying the depth and nature of it. The significant impact on this consideration can have a chosen language. Programs which uses dynamically typed paradigm are much more difficult to analyse than the strongly typed ones.

3. Recommendation engines

On a day to day basis, people are leaving thousands of footprints all around the network. These leads are the most valuable product on the whole internet. The product that everybody wants to possess. Tracks and personal data are the currency for the features that are in theory free of charge. This kind of knowledge is extraordinarily important for machine learning engineers, whom can assign an appropriate group to target and predict preferences. This data can increase the number of sales (conversion rate) significantly, so the companies are willing to pay a lot for it. With enough information, one can feed the users with a proper data and offer special occasions. However, in order to be able to do it, the data has to be filtered and then grouped around some similarities. The resemblance could be defined as a similarity between items or between users actions. Each of association can provide slightly different outcome and be used in various contexts. For example, an additional lens can be recommended when buying a camera, or some different object can be recommended because friends of user bought it also.

3.1. Collective intelligence

Collective intelligence appears in large groups of people. It is a phenomenon where a shared intelligence emerges from the collaboration and competition of many individuals[30]. Such intelligence in animal kingdom in forms of swarms, flocks or herds. The similar behaviour also appears among humans. Formerly a questionnaire or census provides an insight into the collective intelligence of some group of people. Nowadays, thanks to the internet revolution it became easier than ever. Social networks, e-commerce websites or web browsers contains probably even exabytes of data[31]. This data allows to generalise users behaviour, forecast their needs and improve conversion rates or keep users at websites. On the lowest level teamwork can be treated as an example of collective intelligence as well. Such teamwork can be spotted at *Wikipedia*[32] website for instance. A collaboration of thousands of users resulted in the largest encyclopedia available. The similar effort, would not be possible for any coordinated group.

3.2. Content-based filtering

Content-based filtering is a recommendation based on a comparison between product content and user profile preferences. It tackles the problem of items from an extensive collection which can be useful or interesting for the end users. These filters try to calculate a correlation between previous user decisions and other available items[33]. If the items have a common factor, and one of them was bought or highly rated by the user, the second one should be recommended. The main benefits of this approach is that is bypass problem of a cold start. Since the system delivers information based only on one user behaviour, it can become robust much faster compared to collaborative filtering. Likewise, new items in a database could be recommended instantly, as the only similarity to previous items is taken into account.

3.3. Collaborative filtering

This recommendation paradigm is based on a similarity between like-minded users. It is based on the assumption that users with similar past action are more likely to proceed similar actions in the future as well[34]. The similarity between users can be evaluated based on their social networks or actions on the same website. The main benefit of this approach is that not only similar items are taken into account. This advantage may be invaluable for business not heavily focused on one topic such as marketplaces like *Amazon*. Variety of products promotes solution where similarity between them is not a key.

3.4. Similarity measurement

A crucial step in each recommendation is to compute a similarity between items and then choose the most similar ones. Firstly we need to isolate items that could have something in common and then compute a similarity between them[34].

There are a number of methods of items similarity computation. Below, two of them would be described. They are:

- Cosine correlation
- Pearson correlation

3.4.1.Cosine correlation

Cosine correlation treats items as a vector in m-dimensional space, where the similarity is expressed by the cosine of the angle between these two vectors. If the vectors have the same orientation, then the cosine similarity is equal to 1, on the contrary, when they are unlike, then cosine similarity value is equal to 0. This value basically can qualify how close users preferences are.

3.4.2.Pearson correlation

Pearson correlation is pretty similar to cosinus correlation with one notable improvement: it takes into an account the fact that different user will have different mean ratings. The similarity in ratings can be computed with better results if both users are similar, where one just tend to be a more harsh critic.

4. Implementation

Best practices mentioned in chapter 2 has a significant impact on the code and folder structure. The theoretical background is much easier to understand when there is an example implementation of it. Defining a sample application project scope, it should allow to:

a) Register a new account
b) Log in
c) For logged in user
- Add new beer to catalogue
- Rate available beer
- Fetch recommended beers

d) For all users
- Browse beers catalogue
- Check given beer details

4.1. Definition of "Done"

The project is defined as done (*DoD*) when each feature will be covered by automated test (*TDD, BDD, FIRST*), application code will be clean, and high quality (defensive programming, *SOLID, Object Calisthenics*) and communication will be done with *RESTful API*.

4.2. Chosen tools

4.2.1. Core language - PHP 7.2

The platform will be implemented in *PHP* (Personal Home Page) language. *PHP* is a scripting language created in 1996. It became the most common solution for web development. It has been powering 83.1% of all the websites whose server-side programming language was known (data relevant for February 11, 2018)[35]. It is used mostly for low-to-middle traffic websites when *Java* and *ASP.Net* are mostly used for high traffic enterprise solutions. Since the version, 7.0 *PHP* introduced scalar type hints and a

possibility to check them strictly[36]. It means that wrong type value will throw a fatal error during code execution. More than that, *PHP* 7.0 brought a noteworthy performance improvement[37] (up to 100% improvement compared to *PHP* 5.6 - last minor version of *PHP5*). Current recommended version for all application is *PHP* 7.2 published in November 2017 contribute to even more performance improvements and some generic type hints (like "object")[38]. Another essential factor of *PHP* language is its open source community. Thanks to it many libraries inspired by other languages are ported to *PHP* environment.

4.2.2. Dependency manager - Composer

Composer is a dependency manager tool crafted for *PHP*. Like *Maven* for *Java* or NPM (*Node Package Manager*) for *JavaScript*, it downloads third-party libraries to the specific folder. The whole idea behind it is that depended libraries should not be changed in vendor folders. They should be extended with Object-Oriented programming paradigm. Before now, dependencies were copy-pasted to vendor folder which gave an opportunity to developers to change something in their source code. This practice worked at the begging, but it becomes a nightmare to maintain. The new version of a changed library could break everything. Outdated libraries cause security issues and are hard to manage. *Composer* composed together with semantic versioning resolves that problem. This library provides a framework, which downloads all dependencies each time application is set up. It blocks a possibility of depending on changes in third-party libraries.

4.2.3. Testing frameworks

The application will be tested on different levels, which requires different libraries specialised in given work.

a) Behat

Behat is an official *Cucumber test framework* implementation for *PHP* which embraces *StoryBDD*. *Cucumber* is a multi-platform test framework which takes advantage of *BDD* practices and builds a perfect ground for its adoption. It is available for a dozen of software platforms, from *Java*, throughout *Ruby* up to *Javascript* or *Python*. In the root of

BDD, there is an understanding of business goals, not testing itself. Interpreted feature scenarios which can be used for automated testing is just a side effect of its this practice. *Cucumber* framework and its *PHP* implementation - *Behat*, makes it easy and smooth. For a reason explained in the chapter 2.6.6 all feature scenarios are written in with *Gherkin* syntax.

b) PHPSpec

PHPSpec is a tool for *SpecBDD* and unit testing. A fundamental feature of this library is mocking all external dependencies out of the box. Each element defined as a constructor parameter will be mocked by default. This feature allows focussing only on the expected result from the tested class. Its main aim is to allow a developer to describe expected result and then bootstrap the class scaffolding. *PHPSpec* will generate a class file and required methods, including public or named constructors. Thanks to template system developer can define default class implementation which can help to keep some standards, such as making a class final by default or adding the "strict_types" definition by default.

4.2.4.Static analysis - PHPStan

PHP as a language with wick type hinting and script nature is easy to be broken. Defensive programming, tests and static analysis are just a few concepts that help to maintain its code. In the interest of its community, a lot of static analysis tools has been developed, to name a *PHPStan* or *Phan* as an example. Most of them are focusing on finding broken code samples and prove incorrectness rather than correctness. They usually work on *PHP Abstract Syntax Tree* (*AST*) to validate if expected arguments and returned values match. The project will be tested against *PHPStan*, the library used by several Open Source Software (*OSS*). An additional bonus of this library is a fact that it is battle tested and has several predefined check suites. *PHPStan* can be run on one of eight levels of strictness: from 0 to 7, where 0 is the weakest check, and 7 is a value impossible to reach at current stage for most of the existing projects.

4.2.5.Service buses and event store - Prooph

Architectural choices such as onion architecture and *CQRS* paradigm described in the previous section are much easier to implement with some libraries. Service bus is present in the form of two separated buses:

- Command Bus
- Event Bus

Both of them allow for smooth usage of *Command* pattern - a behavioural pattern described in "Design patterns: Elements of reusable object-oriented software"[39]. On each of busses, a different class is dispatched, *Command* on *Command Bus* and *Event* on *Event Bus*. These two models are behaving as a command in the way described in *Design Patterns*. The distinction between them is based on intentional limitation. *Commands* which are a representation of user actions can be interpreted by one, and only one *Handler*. It means that only one class can react to user actions. The main idea behind it is to use split how a customer interacts with an app from an interpretation of this action. Thus, the user action can come to the system from different parts of a system, but it will always be handled the same way. These classes follow the naming convention and are always written in present simple tense, for example, *RegisterConnoisseur*[1]. A class that will react to this command will have corresponded name *RegisterConnoisseurHandler*. As a result of its execution, one or more *Events* can be dispatched to *Event Bus*. *Events* encapsulate information about past action that has been finished. By their very nature, their names are written in past simple tense, for instance, *ConnoisseurRegistered*. They can be interpreted by many listeners, which can store some critical information about them or just remark its existence.

An additional profit of this separation can be achieved if all these events are stored in some database. Commonly such a database is named *Event Store*. Storing all these events may give a notable possibility for future audits, as each event is a record of user action results. It also allows rebuilding all instances which react to events.

---

[1] Connoisseur - a domain name of the application user.

Among all possible solutions available for *PHP*, only two of them can be considered as sufficient implementation for both requirements. There are *Prooph* and *Broadway* libraries. Other great libraries usually support command bus part only. This project will be based on *Prooph* implementation as it seems to be more actively supported at the time of writing.

For the sake of current implementation on tests environment, an *Event Store* uses in-memory implementations.

4.2.6. Framework - Symfony 4.0

Highly used languages usually provide several alternatives for some problems. *PHP* world is rich in frameworks and micro-frameworks implementation. The most advanced and usually considerable are *Symfony*, *Zend* and *Laravel*. All are great and allows to achieve similar goals. Undoubtedly the developer, not the framework is responsible for code quality and project state. Newest changes in Symfony ecosystem together with the release of version 4.0 (published November 2017) allows for incremental and lean changes to the project. Thanks to *Flex* library developer can start with just a micro-framework implementation and adds only necessary libraries. *Symfony* is also a project highly inspired by *Spring* framework - one of the most popular *Java* frameworks. *Symfony*, similar to the most of the used libraries, is available under *MIT license* and is developed as *OSS* by many volunteer contributors all over the world. It provides an excellent background for Model View Controller (*MVC*) or Action Domain Response (*ADR*) architectural patterns. This framework contains trustworthy abstraction over *HTTP Request-Response* handling with the support of its methods and codes. It contains a decent Dependency Injection (*DI*) component that promotes applying *SOLID* principles described in section 2.1.4. Dependency Injection is a pattern that allows removing the dependency from the application class and changes the way how one object acquires a dependency. Therefore promotes Dependency Inversion Principle, as constructor parameters are preferred to be defined as an abstraction to provide an easy way of changing their implementation.

