Master Thesis

Czech
Technical
University
in Prague

F3

Faculty of Electrical Engineering
Department of Computer Science

Web Editor of the Model of the System Under Test Processes

Bc. Ruslan Bakeyev

Supervisor: doc. Ing. Miroslav Bureš, Ph.D.
Field of study: Open informatics
Subfield: Software Engineering
May 2019

ZADÁNÍ DIPLOMOVÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Bakeyev** Jméno: **Ruslan** Osobní číslo: **399361**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra počítačů**

Studijní program: **Otevřená informatika**

Studijní obor: **Softwarové inženýrství**

II. ÚDAJE K DIPLOMOVÉ PRÁCI

Název diplomové práce:

Webový editor modelu procesů testovaného systému

Název diplomové práce anglicky:

Web Editor of the Model of the System Under Test Processes

Pokyny pro vypracování:

Navrhněte a implementujete webový editor, pomocí kterého bude možné vytvářet a upravovat modely procesů testovaného softwarového systému. Tyto modely budou založeny na orientovaných grafech rozšířených o konfigurovatelnou množinu metadat. V editoru bude možné paralelně pracovat s několika projekty, které budou uložené v databázi systému. Nad vytvořenými modely možné spouštět různé algoritmy pro generování testovacích scénářů. Tyto algoritmy bude do systému možné připojit pomocí otevřeného rozhraní. U této části systému se předpokládá vysoké zatížení z hlediska výpočetního výkonu, proto navrhněte architekturu celého systému tak, aby umožnila škálování této zátěže. Samotné algoritmy pro generování testovacích scénářů nejsou předmětem práce a jejich implementace bude dodána vedoucím práce. Systém implementujte pomocí technologií J2EE a Javascript. Pro implementaci uživatelského rozhraní editoru použijte již existující vhodnou knihovnu pro vizualizaci grafů. Výsledné řešení publikujte jako open-source projekt a otestujte jej sadou vhodných testů.

Seznam doporučené literatury:

P. Ammann, J. Offutt. Introduction to software testing. Cambridge University Press, 2016.
C. Walls. Spring in Action, Fourth Edition. Manning, 2014.
I. Robinson, J. Webber, E. Eifrem. Graph Databases. O'Reilly Media, 2015.

Jméno a pracoviště vedoucí(ho) diplomové práce:

doc. Ing. Miroslav Bureš, Ph.D., laboratoř inteligentního testování softwaru FEL

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) diplomové práce:

Datum zadání diplomové práce: **18.09.2018** Termín odevzdání diplomové práce: **24.05.2019**

Platnost zadání diplomové práce: **19.02.2020**

doc. Ing. Miroslav Bureš, Ph.D.		prof. Ing. Pavel Ripka, CSc.
podpis vedoucí(ho) práce	podpis vedoucí(ho) ústavu/katedry	podpis děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

Diplomant bere na vědomí, že je povinen vypracovat diplomovou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací.
Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v diplomové práci.

.

Datum převzetí zadání	Podpis studenta

© ČVUT v Praze, Design: ČVUT v Praze, VIC

Acknowledgements

I would like to thank my supervisor doc. Ing. Miroslav Bureš, Ph.D. for his valuable advice and knowledge shared for the duration of the diploma thesis. I am also grateful to my family and friends for the support, which helped me to complete this work.

Declaration

I declare, that I am the only author of the submitted work and all the information sources have been listed and used in accordance with the Methodological guideline on compliance with the ethical principles for the university final theses.

In Prague on 21. May 2019

Abstract

The goal of this master thesis is to describe the full development lifecycle process of a web editor allowing to model the system under test processes and generate an optimized set of test cases based on the provided model. In the beginning, this work describes the motivation and defines the fundamental concepts from the software testing area, like the Model-based testing, and continues with the review of core Java and JavaScript technologies, which has been used to implement the solution. Core architectural concepts, description of design patterns used, and successful engineering decisions provided at the end of the thesis, can be useful for anyone trying to build a highly interactive web application. The work finishes with a short conclusion summarizing the results achieved.

Keywords: Model-based testing, process testing, automated test case generation, web editor, Cytoscape.js, React, Redux, Spring framework, Spring WebFlux, MongoDB

Supervisor: doc. Ing. Miroslav Bureš, Ph.D.
Laboratoř inteligentního testování softwaru,
Katedra počítačů,
Karlovo náměstí 13,
121 35 Praha 2

Abstrakt

Cílem této diplomové práce je popsat celý vývojový cyklus webového editoru, umožňujícího modelování procesů testovaného systému a generování optimalizovaných množin testovacích scénářů na základě poskytnutého modelu. Na začátku je popsana motivace a definice základních pojmů z oblastí testování softwaru, například testování založené na modelu. Práce pokračuje vyhodnocením základních technologií ze světu jazyků Java a JavaScript, které byly použity k implementaci řešení. Základní architektonické koncepty, popis použitých návrhových vzorů a užitečná technická rozhodnutí uvedená na konci práce mohou být přínosné pro každého, kdo se snaží vytvořit vysoce interaktivní webovou aplikaci. Diplomová práce končí krátkým závěrem a diskuzí dosažených výsledků.

Klíčová slova: Model-based testing, testování procesů, automatické generování testovacích scénářů, webový editor, Cytoscape.js, React, Redux, Spring framework, Spring WebFlux, MongoDB

Překlad názvu: Webový editor modelu procesů testovaného systému

Contents

Chapter 1

Introduction

Software testing plays a big role in the software development process. The main goal of testing is to verify, that requirements specified by customers are met, the quality of the built product is acceptable according to the predefined criteria, and application behavior is correct and free of bugs. When it comes to creating test scenarios, it is important to identify and understand what is going to be tested, which is usually referred to the system under test (SUT[1]). In some cases, it could be a small function (then we are talking about unit-testing), or a complex system. The latter situations usually require modeling of business logic or flow, which is hidden inside the SUT. Such models help testers to create more efficient and optimized test cases, covering all the branches of the tested process. There are a lot of tools available on the market, like draw.io[2], which can be used to model and graphically represent the system under test, but neither of them has been created specifically for software testing area and free at the same time.

The goal of this work is to provide expertise to the reader on the possible way how to implement the tool, which generates and optimizes test cases. It is obvious, that such optimizations increase the efficiency of the whole quality assurance team, thus reducing the overall cost of the project and time to market. The overall process could be divided into two parts:

- Finding a set of algorithms, which can analyze, optimize and derive test cases from the system under test. Since this problem is a big and independent area of software testing, it is outside of the scope of this thesis. STILL (Software Testing Intelligent Lab) research group has developed an algorithm optimizing test case generation, which is used in this work and called Process Cycle Test [1]. But the system is designed to accept a set of different algorithms, which can help to solve this task, like Prioritized Process Test [2, 3], Set-covering Based Solution or Prefix-graph based solution [4, 5]. Chapter 3 describes this problematic from a high-level perspective, because it is a core of the final software product and all of the outcomes, like input data model validation, and performance influence on the system, should be known ahead and taken into account during the software development process.

[1]A better definition of the term is given in the next chapter

[2]https://www.draw.io/

- Implementing software product, which serves as a platform for the system under test modeling, providing the end user good user experience. The process consists of the following activities:
  - analysis of the application requirements (chapter 4),
  - analysis of existing software libraries and frameworks, which can speed up the development process and ensure best practices during the implementation phase (chapters 5 and 6),
  - definition of software architecture and module structure (chapter 7),
  - user experience and user interface design.

The idea of the product takes inspiration from the concept of Oxygen platform (formerly known as PCTGen [1]), but enhanced with the following features:

- cross-platform support due to the web nature of the tool,
- synchronization between devices, since the data are stored in a web database and attached to the user account,
- data backups in the background every 3 seconds after inactivity in order to improve user experience and minimize the aftermath of unexpected situations, like internet connection loss,
- software architecture that allows extension of the product in the future with different types of diagrams, like state machines or UML class diagrams,
- data model, specifically designed to share projects between users in the future.

The last chapter of the work summarizes achieved results and describes the product roadmap.

Chapter 2

Definition of basic concepts

This chapter explains essential terms, which are used throughout the thesis, but their proper explanation is out of the scope. Table 2.1 shows a brief definition with a short form of the term, which the reader can face in the future chapters.

Term	**Abbreviation**	**Definition**
Test Case Values	-	The input values necessary to complete some execution of the software under test [6].
Prefix Values	-	Any inputs necessary to put the software into the appropriate state to receive the test case values [6].
Postfix Values	-	Any inputs that need to be sent to the software after the test case values are sent [6].
Test Case	-	Composition of the test case values, expected results, prefix values, and postfix values necessary for a complete execution and evaluation of the software under test [6].
Test object	-	The component or system to be tested [7].
System under test	SUT	A type of test object that is a system [7].
Test requirement	TR	A condition or an objective that must be met or covered by a test case [6].
Continuous integration	CI	An approach, which aims to automate build and testing processes after every change committed to the source code repository [8].

Term	**Abbreviation**	**Definition**
Unified Modelling Language	UML	A set of graphical notations with the unified metamodel, which are used to design, model, or document an aspect of the software system [9].
Application programming interface	API	A set of methods, which can be used by one system to communicate with another.
Document Object Model	DOM	A platform-neutral interface exposed to the programs and scripts to dynamically manipulate with the content, attributes, and style of XML-based document [10].
User interface	UI	A set of interfaces, which allows a user to interact with the elements of the system.

Table 2.1: Definition of used terms.

Chapter 3

Introduction to the testing based on a model

Testing is not a separate activity in software engineering. Instead, it is tightly coupled with every phase of the software development lifecycle, starting from requirements analysis and management until the release to production. The main goal is to verify on each stage, that the developed product is correct and respects the specification and client needs. Most of the errors are hidden in the requirements, while the cost of the fixes increases with time. That is the main reason why the test process should be established as early as possible.

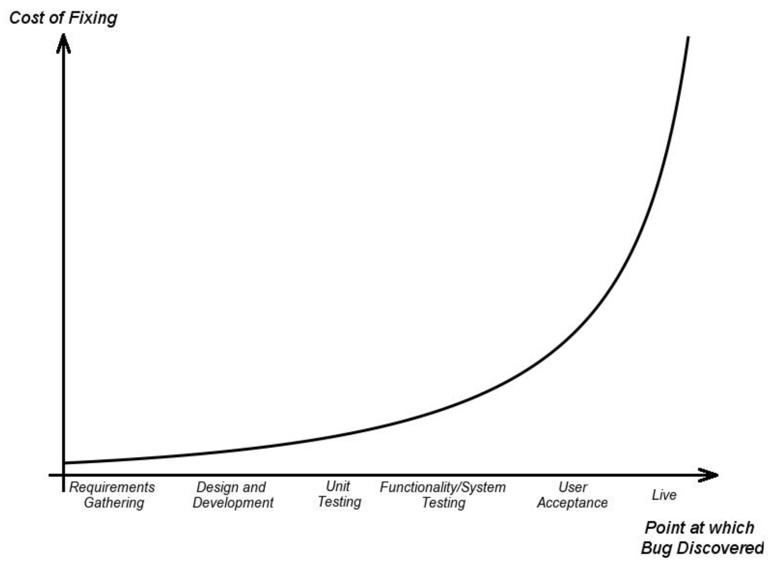


Figure 3.1: Cost of the bug fix with respect to time [11].

3.1 Fundamental definitions in software testing

Each phase of the software development lifecycle has a corresponding **test level**, that relates to it. P. Ammann and J. Offutt define the basic test level hierarchy shown in Figure 3.2. Most of the test levels require different parts of the team to participate. For example, software developers mostly involved in unit testing, while stakeholders or product owners in agile environments are responsible for creating acceptance criteria.

