Behaviour Driven Development for the ServiceNow Application Platform

Low-Code Application Development using ServiceNow and Cucumber

Bachelor-Thesis

zur Erlangung des akademischen Grades B.Sc.

Magdalena Lucreteanu

1882817

Hochschule für Angewandte Wissenschaften Hamburg
Fakultät Design, Medien und Information
Department Medientechnik

Erstprüferin: Prof. Dr. Larissa Putzar

Zweitprüferin: Prof. Dr.-Ing. Sabine Schumann

Hamburg, 27.06.2022

Contents

Abstract

Low-Code, as a development approach, is now supported by many software development platforms and requires rethinking the approach to Quality Assurance and value-based software. This paper investigates the opportunities and challenges of developing a Low-Code application using the Behaviour-Driven Development methodology. For this purpose, scenarios with concrete examples are written in a domain-specific language to describe an application's requirements. They are then used to build the application and implement automated tests to validate it. An analysis of the results reveals whether a better quality (functional suitability and maintainability) is achieved. The analysis also provides information related to the change in the development effort and how well existing Behaviour-Driven Development tools integrate with a Low-Code platform.

Zusammenfassung

Low-Code als Entwicklungsansatz wird heute von vielen Softwareentwicklungsplattformen unterstützt und erfordert ein Umdenken bei Qualitätssicherung und wertorientierter Software. In dieser Arbeit werden die Chancen und Herausforderungen der Entwicklung einer Low-Code-Anwendung unter Verwendung des verhaltensgetriebenen Softwareentwicklungsmethodologien untersucht. Szenarien mit konkreten Beispielen in einer domänenspezifischen Sprache werden geschrieben, um die Anforderungen einer Anwendung zu beschreiben. Szenarien werden dann verwendet, um die Anwendung zu erstellen und automatisierte Tests zu implementieren, um diese zu validieren. Eine Analyse der Ergebnisse zeigt, ob eine bessere Qualität (Funktionalität und Wartbarkeit) erreicht wird. Die Analyse liefert auch Informationen über die Änderung des Entwicklungsaufwands und darüber, wie gut die aktuellen Tools für eine verhaltensgetriebene Softwareentwicklung in einer Low-Code-Plattform sich integrieren lassen.

Acronyms

AAA Arrange, Act and Assert

API Application Programming Interface

ATDD Acceptance Test-Driven Development

ATF Automated Test Framework

BDD Behaviour-Driven Development

CSS Cascading Style Sheets

DOM Document Object Model

DSL Domain-Specific Language

ES ECMAScript

HTML HyperText Markup Language

ID Unique Identifier

IDE Integrated Development Environment

ISTQB International Software Testing Qualifications Board

ITSM Information Technology Service Management

JDK Java Development Kit

JRE Java Runtime Environment

JSON JavaScript Object Notation

LCMDD Low-Code Model-Driven Development

PaaS Platform as a Service

PDI Personal Developer Instance

POM Project Object Model

QA Quality Assurance

REST Representational State Transfer

SOLID Single responsibility, Open-closed, Liskov substitution, Interface segregation, Dependency inversion

STLC Software Testing Life Cycle

TDD Test-Driven Development

UI User Interface

UX User Experience

UML Unified Modelling Language

XPath XML Path Language

1 Introduction

The world of technology is entering a new age. Low-Code, as a development approach, is now supported by many software development platforms. Its main characteristic is that it brings non-technical domain experts, named Citizen Developers, into the application development lifecycle. Teams can fulfil the significant demand for application delivery using this approach.

Many Low-Code systems work with visual tools that allow people with less technical knowledge to connect specific visual components and build a mobile or web application. Users can create an application interface just by using UI components. They can model a UI application and calibrate it to data as if they were building a flow chart. They can write customized logic just as quickly as writing Excel functions. This kind of visual programming language facilitates design and integration in a rapid development manner. ServiceNow is one example of such a Low-code platform when looking at the (Gartner 2021) report.

1.1 Motivation

The survey (Dimensional Research 2019: 12) shows that most IT (77%) and business (71%) leaders report that their IT teams have many projects that cannot be implemented because of the lack of qualified resources. The need for software outpaces the available qualified developers. One key finding of the survey is that 99% of the participants think that their organization can benefit from the capabilities of Low-Code platforms. That shows a considerable industry interest in Low-Code technologies.

Citizen developers are non-technical professionals that build and deploy software applications using Low-Code platforms. They are people without any training in software development. As the report (Unisphere Research 2017: 18) regarding citizen developers shows, the lack of training brings problems. The participants in the report identified several challenges. Among the challenges is the lack of knowledge, identified by 51% of IT respondents and 33% of non-IT respondents. Lack of knowledge refers to the missing ability of the non-developers to build professional applications that have the expected quality. Applications either have too many bugs or software which did not fulfil all the requirements has been delivered.

In software engineering, Behaviour-Driven Development (BDD) is an agile process that encourages collaboration among the different roles in a software project. It encourages creating and using concrete examples, called scenarios, representing a shared understanding of how the application should behave. The scrutiny of a discovery ses-

sion for creating BDD scenarios helps the developers better understand what they build. It also reveals low-priority functionality that can be deferred from the scope of a user story. The created scenarios are then used for test automation, improving the quality of the software as bugs are discovered as soon as possible. Additionally, the BDD scenarios serve as living documentation for the produced software.

There is little academic research that reveals the fundamental challenges while testing and assuring quality in Low-Code development. This paper investigates the opportunities and challenges of developing a Low-Code application using a BDD approach.

This paper analyses the BDD process from a single person's perspective and focuses exclusively on the impact of the BDD Domain-Specific Language (DSL) on product development. It does not assess whether the BDD approach provides better communication inside a team or between stakeholders.

There are two presumptions that this paper verifies. A first presumption is the delivery of better functional suitability of an application using BDD compared to the development without BDD. A second presumption is that increased maintainability, like reduced maintenance efforts, counteracts the additional development time needed to create and automate BDD scenarios.

1.2 Structure

This work is organized as follows.

Chapter 2 addresses the main concepts for this work, including software product quality, automated testing, test-driven software development methodologies and Low-Code platforms.

Chapter 3 explains why there is a need for a BDD approach in Low-Code development.

Chapter 4 presents the ServiceNow Low-Code platform architecture, application architecture, development instance, and test automation support.

Chapter 5 analyses the features of several BDD frameworks, compares them and chooses the one that best suits this work.

Chapter 6 defines the technical prerequisites required to develop automated tests using the chosen BDD framework. It offers information on how to install all the needed tools and describes best practices for writing test code.

Chapter 7 illustrates the implementation of a ServiceNow Low-Code application using BDD and compares the results with the same application developed without BDD.

Chapter 8 presents the results of this work.

Chapter 9 provides conclusions on whether the work presumptions were achieved and gives possible solutions for encountered problems.

2 Foundations

This chapter introduces the foundations needed for developing a Low-Code application using BDD. First, an overview of software product quality and software testing is given. Then, test-driven software development methodologies are described and compared with each other. Finally, a brief introduction to Low-Code platforms is conducted.

2.1 Software Product Quality

According to (ISO 25010 n.d.) "The quality model is the cornerstone of a product quality evaluation system. The quality model determines which quality characteristics will be taken into account when evaluating the properties of a software product." ISO/IEC 25010 defines eight quality characteristics as shown in Figure 2.1.

Figure 2.1: ISO/IEC 25010 Product Quality Model (ISO 25010 n.d.)

This work focuses only on sub-characteristics of functional suitability and maintainability that are improved by the BDD process.

Functional suitability represents the degree to which the delivered software or product matches the requested functionalities. Two sub-characteristics are relevant for this work: functional completeness and functional correctness. Functional completeness is the extent to which the delivered product covers all user requirements. Functional correctness is the extent to which the delivered product provides correct results based on the user requirements.

Maintainability represents the degree to which the delivered software or product can be improved, corrected or adapted to changes, like changes in user requirements. The sub-characteristic relevant for this work is modifiability. Modifiability is the

extent to which the delivered product can be changed without degrading the product quality or introducing defects.

2.2 Software Testing

According to (Glenford 2004: 10-11) "Testing is the process of executing a program with the intent of finding errors." This definition emphasizes the difference between software developers and testers. Software development is a constructive process that builds a product, while testing is an opposite process that aims to find errors and problems with the product.

The role of testers is to add value to a product. They can do that by increasing the quality and reliability of the product and by finding and removing errors while testing the product's functionalities. An important note is that testers cannot find all bugs (Kaner 2002: 6-7) and must use their expertise to decide what and how to test as it is impossible to test all permutations (input and output combinations) of a program. Even straightforward programs can have thousands of possible permutations.

Testing of a software project can be categorized into a sequence of phases (Black 2009: 4-8): unit test, component test, integration test, system test, and acceptance test.

A unit test tests an isolated code piece like a method or a function. Developers write unit tests to test their code.

The component or subsystem test tests the system's pieces (components). The login functionality of a web application is such a component. Automated tests written for components can be reused in later phases, ex: integration or system testing.