4.2.7.Database - Doctrine2

With a proper application design, a database can become an implementation detail. Libraries like *Doctrine2*, *Propel* or *Eloquent* provides a useful abstraction over database connection, so the developer does not need to worry about it. Probably the most famous database handling library for Symfony framework is *Doctrine2*. This component has been widely inspired by *Hibernate* (*Java ORM* implementation). It is a composite of several libraries comprehensive database management. It makes defining entities mapping smooth and painless. Its repository implementation provides default hydrators which resolve the problem of recreating object based on data fetched from a database. As a result of using reflection to fetch data from an object, it allows for proper state encapsulation. Nonetheless, usage of this library imposes removing "final" keyword on entities in exchange for making class proxies which improve library performance. Doctrine2 also provides a few default database abstractions such as *Object-Relational Mapping (ORM)* for *SQL* databases or *Object Document Mapper (ODM)*.

a) SQLite and ORM

Part of the *Doctrine2* library is an *Object-Relational Mapping (ORM)* component. It is a unit responsible for abstraction over *SQL* databases. Gives a possibility to define an entity mapping in several ways, such as code annotation or in *XML* files. It has adapters for several database implementations like *MySQL*, *MS SQL Server*, *PostgreSQL*, *IBM BD2* or *SQLite*. All these features allow postponing choosing an actual implementation. The project has been created and tested under *SQLite* database implementation. This database required the least amount of setup and was enough for current needs.

b) Neo4j and OGM

A key feature of this application is making a recommendation. According to the description from the third chapter, there are several possible implementations of recommendation engines. However, choosing given paradigm, it gives a freedom of choice over implementation details. One of the possibilities was to develop recommendation engine in *Python*, most common choice in this field. Choosing *Python* as a

recommendation engine would require a whole application to be written in *Python* with *Django* web framework or to be integrated with the current *PHP* application. Such integration can be done for instance with *HTTP* protocol and communication over *RESTful API* or using messaging pattern. The drawback of this solution is that it would noticeably increase the project complexity. Another possible solution is to use graph databases. As the official *Neo4j* website states: "Graph databases easily outperform relational and other *NoSQL* data stores for connecting masses of a buyer and product data (and connected data in general) to gain insight into customer needs and product trends"[40].

Project event-based architecture allows for multiple read data projection. Consequently, feeding two different databases with required data to operate, become effortless. Thus graph based recommendation engine implementation become flawless inside project. *SQL* based data allows for simple data management and listing available resources. When the end user asks for a recommendation, then data can be fetched from a graph database.

In February 2018 *Neo4j* was most popular graph database implementation according to DB Engines ranking[41]. It is a *NoSQL* database. Therefore it stores data in *JSON*-like key-value format. It has an adapter for *PHP* as well as *PHP Object-Graph Mapper* (OGM) implementation which enhance graph related entities implementation.

What is more, *Neo4j* allows for the elegant presentation of interconnected nodes, which is helpful for business analytics and developers, what is presented in *Fig. 4*.

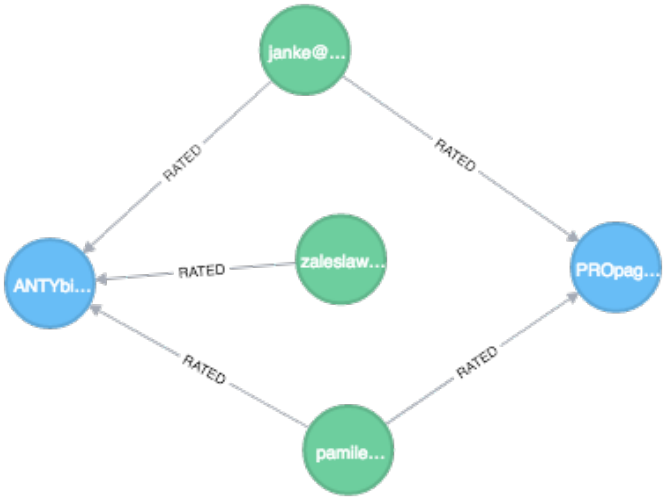


Figure 4. Sample Neo4j query result (source own)

a) Cypher

Cypher is open, vendor-neutral graph query language implemented in *Neo4j* database. It allows for a readable way to match patterns or relationships within graph database. It is declarative query language which allows to state, what should be done instead of how it should be done. It is easy to learn and intuitive thanks to English prose and intuitive iconography. It is highly inspired by *SQL* and borrows collection semantics from *Python*[42,43].

```
MATCH (c:Connoisseur)-[r:Rates]-(m:Movie)
RETURN c, r, m
```

Listing 5. Sample Cypher query

Cypher queries are self-explanatory and easy to comprehend. The query presented in *Listing 5* matches all *Connoisseurs* and *Movies* pairs connected with *Rate* relation. It is also worth mentioning that relations are first-class citizens in *Neo4j* databases, and they may contain own properties.

4.3. Project Development

Sample project aims to present modern methods of software development on an example of recommendation platform. The application is named *Beery* and allows to share details and opinions about crafted beers with other users of the platform.

4.3.1. Development setup

An early stage of project development starts with setting up project scaffolding. Minimal configuration contained initialled *Git* repository, bootstrapped *README* file with project description and configured composer to follow the *PSR-4* autoloading standard. The *README* file contains elemental description of project goals, road map and setup information.

4.3.2. Product backlog

In the introduction to this chapter following requirements have been defined for the project:

a) Register a new account
b) Log in
c) For logged in user
- Add new beer to catalogue
- Rate available beers
- Fetch recommended beers

d) For all users
- Browse beers catalogue
- Check given beer details

These requirements can be called a product backlog. Product backlog is a list of product requirements usually in form of User Stories. Important factor is that this list is never closed and never done. It contains ordered a list of all features that can be added to the platform. This list evolves together with the project, when the domain knowledge grows in team members[44]. It is a part of *SCRUM* methodology, which is "a framework within which people can address complex adaptive problems, while productively and creatively delivering products of the highest possible value"[27].

In the first stage of a project, requirements have been translated into user stories. User stories define background and action required to achieve given functionality. As was described in chapter 2.2.6, user stories are defined with *Gherkin* syntax and are written in ubiquities language. Scenarios should be written in the close collaboration with the client, should express a real business value gained at the end and benefit for the end user.

First two features offer a possibility to register and log in. The user needs to register and then to log in, in order to get access to the full version of the application. Guest users do not have access to some functionalities, such as adding new beers to the catalogue or rating them. Because logging in functionality is closely related to registration both of them will be expressed in one story presented in *Listing 6.*

```
@connoisseur @application @api
Feature: Registering a connoisseur
    In order to be able to use application fully
    As a Connoisseur
    I want to register

    Scenario: Registering a connoisseur
        When I register the "Rick Sanchez" connoisseur with the
"rick@morty.com" email and the "birdperson1" password
        Then I should be able to log in as "rick@morty.com" with
"birdperson1" password
```

Listing 6. Feature: Registering a connoisseur

When the users are logged in, they can add a new beer to the catalogue. In the current implementation, only name and ABV value (Alcohol By Volume) is required for beer creation. As was written before, the customer has to register and log in to the platform to have access to this functionality. These preconditions have been expressed in the scenario presented on *Listing 7*.

```
@beer @api
Feature: Adding a beer
    In order to share details about the beer with the community
    As a Community Member
    I want to add a beer to catalogue

    @application
    Scenario: Adding a new beer
        Given I registered as "Rick Sanchez" with the "rick@morty.com"
email and the "birdperson1" password
        And I am logged in as "rick@morty.com" with "birdperson1"
password
        When I add a new "King of Hop" beer which has 5% ABV
        Then the "King of Hop" beer should be available in the
catalogue

    Scenario: It is impossible to add a new beer if customer is not
registered
        When I try to add a new "King of Hop" beer which has 5% ABV
        Then I should be notified that I'm not allowed to do it
```

Listing 7. Feature: Adding a beer

Once the beer is added every registered user can rate this beer. A client with whom the features were defined also requested an average rate displayed on beer details page. Undoubtedly, it can be valuable information for the user. Nevertheless, scenarios become a bit more complicated from this moment. For sure, it is essential to check if different rates are not mixed up and if the average rate is calculated correctly. It was covered by feature presented in *Listing 8*.

```
@rate @api
Feature: Rating a beer
    In order to share my opinion about a beer with the community
    As a Community Member
    I want to rate a beer

    Background:
        Given the "King of Hop" beer with 5% ABV has been added

    @application
    Scenario: Rate from single customer
        Given I registered as "Rick Sanchez" with the "rick@morty.com"
email and the "birdperson1" password
        And I am logged in as "rick@morty.com" with "birdperson1"
password
        When I rate the "King of Hop" beer 5
        Then the "King of Hop" beer should have average rate 5

    Scenario: It is impossible to rate a beer if customer is not logged
in
        When I try to rate the "King of Hop" beer 5
        Then I should be notified that I'm not allowed to do it

    @application
    Scenario: Multiple rates from single customer
        Given the "Primátor Exkluziv 16°" beer with 7.5% ABV has been
added
        And I registered as "Rick Sanchez" with the "rick@morty.com"
email and the "birdperson1" password
        And I am logged in as "rick@morty.com" with "birdperson1"
password
        When I rate the "King of Hop" beer 5
        And I rate the "Primátor Exkluziv 16°" beer 3
        Then the "King of Hop" beer should have average rate 5
        And the "Primátor Exkluziv 16°" beer should have average rate 3

    @application
    Scenario: Rates from multiple customers
        Given the "King of Hop" beer with 5% ABV has been added
        And there is a "Rick Sanchez" connoisseur
        And the "Rick Sanchez" connoisseur rated the "King of Hop" beer
5
        And I registered as "Mr Meeseeks" with the "mr@meeseeks.com"
email and the "lookatme" password
        And I am logged in as "mr@meeseeks.com" with "lookatme"
password
        When I rate the "King of Hop" beer 4
        Then the "King of Hop" beer should have average rate 4.5
```

Listing 8. Feature: Rating a beer

However, the core of application can be only used when several users will rate different beers. Recommendation engine requires entrance data to be able to calculate a similarity between users. This similarity allows presenting some suggestions to the customer. In this scenario shown in *Listing 9*, the biggest setup is required.

```
@recommendation @api
Feature: Recommending best beers for connoisseur
    In order to make better choices when trying new beer
    As a Connoisseur
    I need an advice from application

    Background:
        Given the "ANTYbiotyk" beer with 7% ABV has been added
        And the "PROpaganda" beer with 6.8% ABV has been added
        And the "Kasztelan Niepasteryzowane" beer with 5.7% ABV has
been added
        And there are "armin@van.kraken", "pamile@krawczyk.pl",
"janke@mops.com", "zaleslaw@middleage.com" connoisseurs

    Scenario: It is impossible to get a beer recommendation if customer
is not logged in
        When I try to ask for a beer recommendation
        Then I should be notified that I'm not allowed to do it

    Scenario: Getting a simple recommendation
        Given the "armin@van.kraken" connoisseur rated the "ANTYbiotyk"
beer 4.5
        And the "armin@van.kraken" connoisseur rated the "PROpaganda"
beer 4
        And the "armin@van.kraken" connoisseur rated the "Kasztelan
Niepasteryzowane" beer 2
        And I registered as "Rick Sanchez" with the "rick@morty.com"
email and the "birdperson1" password
        And I am logged in as "rick@morty.com" with "birdperson1"
password
        And I rated the "PROpaganda" beer 4
        When I ask for a beer recommendation
        Then the "ANTYbiotyk" beer should be suggested

    Scenario: Getting an advance recommendation
        Given the "armin@van.kraken" connoisseur rated the "ANTYbiotyk"
beer 3.5
        And the "armin@van.kraken" connoisseur rated the "PROpaganda"
beer 4
        And the "armin@van.kraken" connoisseur rated the "Kasztelan
Niepasteryzowane" beer 2
        And the "pamile@krawczyk.pl" connoisseur rated the "ANTYbiotyk"
beer 4
        And the "pamile@krawczyk.pl" connoisseur rated the "PROpaganda"
beer 4.5
```

```
        And the "pamile@krawczyk.pl" connoisseur rated the "Kasztelan
Niepasteryzowane" beer 1
        And the "janke@mops.com" connoisseur rated the "ANTYbiotyk"
beer 2
        And the "janke@mops.com" connoisseur rated the "PROpaganda"
beer 4
        And the "janke@mops.com" connoisseur rated the "Kasztelan
Niepasteryzowane" beer 3
        And the "zaleslaw@middleage.com" connoisseur rated the
"ANTYbiotyk" beer 4
        And the "zaleslaw@middleage.com" connoisseur rated the
"Kasztelan Niepasteryzowane" beer 1
        And I registered as "Rick Sanchez" with the "rick@morty.com"
email and the "birdperson1" password
        And I am logged in as "rick@morty.com" with "birdperson1"
password
        And I rated the "PROpaganda" beer 4
        When I ask for a beer recommendation
        Then the "ANTYbiotyk" beer should be suggested
```