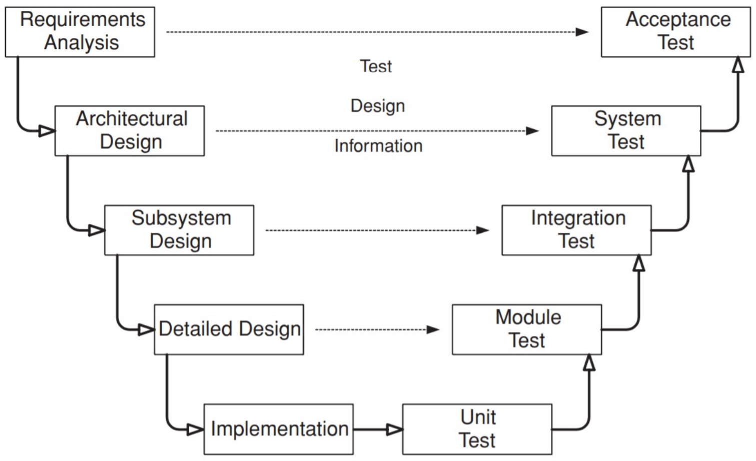


Figure 3.2: Basic test level hierarchy, the "V model" [6].

There are more criteria, which help to classify the testing process and formulate the problem of model-based testing. One of the essential aspects is the target characteristic of the tested system, which requires functional, performance, usability, security, and many more test types.

The knowledge about the internal implementation of the system can significantly change the design of test cases. If the test is derived from the source code, then the method is called **white-box testing**. It can be applied on different levels of the testing process, but usually takes place in unit testing. Such an approach can be easily automated and allows coverage of all branches of the source code. On the other hand, the resulting test is tightly coupled to the existing implementation and requires a deep understanding of the internal implementation from the test engineer or developer. **Black-box testing** is the opposite technique, which relies on system specifications and requirements. Unlike the white-box method, it can be applied to all levels of the testing process and more focused on the functionality instead of internal implementation. **Grey-box testing** is a combination of the methods described above. It usually requires both the system specification and partial knowledge about its internal structure. The main idea behind it is to verify the correctness of software implementation relative to the application requirements.

All the methods mentioned earlier assume, that the target code is **executed**, and differs only with the level of knowledge about system implementation. This is the necessary precondition for **dynamic testing**. Andreas Spillner et al. define a set of problems that can be only found by dynamic testing [12]. Faults in the data exchange or communication between systems are the typical example. Static testing assumes an analysis of the test object, rather than execution [12]. The process could be initiated even in the early stages of the software development lifecycle by analyzing the relevant documentation of the software product. The goal is to find any inconsistencies and defects in corresponding specifications or requirements. Figure 3.1 shows, that early identification of a bug significantly reduces the cost of future rework [12]. Apart from that, static analysis can be performed on the source code during the implementation phase. There are a variety of tools on the market, that can help to automate this process. Andreas Spillner et al. define the following issues, which can be detected by static analysis [12]:

- programming language syntax error,
- violations of code conventions or standards
- security holes or danger constructions, which can potentially lead to security issue, like lack of buffer overflow protection,
- control flow anomalies,
- data flow anomalies.

3.2 Test automation

A proper way to increase testing efficiency is automation [13, 14]. One of the main advantages is the overall process speed up, but, on the other hand, there is a need to maintain the source code of testing scripts [15, 16, 17]. Nowadays, a variety of modern concepts have the goal to decrease the required effort [13, 14, 18, 19, 20]. E. Dustin et al. define automated testing as the use of an automated test tool to manage and perform test activities, including test precondition setup and verification of the outcomes with the expected results [14]. Since software testing can be expensive and require much effort, the main goal is to automate as much, as possible. Thus not only achieving cost reduction but also reducing the probability of human error by reducing the amount of manual work needed. It is especially valuable when repeated execution of the same test is required, i.e., while performing regression testing. The purpose of regression testing is to verify new changes haven't broken a piece of functionality, which is supposed to remain untouched. Unlike manual testing, automated testing allows the execution of the regression tests in a simple and error-free manner. The same effect can be achieved in agile environments while adopting continuous integration practice. The basic idea is to execute a set of automated tests after the developer has pushed the source code to the repository [6]. CI service automatically performs test execution after the system rebuild. Any mistake made by a developer can be easily detected, while the rest of the team gets the notification of the issue.

A. Rafi et al. in the research define 9 main benefits of test automation [17]:

- product quality improvement by reducing the number of defects,
- better test coverage,
- decreased time required for testing,

- improved reliability,
- increase in confidence of the product quality,
- possibility to reuse tests,
- less human effort,
- costs reduction by minimizing manual work required,
- better fault detection ability.

Despite the advantages mentioned above, the decision about automation should be made carefully, taking into account the limitations defined by A. Rafi et al. [17]:

- manual testing is needed anyway, i.e., automation can't replace every test activity.
- The real benefit is not always achievable by automation.
- With every change in the software product, there is a need to maintain the corresponding test.
- It takes time to implement the test and fulfill infrastructure requirements.
- Trying to save as much cost as possible, organizations face with false expectations.
- It is hard to decide what should be automated. Inappropriate strategy leads to the lost benefits of automatization.
- Test engineers should be skilled enough to automate the test because it requires knowledge from different areas.

According to A.Spillner et al., test automation is a bad option for projects with a low maturity level of the test process, e.g., when the documentation doesn't exist or contains inconsistencies [12]. In such cases, the better option would be to establish a manual testing process first. The main idea is to prefer the effectiveness of testing, rather than improving the efficiency of poor testing [12].

3.3 Coverage criteria

Coverage criteria are the metrics used in software testing. P. Ammann and J. Offutt define coverage criterion as a rule or a set of rules that a test activity should satisfy [6]. The research literature identifies a lot of coverage criteria, but the following are supposed to be the most popular [21]:

- function coverage, determining, if all the functions have been called in the application code,

- statement coverage, identifying, if all the lines of the source code have been executed,
- condition coverage, showing if all the possible results of condition expression are tested,
- edge coverage, measuring the ratio of executed edges in the control flow graph,
- branch or decision coverage, verifying if all the branches of the target software, e.g., usually if-else or switch statements, has been covered.

The type of coverage criteria often determined by the test level. While statement coverage is a proper choice for unit testing, it can be hard to satisfy during the system tests [12]. On the other hand, the ratio of tested requirements can determine system test coverage.

Some of the criteria could be applied during the test automation process without spending any additional effort. It is possible due to the available tools, which include coverage analyzers, comparators, and many more. After the test execution, the coverage report is created. Test engineer, developer, or any other person can analyze the produced log and evaluate test efficiency. It is crucial to keep in mind when interpreting the results, that coverage metrics algorithms depend on the chosen tool [12].

3.4 Model Based Testing

Section 3.2 shows, that test automation is not a solution to all the problems. One of the main disadvantages is the necessity to maintain the source code of the test in case of any change. The situation becomes even worse with the new functionality added on a regular basis. As the number of requirements increases, the software system becomes more complicated. The volume of test cases explodes, and it is hard to keep an overview of what has been already tested [22]. Moreover, the number of errors in the specification grows. Figure 3.1 shows that the price of such mistakes significantly increases as the application becomes closer to its release phase, while the industry demands cost-reduction and improved time-to-market. **Model-Based Testing** (MBT) can be a solution to these problems, as it reduces the effort of test scenario generation and maintenance while achieving desired coverage criteria.

The model is the heart of the MBT method and the main challenge at the same time. Robert V. Binder et al. define a model as a set of assumptions and relationships between them, which allow approximating of a specific aspect of reality [22]. In other words, a model is usually used to represent the expected behavior of the system under test. In most cases, a model is abstract and should cover only the main aspects of the SUT. That allows preventing the explosion of generated test cases. Figure 3.3 shows an example of a model.

MBT approach can be used on different test levels. However, it is usually applied on higher levels, starting from integration tests. Figure 3.4 shows the

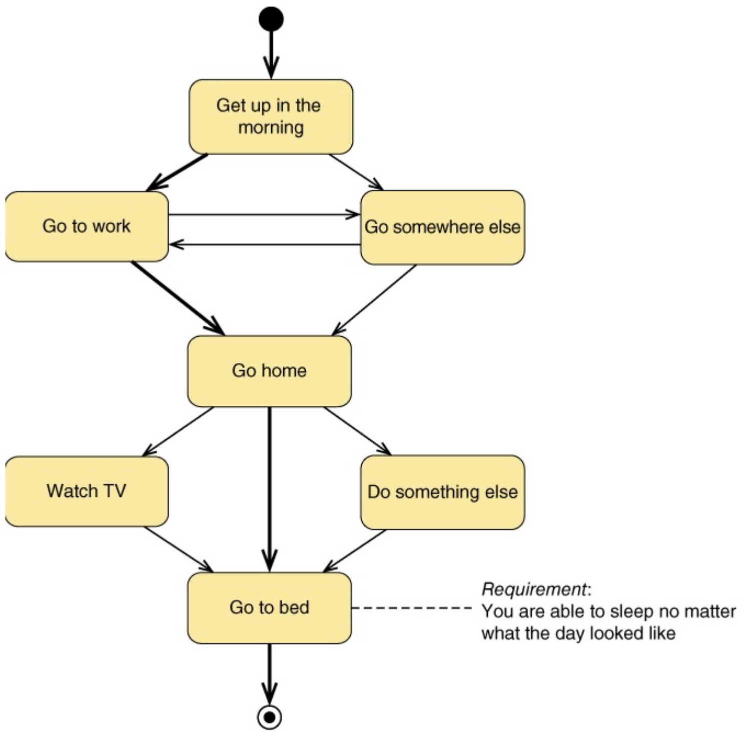


Figure 3.3: Basic example of a model [22].

result of the MBT User Survey from the year 2014 [22]. The main reason refers to the key advantage of the model - the capability to abstract the target user from the complexity of the functional requirements. The use of the 'target user' term is not an accident. Test engineers are not the only people involved. Software developers could be also interested and take part in the process for test automation purposes. Stakeholders can verify the correctness of the requirements. Additionally, any member of the team, who lacks the knowledge about the target product, can benefit from reviewing the model.

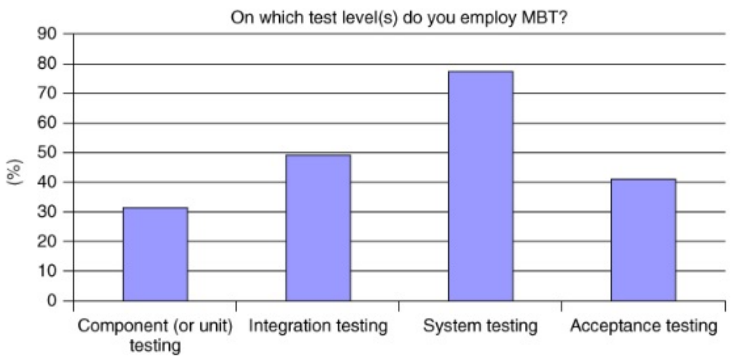


Figure 3.4: Test level MBT approach applied on. MBT User Survey, 2014 [22].

The typical MBT process includes 4 main activities [22].

1. Creation of the model of the SUT. The result is a structured representa-

tion (either text or graphical) of the target aspects of the system. The model is usually derived from the requirements. That is the reason why MBT is a form of black-box testing.

2. Generation of test cases and selection of the test set, which satisfies predefined criteria, as it is possible to generate a variety of tests from the same model by using different test selection criteria. The only precondition is the model from step 1, which should be precise enough to allow the test case generation.