The integration or product tests test the interaction between different system components. It is the first test phase where functionalities developed by several teams are tested together. The goal is to integrate the entire system by adding components a few at a time.

The system test tests the entire system, with all components integrated.

The acceptance or user-acceptance test is testing done to verify that the system meets the end-user requirements. They can also be categorized into alpha tests, which internal users execute, and beta tests executed by a limited number of potential customers. A beta test is executed after the alfa-test is successful. The beta test is the first phase that will involve live customer data and user scenarios.

ISTQB (International Software Testing Qualifications Board) defines the following generic types of testing (Spillner et al. 2007: 65-67): functional testing and non-functional testing. Functional testing tests that a system behaves as expected by verifying input-output behaviour based on user requirements. Functional testing is usually done during system and acceptance tests. Non-functional testing verifies the non-functional aspects of a system, like scalability, usability or security.

Testing software structure uses information about the system's internal structure (ex: code, classes, functions). The tests are designed so that they cover all the internal

structures.

Testing related to changes, also called regression testing, verifies that newly implemented functionalities did not break existing functionalities in the software.

There are two ways of executing tests: manual, where the tester manually executes the test cases and automatic, where a program automatically executes the test cases.

2.2.1 Manual Testing

In manual testing, a human (generally the tester) manually executes the tests to find bugs and defects in the application. There is no use, or little use, of automation tools or software. The tester executes the test steps described in the test case. Manual testing has been the preferred way of testing software for a long time, but that has changed, and most projects nowadays use a combination of manual and automated testing.

Manual testing has several advantages over automated testing. There are no programming skills involved, making it more accessible to testers with little or no technical background. It allows random testing without too much effort, either by using test data not defined in the test case or by checking things not explicitly documented in the test case. It allows testers to check for scenarios that are either not covered by automated testing or are very hard to automate, like correct display of user interface elements or other types of visual feedback, ex: images or videos. It is sometimes cheaper than automated testing because there is no need to write additional software.

Manual testing also has some disadvantages when compared to automated testing. It is more error-prone than automated testing because humans conduct it, and human activities are not deterministic and are more susceptible to errors. It is very time-consuming, especially if the test cases are executed regularly. The tester must manually generate reports after each successful (or unsuccessful) test case, thus increasing the consumed time. Manual testing takes up human resources that can be used for other project areas.

After weighing the advantages and disadvantages of manual testing, it is easy to see that manual testing cannot successfully cover the testing requirements of modern projects. However, it is still an important part and will not disappear.

2.2.2 Automated Testing

Automation tools/software execute the test cases in automated testing. According to (Dustin et al. 2009: 4), the definition of automated software testing is: "The application and implementation of software technology throughout the entire software testing lifecycle (STLC) with the goal of improving STLC efficiencies and effectiveness".

Testers or developers are responsible for writing test scripts to automate the execution of test cases. Tools and libraries can help them to automate faster, but in the end, automated testing is still software development. Automated testing builds on the know-how and testing techniques of a tester. They can define the test cases

that the developers will then automate. Generally, all manual testing tasks can be automated, but it is not always beneficial, as automation can be costly. Automated and manual testing are often intertwined, as they complement each other.

Test automation comes with several advantages. It is reliable because test scripts are code, and their execution is deterministic. Test execution is much faster than manual tests, as computers are faster than humans. Test automation can save time and money in a project if correctly used.

Test automation also has some disadvantages. It has a high initial cost because the test scripts must be developed. Changes to existing tests can be expensive.

There are three primary types of automation according to (Clark M. 2004: 4-6): commanded automation, scheduled automation, and triggered automation.

Commanded automation is the simplest of all automation, and it is the first step in creating the automation. A command is generally a test script and is manually executed by a developer. The computer performs some tasks each time a command is executed. A straightforward command can be changing the current directory using the 'cd' command.

Scheduled automation runs the command (test script) on a schedule at predefined intervals, like hourly or daily. Developers do not need to run the test script anymore manually.

Triggered automation runs the command when some event happens. An example of triggered automation is a build process that starts each time a file is committed into a git repository.

Practical usage of all the test automation types described above ensures successful test automation in a project. Test automation is software development because test scripts are code, and as such, it comes with all challenges associated with normal software development.

There are, however, some specific problems (Fewester 1999: 191-202) that differentiate test automation from a typical software development project.

Automated tests must be maintained, as they are software code. They cannot react to unexpected things like a human (manual) tester. An example is an unexpected field in a web application. A manual tester (human) can interpret the meaning of the field and react to it accordingly. However, an automated test does not have that logic unless a developer specifically implements it. Additionally, small changes in the application under test may break the tests, and the tests must be updated.

The number of tests to be automated depends on the application and its constraints. Each new automated test needs to be developed before it can be used to test the application. On the one hand, too many automated tests may result in high development costs. On the other hand, too few tests may not provide enough coverage.

Some tests are interdependent, which means that one test's outcome may be used as the input for another test. That creates a chain of tests that must be executed in a predefined order. A failed test in the chain means the test execution will be stopped and marked as failed. The failed test provides an incorrect assessment of

the application, as some subsequent tests will be successful, but they are marked as failed.

A machine will execute an automated test, so theoretically, the runtime of the automated tests is of no importance. However, long-running tests are harder to understand, develop and maintain. A solution is to write shorter automated tests.

There is no ideal test automation tool. All tools have advantages and disadvantages, and picking a tool must be carefully done. The selection of a tool for the automated tests must be done at the start of the project.

2.3 Test-Driven Software Development Methodologies

The test-driven software development methodologies build on existing software testing and development knowledge. Their main focus is finding potential pitfalls in requirements and reducing the number of errors before starting the actual development. This section describes the best known methodologies.

2.3.1 Test-Driven Development (TDD)

TDD is the subject of software development where developers write automated test cases to implement new features or update existing ones before writing any code. The main idea of TDD is that developers begin by writing a failing test for a specific fragment of service that they need to implement. Next, they write a clean and simple code to pass that test. After that, the new code is refactored to make sure it meets the required standards of the code.

In short, TDD is about driving development from the test. Compared to the traditional embedding of software testing, TDD first specifies the test cases before implementing the program logic.

(Beck 2003: 5f) describes TDD into five episodes, as shown in Figure 2.2:

1. Create tests for the requirements which are to be implemented.
2. These tests must fail because no functionality has been implemented yet.
3. Make changes to the code: eliminate errors and implement the requirements.
4. Run tests until they are successful: all tests must be passed here. If the test cases are failed, then the third phase has to be redone.
5. Refactor the code: Here, the code is optimized. Since the code was changed, the test cases must be rerun.

The benefit is that TDD helps ensure quality by focusing on requirements before writing the code. It assists in keeping the code clear, simple, and testable by breaking it down into small achievable steps. It builds a suite of repeatable regression tests and acts as an enabler for rapid change. It also provides documentation about how the system works for anyone coming onto the team later.

Putting TDD to work means generating many automated test cases using a test-first development approach that dictates that a unit test is implemented before new

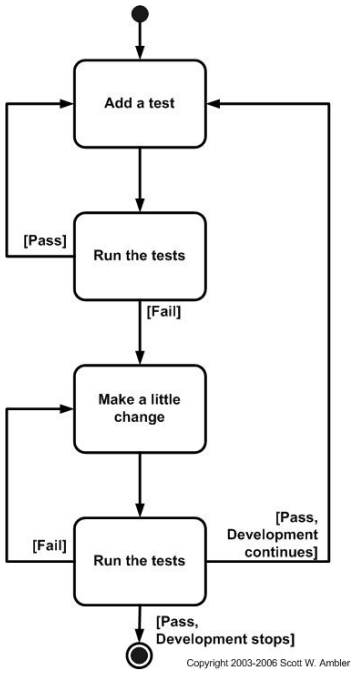

Figure 2.2: Test-Driven Development flow (agiledata.org 2003)

code can be written. It is advocated that this kind of test-first approach be applied to all types of agile software development, as the test suite succeeds in lowering regressions that appear from refactoring (Beck 2003: 10-12).

Consequently, TDD depends on unit tests and is often associated with Unit Testing. A valuable created test case must have the property of failing initially. Therefore, it can be verified that the added code is accountable for passing the test and that the test is practical (Beck 2003: 10-12).

2.3.2 Behaviour-Driven Development (BDD)

BDD, sometimes called Specification-by-example, appeared as an extension of TDD with the intent of improving the communication between developers and business-people. Using Specification-by-example, misunderstandings will be eliminated and "developers know where to start, what to test, how to name their tests and why tests failed" (skillsmatter 2009).

The BDD methodology is well known for its two practices. Firstly, it uses written examples, or behaviours, in a pervasive DSL language to illustrate how users will interact with the product. The examples ensure that everyone involved in the project understands how the product functionalities behave. The second practice is using examples as the basis for automated tests, as shown in Figure 2.3.