Listing 9. Feature: Recommending best beers for connoisseur

Nevertheless there a few functionalities available for guest users. It provides a demonstrative version of the application for new users and may encourage them to register in the application. Anonymous users are allowed to browse beers catalogue and check details of chosen beer, what is presented on *Listing 10* and *Listing 11*.

```
@beer @api
Feature: Browsing beers
    In order to being aware of current beers catalogue
    As a Community Member
    I want to browse all available beers

    Scenario: Browsing beers
        Given the "King of Hop" beer with 5% ABV has been added
        When I browse the beers catalogue
        Then I should see the "King of Hop" beer
```

Listing 10. Feature: Browsing beers

```
@beer @api
Feature: Showing beer details
    In order to check more information about the beer
    As a Community Member
    I want to see beer details

    Scenario: Checking details of newly created beer
        Given the "King of Hop" beer with 5% ABV has been added
        When I check the "King of Hop" details
        Then I should see that the "King of Hop" beer has 5% ABV, 0 ra-
tes and its average rate is 0

    Scenario: Checking details of beer with rates
        Given the "King of Hop" beer with 5% ABV has been added
        And there is a "Rick Sanchez" connoisseur
        And the "Rick Sanchez" connoisseur rated the "King of Hop" beer
5
        And there is a "Mr Meeseeks" connoisseur
        And the "Mr Meeseeks" connoisseur rated the "King of Hop" beer
4
        When I check the "King of Hop" details
        Then I should see that the "King of Hop" beer has 5% ABV, 2 ra-
tes and its average rate is 4.5
```

Listing 11. Feature: Showing beer details

4.3.3. Project structure

The project started with only one folder which contains minimal Symfony 4 structure, features folder and one feature description inside. For the project developed with the incremental approach in mind, it is enough to start development. Rest of required dependencies could be added based on project needs.

Some rules are common for the whole project. Every class is finalized unless there is a good reason to not do it, such as limitation of the Doctrine library. Each class has been declared with "declare(strict_types=1);" header, which forces classes to return only specified kind of data. No setters were needed for any of classes. These rules are needed in consideration of defensive programming paradigm and object calisthenics.

According to Symfony standards whole application has been placed inside the "src" folder, and it is configured in "config" folder. The "src" folder contains three subfolders according to onion architecture described in chapter 2.1.6.

A core of the application is defined inside "domain" folder. It contains two domains: *Beer* and *Connoisseur*. Both of them do not have an intersection. Also, both of

them are standalone components reusable in other places. The perfect implementation assumes total lack of outside dependencies, but this rule has been violated in the interest of being pragmatic. *Ramsey/Uuid* library is used to validate unique universal identifiers (*UUID*) of objects, and Prooph related dependencies are required for event bus support. Almost all classes in this folder are finalized. An exception has been made to *Beer* and *Connoisseur* classes which are root classes for each domain. Both classes cannot be finalized because they are entities mapped by *Doctrine*. Nevertheless, the state for both classes changes with events.

Domain layer is surrounded by an application. This layer contains the definition of all possible user actions and related handlers according to *CQRS* paradigm (chapter 2.1.7). Handlers from these actions operate on domain models from the inner layer, then they add them to a repository. Changes done on a domain models triggers events which are kept on models at this moment. Application layer contains only repository interfaces because it is an infrastructure concern. Whole application layer was created before the infrastructure layer. As a result, it is infrastructure agnostic. It is not possible to define from which source commands are coming. No matter if they are dispatched in tests, during fixtures loading (a process of loading default data to application) or the result of API call. Result and database state will be the same. It makes setting up test environment smooth and prepares the application for future changes.

The outermost layer is placed in "infrastructure" folder. Here are present all implementation details such as database implementation or framework related logic. It is the place for controllers - classes which are an entry point for the application. Each controller is responsible for only one action per class. Therefore every controller is invokable. *Invokable* is a special *PHP* construct, that allows a class to be used as a function. Whole application has implemented *RESTful API* with consideration to its methods and return codes. Each query to the system is handled with *GET* action and returns code 200 - *HTTP_OK*. Therefore this actions cannot change internal system state. Changes can be done with the *POST*, *PUT* or *DELETE* methods. The action triggered with this methods returns either 201 - *HTTP_CREATED* or 204 - *HTTP_NO_CONTENT*. Another important aspect of this layer is database connection handling. An application

layer does not know anything about persistence, thus no "save" or "flush" method are executed inside of it. However, each state change should be persisted to the database in order to save it between requests. Another constant is that state of object has to be saved in database before domain events will be saved to event store. Otherwise, events can trigger some external logic execution, when database error blocks change of domain model. As a result, the application will be in inconsistent state, which can have only bad consequences. This problem is handled by two listeners. One is attached to command bus and triggers flush method on the database connection after execution of a command. The second one tracks changes on *Doctrine* events and after each successful update will save all produced events to event store. After saved to event store all events will be dispatched to event bus so that other logic can be executed. For instance, two projectors listen to the *ConnoisseurRegistered* domain event. The first one is responsible for projection to *ORM* database and the second will create a projection to a graph database.

4.3.4. Testing application

The application has been created according to *TDD* (chapter 2.2.5) and *BDD* (chapter 2.2.6) methodologies. First, the requirements have been defined and expressed as user stories (*Product Backlog* chapter 4.2.2). With defined requirements, it was possible to start development. The test suite contains acceptance tests on application and infrastructure defined with the StoryBDD approach, and unit tests written according to the SpecBDD.

The application behaviour has been described inside of six feature files described in *Product Backlog* (chapter 4.2.2). Not all of that features tackles domain itself. Several of them are related only to the read model. As a result, only a few of them have an "@application" tags. These features change a state of the application. Therefore they are a result of user interaction with data. These operations are registration, adding and rating beers. Showing beers catalogue or beer details, or recommendation required only fetching data from appropriate storage. On the application level, commands (representation of user actions) are dispatched, what should result in domain events in the event store.

All features, however, have been tested on the infrastructural level. For now, it is the only real bridge between application and users. On these level, all features described in

chapter 4.2.2 have been tested. Tag "@api" on a top of each feature file defines the context in which scenarios would be interpreted. These tests translate steps of scenario into specific *HTTP requests*. Each request will be routed to the specific controller, where it will be interpreted. It can either be a command in case that it should change a state or appropriate query to the database when the read action will be requested.

All the previous tests are acceptance tests. They ensure high-level application correctness. Considering the chapter 2.2.1 and subchapters, they also ensure that client requirements have been met. Nevertheless, as stated in chapter 2.2.2, this kind of tests is a peak of tests pyramid. As stated before, classes require their condition checks with unit tests. Low-level quality checks are cheaper to run and maintain. Unit tests are written in the "Beery" project tests entities and value objects correctness and proper behaviour of services. Entities are checked if it is possible to execute needed actions. What is more, domain entities are checked if they produce proper events when calling their methods. They are also checked to ensure that they protect their invariants. Value objects are checked if it is possible to create them with correct data and if they will throw an exception when the value is invalid. Services are tested if they provide correct results for entering data or if they are interacting with their dependencies.

4.3.5.Static code analyse

Two static code analysis tools have been added to the project. First one, *EasyCodingStandard* helps to keep code clean and tidy. This library will complain each time some leftovers have been left after a refactor, or if one of the clean-code rules is broken. Project configuration ensures that it follows the object calisthenic (chapter 2.1.3), industry standards (*PSR-2*) and several other clean code checks (chapter 2.1.2). Correspondingly, the code has maximal indentation level of 2 for all functions, "else" keyword is not used anywhere, names are not abbreviated, classes are small (maximal length of 200 lines with each line shorter than 130 characters) and so on. An additional benefit of this library usage is that it can fix many common clean code violations. Executing one command in a terminal makes clean code maintenance effortless.

The second library is responsible for asserting that the code will not throw unexpected exceptions during executions. *PHP*, as a scripting language is not compiled, so despite hints from the *IDE* (Integrated Developer Environment), static code analyse is the only way to check code correctness without it execution. Tests, even if essential, don't provide 100% certainty, that code is correct. Some unhappy paths can be skipped or edge-cases ignored. Static code analyse ensures, that when the variable can be either set or have a null value developer secured both possible scenarios in code. In order to provide the highest quality in the example application, static analyse run on the highest possible level of strictness - level 7th.

4.3.6. Version control system

The project has been developed over many days. From the day one Git repository has been initialised, in the interest of keeping track of code changes. Remote code repository was also insurance in case of a problem with a local laptop. Besides remote code repository allows setting the rule, that changes can be added to the master only after manual review. Checking the code after a break may provide a different insight into code. As a result, bugs in the code may be spotted more easily. Moreover, *Git* allows for working on several concepts at the same time, even in single developer team. During development, many times developer can face some mistakes or lines of code that should be fixed. When one is in a middle of work it can be tempting to fix the code together with adding a new feature. Nonetheless, this approach may be problematic during code review. Too big batches of code to merge are hard to review and check. When working with the *Git VCS*, it is easy to branch work, add a separate feature, and come back to primary task. Just created fix will be added to the repository when there will be time to review it.

4.3.7. Continuous integration

Considering project aims and quality importance continuous integration server has been connected to remote *GitHub* repository. This integration allows blocking the possibility of merging new parts of the code if they do not pass tests. It also allows for checks after each merge, to ensure that nothing changed in the main codebase in the meantime. "Beery" continuous integration pipeline includes running all tests (application

tests, infrastructure tests and unit tests), dependencies validation and code quality check. It means, that if the code would violate any rule of clean code, object calisthenic or will not pass the PHPStan check the build will be "red" as well. It will not be possible to add it to the central repository without fixing all spotted problems. All these practices are on purpose of keeping quality over time.

4.3.8. Feature implementation

This section will describe a process of adding a new feature to the platform. As a matter of convenience, a service declaration and tools configuration has been skipped in the following description.