3. Implement abstract tests to make them executable. Since the model is an abstract representation of the SUT, it is not possible to directly execute the generated test set.

4. Test execution and further result analysis. Assuming, that the model from step 1 is correct, failed tests can signalize about the differences between actual and expected behavior of the system caused by incorrect implementation or error in the requirements.

3.4.1 Advantages of the MBT

Section 3.4 introduces the main benefits of the adoption of the MBT approach. Every test engineer tries to get maximum effectiveness and efficiency from the test activity. The modeling process encourages better communication both with the stakeholders and throughout the team, improves application domain knowledge, and continuously supports the level of understanding of the application requirements for every member involved. These factors directly influence the effectiveness of the test activity. The efficiency is significantly increased by the reduction of maintenance effort when the application requirements change. Rather than maintaining test cases itself, the focus is shifted to the model, which allows generating test cases whenever needed. Moreover, the same model provides the possibility to create different test sets based on the test level, coverage criteria, and test selection strategy.

Apart from that, Robert V. Binder et al. define the further benefits, which the MBT approach can bring [22]:

- MBT models establish a focus on what is tested, thus helping in managing complexity. That follows directly from the definition of term model, which, according to the recommendations [22], should be abstract and much smaller than the SUT.

- MBT models visualize tests, allowing more effective team involvement and more productive discussions with non-technical stakeholders.

- The model can be created in the early stages, helping to verify requirements and avoid bugs in later phases of the software development lifecycle.

3.4.2 Disadvantages of the MBT

The previous section described the main advantages of the MBT approach, but every test team should also be aware of the limitations and costs required for the adoption of the process. One of the biggest challenges of MBT is a modeling activity. First of all, the model should be correct. Otherwise, the interpretation of failed tests would be misleading and confusing. It is also necessary to choose the proper level of granularity and abstraction of the model, which requires experienced model designers and time. Each of these factors leads to the increased cost of the overall testing process. According to Robert V. Binder et al., MBT can also lead to the test case explosion 3.5 shows a simple model of a chat room. There are no limitations on the number of messages sent per person, as well as the order of chats received is not predefined by the system. The number of possible variations grows exponentially and equal to $4^4 = 256$ for the 4 participants in the chat room. Fortunately, defining test selection criteria can help in this situation [22].

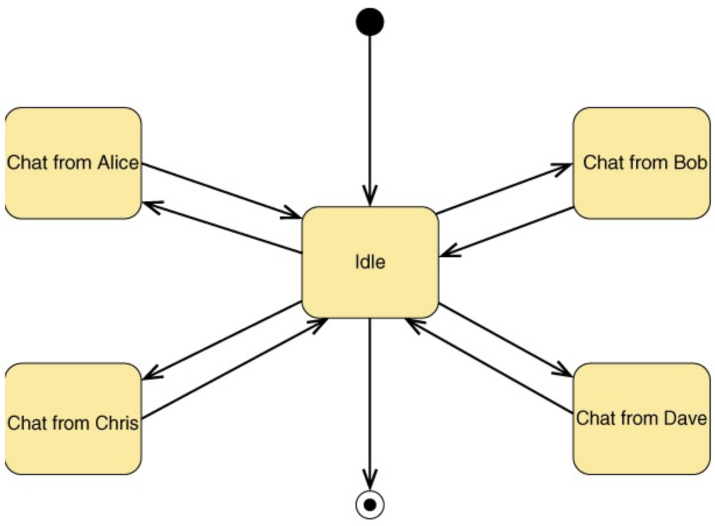


Figure 3.5: Model for testing a chat room [22].

3.5 Process testing

Process testing is considered to be a discipline of the abstract model-based testing concept. The main difference is in the model, which usually represents an application workflow or a process. Bures, Ahmed, and Zamli state that process test uses a sequence of actions in a system under test to identify possible inconsistencies in SUT processes [3]. An automated generation of process tests is significant both for the software systems testing [6, 23, 24], and IoT systems testing, where quality assurance is a common technological and organizational challenge [25, 26] or specific platforms like SmartTV [27, 28].

There exist a lot of notations, which can be used to describe the model

[22]. Moreover, the choice of the modeling language directly depends on a modeled aspect of the application. For example, UML package diagrams are widely used to describe structural MBT models, while UML activity diagrams can be a proper choice to represent behavioral MBT models [22]. The latter, together with Business Process Model Notation (BPMN), is also the commonly used option for the modeling of workflows and processes of an application [3]. However, for test case generation purposes, these two notations should be transformed into a more suitable data structure. A directed graph is a preferred choice [3] due to the following reasons:

- A lot of existing algorithms for finding a path in a directed graph, like Depth-first-search (DFS) or Breadth-first-search (BFS), can be adapted for an automated test generation.
- Both the UML activity diagram and BPMN can be transformed into a direct graph. An example is shown in figure 3.6

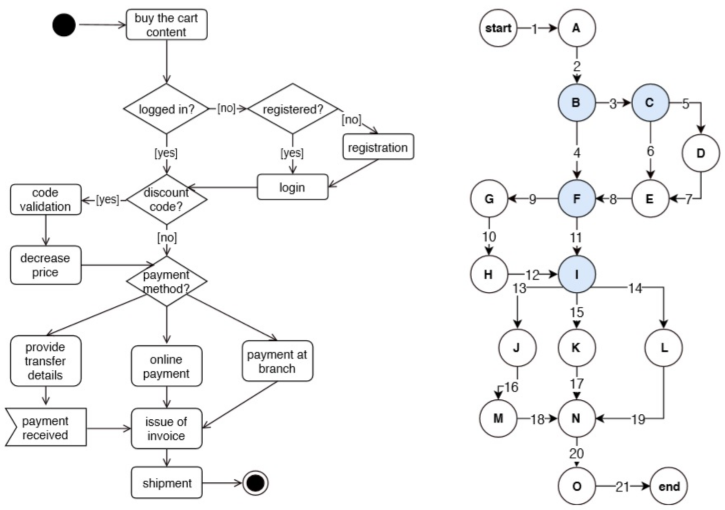


Figure 3.6: An example of conversion UML activity diagram into a directed graph [3].

3.5.1 Process model formal definition

As has been mentioned earlier, a directed graph $G = (V, E)$ can represent the SUT flow or a process. V stands for a set of vertices of G, where each vertex has a meaning of a decision point or an action within the workflow [1]. It is up to a model designer to choose an appropriate granularity of the actions, which are additionally determined by the test level. Graph edges E is a subset of $V \times V$, representing transitions between distinct decision points [1] or actions. Additionally, graph G must fulfill the following conditions:

- there should exist one initial vertex $v_s \in V$,

- there should exist at least one end vertex, i.e., $V_e \subseteq V$,
- $D \neq \emptyset$.

The generated test case t is a sequence of vertices $v_1, v_2, \cdots, v_n$, satisfying the following constraints:

- $v_1 = v_s$,
- $v_e \in V_e$,
- there should exist edge e_i, which connects v_i and v_{i+1}.

3.5.2 Coverage criteria

One of the primary factors, defining the number of generated test cases, is selected coverage criterion. Path coverage is a commonly used strategy for process testing. Table 3.1 reviews the most common coverage criteria, which are ordered by the level of intensity of the generated tests.

Criterion	Description
All Node Coverage	Each node $v \in V$ should be present at least in one test case t from the test set T.
All Edge Coverage	Each edge $e \in E$ should be present at least in one test case t from the test set T. This criterion is usually applied for smoke or regression tests.
Prime Path Coverage	Each reachable prime path (a simple path, that is not a sub-path of any other simple path) in graph G should be a sub-path of some test case t from the test set T [3].
All Paths Coverage	Each possible path in graph G should be present in the test set T. This criterion is rarely used in testing because it is too exhaustive.

Table 3.1: Common coverage criteria in process testing [3] .

As an alternative to the criteria shown in table 3.1, **Test Depth Level** (TDL) concept can be used to determine the test coverage level. The main advantage is the flexibility it provides. For instance, $TDL = 1$ is equivalent to All Edge Coverage from table 3.1, while $TDL = n$ represents all possible combinations of n following edges in graph G [3]. In other words, the value of the TDL parameter determines the intensity of the generated test set. A more strict definition is given, for instance, in *Prioritized Process Test: An Alternative to Current Process Testing Strategies* [3].

3.5.3 Algorithms used for test case generation

The TDL coverage criterion is widely used in process testing, but it only helps to generate a sequence of n edges in a graph. The next step is to assemble the generated combinations to a path in an efficient way. Usually, the goal is to minimize the number of steps to execute in the output test case, covering all the actions required to test at the same time. **Process Cycle Test** (PCT) is an example of a technique solving this task. The internal implementation is described in *TMap Next* [23], and out of the scope of this work. However, the algorithm doesn't respect other aspects, which should be kept in mind during the test generation process. For instance, certain actions in the workflow could have higher business priority or potential risk from the technical point of view. Bures et al. implemented an algorithm called **Prioritized Process Test** (PPT) [3], which respects the concept of prioritization. A detailed description of the internal implementation can be found in *Prioritized Process Test: An Alternative to Current Process Testing Strategies* [3].

Chapter 4

Web Application Requirements

In the previous chapters, there were a lot of mentions about the importance of application requirements. Figure 3.2 shows that the whole lifecycle of software development starts with the definition of the system specification. In an agile environment, this process can run simultaneously with other phases of the lifecycle.

There are two basic types of requirements in software engineering. R.Young summarizes that functional requirements describe the behavior of the system, whereas non-functional requirements define system properties [29], that is application availability, security, scalability, response time, and many others. The document providing both functional and non-functional requirements to the rest of the team is called specification [29].

Tables 4.1 and 4.2 shows the most important requirements to the web application.

#	**Requirement**
1	The application should provide the possibility to add new types of model diagrams, like UML activity diagram or state machine charts.
2	The application should provide the possibility to add new types of the test set generating algorithms, like PPT or PPC.
3	Server-side of the application should be implemented in Java.
4	Client-side of the application should be implemented in Javascript.
5	The application should use Cytoscape.js as a graph visualization library.
6	The application should be both horizontally and vertically scalable.
7	The test case generator module is computation-intensive. The application should handle this loading factor.
8	The application should be implemented as an open source project, that is source code should be properly documented.
9	The application should be secured. Main security vulnerabilities, like SQL-injection or XSS-scripting [30], have to be prevented.

Table 4.1: Non-functional requirements.

#	Requirement
1	The application should allow a user creating, updating, and deleting a model of SUT processes represented in the form of a directed graph. Every element of the graph can be extended with a configurable set of metadata. An element's priority should be visualized using appropriate colors. Low priority is considered to be a default.
2	The application should allow a user to change the graph element name and any other attribute except the node/edge id.
3	The application should allow a user to create, update, and delete projects. A project is a collection of the models of SUT processes, which is identified by the name and optional description.
4	The application should allow a user to execute algorithms for test case generation and display the output in the editor.
5	The application should mark the generated test case set as invalid in case of any inconsistencies, which could be caused by node or edge deletion or missing start point.
6	The application should allow a user to review previously generated test cases and delete it if necessary.
7	The application should allow a user to register and authenticate to the system.
8	The application should restrict unauthorized access to the created entities, which are owned by another user.
9	The application should persist data into the database.
10	The application should persist data on: ■ "Save" button click, ■ after a new graph is selected, ■ after 3 seconds since the last user action. In case of any synchronization issues, alert notification should be displayed to the user.

Table 4.2: Functional requirements.