John Ferguson Smart explains in his book that using BDD, the process of software development begins with the vision statement. This vision is the reason for

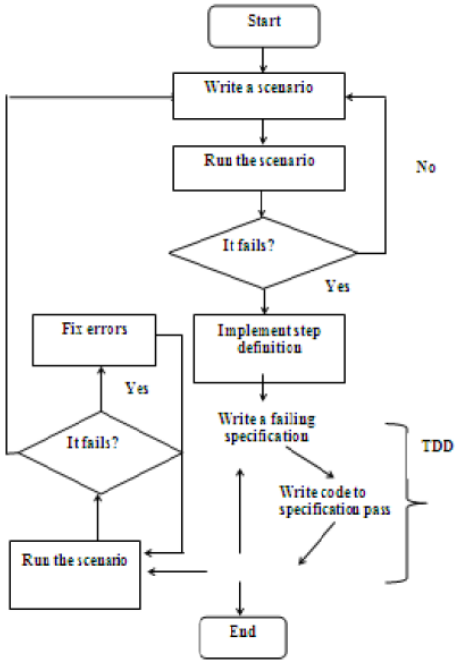

Figure 2.3: Behaviour -Driven Development flow (Myint Myint Moe 2019)

the implementation of new software. Moreover, it can be achieved by also defining reasonable business goals. Following the business goals and trying to fulfil them will correspondingly achieve value for the customer.

Building software needs the so-called feature to achieve the business goals. This feature is a fundamental functionality for the end-user and helps to satisfy the business goals.

Feature injection workshops take place at the beginning of a software project according to (Smart 2015: 108-111). All parts involved in the development process define features. This team, also called the Three Amigos, includes a Business Analyst, a Developer, and a Tester. They bring different opinions and concerns to the discussion table, resulting in the definition of a feature. Everyone understands it and meets the business goals. Illustrative user stories must be created to aid in the definition of features. The goal of a user story is to simplify and create smaller portions of a complex feature. (Smart 2015: 38-42) also recommends the practice of a pattern when writing user stories.

A user story follows a pattern **As a...I want...So that**.

<Story Title>
As a <stakeholder>
I want <something>
So that <I can achieve a business goal>

A user story is broken down into acceptance criteria. Moreover, because these can

be misinterpreted, they are enriched with examples, making it easier to understand them. These examples are called scenarios, enabling the team to investigate the various interpretations of acceptance criteria.

The process of writing automated tests happens in step definitions prior to developing pieces of software. After all scenarios and user stories are communicated, the team assesses if the features carry business value. Eventually, a test engineer and a developer turn the examples into scenarios that the product owner and the business analyst will review and approve for development. Once scenarios are formulated, they will also serve as a base for automated testing, which is the next BDD step that (Smart 2015: 180-188) describes. This way, business people can understand the scenarios, which are then integrated into automated testing. This practice is inherited from TDD.

Gherkin is an executable DSL language that enables to take the scenarios to automated testing. Gherkin is in natural language format, containing keywords like Given, When, Then, And, But. The syntax template from Gherkin can express the scenarios. It is designed to depict software in behaviour terms and make features easier to understand.

Initial State
Assumed (GIVEN) <precondition> describes the current situation, the context in which to act.
There can also be an **(AND)** used to link more preconditions.

Input
If (WHEN) <action> describes the specific action that is taken (e.g., a button is clicked, an address or resource is called).

Output
THEN <result> describes the expected result.

```
Feature: Create new account

Scenario: Successful account creation

Given I am a new user on the create new user page

When I enter a valid full name, username, and password

Then the "successful user created" page is displayed
```

Figure 2.4: Gherkin scenario example (agility.im n.d.)

The described behaviour is implemented as test code and evolves into documentation, with scenarios becoming acceptance tests and then regressions tests. Figure 2.4 shows an example of such a Gherkin scenario.

Scenarios are executed using a BDD tool that interprets the Gherkin syntax and runs the automated tests. The tests fail when the code is initially executed because

no software is yet developed. Then, the piece of software that permits the tests to pass is written. The tests will be automated and run daily, providing living documentation. This ensures that the following software parts will work after they are deployed. All tests that relate to it must pass to deliver a feature. These automated tests also serve naturally as acceptance criteria. Thanks to the test automation process, other software components can be developed by applying the same procedure. It is guaranteed that the last deployed software parts will always run as planned throughout the implementation (Smart 2015: 180-188).

BDD scenarios can be easily integrated into an automated testing tool like Cucumber or Specflow:

1. The framework reads a specification file with the scenario descriptions.
2. It translates the formal parts of the scenario's language, breaking each scenario into steps.
3. Each step is then transformed into a method for testing. The developers then implement the generated methods (step definitions).
4. The framework allows test execution and reports the results at the end. The report contains information about the scenarios that passed and those that failed.

(Smart 2015: 28ff) points out very well the advantages and disadvantages of the BDD process. Waste and costs are reduced because there is more focus on finding the right features which bring business value. Consequently, there are cost savings by reducing the effort made for fixing bugs or any other delays in building the software the customer wants. This will save time from revising code that does not meet the requirements, thus a waste for the business. Thanks to the living documentation, it is easier to accommodate changes to the software or an application. This kind of documentation generated by the executable specifications in a language all stakeholders understand makes it easier to comprehend what each feature represents, the meaning of the tests, and why they fail. The release cycle is sped up. Test automation makes it no longer necessary to run many tests manually before new releases. Therefore, more time can be invested in exploratory or useful tests. Releases may come out faster once the testing process is simplified.

BDD requires high business engagement to be productive, and optimal communication might be hard to achieve in larger companies. BDD is focused a lot on functional requirements and assumes that these will evolve once the stakeholders and the team are getting more acquainted with the project. This means the process can be shaped to the context or a certain reality. "A BDD practitioner is defined in the principles, values, and goals that the BDD community holds to be central." (Rayner 2015). Users who want to practice BDD are looking for tools and frameworks that assist the process. Sometimes, such tools can only be deceitful and often only create the illusion that the BDD process is being followed. Also, writing automated acceptance tests can be sometimes very challenging and demands specific capabilities. Tests must be thoughtfully designed and require the right level of abstraction. Otherwise, they will be hard to maintain.

2.3.3 Acceptance Test-Driven Development (ATDD)

Software should bring value to the customer; therefore, the software has to comply with customer/stakeholder requirements. ATDD is a software development methodology based on the communication between the business customer, the developers, and the testers.

Mostly, stakeholders want the functional system or software to fulfil the technical tasks. These are also called functional requirements. The software should also have non-functional requirements, for example, performance or scalability. The benefits of software should significantly exceed the costs for its creation and maintenance. ATDD embraces acceptance testing, focusing on writing acceptance tests.

The ATDD process follows these steps:

1. Select a user story
2. Write acceptance tests
3. Implement user story
4. Run acceptance tests
5. Make a small change/refactor

Acceptance tests have advanced from what was first called customer testing in the agile approach of eXtreme Programming (Beck 2003: 207-208). These automated test cases can serve as criteria for meeting customer expectations. If these test cases run error-free, the software is considered accepted. Figure 2.5 shows the ATDD development flow.

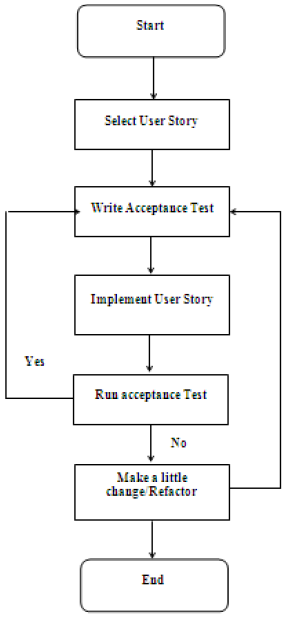

Figure 2.5: Acceptance Test- Driven Development flow (Myint Myint Moe 2019)

An acceptance test describes an expected behaviour of a software product, usually

revealed by a scenario meant for automation and documentation. These specifications are then executed with automation frameworks.

ATDD is a collaborative method of testing, bringing together different perspectives from different team members (customers, testers, and developers) to create and write proper acceptance tests which implement the correct functionality. It is a way to warrant that stakeholders understand what needs to be implemented. ATDD is an approach where customers are involved in the test design process before code may be written.

2.3.4 Comparison of the Test-Driven Development Methodologies

All three test-driven development methodologies share some common traits, but there are significant differences between them.

TDD focuses on the low level, while ATDD and BDD on the high level. (Myint Myint Moe 2019: 1-3).

TDD leans towards the developer-focused side of things. It focuses on the implementation aspect of the system and it only provides the developer with a limited understanding of what the system should do.

ATDD and BDD are the steps of making the development more focused on customers.

ATDD captures requirements in acceptance tests and uses them to drive the development. It focuses on the external quality of the software. BDD focuses on the behavioural aspect of the system and it provides a clearer understanding of the system functionalities from the perspective of both developers and customers. BDD is done in an English-like language and often with additional tools to make it easy for non-techies to understand. This permits much easier collaboration with non-techie stakeholders than TDD or ATDD.

2.4 Low-Code, No-Code Application Platforms

(Gartner 2021) estimates that by 2024, low code application development will account for 65% of all application development activity, primarily for small and medium-sized projects.