As was stated in previous chapters, BDD with StoryBDD and SpecBDD promotes a top-down approach to development. Implementation of a new functionality should start from the story. A *Listing 6* describes registration of a new connoisseur in the platform. It contains two steps:

- An action - When I register the "Rick Sanchez" connoisseur with the "rick@morty.com" email and the "birdperson1" password
- An assertion - Then I should be able to log in as "rick@morty.com" with "birdperson1" password

Moreover, it is interpreted in a domain of *Connoisseur*, because of "@connoisseur" tag and will be implemented on application and infrastructure level, as a consequence of "@appliation" and "@infrastructure" tags. This feature will change internal state of the application, so at least two layers will be involved in user action interpretation. First of all, the feature has to be implemented on the inner layer - the application layer. Both steps have to be translated to *PHP* code. As a result, a new *Behat* context has to be added - *ConnoisseurContext*, what is presented in *Listing 12*.

```
<?php

declare(strict_types=1);

namespace Tests\Behat\Context\Application;

use Behat\Behat\Context\Context;

final class ConnoisseurContext implements Context
{
}
```

Listing 12. An empty context class

The context class has to follow basic assumptions of defensive programming (chapter 2.1.5) and clean code (chapter 2.1.2). It has to have a strict type declaration, to provide type safety and is declared as final, as it should not be extended. Now, steps mapping should be added to the class. *Behat* allows for autogenerating steps definitions. The result of step mapping generation can be seen in *Listing 13.*

```
final class ConnoisseurContext implements Context
{
    /**
     * @When I register the :name connoisseur with the :email email and
the :password password
     */
    public function
iRegisterTheConnoisseurWithTheEmailAndTheEmail(string $name, string
$email, string $password): void
    {
        throw new PendingException();
    }

    /**
     * @Then I should be able to log in as :email with :password pas-
sword
     */
    public function theConnoisseurShouldBeCreated(string $email, string
$password): void
    {        throw new PendingException();

    }
}
```

Listing 13. Context with generated steps

Now execution of *Behat* scenario will show information presented in *Fig. 5.*

Figure 5. First Behat run

Next step is to implement enough code, to make a `When` step pass (green). This step is describing an intention of the application *User* to register a new connoisseur. Therefore a new *Command* named *RegisterConnoisseur* should be created. This command will be a *VO*, dispatched over *Command Bus*, and will deliver encapsulated intention to *Command Handler* in some other part of the system. However, this *VO* has to be created before it can be dispatched. Following a *SpecBDD* paradigm, before the class will be created a test, or to be more precise a specification should be defined first. *PHPSpec* makes it effortless to bootstrap a new specification class. With a defined set of templates, when the library will be asked to generate a class description it will produce a *Listing 14*

```
<?php

declare(strict_types=1);

namespace spec\App\Application\Command;

use PhpSpec\ObjectBehavior;

final class RegisterConnoisseurSpec extends ObjectBehavior
{
}
```

Listing 14. Empty PHPSpec class

Of course, this class has to follow same rules as rest of code. Hence, the class is declared as final and has a strict type declaration. This *VO* will represent a register connoisseur intention and will be a command. Both features of the class should be described inside of this class, what is presented in *Listing 15.*

```
final class RegisterConnoisseurSpec extends ObjectBehavior
{
    function it_represents_register_connoisseur_intention(): void
    {
    }

    function it_is_command(): void
    {
    }
}
```

Listing 15. RegisterConnoisseur class with named examples

Now, executing of PHPSpec library for will showed information about pending examples what can be seen in *Fig. 6*.

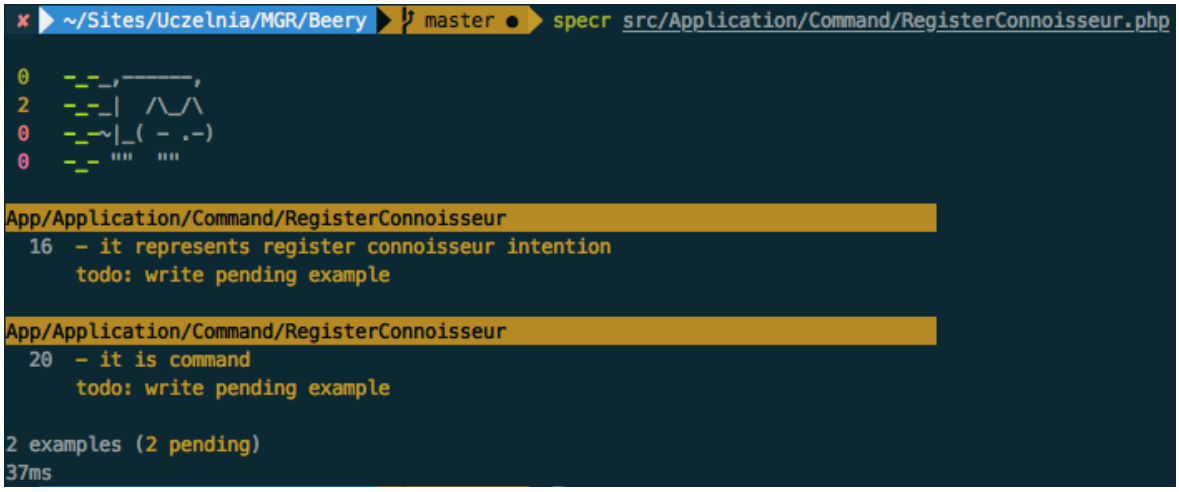


Figure 6. First PHPSpec run

As a next step, a test should be filled with expected behaviour. The implemented example will look like on *Listing 16*.

```
final class RegisterConnoisseurSpec extends ObjectBehavior
{
    function it_represents_register_connoisseur_intention(): void
    {
        $this->beConstructedThrough('create', [
            new Id('e8a68535-3e17-468f-acc3-8a3e0fa04a59'),
            new Name('Harvey Specter'),
            new Email('harvey@specter.pl'),
            new Password('$2a$04$N2x1MTIgy8fth66TdWZ1NeHIjJIrK7Ns09I-
9xk1PDRn8IqkQSckua'),
        ]);

        $this->id()->shouldBeLike(new Id('e8a68535-3e17-468f-acc3-8a3e-
0fa04a59'));
        $this->name()->shouldBeLike(new Name('Harvey Specter'));
        $this->email()->shouldBeLike(new Email('harvey@specter.pl'));
        $this->password()->shouldBeLike(new Password('$2a$04$N2x1MTIgy-
8fth66TdWZ1NeHIjJIrK7Ns09I9xk1PDRn8IqkQSckua'));
    }

    function it_is_command(): void
    {
        $this->shouldHaveType(Command::class);
    }
}
```

Listing 16. Implemented RegisterConnoisseurSpec

If the library is executed, the test will fail of course. This class requires a few more *VO* to be completed. Thus, same implementation cycle should be executed:

- Generate the class description
- Implementation of it

The process is recursive, so each time a new object is needed a new description is defined. Then it has to be fulfilled, and one by one the tests should start passing. When all of required *VO* are described and implemented, the test can be safely executed. The library will offer to generate a class scaffolding. As a result of *beConstructedThrough* method usage, the library will understand the usage of named construct pattern and make a regular constructor private. It will also generate a naive implementation of first id method, as it is a first one tested. Rerunning the same library will end up with a failure presented in *Fig. 7*.

Figure 7. Second PHPSpec run

The newly generated class is presented in *Listing 17*.

```
<?php

declare(strict_types=1);

namespace App\Application\Command;

final class RegisterConnoisseur
{
    private function __construct()
    {
    }

    public static function create($argument1, $argument2, $argument3,
$argument4)
    {
        $registerConnoisseur = new RegisterConnoisseur();

        // TODO: write logic here

        return $registerConnoisseur;
    }

    public function id()
    {
        // TODO: write logic here
    }
}
```

Listing 17. Empty RegisterConnoisseur class

Now, the class has to be implemented. The result is shown in *Listing 18.*

```
<?php

declare(strict_types=1);

namespace App\Application\Command;

use App\Domain\Connoisseur\Model\Email;
use App\Domain\Connoisseur\Model\Id;
use App\Domain\Connoisseur\Model\Name;
use App\Domain\Connoisseur\Model\Password;
use Prooph\Common\Messaging\Command;
use Prooph\Common\Messaging\PayloadTrait;

final class RegisterConnoisseur extends Command
{
    use PayloadTrait;

    private function __construct(array $payload)
    {
        $this->init();
        $this->setPayload($payload);
    }

    public static function create(Id $id, Name $name, Email $email,
Password $password)
    {
        return new self([
            'id' => $id->value(),
            'name' => $name->value(),
            'email' => $email->value(),
            'password' => $password->value(),
        ]);
    }

    public function id(): Id
    {
        return new Id($this->payload()['id']);
    }

    public function name(): Name
    {
        return new Name($this->payload()['name']);
    }

    public function email(): Email
    {
        return new Email($this->payload()['email']);
    }

    public function password(): password
    {
        return new Password($this->payload()['password']);
    }
}
```

Listing 18. Implemented RegisterConnoisseur class

With the generated class a tests can be executed once again, with a *fpretty* flag to present expected behaviour of the class. The result is present in *Fig. 8.*

Figure 8. Third PHPSpec run

Once the class is implemented, it is investigated if its implementation meets expectations. If so, the next test should be made green. It is time move back to Behat test and finish implementing user action. With created *RegisterConnoisseur* class the When step can be implemented as presented in *Listing 19*.

```
/**
 * @When I register the :name connoisseur with the :email email and the
:password password
 */
public function iRegisterTheConnoisseurWithTheEmailAndTheEmail(string
$name, string $email, string $password): void
{
    $this->commandBus->dispatch(RegisterConnoisseur::create(
        new Id($this->uuidGenerator->generate()),
        new Name($name),
        new Email($email),
        new Password(password_hash($password, PASSWORD_BCRYPT))
    ));
}
```

Listing 19. Definition of first Behat step

Executing a *Behat* test suite will also result in failure, what is presented in *Fig. 9.*

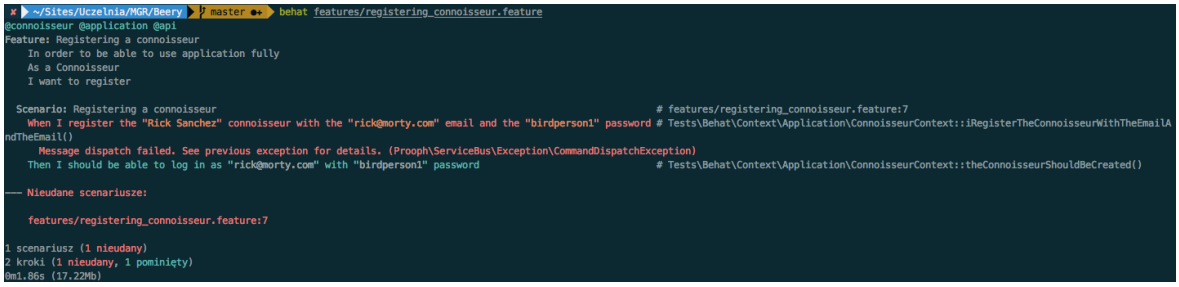


Figure 9. Second Behat run

In order to make this test pass, an associated *Command Handler* is required, which will translate a *RegisterConnoisseur* in some action on the application level. Therefore, the same cycle as explained previously has to be repeated for *RegisterConnoisseurHanlder*, some *Connoisseur* implementation in the domain layer, and for the *Event* that will be triggered. Once all of these classes are created, the next step will be fulfilled. The result of executing the *Behat* library is presented in *Fig. 10*.

Figure 10. Third Behat run

As a next step, the whole process is repeated for the next Behat step. When the whole scenario works, another round of code refactor takes place. Implementation should be checked and adjusted to project requirements. Most of the problems can be tracked with the help of the *PHPStan* and *EasyCodingStandard* libraries, which fail the build if some refactor leftovers would be found, some return types are missing, or something is wrongly compared. If this step is skipped, then the *CI server* will not allow merging a newly created feature if the code will not fulfil defined rule set. When the expectations are met, new functionality can be merged into the master branch, and the process is repeated on the infrastructure level. When it is done, then the new feature can be implemented. This loop is continued until all required feature will be developed.