Chapter 5

Defining server-side technology stack

Choosing core technologies for a project is a complex process, which requires continuous analysis of both the application requirements and functionality provided by the target technology. This chapter provides a review of the most important libraries and frameworks, which help to build a reliable and maintainable software product. In the context of this work, the following evaluation criteria for the technologies have been defined:

- Java language support for the web-server and JavaScript language support for the web client;
- availability of the tutorials and a gentle learning curve to involve new contributors more effectively;
- the level of flexibility provided by the tool, since the web editor should support various types of the diagrams and algorithms in the future;
- the effectivity with which technology solves the problem;
- a type of the license authorizing free use of the application.

5.1 Client-server communication

5.1.1 Simple Object Access Protocol

Simple Object Access Protocol (SOAP) is a communication protocol specification extensively using XML to structure the message format. It mostly relies on Hypertext Transfer Protocol (HTTP) as a transport protocol for message exchange. The complexity of the specification, performance implications due to the message size, and the necessity to deal with XML format makes SOAP inappropriate technology to use at the moment. Nowadays, the protocol is primarily used in systems that have strict security policies, which are banks, payment gateways, or billing companies.

5.1.2 Representational State Transfer

Representational State Transfer (REST) is an architectural style, which defines a set of constraints applied to the distributed systems. Nowadays, nearly 80% of the applications are following this architectural style [31] due to its simplicity and scalability. REST defines a concept of resources, which relates to a piece of the application state. The HTTP protocol is widely used as a communication protocol, allowing to transfer messages and operate with the resources using the following basic request methods:

- `GET` - used to retrieve a resource or a collection of the resources;
- `POST` - used to create a new resource;
- `PUT` - used to update a resource or a collection of the resources;
- `DELETE` - used to remove a resource or a collection of the resources.

The web service is called RESTful, if it doesn't violate the following architectural constraints [32]:

- client-server architecture,
- statelessness,
- cacheability,
- layered system,
- code on demand,
- uniform interface.

5.1.3 GraphQL

GraphQL is a query language providing the possibility to request for data required by the client. The main idea is the opposite principle of the traditional paradigm, where it is up to the server to decide what information to expose to the client. The technology was released by Facebook in 2015 and published as an open source project.

GraphQL is especially useful when the data needs to be fetched from different sources, which is a usual case for enterprise-level applications or the systems following microservices approach. On the other hand, it adds another level of complexity for smaller projects.

5.2 Spring framework

Spring is an open source Java application framework. Before Spring was created, Java applications had been built using a traditional Java EE approach with the following unresolved problems:

- complex implementation,
- huge specification document to learn from,
- heavyweight deployment artifacts.

Spring, which initially served as an inversion of control (IoC) container, can solve nowadays a variety of tasks from different domains:

- dependency injection,
- data access,
- system integration,
- web applications,
- AOP (Aspect Oriented Programming),
- testing.

It can perfectly achieve any goal, starting from developing a small proof-of-concept application until the complex modern solutions built on top of the micro-services [33].

The following benefits make the Spring framework a perfect candidate to use in the project:

- time-tested architectural decisions and best practices enforced by the framework;
- an open source nature of the project, which is improving continuously due to the community support;
- detailed documentation enriched by the guidelines aiming both the new and experienced developers.

Starting from version 5 Spring offers two different approaches for building web applications [34]:

- based on synchronous blocking I/O architecture,
- served by an asynchronous, non-blocking stack built on the reactive design principles.

The whole architecture of WebFlux is event-driven. As shown in figure 5.1, Spring WebFlux uses reactive streams instead of servlets to handle the asynchronous flow. The application built on the top of WebFlux is able to handle a relatively big amount of concurrent requests, which is exactly the case of a typical web editor application. Project Reactor, which is Spring WebFlux based on, includes the creation of reactive controllers, repositories, web clients, and many other components [33]. However, the solution has the following disadvantages:

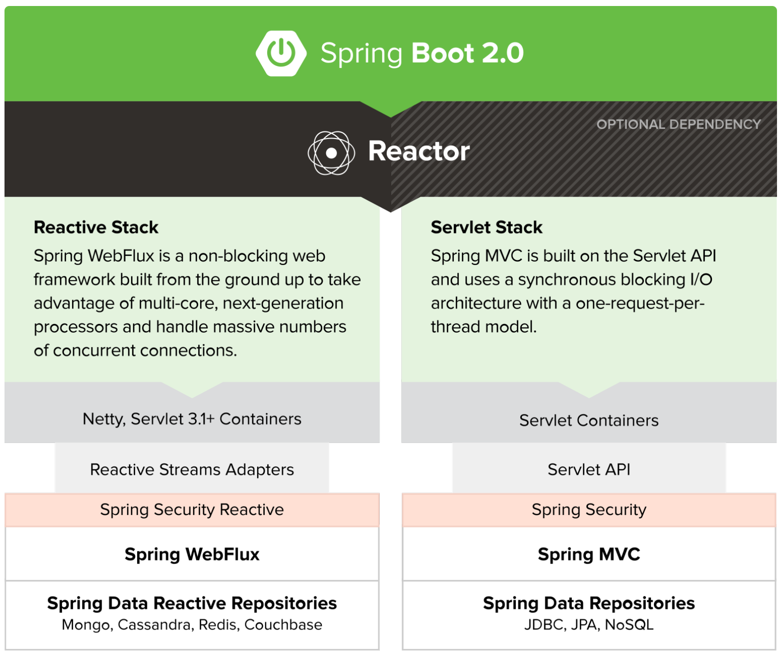

Spring Boot 2.0
Reactor
OPTIONAL DEPENDENCY
Reactive Stack
Spring WebFlux is a non-blocking web framework built from the ground up to take advantage of multi-core, next-generation processors and handle massive numbers of concurrent connections.
Servlet Stack
Spring MVC is built on the Servlet API and uses a synchronous blocking I/O architecture with a one-request-per-thread model.
Netty, Servlet 3.1+ Containers
Servlet Containers
Reactive Streams Adapters
Servlet API
Spring Security Reactive
Spring Security
Spring WebFlux
Spring MVC
Spring Data Reactive Repositories
Mongo, Cassandra, Redis, Couchbase
Spring Data Repositories
JDBC, JPA, NoSQL

Figure 5.1: Comparison of Spring WebFlux and MVC architectures [34]

- increased complexity of the projects, due to the paradigm shift caused by the reactive approach;
- less amount of documentation, since the WebFlux module is relatively new
- some bugs can be still present.

Chapter 6

Defining client-side technology stack

This chapter analyzes existing Javascript libraries and frameworks available for software developers. Since the Javascript projects usually have a bunch of dependencies, the next sections cover the most important libraries only. Chapter 5 defines the fundamental criteria helping to choose proper technology. However, for the web client, several additional factors should be taken into account:

- browser support,
- the size of the package,
- the type of web application, for example, a single page application (SPA).

6.1 UI libraries

React, Vue.js, and Angular are the most popular Javascript libraries used for building large-scale single-page applications. Figure [35] illustrates the interest of developers to these tools.

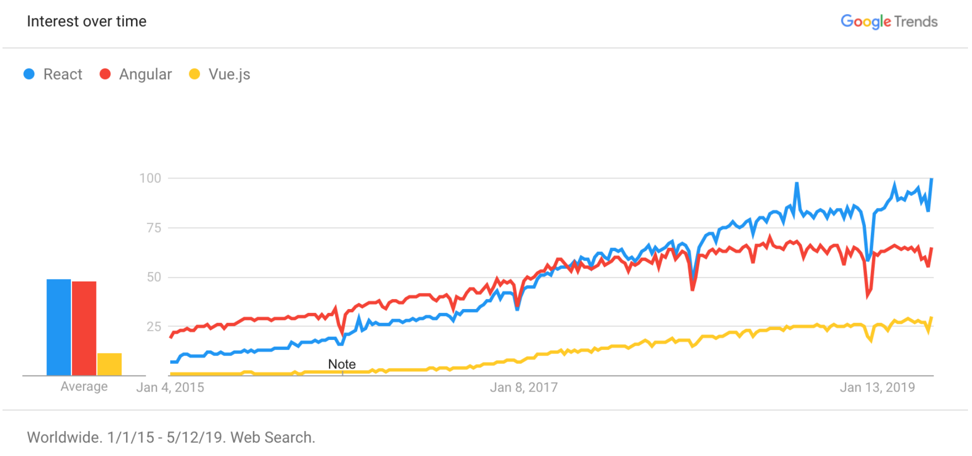


Figure 6.1: React, Angular and Vue.js trends comparison [35]

It is not entirely accurate to compare React with Angular or Vue.js, because, technically - React is a library, while the latter two are full-fledged frameworks.

The difference between a library and a framework is in the amount of the tasks they try to solve. Usually, the latter provides the foundation and tools often complete in functionality to deliver full solutions. In other words, picking a library requires an analysis of further tools specific for different tasks.

6.1.1 Angular

Angular is a framework created by Google in 2010. The first version, called AngularJS, was the first of its kind, i.e., a full-blown solution for highly interactive single and multi-page web applications. In 2016, due to vast advancements in JavaScript language and its ecosystem, the company decided to completely redesign and rewrite the framework to fit in the modern environment.

Angular includes everything needed to implement a complete solution. It uses the TypeScript language to make the code type-safe and eliminate the most common bugs that most pure JavaScript applications have. A typical Angular application is structured into modules including components and services, which are building blocks of every Angular-based application. The architecture enforced by the framework focuses on building scalable web applications without losing control of the source code.

Between the main disadvantages are a steep learning curve and a high level of abstraction, which can prevent other developers from contributing to the open-source application.

6.1.2 Vue.js

Vue is a web application framework that focuses on incremental adoption, declarative rendering, and component composition [36]. It takes inspiration from Angular and React to deliver an Angular-like experience with React rendering model. The framework was created by Evan You in 2013. However, the first version was released two years later. Vue.js aims to be more comprehensive than React by providing official packages for routing and state management, and less complicated than Angular at the same time. Unfortunately, the level of popularity of the framework makes it an inappropriate choice for developing an open-source project.

6.1.3 React

React was initially released in 2013 by Facebook as a competitor to Angular.js, and quickly gained in popularity due to the following reasons:

- a gradual learning curve,
- better performance,
- the paradigm shift from imperative to the declarative approach.

The main building block of the library is a component. React components are reusable and self-contained pieces of code, that accept parameters, called properties, and optionally can have a state. A component changes its state as a response to some event, for example, button click. React organizes components into a tree, which is called virtual DOM. The idea behind it is to minimize the number of operations with a real DOM to improve the overall performance. Managing the virtual model is a considerably cheaper operation. The algorithm which prevents re-rendering of the whole component tree is called reconciliation. Its goal is to identify and render the parts of the tree, which require updates, leaving all the rest untouched. Such an approach makes React very performant.

Another innovation, which made the library successful is JSX. It stands for **J**ava**S**cript **X**ML and provides the capability to define the output of the components in an HTML-like representation. The purpose of JSX is to improve code readability. However, JSX is just syntactic sugar converted to the plain Javascript during the compilation phase.

At first sight, the main disadvantage of React is a lack of additional tools shipped with it. That is the main difference between a library and a framework. However, React perfectly fits the needs of this work due to the following reasons:

- flexibility to choose the appropriate library, since the process modeling tool is not a typical web application, and the tools shipped with the frameworks may not help;
- the size of the community, as shown in Figure 6.1, and the number of learning materials available;
- performance increased by applying the virtual DOM concept.

6.2 Introduction to the Flux architecture

The core of every web application is its state, so, along with React, Facebook also introduced an architectural pattern, which helps to build scalable web applications - Flux. It is best suited to work with React, but can be used in other contexts as well. Flux mainly addresses a fundamental part of every architecture - the data flow. The pattern ensures that data flow only in one direction, which makes the application more predictable and easier to understand.