Low-Code refers to the development environment used to create a program. Software developed within such a platform is done partly or mainly via graphical user interface or drag and drop tools instead of using conventional programming languages. On the other hand, No-Code applications are created exclusively via a graphical interface: everything is done via drag & drop, and no programming language skills are required. Very few platforms are No-Code.

The Citizen Developer plays an essential role in application development and testing using a Low-Code platform. A citizen developer is the primary user of a Low-

Code Development Platform. It is a domain expert who knows what scope lays for the needed software but does not use an actual programming language and has little or no programming skills.

In 2014, the Citizen Developer term appeared especially in an environment characterized by topics such as agility and the New World. Due to the necessity to step toward the digital world, there is a great need for many different digital products. Unfortunately, the need for software programmes is bigger than the number of available developers that can fulfil the need (Dimensional Research 2019: 12).

Everyone can develop tools and participate in the task of developing digital products using Low/No-Code. More people are now involved in a process previously reserved for the development teams and specialists with more extensive Quality Assurance (QA) processes. Therefore, one can even speak of democratization of development. This kind of approach and the absence of technical IT knowledge comes with several challenges according to (Unisphere Research 2017: 18), like the development of buggier or incorrect products. This work investigates whether a BDD approach can conquer some of these challenges. Security is an additional issue when "non-specialists" create applications, but this work does not take into account that aspect.

3 The Concept of BDD for Low-Code Platforms

This chapter provides information regarding the current status of the research done for BDD as a methodology for implementing Low-Code applications.

Low-Code development follows a black-box approach, where the software component is a system whose internal workings are hidden and only external properties are visible. It is evident what the system does, but how a task is executed is unknown. Such black-box design hides the complexities of code-based software development from the user. Nevertheless, the assembled application must fit its mission. The software can break if the functionally correct components that make up the application are not assembled in a logical path. Drag-and-drop, low-code applications require that each component works with the output of the previous component. To maintain the quality of the developed product, a strategy to keep the software on track at the concept/design, implementation and QA levels is needed. Even low-code programming needs well-defined business requirements, captured in terms Citizen Developers can understand and implement.

Adopting Low-Code platforms requires rethinking the approach to QA. TDD is known for its low-level tests, which mostly do not correspond to what the business hopes to achieve. Within BDD, multiple stakeholders can work together to create scenarios describing the application's behaviour. Furthermore, BDD is a solid methodology for incorporating more software quality within low-code application development. Therefore, BDD seems an appropriate enhancement for Low-Code platforms.

Notable research relating Low-Code technologies to BDD is the work of Stephan Braams, which analyses how BDD is practised in Low-Code Model-Driven Development (LCMDD). Braams contemplates the fact that Unit Testing is the foundation of testing processes. However, observing Low-Code technology, the code units are not very easy to access because the code is not directly tested. Furthermore, a citizen developer does not seem to define priorities among features clearly and does not measure the value correctly for the client; consequently, many useless features are being built. One problem is that organizations spend half of their resources on testing and that an outcome is still software that is not what the customer requires.

Braams also mentions that low-level tests like unit and component testing and non-functional requirements like performance and security are already covered by the Low-Code platform and application architecture. Therefore, it seems justified that BDD would be a good complement to LCMDD, as BDD focuses more on functional testing, acceptance testing and exploratory testing. (Braams 2017: 2ff)

4 ServiceNow Platform

This chapter introduces the ServiceNow platform and explains why it was chosen as the Low-Code platform for this work. It then illustrates the platform and application architectures and afterwards developer instances. The last section presents the ServiceNow test automation support and highlights its shortcomings.

ServiceNow is an information technology service management (ITSM) tool capable of providing all types of services in the organization, including human resources, facilities, and project management. The reason for choosing ServiceNow as the Low-Code platform for this work is that ServiceNow has been chosen in (Gartner ITSM 2021) as a leader for the eighth consecutive year, as shown in Figure 4.1.

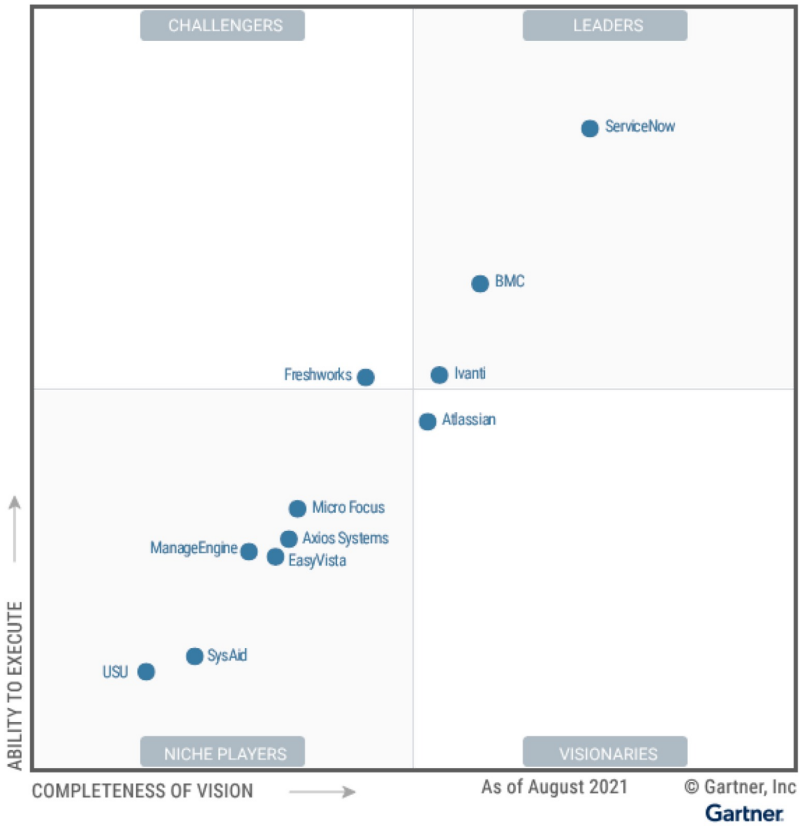

Figure 4.1: Gartner ITSM Magic Quadrant 2021

ServiceNow is a platform as a service (PaaS). PaaS is a cloud platform where a company provides hardware and software for users to use over the internet. A general

problem for customers of a PaaS is vendor lock-in. Vendor lock-in is a problem when customers depend on the solution provided by a specific vendor and cannot move to a new vendor without high costs and legal constraints. Migrating a ServiceNow application to another vendor can only be done by rewriting most of the application, making a vendor change almost impossible.

Many of the processes in different companies are similar throughout the same business branch or even between business branches. An example would be the return of an item in case it is not defective. Most companies have the same processes for such a problem. ServiceNow identified areas where the same solution can be applied for similar issues. It provides predefined solutions by delivering modules that can be used and extended. In addition, it offers various third-party application integrations with tools like Slack or Microsoft Teams. These applications can be purchased through an app store, and they help expand the platform's capabilities. Each new version of ServiceNow adds new modules or improves the existing modules.

The ServiceNow platform is continuously developed, and new versions of the platform are released twice a year, each with a unique name. San Diego is the current release at the writing of this work (Q2 2022). The following release will be called Tokyo (Q4 2022).

4.1 Platform Architecture

The architecture of the ServiceNow Platform (Figure 4.2) is based on four key concepts: single architecture, shared resources, single data model, and custom applications.

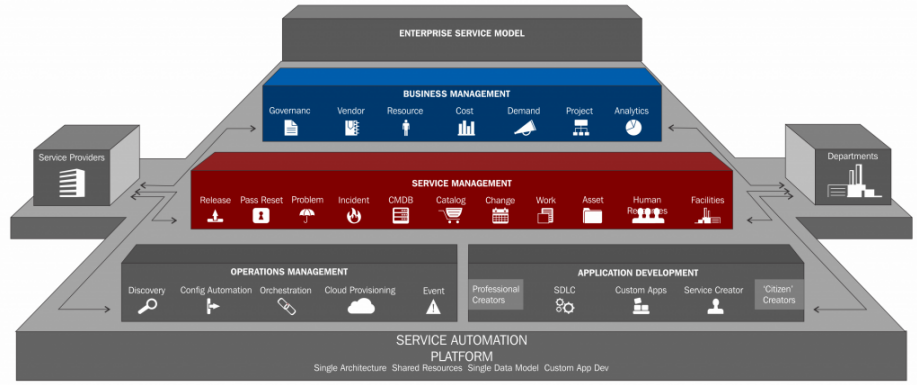

Figure 4.2: ServiceNow Architecture (Gupta 2017: 180-188)

Single architecture. Each software application in typical software development has its architecture. The development team can decide on each aspect of the architecture. Different development teams will have different architectures for their

projects, based both on the project requirements and the skill set of each team. Moving developers between applications is costly as each developer needs to understand the architecture of the new application. Also, some of those architectures may not be ideal for the problem they solve. This results in increased costs and additional development time. The ServiceNow Platform eliminates these issues by enforcing a single architecture for all the developed applications.