4.3.9. Recommendation algorithm

The main functionality of this application is a recommendation system. As was described in chapter 3, the recommendation algorithm can be implemented in several ways, mostly with the usage of *Content-based filtering* or *Collaborative filtering*. What is more, this algorithm can be added to the application also in many ways. It can be implemented in *Python* what would require the whole application to be written in this

language or some communication layer like *Message Queue* or external *API* calls. Another possible approach is to used software crafted for this purpose. One of the possibilities is to use a *Graph Database*. The *Neo4j* implementation of this paradigm has been used in this thesis. This database has an official client implementation in *PHP* what makes its integration effortless. With the lean and incremental way of development, the *Collaborative Filtering* approach has been developed. It requires the smallest amount of details about the single entry. On the other hand, this algorithm needs the significant amount of starting rates. It is a typical problem of cold start for this type of filtering. It is an essential factor and tradeoff that should be considered during project development. Nevertheless, it is a satisfying solution regarding current application stage. The good quality of code allows for almost effortless changes in the algorithm. *CQRS* paradigm (chapter 2.1.8) used in the application is entitled to the uncomplicated transfer of only required information to the *Graph Database*. *SQL* database manages beers catalogue, register customers and theirs beer rates while a graph database contains the only beers and connoisseurs identifier together with a relation between them - rate. Each time a new rate is added by the connoisseur, the domain layer produce the *BeerRated* event which is projected to both databases. Therefore, the database becomes an implementation detail, and it is uncomplicated to adjust information persisted to them. The first iteration of recommendation engine implementation was trivial. As presented on *Listing 20* the algorithm was recommending beers which have never been tried by the user. Also, there has to be at least one other connoisseur who rated recommended beer by at least 4. What is more, both of the user and other connoisseur rated some other beer by at least 4.

```
MATCH (:Connoisseur{email: {email}})-[r1:RATED]->(rated:Beer)<-[r2:RA-
TED]-()-[r3:RATED]->(unrated:Beer)
WHERE r1.rate >= 4
AND r2.rate >= 4
AND r3.rate >= 4
AND NOT ((:Connoisseur{email: {email}})-[:RATED]->(unrated))
RETURN unrated
```

Listing 20. Trival recommendation algorithm implementation

When the recommendation algorithm was working, and implementation has been tested, it could be improved to provide more sophisticated recommendations. One of the possible improvements was to use a cosine similarity between nodes to determine the

closest relationships between them and then recommend beers only on the option on the closest connoisseurs. Firstly, a new relation between nodes has to be added - similarity. As was described in chapter 3.4 one of alternative is to use cosine similarity as similarity measurement. According to "Item-Based Collaborative Filtering Recommendation Algorithms" the cosine similarity "is measured by computing the cosine of the angle between these two vectors."[34]. In *Beery*, this calculation is done on database level. Cosine similarity calculation in *Cypher* can be expressed as presented on *Listing 21.*

```
MATCH (c1:Connoisseur)-[x:RATED]->(b:Beer)<-[y:RATED]-(c2:Connoisseur)
WITH SUM(x.rate * y.rate) AS xyDotProduct,
SQRT(REDUCE(xDot = 0.0, a IN COLLECT(x.rate) | xDot + a^2)) as xLength,
SQRT(REDUCE(yDot = 0.0, b IN COLLECT(y.rate) | yDot + b^2)) as yLength,
c1, c2
MERGE (c1)-[s:SIMILARITY]-(c2)
SET s.similarity = xyDotProduct / (xLength * yLength)
```

Listing 21. Cosine similarity expressed in Cypher query language[48]

With the calculated similarity, the recommendation algorithm can be improved. A more advanced version of it is presented in *Listing 22.*

```
MATCH (unrated:Beer)<-[r:RATED]-(c:Connoisseur)-[s:SIMILARITY]-(sub-
ject:Connoisseur{email: {email}})
WHERE NOT ((subject)-[:RATED]->(unrated))
WITH unrated, s.similarity AS similarity, r.rate AS rate
ORDER BY unrated.name, similarity DESC
WITH unrated as beer, COLLECT(rate)[0..3] AS rates
WITH beer, REDUCE(s = 0, i IN rates | s + i) * 1.0 / LENGTH(rates) AS
recommendation
WHERE recommendation >= 3
RETURN beer
ORDER BY recommendation DESC
```

Listing 22. More sophisticated recommendation algorithm[48]

New implementation of recommendation engine takes into account similarity between connoisseurs. Once it is calculated, it gets all unrated beers of connoisseurs similar to the user, next it calculates an average rate from the three closest ones, and recommend all of them, which average rate is more significant than 3.

5. Result

The project can be considered as succeeded, because the given backlog requirements has been fulfilled. What is more, the definition of "Done" (chapter 4.1) have been fulfilled as well. The project allows to register and login to the application. Browsing beers catalogue and checking beer details are available for every API user. Additionally, logged in users can add new beers to the catalogue, rate existing ones and receive a beer recommendation based on these rates.

5.1. Project quality

The project quality has been ensured with static code analyse. Defined rule set did not allow for the violation of the clean code, object calisthenics or *PSR-1* and *PSR-2* principles. The result of coding standard check is presented in *Fig. 11*.

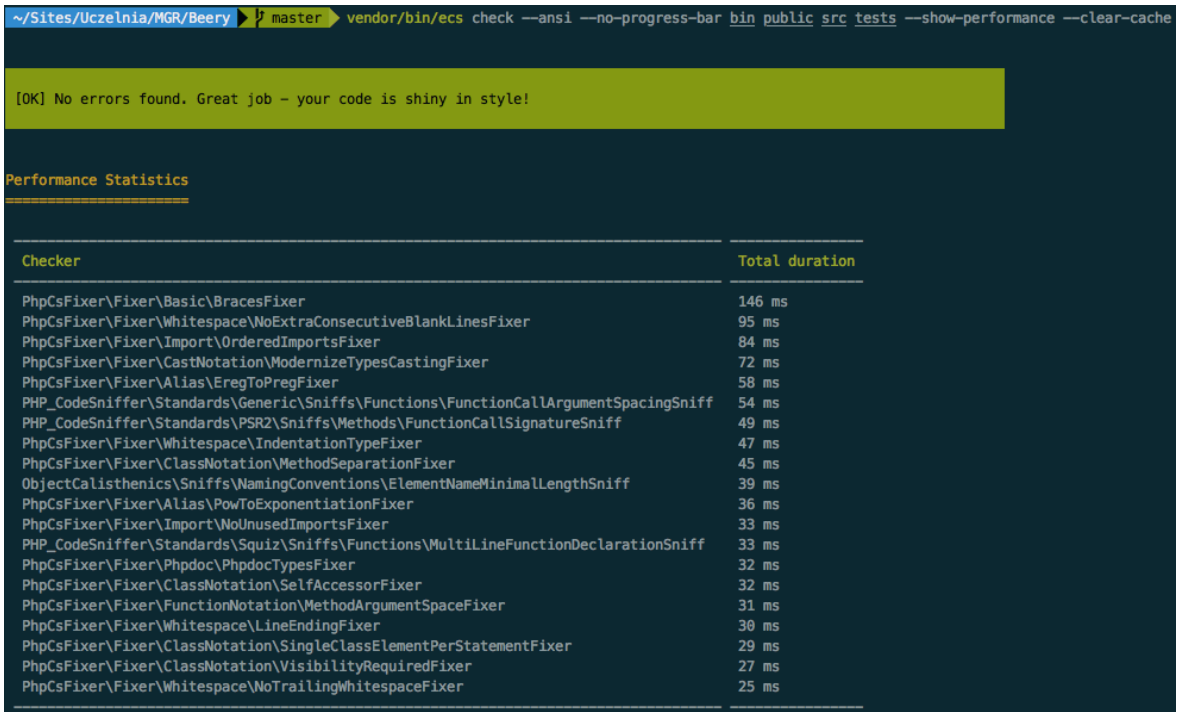


Figure 11. Results of checking coding standards

Each of 148 checkers were run each time *EasyCodingStandard* library was executed. Another part of code quality assurance was done with the *PHPStan* library. This library tested if the code is semantically correct, all null pointers are handled, or expected method exists on the returned object according to the return type. As presented on *Fig. 12* all checks have passed.

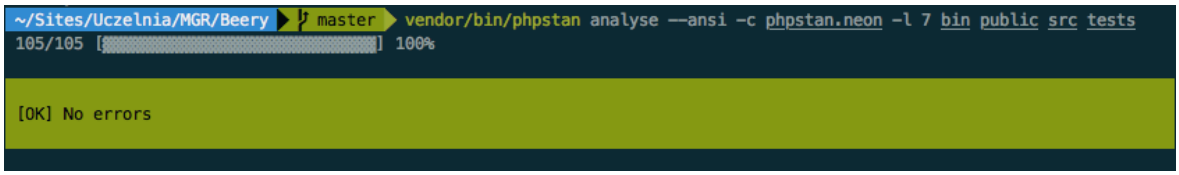


Figure 12. Results of static code analyse

The project source code consists of 73 files from 27 folders. It is built from 2695 lines of code, where about 30% is strictly related to project logic. Rest of the code is related to white spaces, imports, static type declaration, namespaces, brackets line and so on. Classes are small and readable. The largest class contained only 18 lines of logical code, where average size was 1/3 of it what is shown in *Fig. 13*.

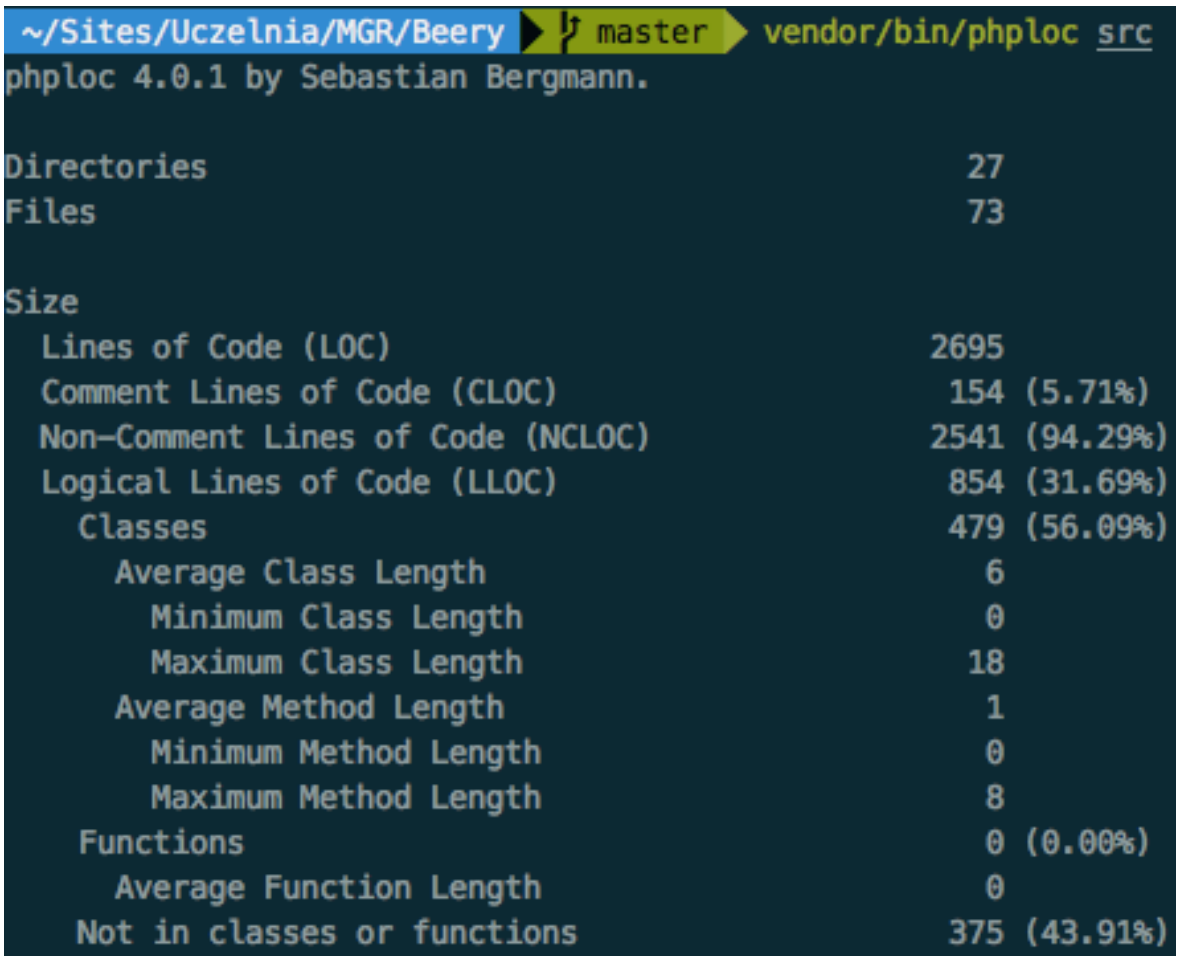


Figure 13. Results of checking project size

According to *Fig. 13* and *Fig. 14* in 73 files, 64 classes where declared, none of them was abstract-type, eight interfaces and one trait. Consequently, most of the project is build upon concrete implementation of logic without leaking abstraction to other classes. Thanks to DI component of Symfony framework *Commands*, *Events* and related handlers can be declared just as a final class without interface requirement. Trait provides the useful

logic of translation of *Command* or *Event* name to the related method in the class with "handle" prefix. For example, if the class has logic related to *ConnoisseurRegistered* event, then it has to have *handleConnoisseurRegistered* method.