As shown in figure 6.2, the main components of Flux are:

- Dispatcher, which is responsible for receiving actions and sending them to the stores.
- Store, which holds the application data and mutates them upon receiving an action from the dispatcher. After modifying the data, the store must emit a change event, to let the views know that they need to re-render.
- Action, which is usually a simple object containing data and a type field to identify itself. Dispatching an action is the only way how to interact with the application state.

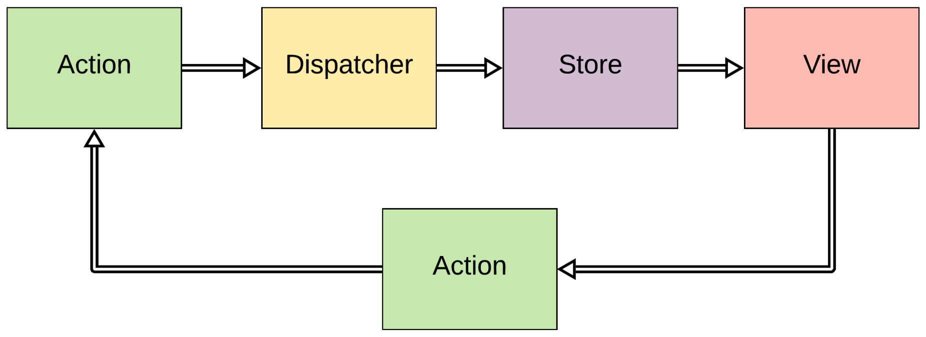


Figure 6.2: Flux architectural pattern [37]

- View, which displays data from the stores and sends actions to the dispatcher in response to some events, such as a button click.

Flux architecture is widely used with React to organize the data flow and the structure of event-driven applications, making it a suitable candidate to use within this work.

6.2.1 Redux

Redux is a Javascript library helping to organize and manage application state. It takes inspiration from the Flux architectural pattern and is often thought of as its implementation. However, there are some aspects in which Redux differs. They are as follows:

- the dispatcher component is missing in the Redux;
- the number of stores is limited to one, following the concept of a single source of truth;
- every store subscriber is required to provide a callback, which is executed on every update of the state.

Redux is based on the concept of reducers, which are the functions calculating a new state based on the current application state and the action dispatched. Such an approach has a great scaling potential, by utilizing the principle of function composition, that allows managing the state trees of any complexity by slicing it into small maintainable chunks. In addition to the benefits described above, Redux enforces the entity immutability principle to minimize the number of side effects, which are hard to debug. Following the best principles from the Flux pattern and functional programming, Redux is a powerful solution for application state management, which is widely used with React to build highly interactive, event-driven maintainable applications.

6.3 Babel

Babel is a JavaScript compiler that converts newer ECMAScript standards into the code supported by older browsers to ensure backward compatibility. That gives the developers flexibility to use cutting edge language features, and at the same time support outdated JavaScript engines used in browsers. Babel is widely used by most of the open source frameworks and libraries, including React.

6.4 ESLint

Since JavaScript is a dynamically typed language, keeping bugs out is a hard task. ESLint is a static code analysis tool helping to find defects that are usually hard to detect, like a typo or missed comma. One of the main advantages of ESLint is a pluggable architecture, providing the possibility to adapt the tool to different JavaScript-based languages, like TypeScript. The plugins are actively developed by the community and are free to use. Additionally, ESLint can be used to validate the source code style against the style guides created by different organizations, enforcing consistency within the project.

6.5 Webpack

Webpack is a static module bundler for web applications. It analyzes the libraries used by an application and builds a dependency graph. The output of the process is one or more bundle files, containing all the necessary dependencies and assets packaged with it. A typical example of an asset is HTML, CSS, or an image. Apart from that, Webpack supports the following features [38]:

- source code transformations, like compilation, minification, obfuscation, and dead code elimination;
- dependency graph optimization dramatically decreasing the output bundle size;
- built-in optimizations based on the target environment, like an injection of the environment variables.

The scope of the tasks solved by Webpack is defined by a set of declared plugins in the configuration file. As with the ESLint tool, the project, plugins, and loaders are developed and maintained by the community.

6.6 Create React App (CRA)

Configuring a modern React-based web application requires knowledge of tools like Webpack, Babel, ESLint, and many others. To simplify the process, CRA has been created. The benefits of its usage are as follows:

- automatic configuration of Babel, Webpack, and ESLint;
- automatic compilation of JSX to the plain JavaScript code;
- setting up build scripts for development and production environments;

In other words, CRA provides a modern build setup with no effort required. It is especially useful for new developers trying to get acquainted with React.

Chapter 7
Implementation of the web application

This chapter describes the most relevant implementation aspects of the software product. It consists of several sections, including the overall application architecture review, API description, the data model definition, server, and client internal implementation details.

7.1 Overall architecture review

Figure 7.1 illustrates the overall application architecture. It consists of three main modules, which are React-based client application, Spring-based server application, and the MongoDB database. The presented solution is a typical implementation of the client-server architecture, which refers to requirement #3 from table 4.1.

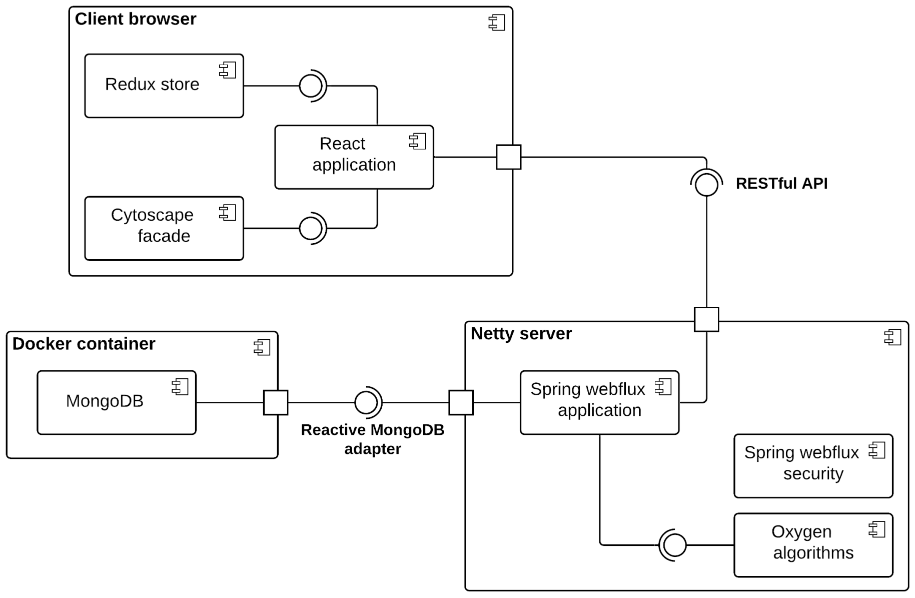


Figure 7.1: Web application architecture diagram

7.2 API endpoints definition

It is a good practice to think in terms of resources working with the RESTful APIs. In the scope of this work, the following resources can be defined:

- user,
- project,
- graph,
- test case.

Table 7.1 shows authentication endpoints, which violate the concept of resources. The choice is made in favor of simplifying the integration process for the third-parties.

URI path	Method	Semantics
/login	POST	Retrieves authentication token, which should be sent with every request for the protected resources.
/register	POST	Creates a new user in the system. An authentication token is sent as a part of the response.

Table 7.1: Description of the authentication endpoints

The authentication token obtained, allows the user to request resources defined in table 7.2. Note, that all endpoints are prefixed with the `/api/v1` path. Incrementing the version of the prefix is a simple way how to manage any changes made to the API. Since the designed API follows conventions and best practices defined for RESTful web services, the semantics of request methods reflects standard behavior and doesn't require any additional description.

URI path	Available Methods
/users/:id	PUT
/projects	GET, POST
/projects/:id	GET, PUT, DELETE
/projects/:id/graphs	GET, POST
/projects/:id/graphs/:id	GET, POST, PUT, DELETE
/projects/:id/graphs/:id/test-cases	GET, POST
/projects/:id/graphs/:id/test-cases/:id	DELETE

Table 7.2: Description of the main application endpoints

7.3 MongoDB data model

The data modeling process for MongoDB slightly differs from the approach for relational databases. Even for NoSQL databases, the references between entities is a necessity. Fortunately, MongoDB manual [39] provides comprehensive information about the best practices for data modeling. The key idea is to prefer nesting of the sub-documents instead of references due to the following reasons:

- The support of object-relational mapping is limited for MongoDB. Spring Data MongoDB reference documentation suggests using multiple queries to fetch required entities [40], which is an expensive operation.
- The transaction mechanism was introduced since MongoDB 4.0, while the implementation phase of this work had finished earlier. Hence, updating multiple documents at once can lead to inconsistencies.
- Nested documents are simpler to analyze and understand. As the application is an open-source project, this is especially useful for newcomers.

Listings 7.1, 7.2, 7.3, and 7.4 show JSON-like representation of the main application entities. `Project`, `graph`, and `test case` structures are split for better readability. Note that the `user` entity uses an id to reference projects. Such a design decision could be useful in the future for implementing project sharing functionality.

```
{
    "_id": "5cc64ce586aa4826279ba1d6",
    "firstName": "Ruslan",
    "lastName": "Bakeyev",
    "email": "bakeyrus@fel.cvut.cz",
    // Password is hashed for security reasons
    "password": "$2a$10$CpYC7Kj2LLTcKUH2"
 }
```

Listing 7.1: Example of `user` entity

```
{
   "_id":ObjectId("5cd56d7f86aa485c9ffae41b"),
   "name":"Test project #1",
   "description":"With description",
   // graphs attribute is omitted for better readability
   "graphs":[]
}
```

Listing 7.2: Example of `project` entity

```
{
   "id":"5cd56d8886aa485c9ffae41c",
   "name":"Test graph #1",
   "type":"FLOW",
   "nodes":[
      {
         "data":{
            "startNode":true,
            "id":"2124ab2f-b68d-4c87-b2fd-d5516cc956b5",
            "name":"a",
            "priority":"LOW"
         },
         "group":"NODES"
      },
      {
         "data":{
            "startNode":false,
            "id":"935dd698-a99b-48d8-8343-57664a0e67ef",
            "name":"b",
            "priority":"LOW"
         },
         "group":"NODES"
      }
   ],
   "edges":[
      {
         "data":{
            "source":"2124ab2f-b68d-4c87-b2fd-d5516cc956b5",
            "target":"935dd698-a99b-48d8-8343-57664a0e67ef",
            "from":"a",
            "to":"b",
            "id":"7091a264-28a6-4608-87cc-cbb196793d21",
            "name":"0",
            "priority":"LOW"
         },
         "group":"EDGES"
      }
   ],
   // testCases attribute is omitted for better readability
   "testCases":[

   ]
}
```

Listing 7.3: Shortened example of `graph` entity

```
{
   "id":"5ce335b086aa4863b4851998",
   "name":"Test set #1",
   "tdl":1,
   "algorithm":"PCT",
   "paths":[
      [
         "7091a264-28a6-4608-87cc-cbb196793d21"
      ]
   ]
}
```

Listing 7.4: Example of `test case` entity

7.4 Web server

The web server is built on top of the Spring framework, which incorporates time-tested best practices and enforces the layered architecture pattern. Figure 7.2 illustrates the UML package diagram together with the main application layers highlighted.