Shared resources. Resources are entities needed by the applications to run. Such entities can be memory or hard disk storage of computers found in a data centre on-premise or the cloud. Resources are generally organized into a group to simplify their management. Resources are allocated to an application in order for it to run. A typical application may have an application server and a database, and these require hardware (resources) to run. A shared resource is made available from one computer to another over a network. The resources in ServiceNow are shared, and all developed applications receive needed resources from a shared pool that is available for all applications.

Single data model. The single data model of ServiceNow is a best practice in data modelling and data management. It uses a shared data model with standard semantics, format, and quality standards. This model comes with many predefined tables and columns that can be directly used or extended by developers with just a few mouse clicks. Such predefined tables can be used by many applications that share similarities. New tables can be created just as quickly. Additionally, the developers can focus on creating their application without worrying about how the database is set up or how other database elements like tables or views are structured.

Custom applications. The applications are deployed on top of the ServiceNow platform as packaged software that solves business problems and processes, for example, Change or Incident Management.

4.2 Application Architecture

All ServiceNow applications and modules are designed using the same architecture. Figure 4.3 shows the architecture of ServiceNow applications.

Each application uses features of ServiceNow like UI or tables and is composed of the following artefacts: tables, UI elements, application files, integrations and dependencies.

Tables are collections of data held in a database. ServiceNow makes use of relational databases to store the data. The platform tries to remove the effort needed for the object-relational mapping so that developers no longer must maintain their database scripts and SQL statements but instead use visual tools to create their data model.

UI elements define the application's graphical user interface used to create, update, or display data. Each UI element is generally mapped to a field of a table. It is the responsibility of the developer to ensure the correct mappings.

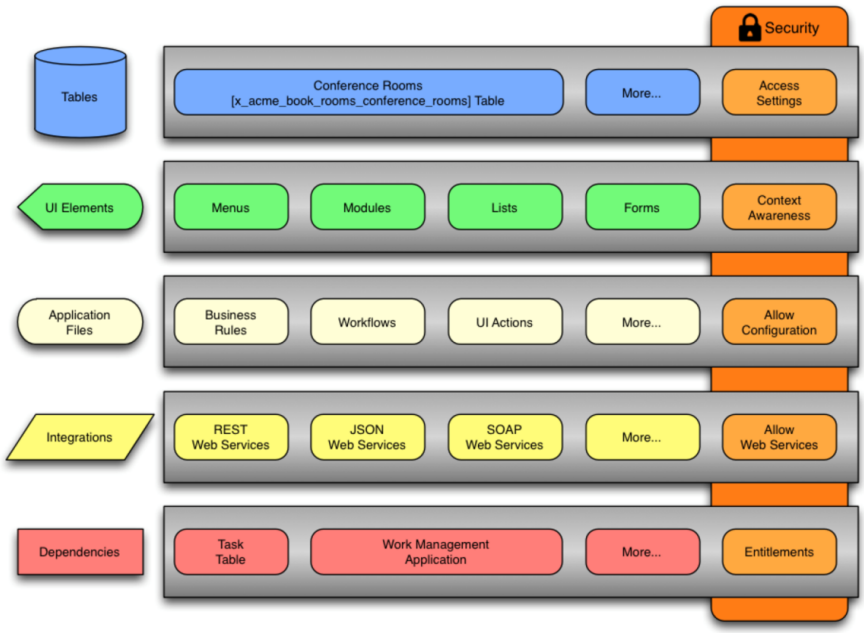

Figure 4.3: Application Architecture (Gupta 2017: 180-188)

Application files contain the application's logic, like what happens when the user interacts with a button.

Integrations allow integration with applications outside the ServiceNow platform, like the Jira project management tool.

Dependencies are other ServiceNow applications, and external ServiceNow artefacts are needed. Like the usage of a standard table that is not defined within the scope of the application but the application references it.

The ServiceNow application architecture allows the developers to create standardised applications, but that does not mean that software engineering and development, relational database modelling or UI design knowledge is obsolete. The following paragraphs describe the disadvantages of developing a ServiceNow application without the aforementioned knowledge.

Software requirements, design, construction, testing, and maintenance are software engineering tasks. They cannot be ignored, even on a Low-Code platform. The ServiceNow platform provides a standardised architecture that helps cover parts of the software design (architecture). It also offers some test automation support (described in the next section), partially covering the unit and integration testing part of software construction. Software requirements (functional and non-functional) and good test cases still have to be defined. The few developed code lines are easier to maintain if they contain no duplicates and if the variables inside the code have meaningful names.

Moreover, maintenance of ServiceNow applications plays an important part. Upgrades of an application to new versions of the ServiceNow platform must be done

regularly as ServiceNow provides support only for the two most recent release families. ServiceNow recommends doing upgrades at least once a year. There is a 7-phase plan for upgrading to a new ServiceNow version: plan, prepare, schedule upgrade, review upgrade, test and validate, remediate, and production upgrade. Upgrades can be significant projects, costing time and money, especially if there is a lot of custom written JavaScript code.

Defining tables in ServiceNow can be done without relational database modelling knowledge as the platform provides a lot of predefined tables that can easily be extended. Nevertheless, this results in poorly designed tables. Data is then inconsistent or ambiguous. It is scattered over several tables, is duplicated or is too complex. The quality of the application suffers from a poorly designed database.

UX knowledge is essential for creating an intuitive and easy to use UI. Creating UI components without UX knowledge results in a poor user experience, making it difficult for users to use the application.

4.3 Personal Developer Instance (PDI)

A PDI is a sandbox used by registered users to develop ServiceNow applications. License information and activation instructions for a PDI are described in Appendix A. Developers can test their applications on a PDI without impacting production or other non-production instances. The PDIs are free to use but have their limitations so that the ServiceNow Developer Program can save resources and provide active members with free instances. Only one PDI can be started for an account. The instance goes into hibernation mode when it is not used for an amount of time. That means the database and the application server are stopped, but all the data is preserved. Such data can be created in applications, tables, or data inside the database. Signing in a hibernating instance will wake it up. An instance will be deleted after ten days of inactivity, and the ServiceNow platform will make a backup. A user can restore the instance by using the backup after more than ten days of inactivity, with the mention that the URL of the application will change.

4.4 Test Automation Support

ServiceNow offers developers two ways to automate an application: write unit tests in JavaScript for the UI framework components and make use of their automated testing framework (ATF) for functional tests.

Custom UI components written by developers must be tested, and one way to do it is by writing JavaScript unit tests. However, it is not always possible to write those tests in ServiceNow, and the platform recommends avoiding them and using functional tests instead. The default unit testing framework in ServiceNow is Jest, a JavaScript tool focusing on simplicity. Jest integrates very well with UI frameworks

like Angular, React or VueJs. It requires little to no configuration, can run tests in parallel for better performance, has an intuitive, well-documented API and provides code coverage reports. The Jest fluent API (an API that uses method chaining to provide concise and elegant code) makes tests easy to write and understand, see Listing 4.1.

Listing 4.1: Simple Jest test

```
test('three plus three is six', () => {
  expect(3 + 3).toBe(6);
});
```

A particular feature of Jest that is very useful for ServiceNow testing is mocking. A unit test aims to focus only on a small piece of code and test it in isolation. The piece of code under test is not written in isolation but is part of a more extensive software, and it can have dependencies on other objects. It is impractical to instantiate all external dependencies of a piece of code, and this is where mocking comes into play. Mocking replaces these external dependencies with other objects that mimic their behaviour. A simple example would be mocking calls to a database by creating a mock object that does not interact with the database. Mocking in ServiceNow unit tests can be used to mock specific components, like in Listing 4.2 that replaces the now-icon component view with an empty one.

Listing 4.2: Jest mocking example

```
jest.mock('@servicenow/now-icon', () => {});
```

Additionally, ServiceNow has its in-house testing framework for functional testing, built into the Automated Testing Framework (ATF) platform, based on the low-code / no-code paradigm. From the architectural point of view, ATF is just a standard ServiceNow application that gives developers tools to write and execute automated tests on a ServiceNow non-production instance and is free to use inside the ServiceNow platform. ATF aims to reduce development time and improve the overall quality of an application by replacing manual testing with automated testing. The functional testing provided by ATF allows the creation or deletion of records on the data level, setting values in the UI components, and checking the test results.

The main advantage of ATF is that testing is done from within the platform and without the need to learn additional tools. The tutorials and presentations of the ATF on the ServiceNow platform show the main functionalities of the testing tool, together with best practices. They are focused on introducing citizen developers to automated testing as mainly ATF test design is done with little to no development knowledge. ServiceNow's best practices for writing tests include using a standard naming convention, writing small and self-contained tests, writing validations as much as possible, using parameterized tests or organizing tests into test suites.