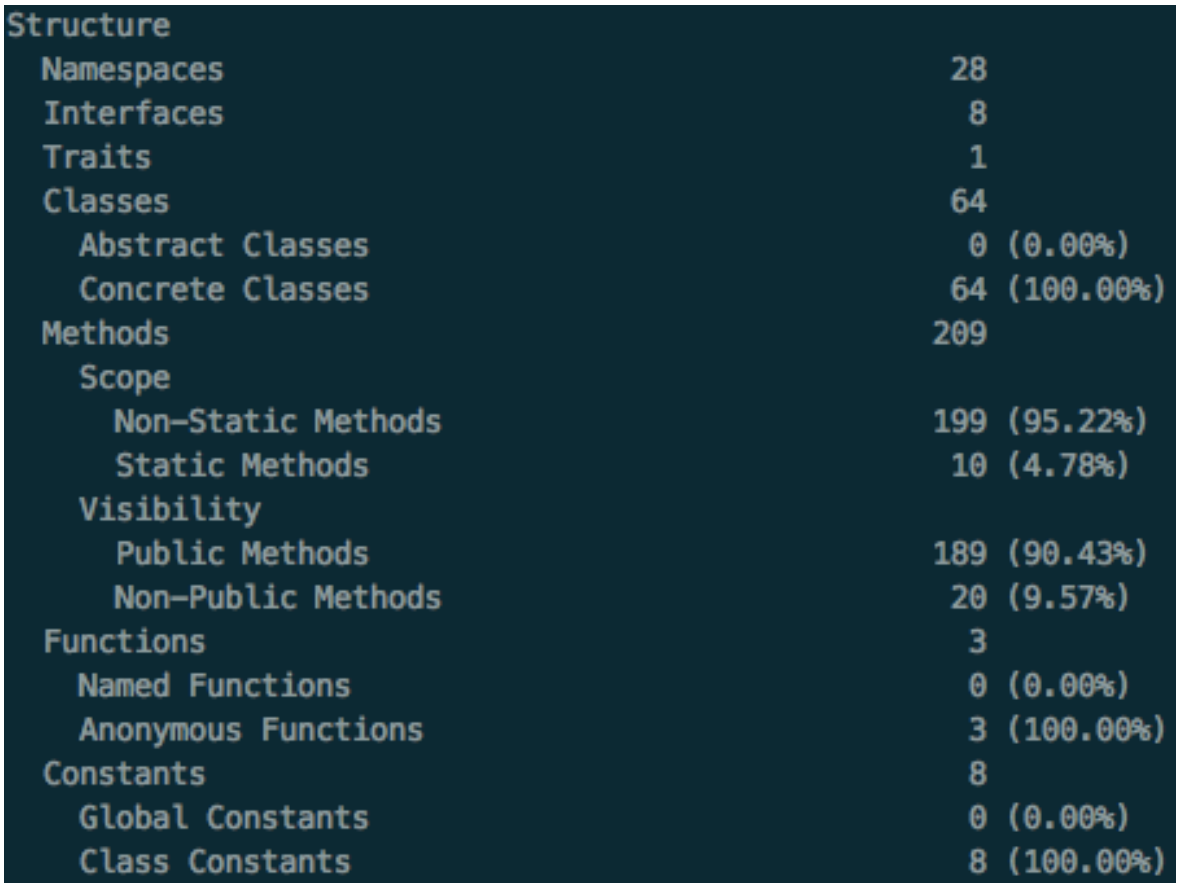


Figure 14. Results of checking methods size

One of the widespread code complexity measurements is *Cyclomatic Complexity*. As it was described in chapter 2.1.3, *CC* defines the amount of all possible paths of method execution. The higher the *CC* value is, the more the complexity increases. Thus, objects are more complicated, and more tests are required to cover their behaviour. *Table 2* presents *CC* thresholds interpreted according to the "Software Technology Reference Guide: A Prototype"[45].

Cyclomatic Complexity	Risk Evaluation
1-10	A simple program, without much risk.
11-20	More complex, moderate risk.
21-50	Complex, high risk program.
Greater than 50	Untestable program (very high risk).

Table 2. Reference table of cyclomatic complexity values[45]

As presented on *Fig. 15*, the project code result in maximum class complexity equal to 9, where the average class had *CC* value around 2 and maximum method complexity equal to 5, where the average method *CC* was around 1.4:

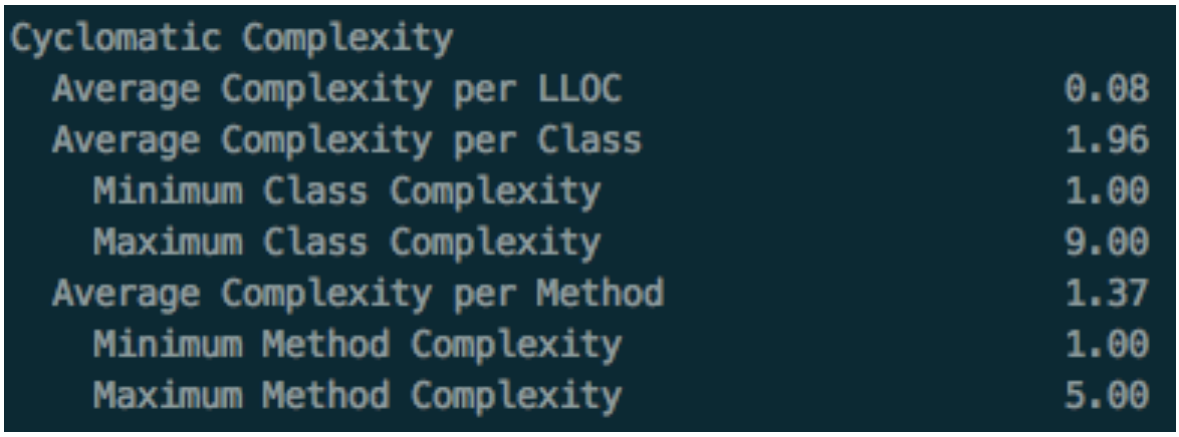


Figure 15. Results of checking code complexity

It is important to mention that *Cyclomatic Complexity* was never taken into account during project development. It is just one of a metric of code quality. It is worth to mention its value, as it is a result of clean code paradigm and other rules from chapter 2.

Another significant factor of the project is that similar rules have been forced on test part of the project. It is natural that not all of the rules were applied in test context and some of them are redundant. Nevertheless, the testing code should also be clean and keep high quality. According to *Fig. 16*, the project, code tests accounted for almost half of project size. Low quality of test code can lead to unexpected fails, missing parts of testing code or problems with running the tests.

```
~/Sites/Uczelnia/MGR/Beery  master  vendor/bin/phploc tests spec
phploc 4.0.1 by Sebastian Bergmann.

Directories                                    18
Files                                          57

Size
  Lines of Code (LOC)                        2386
  Comment Lines of Code (CLOC)                166 (6.96%)
  Non-Comment Lines of Code (NCLOC)          2220 (93.04%)
  Logical Lines of Code (LLOC)                754 (31.60%)
    Classes                                   367 (48.67%)
      Average Class Length                      6
        Minimum Class Length                    2
        Maximum Class Length                   14
      Average Method Length                     1
        Minimum Method Length                   1
        Maximum Method Length                   6
    Functions                                   0 (0.00%)
      Average Function Length                   0
    Not in classes or functions               387 (51.33%)

Cyclomatic Complexity
  Average Complexity per LLOC               0.02
  Average Complexity per Class              1.28
    Minimum Class Complexity                1.00
    Maximum Class Complexity                5.00
  Average Complexity per Method             1.09
    Minimum Method Complexity               1.00
    Maximum Method Complexity               4.00
```

Figure 16. Results of checking tests size and complexity

The assumption of following architectural code best practices described in chapter 2.1 along with the usage of tools described in chapter 4.1 had a good influence in code. As a result, the code has high quality, is not overcomplicated and have low *Cyclomatic Complexity*. Such code is understandable and easy to maintain. These features can noticeably reduce the danger of project failure and reduce the cost of development of long-term projects.

5.2. Tests quality

According to chapter 4.3.2 and 4.3.4 application has been tested with the *Behat* framework for acceptance testing and *PHPSpec* for unit testing. All tests were executed on the local environment, on MacBook Air 2014, with 4GB of RAM, 1,4 GHz Intel Core i5 CPU and SSD hardware.

Product backlog from chapter 4.3.2 contained 12 scenarios. All these scenarios were used in *Behat* to define a base for acceptance tests. Each line of scenario was translated into part of a code, which interprets user intention. Therefore, application level tests were dispatching *Commands* over *CommandBus* when infrastructure level tests were sending an *HTTP* request. Only a few scenarios were interpreted on two levels. It was required in order to ensure application-level independence from infrastructure, for write side of actions only. Thus, six features described in 12 scenarios are interpreted 18 times, where 6 of them are checked twice. As stated in chapter 2.2.4, one of the most significant factors of high-grade tests is speed. In the project, all *Behat* scenarios execution takes around 30 seconds, as was presented in *Fig. 17*. As was stated in chapter 2.2.1 these tests are checking if communication over all layers is proper and if the application, as a whole, works as expected.

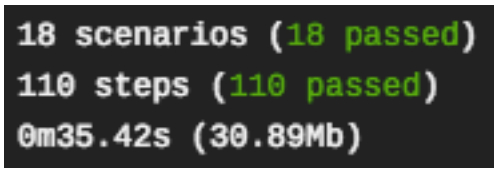


Figure 17. Results and execution time of Behat tests

Unit tests cover all of the features over domain and application level and some from the infrastructure level. These tests are focused on ensuring that each of class and its methods is behaving as planned. They are covering 27 of 64 classes available in code, which is around 42% of the code. However, they covered 100% of classes from application and domain level (accordingly 6 and 14 classes). The total execution time of all unit tests is around 0.5 second (*Fig. 18*).