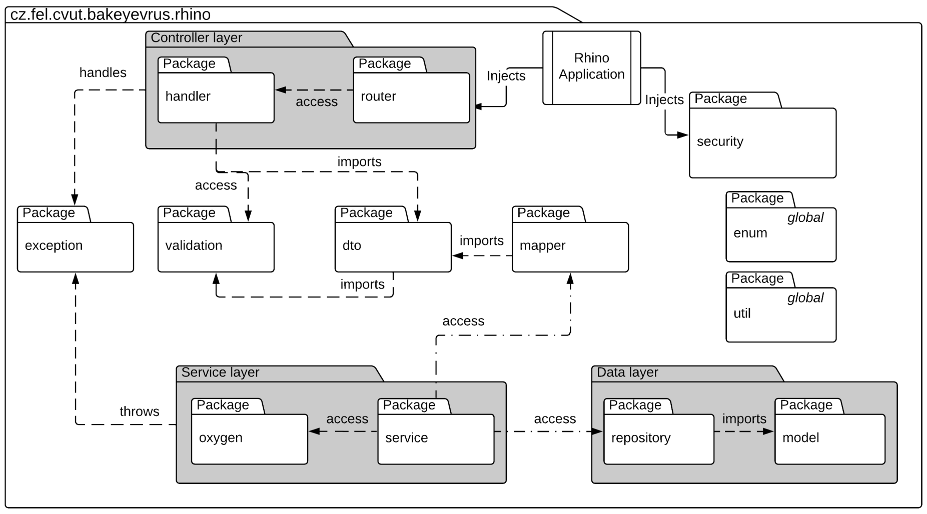


Figure 7.2: Web server package diagram

The detailed description of every package is as follows:

- `model` - contains business entities, used to represent database state.
- `repository` - contains a set of classes, used to interact with the database, e.g., retrieve or persist business entities.

- `service` - contains a set of classes incorporating application business logic.
- `oxygen` - contains algorithms for test cases generation from input graph structures.
- `mapper` - contains classes transforming business entities into the data transfer objects (DTO). The purpose of the DTO is to expose a minimal portion of the information demanded by the upper layers.
- `dto` - contains classes describing DTO, which are serialized into required data format, typically JSON or XML, during transfer over the network.
- `handler` - contains a set of classes, transforming web client requests into web server responses. Other responsibilities of handlers are:
  - DTO serialization and deserialization,
  - DTO validation,
  - mapping exceptions generated by the service layer into appropriate HTTP status codes.
- `router` - contains mapping logic between URIs and corresponding handler methods.
- `validation` - contains custom annotations, which extend the basic functionality of the Java Bean Validation standard (JSR 303: Bean Validation).
- `enum` - contains enumeration types, that are primarily used as constants.
- `util` - contains a set of utility classes, which are widely reused through the application.
- `exception` - contains exception classes, which are created by the service layer and should be handled in the controller layer.
- `security` - a set of classes, which implement fundamental interfaces of the Spring Security module. Subsection 7.4.3 contains more detailed information.

The possibility to add new types of model diagrams is one of the non-functional requirements described in table 4.1. Figure 7.3 illustrates a hierarchy of core entity classes and their interconnection. Such a detailed decomposition provides better flexibility and a capability to extend existing functionality to support new model diagrams.

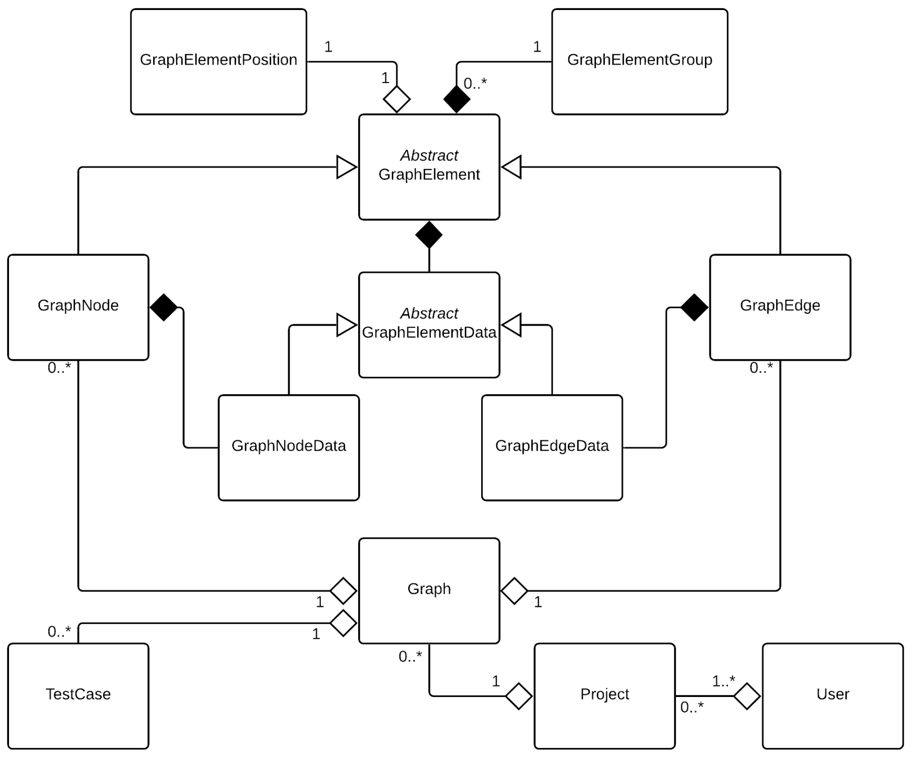


Figure 7.3: Class diagram of the `model` package

7.4.1 Integration with Oxygen library

The purpose of the `oxygen` package is to expose an interface to integrate with the `oxygenCORE` module, which contains a set of test case generation algorithms. However, the tool is tightly coupled to the mxGraph library, as shown in listing 7.5. An **adapter** design pattern has been used to abstract the end user from the difference between expected interfaces [41]. Listing 7.6 shows the definition of the `IOxygenAdapter` interface exposed by the `oxygen` package.

```
public class PCTSituationsGeneratorImpl extends
    AbstractPCTSituationsGenerator {

    public PCTSituationsGeneratorImpl(GraphModelCore model,
    int tdl, Priority defaultEdgePriority, boolean optimize) {
        // constructor implementation is omitted
    }

    // rest of the implementation is omitted
}
```

Listing 7.5: `PCTSituationsGeneratorImpl` constructor signature

```java
public interface IOxygenAdapter {

    List<List<String>> generateTestCases(Graph graph, int tdl,
     boolean optimize);
}
```

Listing 7.6: `IOxygenAdapter` interface

7.4.2 Optimizing resource intensive calculations

Another aspect, associated with Oxygen algorithms, is high-load computations dramatically affecting the overall system performance. The problem has been solved by using a **thread pool**, i.e., a set of pre-initialized threads, waiting for heavy computation tasks to reduce the load on the main application thread. Fortunately, the concept of reactive programming helps to solve such kinds of tasks avoiding the complexity of managing both the thread pool size and threads synchronization. The combination of the `Schedulers` abstraction mechanism and `publishOn` operator [42] provides the possibility to adapt to the increasing load and scale vertically, which refers to the non-functional requirement #5 from table 4.1. Project Reactor reference documentation provides additional information about parallelization capabilities.

7.4.3 Security module

An authentication and authorization concepts are built on top of the Spring WebFlux Security module. Table 7.1 shows the endpoints that the user can utilize to obtain a JWT token, that is necessary for accessing secured resources. Such an approach is known as a **token-based authentication** concept.

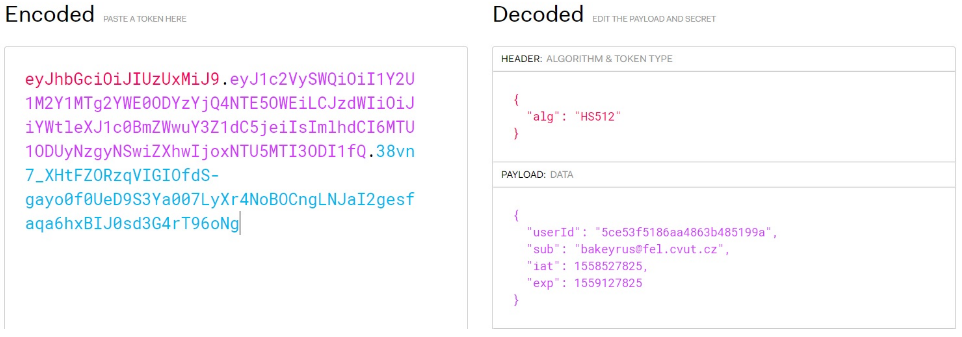
Figure 7.4 demonstrates an example of a JWT token and its based64-decoded representation. It contains necessary nonsensitive information about the authorized user, which allows reducing the number of requests to the database. The JWT token is sent as a part of the HTTP `Authorization` header and verified by the web server on each request sent by a client application. The stateless nature of the approach reduces the effort needed to scale the system horizontally. That fact satisfies non-functional requirements #5 from table 4.1.

Package `security` contains the following configuration classes:

- `SecurityContextRepository` class, which implements Spring Security `ServerSecurityContextRepository` interface;

- `AuthenticationManager` class, which implements Spring Security `ReactiveAuthenticationManager` interface;

- `SecurityConfig` class, which defines the parts of the application needed to secure, and provides concrete implementation of the interfaces mentioned above.


Encoded PASTE A TOKEN HERE
Decoded EDIT THE PAYLOAD AND SECRET
HEADER: ALGORITHM & TOKEN TYPE
PAYLOAD: DATA

```
eyJhbGciOiJIUzUxMiJ9.eyJ1c2VySWQiOiI1Y2U
1M2Y1MTg2YWE0ODYzYjQ4NTE5OWEiLCJzdWIiOiJ
iYWtleXJ1c0BmZWwuY3Z1dC5jeiIsImlhdCI6MTU
1ODUyNzgyNSwiZXhwIjoxNTU5MTI3ODI1fQ.38vn
7_XHtFZORzqVIGIOfdS-
gayo0f0UeD9S3Ya007LyXr4NoBOCngLNJaI2gesf
aqa6hxBIJ0sd3G4rT96oNg
```

```
{
  "alg": "HS512"
}
```

```
{
  "userId": "5ce53f5186aa4863b485199a",
  "sub": "bakeyrus@fel.cvut.cz",
  "iat": 1558527825,
  "exp": 1559127825
}
```

Figure 7.4: Example of JWT token generated by the application

7.5 Web client

Cytoscape.js, React, and Redux are the core libraries used in the web client. Due to the React popularity, lots of guidelines are created helping to organize the application structure. Figure 7.5 shows the hierarchy of main application modules and their interaction, used in this work. The overall communication flow between modules corresponds to the constraints defined by the Flux architecture.

The detailed description of every module is as follows:

- `components` - contains a collection of **reusable** React components.
- `hooks` - contains pieces of reusable logic that other React components can apply to reduce code duplication.
- `LoginPage` and `EditorPage` packages export React components responsible for composing the whole web page from other elements. They are usually at the top of the React component tree hierarchy as described in subsection 7.5.2.
- `actions` - contains Redux actions usually dispatched by React components connected to the store. Dispatched actions describe what is going to change in the state, and trigger the re-rendering process.
- `reducers` - contains small composable functions, which describe how the application state changes in response to the dispatched action.
- `services` - contains a set of classes, responsible for communication with the web server, i.e., sending HTTP requests, accepting HTTP responses, and handling network errors.