ATF is still in development and receives significant upgrades with each new version of the ServiceNow platform. Essential features for automated testing like parameterized or parallel testing were introduced just a few years ago. For example, the

Madrid release added parameterized tests in Q1 2019, and the New York release added parallel testing in Q3 2019. Parameterized tests are used to execute the same test with different test data. A straightforward example would be testing the login functionality for different users. The logic of the test remains the same: open the login page in a browser, enter the username in its UI field, enter the password in its UI field and then click on the button for login. The result would be either a successful login or an unsuccessful login. Without parameterized tests, the same test logic would have to be implemented for each separate user, generating much redundant code. Parameterized tests can rerun the same test using different users and remove much redundancy. Parallel testing allows the execution of different tests in parallel, resulting in a faster execution time. The runtime of automated browser tests can be high. It will impede more extensive applications as any change in code needs to be validated by executing long-running tests before being deployed in production. Running two tests in parallel can half the execution time of tests, and running eight tests in parallel can reduce possible runtime from 8 hours (a full working day) to just one hour.

Another issue with ATF is the test coverage. Test coverage is a software development metric defined as a percentage that measures how much of the source code of a program is run during the execution of a test suite. Several code coverage criteria include function coverage, statement coverage, branch coverage, and predicate coverage. Function coverage verifies whether functions/methods/procedures have been executed. Statement coverage confirms whether each line of code/statement of the program has been executed. Branch overage verifies whether each edge in the program's control-flow graph has been executed at least once. Predicate coverage verifies whether each Boolean condition has been evaluated as true or false.

The values for test coverage vary between 0% (nothing has been covered) to 100% (everything has been covered). A predicate coverage of 100% means that all Boolean conditions in a program have been tested, and each was evaluated at least once true and at least once false. 100% is an outstanding value for all code coverage criteria, but it is not always obtainable as some unique errors cannot be tested using regular automated tests. Figure 4.4 illustrates an example of a coverage report generated by JaCoCo, a tool for Java code coverage. It shows code coverage for different packages of the JaCoCo code base, with values between 58% for the 'examples' package till 98% for the 'ant' package.

ATF does not provide a way of measuring the test coverage as defined above. That is how many lines of code have been covered by the tests. One challenge is the low-code approach of the platform, with the application mainly being developed using drag and drop. All the code coverage criteria defined above (function, statement, branch, or predicate coverage) works on code and cannot be directly used for a ServiceNow application.

ServiceNow uses the term Test Coverage but in conjunction with the number of executed tests. That is essential information regarding the execution status of the tests, showing how many tests there are, but it does not provide any information

JaCoCo

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
org.jacoco.examples		58%		64%	24	53	97	193	19	38	6	12
org.jacoco.core		97%		93%	112	1,410	116	3,376	20	715	2	138
org.jacoco.agent.rt		76%		82%	32	122	69	319	21	74	7	20
jacoco-maven-plugin		90%		80%	37	192	46	414	8	116	0	23
org.jacoco.cli		97%		100%	4	109	10	275	4	74	0	20
org.jacoco.report		99%		99%	4	572	2	1,345	1	371	0	64
org.jacoco.ant		98%		99%	4	163	8	429	3	111	0	19
org.jacoco.agent		86%		75%	2	10	3	27	0	6	0	1
Total	1,372 of 27,573	95%	153 of 2,187	93%	219	2,631	351	6,378	76	1,505	15	297

Figure 4.4: Example of a coverage report generated by the JaCoCo tool (jacoco n.d.)

about the quality of the tests, like how much of the application they check and can lead to parts of the application being completely untested. The Test Management dashboard (Figure 4.5) displays either bar or pie charts illustrating the test case distribution by execution status: passed, failed, blocked, in progress, not executed.

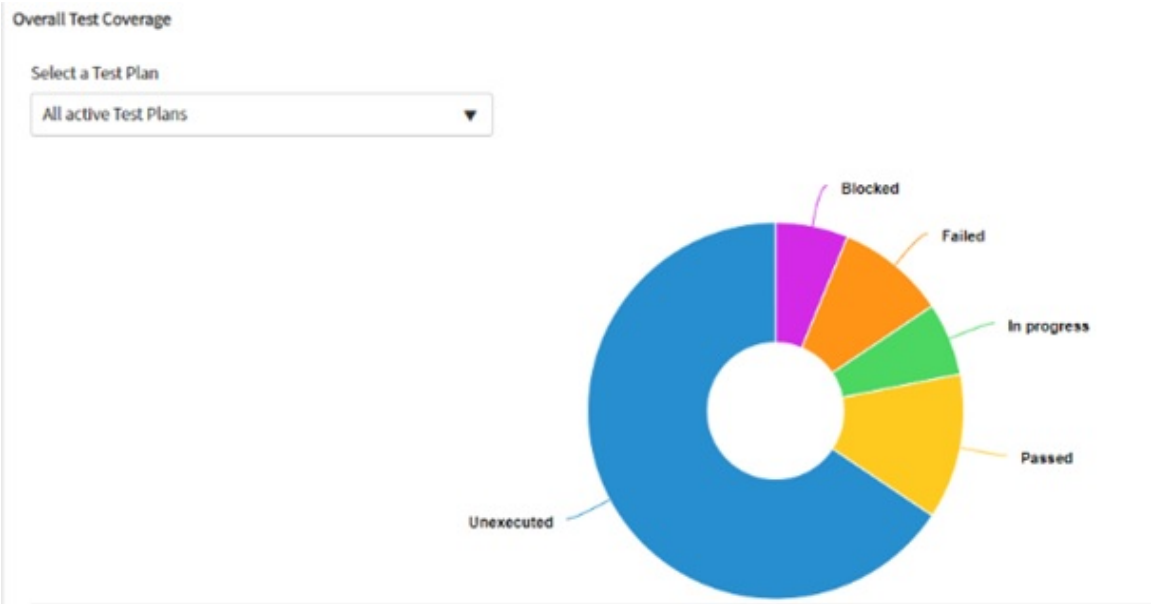

Figure 4.5: Test Coverage pie chart in the ServiceNow Test Management dashboard (docs.servicenow.com 2022)

One major issue of the ATF in earlier releases was the reusability of tests. It was impossible to reuse test steps from another test in another test. The only solution was to copy the whole test into a new one. That violates the DRY (do not repeat yourself) principle, which aims to reduce repetitions in code by abstracting to reduce redundancy. Violating the principle generates much redundant code and increases the maintenance effort and complexity. The problem has been fixed just recently in the Quebec release (Q1 2021) by adding the possibility to reuse tests.

5 BDD Frameworks

This chapter introduces some of the most used BDD automation frameworks: Cucumber, SpecFlow, Behave and Serenity. It then evaluates and compares them with each other to select one that best suits this work's needs.

5.1 Cucumber

Cucumber is one of the most used BDD testing frameworks. It was initially written in Ruby as a companion tool for the RSpec BDD framework. RSpec is a domain-specific language testing framework that can test Ruby code.

Cucumber evolved to support more than 15 programming languages, including Java, Ruby, JavaScript, C++, Go, Kotlin, Python, PHP, and OCaml. It provides specific implementations for each of the supported languages. The implementations can be official, semi-official, unofficial, and unmaintained. Official implementations are hosted on the official GitHub page of Cucumber. Semi-official implementations are hosted in other locations and use components from the official Cucumber. Unofficial implementations are hosted in other locations and do not use components from the official Cucumber. Unmaintained implementations are official, but they are no longer further developed because of the lack of developer maintainers. It is safe to use official or semi-official implementations for a project as they are maintained, and they all use official Cucumber components. Unofficial and unmaintained implementations may be helpful in some use cases, especially if they support a specific programming language not available in the official or semi-official implementations.

Figure 5.1 shows how Cucumber executes scenarios.

Cucumber has two licensing plans: open-source and premium.

The open-source version is free to use and contains all the main functionalities of Cucumber. It has cross-platform implementations for different programming languages. The specifications are kept in text files, called feature files and are written using the Gherkin language. Cucumber reads the files and executes the test scenarios against an application. It generates test reports in different formats: HTML, JSON or user-defined. Cucumber can be used with other libraries, like Selenium, for browser automation. It integrates itself with different editors and IDEs, like Atom, Visual Studio Code or Eclipse IDE and can run tests in continuous integration tools like Jenkins.