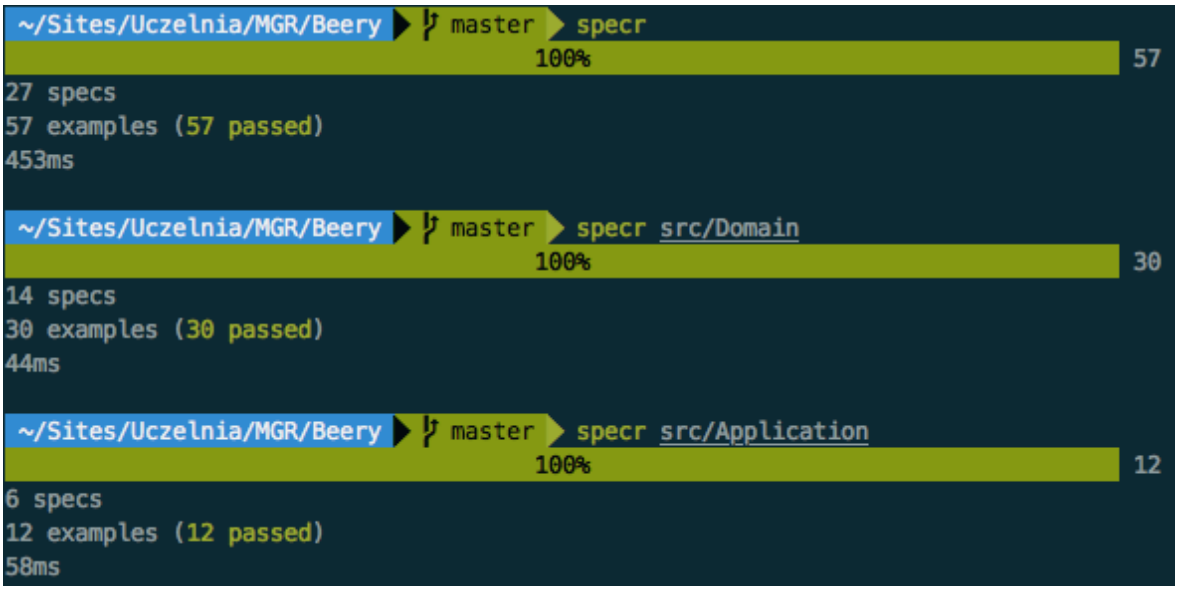


Figure 18. Results and execution time of unit tests

Following the chapter 2.2.2, the number of unit tests is noticeably larger than the end-to-end tests (*acceptance tests*) - 18 to 57. They are also much faster 35s to ~0,5s.

Similar to the code, tests can have their quality as well. When tests are written, not only the quality of testing code is critical, but the quality of tests itself. Poor quality tests do not measure code correctness or mismeasure it. For example, a missing assertion at the end of the test execution results in poor tests quality. Test quality can be measured as the percentage of code executed, compared to tested codebase (*code coverage*) or an amount of caught exception from intentionally broken code base (*mutation testing*). Unfortunately, *Behat* does not provide a toolkit to measure such metrics on a code base. Thus, acceptance tests correctness is hard to measure, and only code review, live demo and empirical testing can provide the final quality confirmation. Nevertheless, these metrics can be collected over application and domain layer of the project, due to PHPSpec usage.

PHPSpec classes covered 100% of classes available in domain and application layers. These tests cover about 94% of all logic lines of code. This value is not perfect, but high enough to recognise tests quality as sufficient. As presented on *Fig. 19,* two value objects do not have enough tests, which could be fixed in the later phase of a project or during some code refactoring session.

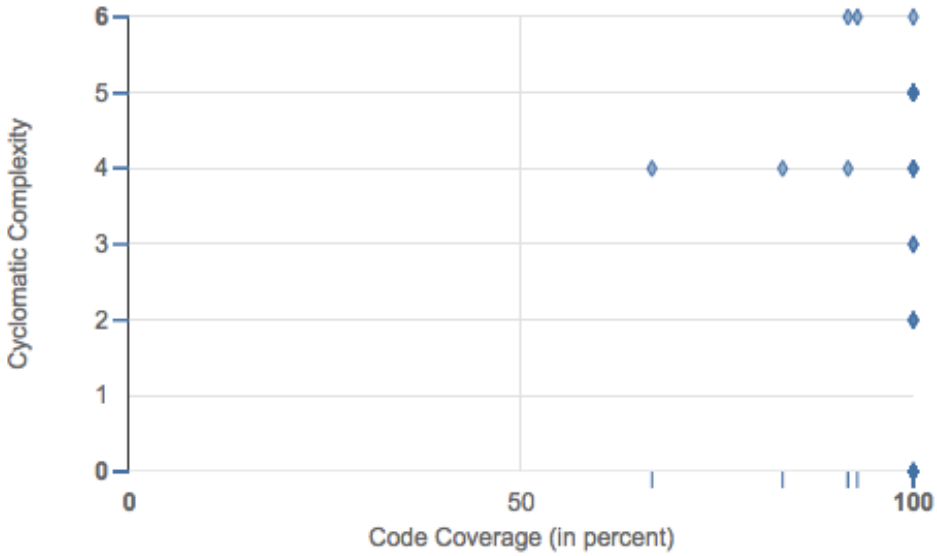
Insufficient Coverage

Class	Coverage
App\Domain\Connoisseur\Model\Id	66%
App\Domain\Beer\Model\Id	83%

Figure 19. Insufficient coverage over some classes

Nevertheless, the most complex classes (ones with highest *Cyclomatic Complexity*) have been tested in 100% as shown on *Plot 2*.

Complexity



Project Risks

Class	CRAP
App\Domain\Connoisseur\Model\Id	4
App\Domain\Beer\Model\Id	4

Plot 2. Code coverage and cyclomatic complexity correlation

However, test coverage can be a too simple metric of code correctness. For instance, single test can report 100% of code coverage even if there were two conditions inside on "if" statement, so two paths should be checked. Better metric results can be

achieved with mutation testing. The "Class Mutation: Mutation Testing for Object-Oriented Programs" describes the mutation testing as "The mutation method is a promising measurement method for test data adequacy"[46]. This method aims to make a typical programming mistake in code which should result in testing suite fails. For example negation of "if" statement can be treated as code mutation. Mutation testing comes with three useful measurements according to "Infection" library description:

- Mutation Score Indicator (MSI) - which calculates the percentage of all defeated mutants. It means that mutant has been covered by tests and has been discovered.
- Mutation Code Coverage (MCC) - percentage amount of covered mutants. It indicates how many generated mutants have been covered with tests.
- Covered Code Mutation Score Indicator (CCMSI) - Real measurement of tests quality. It is a percentage value of discovered mutants only among mutants covered by tests.

According to *Fig. 20*, all written unit tests are reported to cover 87 mutants from 99 available in code. As a result, CCMSI is equal to 90%, only 2% of mutants were not detected, and total MSI for the whole project is equal to 88%.

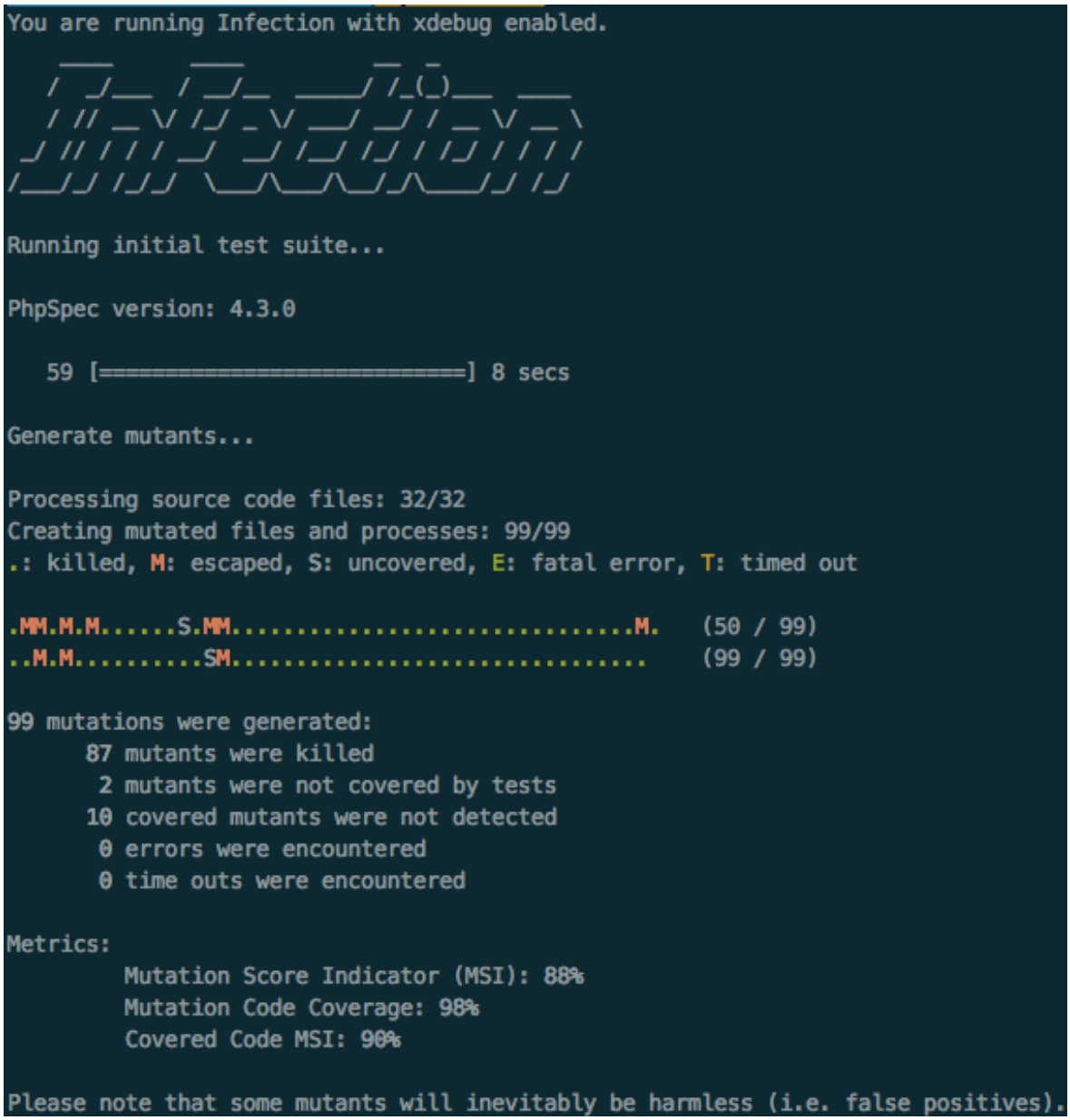


Figure 20. Results of mutation testing

All previously mentioned metrics can be summarised in several conclusions. In the sample project tests are fast, they have high quality and covers most of the logic. Project development process described in chapter 4.2, which aimed to follow best practices from chapter 2.2, had a good influence on the project itself and its quality.

5.3. Repository quality

Another important factor of the code quality assurance had connection to *VCS* repository usage and it remote source - *GitHub*. The code was incrementally written, which

allowed focussing on small, atomic parts of the project at once. Each time, before the code was added to the master branch, it had to pass all tests examination and manual review.

5.3.1.Git

During the development, 79 pull requests have been opened to the main repository, which introduced almost 220 code contributions (*Fig. 21*). In the meantime, five product releases have been done, where each of them introduces new functionality or fix a newly discovered problem.

Figure 21. Results of checking tests size and complexity

Network plot presented in *Fig. 22* shows that even in single development project parallel changes can happen in the codebase.