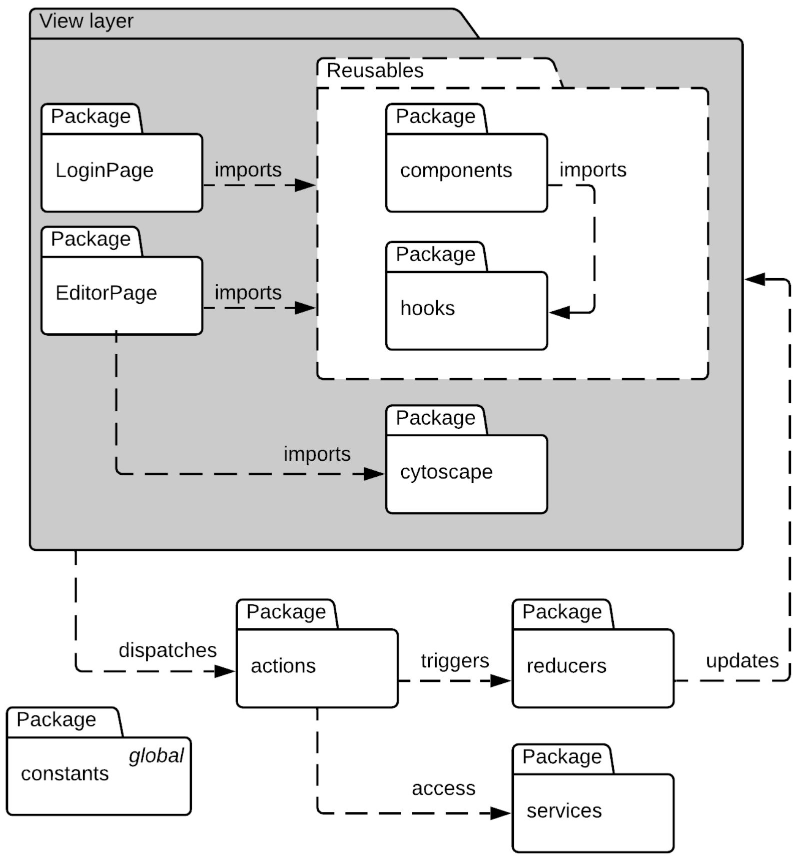


Figure 7.5: Module hierarchy diagram of the client application

- `cytoscape` - contains a set of classes responsible for the integration with the Cytoscape.js library. Refer to subsection 7.5.3 for additional details.
- `constants` - contains a set of immutable variables used through the application.

7.5.1 Organizing application state

An application state is supposed to be a single source of truth despite the library used for its management. Redux additionally enforces a set of best-practices helping to avoid common pitfalls and application bugs. Based on the guidance provided by Redux documentation, the state of the web client is split into the following chunks:

- `projects` (listing 7.7),
- `graphs` (listing 7.8),
- `testCases` (listing 7.9),

- `auth` (listing 7.10),
- `modal` (listing 7.11),
- `editor` (listing 7.12).

```
"projects": {
    "byId": {
      "5cd56d7f86aa485c9ffae41b": {
        "id": "5cd56d7f86aa485c9ffae41b",
        "name": "Test project #1",
        "description": "With description",
        "graphs": [
          "5cd56d8886aa485c9ffae41c"
        ]
      }
    },
    "activeProjectId": "5cd56d7f86aa485c9ffae41b",
    "loading": false,
    "errorMessage": null
}
```

Listing 7.7: Example of `projects` state chunk

```
"graphs": {
    "byId": {
      "5ce333f186aa4863b4851997": {
        "id": "5ce333f186aa4863b4851997",
        "name": "State machine",
        "type": "Flow",
        "elements": {
            // attribute value is omitted
        },
        "testCases": [
          "5ce335b086aa4863b4851998"
        ]
      }
    },
    "activeGraphId": "5ce333f186aa4863b4851997",
    "loading": false,
    "errorMessage": null
}
```

Listing 7.8: Example of `graphs` state chunk

```
"testCases": {
    "byId": {
      "5ce335b086aa4863b4851998": {
        "id": "5ce335b086aa4863b4851998",
        "name": "Test set #1",
        "tdl": 1,
        "algorithm": "PCT",
        "paths": [
          [
            "1ae62e62-69fc-4333-ae16-bb4a640afe22",
            "c3249941-d93e-4cde-853d-278666f37912",
            "1ae62e62-69fc-4333-ae16-bb4a640afe22"
          ]
        ]
      }
    },
    "selectedTestCaseId": "5ce335b086aa4863b4851998",
    "loading": false,
    "errorMessage": null
 }
```

Listing 7.9: Example of `testCases` state chunk

```
"auth": {
    "authenticating": false,
    "loggedIn": true,
    "errMessage": null
},
```

Listing 7.10: Example of `auth` state chunk

```
"modal": {
    "modalType": null,
    "modalProps": {}
},
```

Listing 7.11: Example of `modal` state chunk

```
"editor": {
    "lastSavedTimestamp": 1558394340524,
    "saving": false,
    "errorMessage": null
}
```

Listing 7.12: Example of `editor` state chunk

One of the most effective methods to reduce the complexity of a state tree is **normalization** [43]. The main idea is to use a hash map instead of an array to store domain objects, like graphs or projects. The nested entities are split into separate chunks and referenced using the id attribute. Such an approach helps to think of split pieces as database tables. Listings 7.7 and 7.8 show an example of normalized entities.

7.5.2 React component tree

React stores the hierarchy of components in the form of a tree, which is similar to the DOM concept [44]. Figure 7.6 illustrates the organization of React components. The state of the application is similar to the one from subsection 7.5.1. Note that the diagram shows only the components subscribed to the Redux store and significantly simplified. Nevertheless, it can be useful for new people trying to understand the implementation details of the web client.

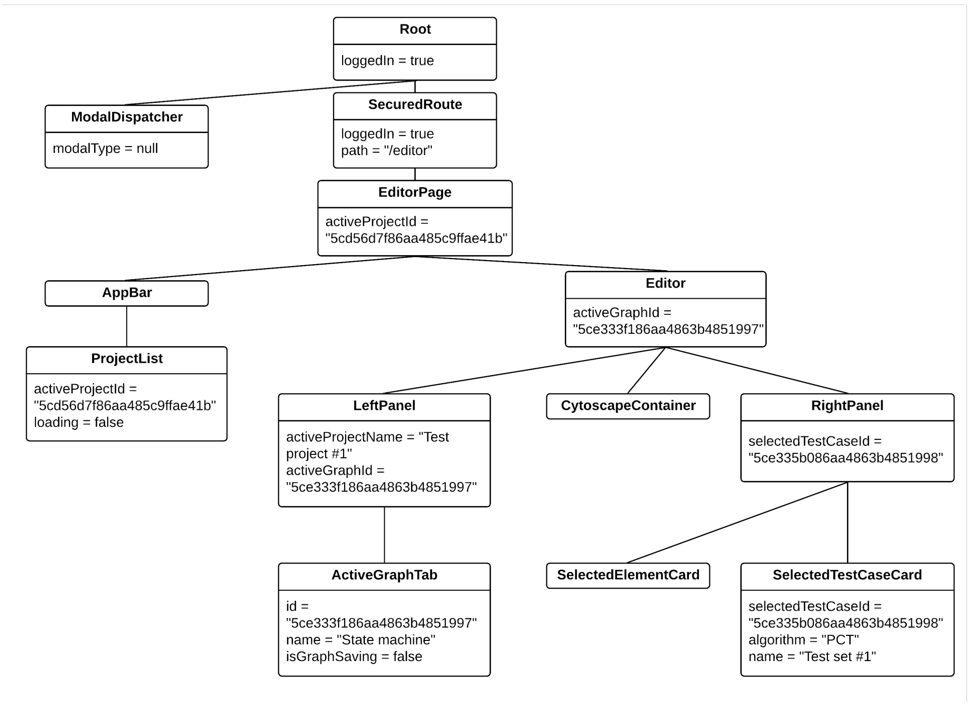

Root
loggedIn = true
ModalDispatcher
modalType = null
SecuredRoute
loggedIn = true
path = "/editor"
EditorPage
activeProjectId = "5cd56d7f86aa485c9ffae41b"
AppBar
Editor
activeGraphId = "5ce333f186aa4863b4851997"
ProjectList
activeProjectId = "5cd56d7f86aa485c9ffae41b"
loading = false
LeftPanel
activeProjectName = "Test project #1"
activeGraphId = "5ce333f186aa4863b4851997"
CytoscapeContainer
RightPanel
selectedTestCaseId = "5ce335b086aa4863b4851998"
ActiveGraphTab
id = "5ce333f186aa4863b4851997"
name = "State machine"
isGraphSaving = false
SelectedElementCard
SelectedTestCaseCard
selectedTestCaseId = "5ce335b086aa4863b4851998"
algorithm = "PCT"
name = "Test set #1"

Figure 7.6: Simplified example of the React component tree

7.5.3 Integration with Cytoscape.js library

Cytoscape.js is the Javascript library helping to create and visualize graph structures. It exposes a well-documented and easy to understand API with a broad set of features supported. However, integrating React with Cytoscape.js was not a simple task.

While Redux encourages the developers to keep an application state as a single tree in one place, including Cytoscape state into the Redux store can lead to performance issues. The reasons are as follows:

- The updates are too frequent. As an example, the element repositioning listener on canvas can dispatch a high number of actions to the store. The cost of the operation is expensive since React re-renders the component tree on each state update.
- An internal Cytoscape state is not fully serializable, that is the violation of the core Redux principles.
- State merging breaks the concept of a single source of truth in the application since the Cytoscape.js is a true owner of its part of the state.

The solution used in this work is based on state merging only in appropriate situations. An example is a graph switch action in the editor. In that case, a particular part of the Cytoscape state is cached in the Redux store and pushed to the server at the same time. If the back-end responses with error status code, the user can switch back and try to save the graph manually. Functional requirement #10 presented in table 4.2 describes other cases when state merging is needed.

The structure of the `cytoscape` package is quite complex since the module has to be easy to extend with the new types of diagrams as described in requirement 3 from table 4.1. Figure 7.7 illustrates core classes of the package and their interconnection.

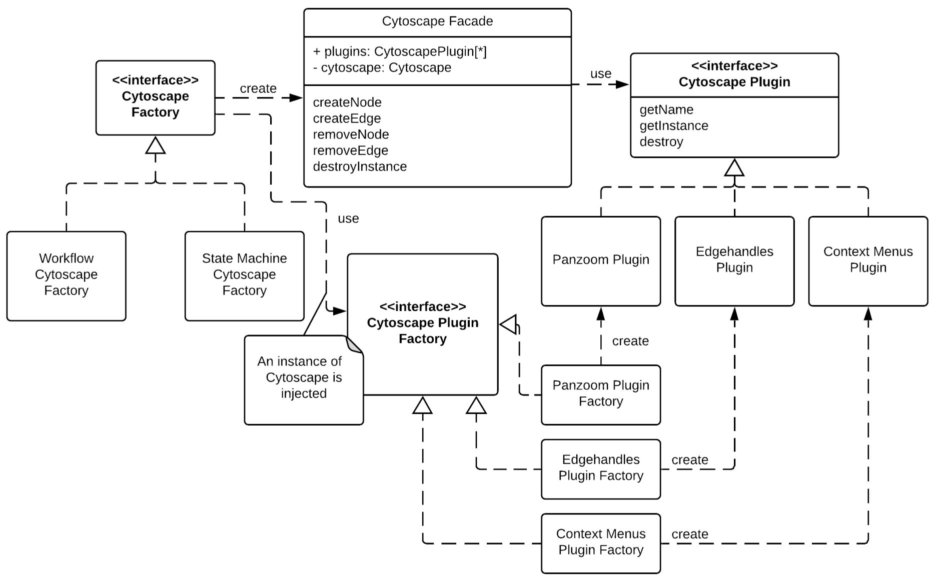

Cytoscape Facade
+ plugins: CytoscapePlugin[*]
- cytoscape: Cytoscape
createNode
createEdge
removeNode
removeEdge
destroyInstance
<<interface>> Cytoscape Factory
create
use
<<interface>> Cytoscape Plugin
getName
getInstance
destroy
Workflow Cytoscape Factory
State Machine Cytoscape Factory
use
Panzoom Plugin
Edgehandles Plugin
Context Menus Plugin
<<interface>> Cytoscape Plugin Factory
An instance of Cytoscape is injected
create
Panzoom Plugin Factory
Edgehandles Plugin Factory
create
Context Menus Plugin Factory
create

Figure 7.7: Class diagram of the `cytoscape` module

The whole concept is based on the **abstract factory** design pattern [41]. Every element of the editor, e.g., a plugin or the `CytoscapeFacade` class, should have a corresponding factory class implemented, that knows how to create an instance of the element according to the target diagram type. After that, these factories are aggregated by a parent class, which can be thought of as a factory of factories. Figure 7.8 illustrates an example flow of how `CytoscapeFacade` class is initialized together with all the required plugins (Panzoom.js and Edgehandles.js in this example) for a particular type of the diagram.