The premium version (Cucumber Studio) contains the open-source version. It includes additional tools aimed at team collaboration: Git integration, displaying

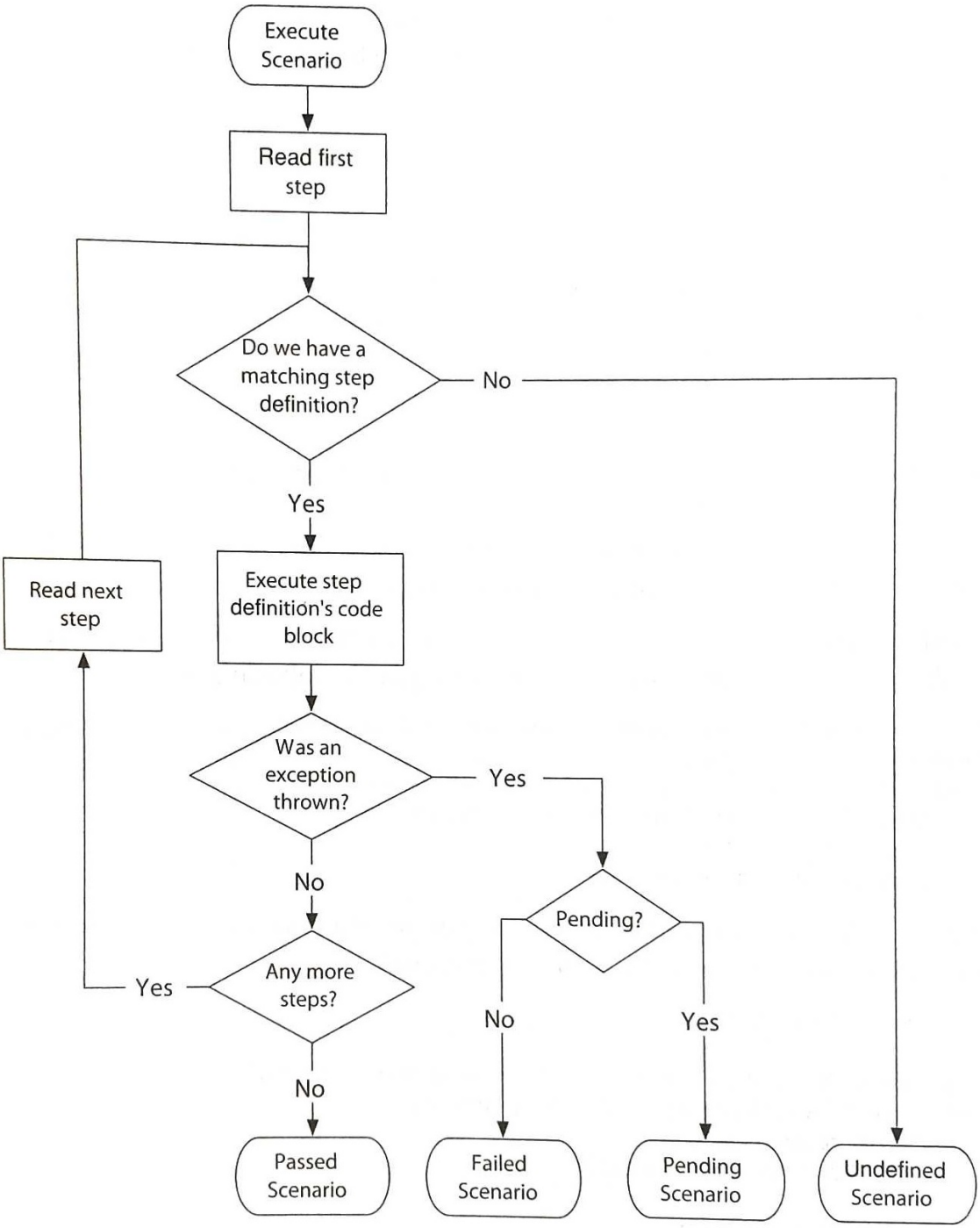

Figure 5.1: Cucumber workflow (Wynne et al. 2012: 53)

the specification files as rendered documentation, and a dedicated editor that offers auto-complete (Figure 5.2) or continuous integration tools for agile test management.

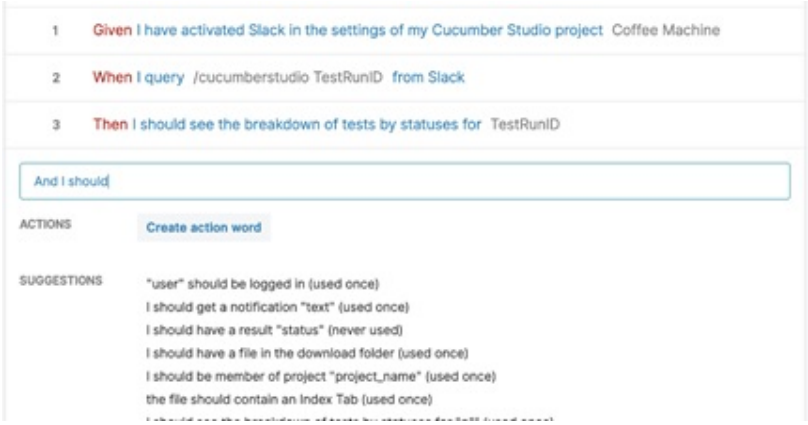

Figure 5.2: Autocompletion feature of Cucumber Studio (cucumber.io n.d.)

Cucumber, especially the premium version, is designed to involve technical and non-technical stakeholders in writing acceptance tests. One such helpful feature is the option to write feature files in the language of the stakeholders by inserting a '#language' comment as the first line of a feature file. Cucumber currently supports over 70 spoken languages.

Listing 5.1: Gherkin example in the German language

```
#language de
Funktionalitaet: Artikel suchen
  Szenario: Einen Artikel suchen
    Angenommen die Hauptseite wurde geoeffnet
    Wenn ich ein den Artikel 'Hose' suche
    Dann sollte eine Liste mit mehreren Eintraegen angezeigt
```

The Listing 5.1 showcases how the stakeholders write their test specifications in the German language. Cucumber uses by default the English language for its feature files when no '#language' comment is present.

5.2 SpecFlow

SpecFlow is an open-source BDD framework for .NET. It was originally created as a porting of the Cucumber framework for the .NET platform and supported only C# as a programming language. SpecFlow has similar functionalities to the open-source version of Cucumber as it keeps the tests in text files written in the Gherkin language. It can be used with other .NET libraries, integrates itself with Visual Studio and Visual Studio Code and can run tests in continuous integration tools.

SpecFlow uses the official Cucumber Gherkin parser and supports Cucumber in over 70 spoken languages.

Despite the similarities, SpecFlow and Cucumber are two different tools. SpecFlow has more extensive Hooks and a dedicated runner SpecFlow+ but lacks the support for different programming languages.

According to (Gamma et al. 1995: 328) hooks "provide default behavior that subclasses can extend if necessary. A hook operation often does nothing by default." In SpecFlow they are steps executed as part of the setup (before) or teardown (after) of the test. SpecFlow allows the configuration of before and after hooks on different levels: features, scenarios, steps, or execution, while the Cucumber hooks may be limited by implementing the specific programming language. It is possible to specify the order for hooks of the same type, as shown in Listing 5.2.

Listing 5.2: Hook order in SpecFlow

```
[BeforeScenario(Order = 0)]
public void LoginUser() {  ...  }

[BeforeScenario(Order = 100)]
public void PlaceOrder() {  ...  }
```

The Order variable contains a number that indicates the order in which scenarios are executed. The hook with the lowest order is executed first. In the example above, LoginUser is executed before PlaceOrder because 0 is smaller than 100.

SpecFlow+ Runner is a test runner for SpecFlow that integrates directly with Visual Studio and replaces general test runners with a dedicated solution that further improves productivity by adding parallel test execution, retry of failed tests and the possibility to run scenarios with different configurations. The development of SpecFlow+ Runner was discontinued in January 2022, according to the developers (specflow.org 2022). The complexity of integrating SpecFlow with the .NET ecosystem increased significantly in the last few years, and the development team does not have enough resources to continue supporting the SpecFlow+ Runner support.

SpecFlow+ LivingDoc is a functionality that generates documentation from automatically validated scenarios to share with the team or other stakeholders.

5.3 Behave

Behave is an open-source BDD framework for python. Like Cucumber and SpecFlow, Behave uses the Gherkin language to define the tests, stores the test definitions in text files and supports different spoken languages.

Behave contains a built-in feature not available in Cucumber and SpecFlow: fixtures. The purpose of a fixture is to ensure that the tests are run using a predefined environment and data so that the test results are repeatable. This is generally ensured during setup/cleanup procedures of tests. Behave goes further by introducing fixtures that simplify the setup/cleanup tasks.

Listing 5.3: Example of a setup fixture in Behave

```
@fixture
def browser_firefox_setup(context, timeout=30, **kwargs):
  context.browser = FirefoxBrowser(timeout, **kwargs)
  yield context.browser
```

The Listing 5.3 defines a Behave fixture that sets up the Firefox browser for the test execution. The fixture for cleaning up the resources (closing the browser) after the text execution looks is shown in Listing 5.4.

Listing 5.4: Example of a cleanup fixture in Behave

```
@fixture
def browser_firefox_cleanup(context):
    context.browser.shutdown()
```

5.4 Serenity

Serenity BDD is an open-source library whose aim is to help developers write acceptance tests faster and more reliable and use these tests to provide living documentation of the application. It supports Java as a programming language. Serenity provides support for both web testing and REST API testing. Web testing uses Selenium, while REST API testing uses RestAssured, a library designed to test HTTP requests.

Serenity does not implement a BDD language as Gherkin but provides integration with BDD tools like Cucumber or Behave. It allows the developers to use their testing libraries and provides additional functionalities and patterns.

Two core features of Serenity are the Step Libraries and the Screenplay Pattern.