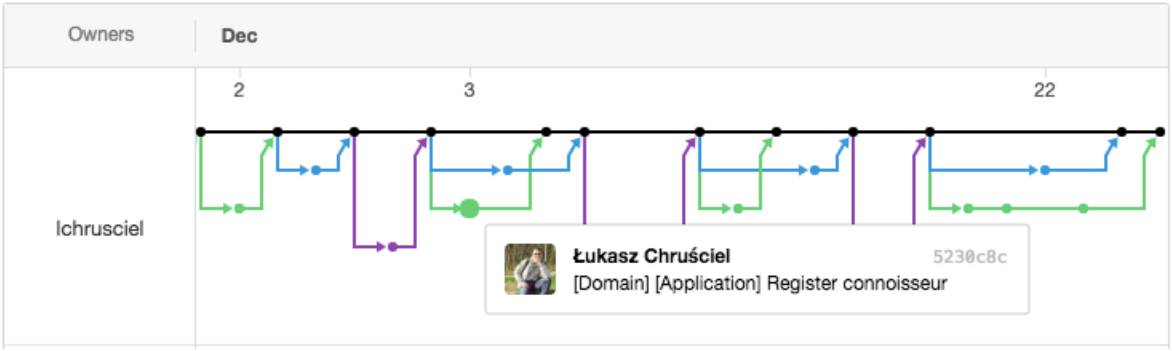


Figure 22. Part of branching network in remote repository

5.3.2.Continuous integration

All of the previous tools would be significantly less effective if they would not be checked regularly. It was essential to connect code development with continuous integration server which will ensure, that proposed changes meet the conditions described in chapter 4.2. Therefore, *CI* server not only ensures that tests passes or code correctness (static analyse), but also most of the clean code assumptions. During project development, the code was tested 299 times, from which 69 builds failed. Most of the fails were due to clean code violations, as presented in *Fig. 23.*

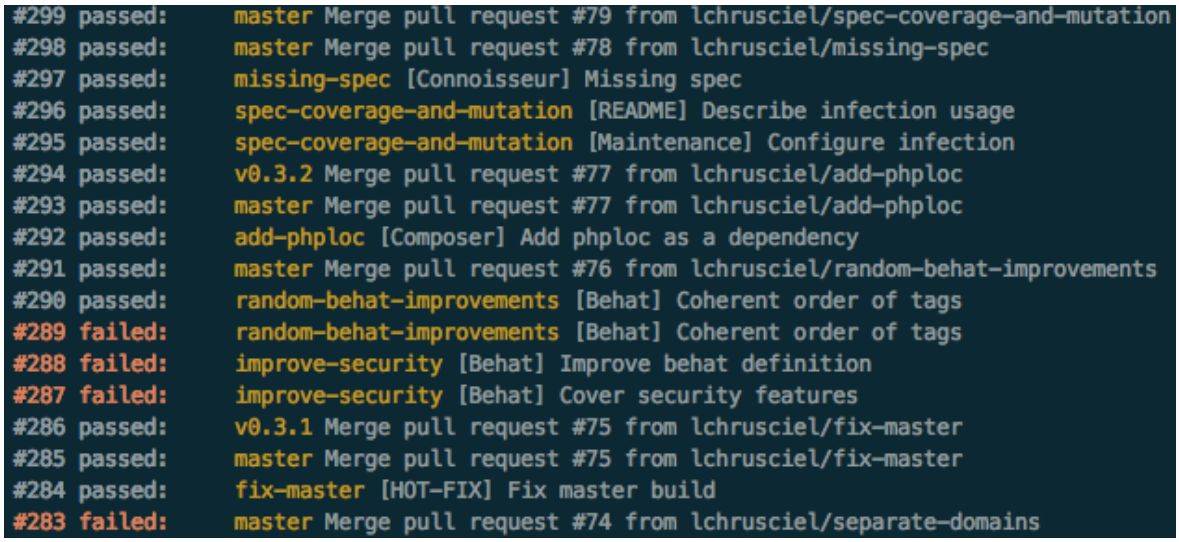


Figure 23. Partial list of all continuous integration builds

6. Summary

The primary aim of this master thesis was to present and study modest methods of software development based o recommendation platform. The work presented in it shows that these methods have a good influence on code quality and can be beneficial in the long term. Following a defined set of rules makes it easier to change team members or redefine some parts of the application. Presented methods require some changes to developers attitude and some skills, but many tools can make this much more accessible. Thus, following this practices can make average industry quality much higher, what potentially can provide better software to the customers.

Work of engineers should always be based on data. Doing experiments or resolving new problems requires measurements to be done. It is a crucial part of engineering, to make a decision based on data. Alternatively, an evaluation is need to check if project meets initial requirements. Software quality can be assessed as well. However, this quality and good analysis results should not be a goal for itself. It is a consequence of following best practice, being prepared for a mistake or changes in requirements. Each project is an investment that usually has to pay off in years, and the only constant in software development industry is the change. Change of requirements, substitution of platform or user needs. Every project is different, and set of rules that is required during development should be adjusted to clients needs and the ability of developers. Developers should be prepared to that. Thus, choosing a right tools set is crucial. Software engineer have to keep in mind, that every time they lower the software quality they are making a debt, that will have to be paid in future. Modest methods of software development aim to reduce the amount of this debts. Nonetheless, when the debt is made these methods reduce the amount of work required to repair them.

Despite the project size, unit tests, defined rule set among the whole project and continuous integration pipeline should be a standard, not an exception in software engineering. Nevertheless, before deciding on some of methods or skipping them one question should be asked: will this decision put the whole project into danger of failure? Only negative answer should be accepted.

6.1. Conclusions

The aim of the thesis was to develop the recommendation application in accordance with modern methods of software development. The recommendation system follows the *Collaborative filtering* approach and uses a cosine similarity to calculate correlation between connoisseurs. The code follows the clean code, *Object Calisthenics*, *SOLID* and defensive programming paradigms. The application was designed according to Behaviour Driven Development technic. Therefore, as a side effect, a Test-Driven Development paradigm has been followed as well. This application is entirely covered with end to end, acceptance tests as well as unit tests. Continuous integration server has been configured to ensure that code quality expectations are met, and the tests are passing. The theoretical background of these methods has been described and proved as well. The process of development presented in this thesis emphasises pros and cons of these methods. The communication with the application is done with RESTful API, so it is ready to handle different kind of web or mobile based application. As a result, the app is very well designed, it achieves all expected requirements, and it is prepared for future changes.

List of figures

List of listings

List of tables

List of plots

Bibliography

1. Ben-Kiki, O., Evans, C. and Ingerson, B., 2005. Yaml ain't markup language (yaml™) version 1.1. *yaml. org, Tech. Rep*, p.23.
2. Bray, T., Paoli, J., Sperberg-McQueen, C.M., Maler, E. and Yergeau, F., 1997. Extensible markup language (XML). *World Wide Web Journal*, *2*(4), pp.27-66.
3. Cunningham, W., 1993. The WyCash portfolio management system. *ACM SIGPLAN OOPS Messenger*, *4*(2), pp.29-30.
4. Kruchten, P., Nord, R.L. and Ozkaya, I., 2012. Technical debt: From metaphor to theory and practice. *Ieee software*, *29*(6), pp.18-21.
5. KNOERNSCHILD, Kirk. Java design: objects, UML, and process. Addison-Wesley Professional, 2002. Composite Reuse Principle (CRP)
6. MARTIN, Robert C. Clean code: a handbook of agile software craftsmanship. Pearson Education, 2009
7. McCabe, T.J., 1976. A complexity measure. IEEE Transactions on software Engineering, (4), pp.308-320.
8. BAY, J. The ThoughtWorks Anthology: Essays on Software Technology and Innovation, Chapter Chapter 6: Object calisthenics. 2008.
9. MARTIN, Robert C. Agile software development: principles, patterns, and practices. Prentice Hall, 2002.
10. NOBACK, Matthias Principles of package design. Leanpub, 2015.
11. Fowler, M., 2002. Patterns of enterprise application architecture. Addison-Wesley Longman Publishing Co., Inc..
12. http://ocramius.github.io/extremely-defensive-php/#, [accessed: February 2018]
13. https://medium.com/web-engineering-vox/the-art-of-defensive-programming-6789a9743ed4, [accessed: February 2018]
14. Keith, M. and Schnicariol, M., 2009. Object-relational mapping. In Pro JPA 2 (pp. 69-106). Apress.
15. Tamai, T. and Torimitsu, Y., 1992, November. Software lifetime and its evolution process over generations. In Software Maintenance, 1992. Proceerdings., Conference on (pp. 63-69). IEEE.
16. https://www.php-fig.org/psr/, [accessed: February 2018]

17.PALERMO, Jeffrey. The onion architecture: Part 1. Retrieved December, 2008, 1: 2014.

18.MEYER, Bertrand Object-oriented software construction. Vol. 2. New York: Prentice hall, 1988.

19.Fowler, M., 2005. Command Query Separation.

20.Young, G., 2010. Cqrs and event sourcing. feb. 2010. URl: http://codebetter. com/gregyoung/2010/02/13/cqrs-and-event-sourcing.

21.Fowler, M., 2012. Test pyramid.

22.BECK, Kent. Test-driven development: by example. Addison-Wesley Professional, 2003.

23.http://butunclebob.com/ArticleS.UncleBob.VehementMediocrity, [accessed: February 2018]

24.SMART, John Ferguson. BDD in Action. Manning,, 2015.

25.https://cucumber.io/docs/reference, [accessed: February 2018]

26.https://github.com/Sylius/Sylius/blob/master/features/account/registering.feature, [accessed: February 2018]

27.Sutherland, J. and Schwaber, K., 2013. The scrum guide. The definitive guide to scrum: The rules of the game. Scrum. org, 268.

28.https://trends.google.com/trends/explore?cat=5&date=all&q=%2Fm%2F077dn,%2Fm%2F03nsxd,GraphQL, [accessed: February 2018]

29.Wichmann, B.A., Canning, A.A., Clutterbuck, D.L., Winsborrow, L.A., Ward, N.J. and Marsh, D.W.R., 1995. Industrial perspective on static analysis. Software Engineering Journal, 10(2), pp.69-75.

30.http://www.dictionary.com/browse/collective-intelligence, [accessed: February 2018]

31.https://www.cirrusinsight.com/blog/much-data-google-store, [accessed: February 2018]

32.https://en.wikipedia.org, [accessed: February 2018]

33.Van Meteren, R. and Van Someren, M., 2000, May. Using content-based filtering for recommendation. In Proceedings of the Machine Learning in the New Information Age: MLnet/ECML2000 Workshop (pp. 47-56).

34.Sarwar, B., Karypis, G., Konstan, J. and Riedl, J., 2001, April. Item-based collaborative filtering recommendation algorithms. In Proceedings of the 10th international conference on World Wide Web (pp. 285-295). ACM.

35.https://w3techs.com/technologies/details/pl-php/all/all, [accessed: February 2018]

36.http://php.net/manual/en/migration70.new-features.php, [accessed: February 2018]

37.https://www.phoronix.com/scan.php?page=news_item&px=PHP-7.2-Beta-1, [accessed: February 2018]

38.http://php.net/releases/7_2_0.php, [accessed: February 2018]

39.Vlissides, J., Helm, R., Johnson, R. and Gamma, E., 1995. Design patterns: Elements of reusable object-oriented software. Reading: Addison-Wesley, 49(120), p.11.

40.https://neo4j.com/use-cases/real-time-recommendation-engine/, [accessed: February 2018]

41.https://db-engines.com/en/ranking/graph+dbms, [accessed: February 2018]

42.http://www.opencypher.org/about, [accessed: February 2018]

43.https://neo4j.com/cypher-graph-query-language/, [accessed: February 2018]

44.Cervone, H.F., 2011. Understanding agile project management methods using Scrum. OCLC Systems & Services: International digital library perspectives, 27(1), pp.18-22.

45.Brune, K., Fisher, D., Foreman, J., Gross, J., Rosenstein, R., Bray, M., Mills, W., Sadoski, D., Shimp, J., van Doren, E. and Vondrak, C., 1997. C4 Software Technology Reference Guide: A Prototype. Software Engineering Institute (SEI), Carnegie Mellon University, Pittsburgh, PA, USA, Tech. Rep.

46.Kim, S., Clark, J.A. and McDermid, J.A., 2000, October. Class mutation: Mutation testing for object-oriented programs. In Proc. Net. ObjectDays (pp. 9-12).

47.https://infection.github.io/guide/index.htm, [accessed: February 2018]

48.https://neo4j.com/graphgist/movie-recommendations-with-k-nearest-neighbors-and-cosine-similarity, [accessed: February 2018]