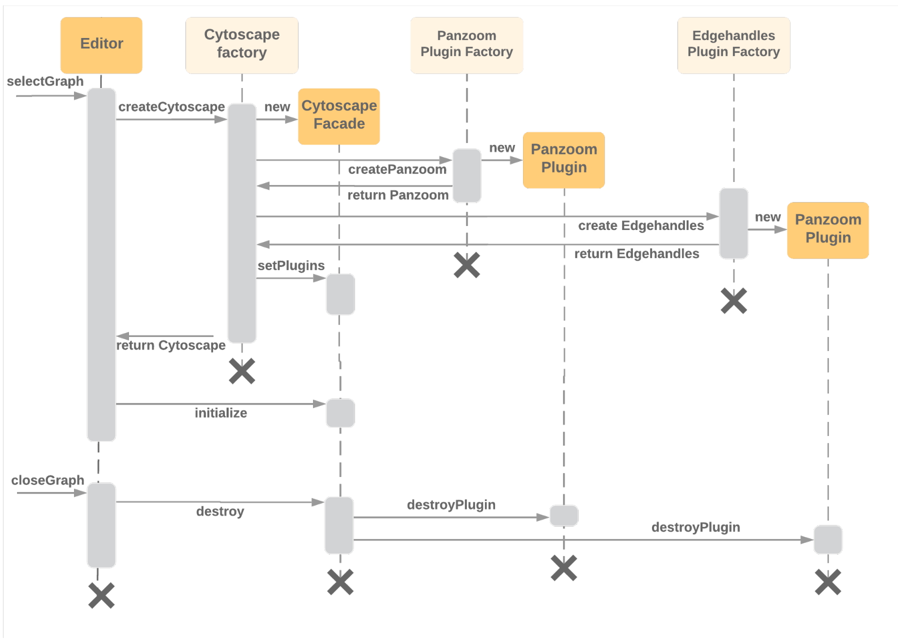


Figure 7.8: `CytoscapeFacade` class initialization flow

Chapter 8
Conclusion

The main goals of this work can be split into two parts. The former is to create a web editor, allowing to model the system under test processes and generate an optimized set of test cases based on the provided model. The latter is to describe the overall process and share the thoughts about common pitfalls that can software engineers face trying to build a highly interactive web application.

Chapters 2 and 3 explained fundamental terms from the software testing area, together with the more advanced concepts like Model-based testing or techniques, that are broadly used to derive and optimize the test cases from the SUT process.

Chapter 4 introduced application requirements, which is usually the first and one of the most important steps in the software development lifecycle process. Provided requirements describe both the functional and non-functional expectations from the system.

Chapters 5 and 6 introduce core libraries and frameworks, which have been used through the work, together with the recommendations on how to choose proper software technology. As a result, the web client is based on React and Redux stack, which allows building highly interactive systems, which is the typical web editor supposed to be. The Spring WebFlux framework is the core technology of the web server, which, on the one hand, provides great scalability opportunities, but on the other hand, it significantly increased the learning curve for new developers.

Chapter 7 is the most valuable part of the whole work, as it provides thoughts and inspiration to other people on how the application architecture, API, and data model can look like, together with the arguments in a favor of the provided solution.

Despite the presented solution solves the problem quite effectively, there are some aspects, which can be improved:

- Horizontal scaling has its limitations. The part of the application, responsible for handling intensive computations, should be scaled vertically. Any cloud PaaS solution, like Amazon Web Services, perfectly addresses such kind of tasks.

- State management synchronization algorithms, described in section 7.5.3, are quite difficult to understand. Some parts of the application logic can be implemented in plain JavaScript without using React and Redux.

- The application user interface can be significantly improved to provide a better look and increase the overall user experience.

Bibliography

[1] Miroslav Bures. Pctgen: Automated generation of test cases for application workflows. In Alvaro Rocha, Ana Maria Correia, Sandra Costanzo, and Luis Paulo Reis, editors, *New Contributions in Information Systems and Technologies*, pages 789–794, Cham, 2015. Springer International Publishing.

[2] Miroslav Bures, Tomas Cerny, and Matej Klima. Prioritized process test: More efficiency in testing of business processes and workflows. In *International Conference on Information Science and Applications*, pages 585–593. Springer, 2017.

[3] Miroslav Bures, Bestoun S Ahmed, and Kamal Z Zamli. Prioritized process test: An alternative to current process testing strategies. *arXiv preprint arXiv:1903.08531*, 2019.

[4] Anurag Dwarakanath and Aruna Jankiti. Minimum number of test paths for prime path and other structural coverage criteria. In *IFIP International Conference on Testing Software and Systems*, pages 63–79. Springer, 2014.

[5] Nan Li, Fei Li, and Jeff Offutt. Better algorithms to minimize the cost of test paths. In *Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on*, pages 280–289. IEEE, 2012.

[6] P. Ammann and J. Offutt. *Introduction to Software Testing*. Cambridge University Press, 2016.

[7] Istqb glossary. `https://glossary.istqb.org/en/search/`. (Accessed on 12/05/2019).

[8] What is continuous integration? `https://docs.microsoft.com/en-us/azure/devops/learn/what-is-continuous-integration`. (Accessed on 19/05/2019).

[9] Martin Fowler. *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. Addison-Wesley Professional, 3rd edition, 2003.

[10] Mdn web docs. `https://developer.mozilla.org/en-US/`. (Accessed on 23/05/2019).

[11] The fail-fast principle in software development. `https://dzone.com/articles/fail-fast-principle-in-software-development`. (Accessed on 13/05/2019).

[12] Tilo Linz Andreas Spillner and Hans Schaefer. *Software Testing Foundations*. Rocky Nook, 4th edition, 2014.

[13] Elfriede Dustin, Thom Garrett, and Bernie Gauf. *Implementing Automated Software Testing: How to Save Time and Lower Costs While Raising Quality*. Addison-Wesley Professional, 1st edition, 2009.

[14] E. Dustin, J. Rashka, and J. Paul. *Automated Software Testing: Introduction, Management, and Performance*. Pearson Education, 1999.

[15] Jussi Kasurinen, Ossi Taipale, and Kari Smolander. Software test automation in practice: Empirical observations. *Adv. Software Engineering*, 2010:620836:1–620836:18, 2010.

[16] Miroslav Bures. Automated testing in the czech republic: The current situation and issues. *ACM International Conference Proceeding Series*, 883:294–301, 06 2014.

[17] Dudekula Mohammad Rafi, Katam Reddy Kiran Moses, Kai Petersen, and Mika Mäntylä. Benefits and limitations of automated software testing: Systematic literature review and practitioner survey. In *2012 7th International Workshop on Automation of Software Test, AST 2012 - Proceedings*, pages 36–42, 06 2012.

[18] Miroslav Bures. Model for evaluation and cost estimations of the automated testing architecture. In *New Contributions in Information Systems and Technologies*, pages 781–787. Springer, 2015.

[19] Miroslav Bures. Metrics for automated testability of web applications. In *Proceedings of the 16th International Conference on Computer Systems and Technologies*, pages 83–89. ACM, 2015.

[20] Miroslav Bures. Framework for assessment of web application automated testability. In *Proceedings of the 2015 Conference on research in adaptive and convergent systems*, pages 512–514. ACM, 2015.

[21] Corey Sandler Glenford J. Myers and Tom Badgett. *The Art of Software Testing*. John Wiley & Sons, 3rd edition, 2011.

[22] Bruno Legeard Robert V. Binder, Gualtiero Bazzana and Anne Kramer. *Model-Based Testing Essentials - Guide to the ISTQB Certified Model-Based Tester*. Wiley, 2016.

[23] Tim Koomen, Leo van der Aalst, Bart Broekman, and Michiel Vroon. *TMap Next, for Result-driven Testing*. UTN Publishers, 2006.

[24] Miroslav Bureš, Miroslav Renda, Michal Doležel, et al. *Efektivní testování softwaru: klíčové otázky pro efektivitu testovacího procesu.* Grada Publishing as, 2016.

[25] Miroslav Bures, Tomas Cerny, and Bestoun S. Ahmed. Internet of things: Current challenges in the quality assurance and testing methods. In Kuinam J. Kim and Nakhoon Baek, editors, *Information Science and Applications 2018*, pages 625–634, Singapore, 2019. Springer Singapore.

[26] Bestoun S Ahmed, Miroslav Bures, Karel Frajtak, and Tomas Cerny. Aspects of quality in internet of things (iot) solutions: A systematic mapping study. *IEEE Access*, 7:13758–13780, 2019.

[27] Bestoun S Ahmed and Miroslav Bures. Testing of smart tv applications: Key ingredients, challenges and proposed solutions. In *Proceedings of the Future Technologies Conference*, pages 241–256. Springer, 2018.

[28] Bestoun S Ahmed and Miroslav Bures. Evocreeper: Automated blackbox model generation for smart tv applications. *IEEE Transactions on Consumer Electronics*, 2019.

[29] Ralph R.Young. *The Requirements Engineering Handbook.* Artech House Print on Demand, 2003.

[30] OWASP. *OWASP Top 10 - 2017. The Ten Most Critical Web Application Security Risks*, 2017.

[31] Which API Types and Architectural Styles are Most Used? `https://www.programmableweb.com/news/which-api-types-and-architectural-styles-are-most-used/research/2017/11/26`. (Accessed on 23/05/2019).

[32] Roy Thomas Fielding. *Architectural Styles and the Design of Network-based Software Architectures.* University of California, 2000. Available at `https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm`.

[33] Craig Walls. *Spring in Action.* 5th edition.

[34] Spring framework reference documentation. `https://spring.io/`. (Accessed on 23/05/2019).

[35] Google Trends. `https://trends.google.com/trends/explore`. (Accessed on 12/05/2019).

[36] Vue Official Guide. `https://vuejs.org/v2/guide/`. (Accessed on 23/05/2019).

[37] Flux Application Architecture For Building User Interfaces. `https://facebook.github.io/flux/`. (Accessed on 23/05/2019).

[38] Webpack Concepts. https://webpack.js.org/concepts. (Accessed on 23/05/2019).

[39] MongoDB Manual. https://docs.mongodb.com/manual/. (Accessed on 20/05/2019).

[40] Spring Data MongoDB - Reference Documentation. https://docs.spring.io/spring-data/data-document/docs/current/reference/html/. (Accessed on 20/05/2019).

[41] Ralph Johnson Erich Gamma, Richard Helm and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1st edition, 1994.

[42] Web on Reactive Stack. https://docs.spring.io/spring/docs/current/spring-framework-reference/web-reactive.html. (Accessed on 22/05/2019).

[43] Redux API Reference. https://redux.js.org/. (Accessed on 22/05/2019).

[44] React API Reference. https://reactjs.org/docs/getting-started.html. (Accessed on 23/05/2019).

Appendix A

Figures

Appendix B

Listings