Steps are an essential principle of Serenity. Steps are the abstraction between the high-level BDD scenarios and the code that interacts with the application under test. They are used to hide the complexity of different parts of a test. Each test is comprised of reusable steps.

The Screenplay Pattern uses SOLID design principles for automated tests. SOLID is an acronym for five object-oriented design principles that establish practices to make the developed software more maintainable and flexible. The five principles are (martin 2000: 4ff): single-responsibility principle, open-closed principle, Liskov substitution principle, interface segregation principle and inversion of control principle. The Screenplay pattern uses the single-responsibility principle (every class or function is only responsible for a single part of a program) and the open-closed principle (the behaviour of an entity can be modified without changing its code).

The Screenplay Pattern in Serenity can be integrated with both Cucumber and regular JUnit tests. It uses actors that perform tasks and verify results. An actor in software engineering is an element that interacts with the system and can be a physical person, like a customer, an organization or another system. The Screenplay

Pattern using BDD has the following structure: an actor can do a task, and then when the actor performs the task, the actor sees the results of her actions and can verify whether the result is as expected.

Serenity generates living documentation from test reports (Figure 5.3). Living documentation is documentation that is always up to date with the development state of the software. Serenity uses the test specifications that are always up to date to generate the documentation.

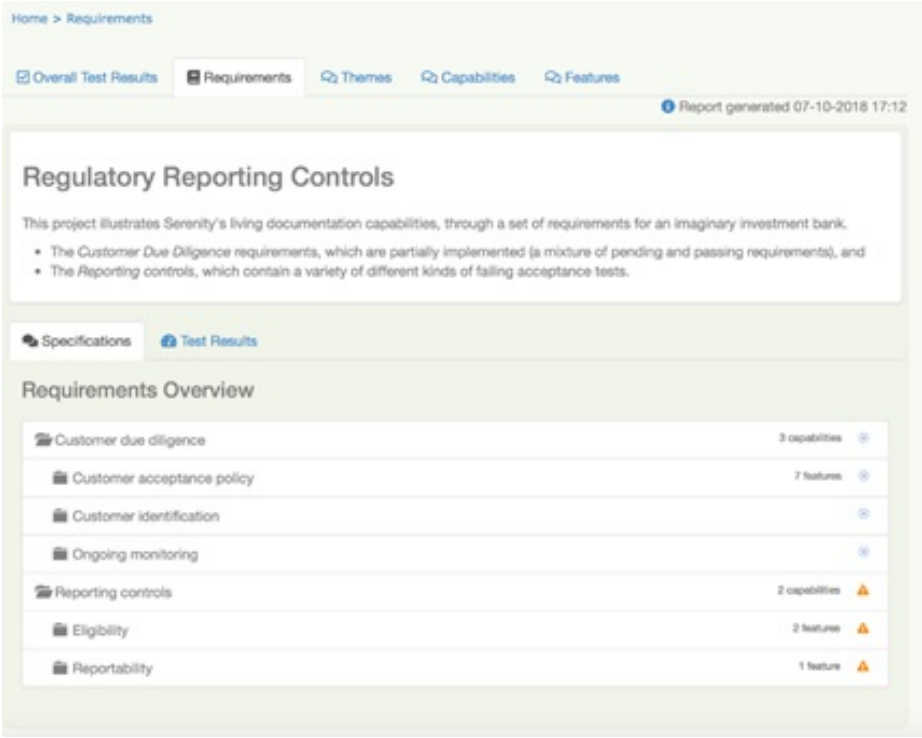

Figure 5.3: A test report generated by Serenity (github.io n.d.)

5.5 Evaluation of Frameworks

When choosing a BDD framework, it is essential to evaluate all the options and choose the best one that suits the requirements. The minimal requirements of a BDD tool for this work are: to be open-source and support the Gherkin language (features/scenarios).

The following criteria were considered for this work:

1. License model, whether the framework is free or has additional costs associated with it

2. Features/Scenarios, whether the features and scenarios are described in a DSL language like Gherkin

3. Living documentation, whether the feature scenarios are kept in separate files and can be used as living documentation of the application under test

4. Supported programming languages, that means which programming languages can be used for the implementation of the features/scenarios

5. Reusable steps show whether the same step can be reused without the need to reimplement it

6. Setup/Teardown steps are essential for preparing the data and configuration needed by the test

7. Custom reports whether the user/developer can create their customized reports

8. Automated execution, whether the BDD scenarios can be automatically executed

10. Integration with other tools shows which other tools can be used together with the BDD framework without additional integration effort, like automation frameworks, continuous integration tools or IDEs

Table 5.1 shows the evaluation of the BDD frameworks based on all the aforementioned criteria.

Feature	**Cucumber**	**Spec Flow**	**Behave**	**Serenity**
License	Open-source / commercial	Open-source	Open-source	Open-source
Living documentation	Yes	Yes	Yes	Yes
Features/Scenarios	Yes	Yes	Yes	Yes
Supported programming languages	Java, Node, Ruby, OCaml, C++, Lua, Kotlin, Scala, Tcl, Go, C#, .NET, PHP, Python, Perl	C#, .NET	Python	Java
Reusable steps	Yes	Yes	Yes	Yes
Setup/Teardown	Yes	Yes	Yes	Yes
Custom reports	Yes	Yes	Yes	Yes
Automated execution	Yes	Yes	Yes	Yes
Integration with other tools	Yes	Yes	Yes	Yes

Table 5.1: BDD Framework Evaluation self made

All the analysed BDD frameworks have all the critical functionalities implemented. The decision of which one to use is based on particular requirements, like the integration with a specific tool or the programming language. C# and Python developers will choose SpecFlow or Behave, while Java developers can choose between Cucumber and Serenity.

I decided to use Cucumber with Java for this work as it fulfils all requirements and is the most known and used BDD Framework.

6 Technical Prerequisites for Implementing BDD Tests

Before implementing a ServiceNow application using BDD, a few technical prerequisites need to be fulfilled. Cucumber with Selenium WebDriver for Java is needed to write the BDD scenarios and develop the Java code to automate these scenarios. This chapter shows how to install and use these tools. Then it describes best practices for writing test code and practices used for the application implementation in the next chapter.

6.1 Cucumber with Selenium WebDriver for Java

Some software must be installed/set up locally to use Cucumber with Selenium WebDriver for Java: Java, Maven, Eclipse, Cucumber, Selenium WebDriver, GeckoDriver using WebDriverManager and a Shadow DOM library. Installation instructions for each of the aforementioned tools are described in Appendix B.

Java

Java is a programming language and a computing platform developed at Sun Microsystems, Inc. in the early 1990s and released in 1995. The platform's objective is to allow programs to run on different platforms without the need to modify the source code or the compiled libraries. Java is a high-level object-oriented programming language.

A JRE (Java Runtime Environment) is needed to run Java programs on a computer. A JRE is an implementation created to provide an execution environment for Java programs. It is used only to run Java programs and not develop Java programs. Most computers have already installed a JRE version, and there are separate installers for different operating systems.

A JDK (Java Development Kit) is used to develop Java applications. A JDK contains a JRE and additional tools needed to develop Java programs, like a compiler, an interpreter or an archiver.

Maven

Maven is an open-source build tool and package manager developed by the Apache Group. It allows developers to set up and build projects, add new java libraries

or other dependencies to the project, update libraries and their dependencies, run unit tests and create reports. Maven POM (Project Object Model) files to store the information related to the project and its dependencies. A POM file is an XML file, generally called pom.xml. Maven is entirely written in Java and requires an installed JDK to run.

Eclipse IDE & Cucumber Eclipse

Eclipse is an integrated development environment (IDE) written in Java. It is mainly used for developing Java applications. However, it has support for many other programming languages, like C, C++, JavaScript, Groovy, Perl, PHP, Python, and R. Installation instructions are provided on the Eclipse page.

Eclipse IDE recognizes Cucumber feature files as simple text files and does not provide by default support for Cucumber development. This can be fixed by installing the Cucumber Eclipse plugin from the Eclipse Marketplace. This plugin provides syntax highlighting and warning messages for Cucumber feature files, making development faster and easier.

Cucumber

Creating a new Cucumber project with maven can be done using the maven cucumber-archetype. A maven archetype is a project templating toolkit. That is a template project from which developers can create projects of the same type. The cucumber-archetype allows the developer to set up a maven project for development with Cucumber. It automatically creates the maven pom.xml file containing the required dependencies, like JUnit and Cucumber, and the expected folder structure.

The successful creation of the Cucumber project can be verified by switching to the project folder in a terminal and executing "mvn test". This command will attempt to start the Cucumber tests found inside the project. Successful execution will show the information "Tests run: 0, Failures: 0, Errors: 0, Skipped: 0Tests run: 0, Failures: 0, Errors: 0, Skipped: 0" as the newly created project does not have any tests yet.

Selenium WebDriver

Selenium is a collection of three tools that simplifies website automation: WebDriver, IDE, and Grid. Selenium WebDriver is a web framework that uses browser automation features to run automated tests and control browsers. Selenium IDE is an integrated development environment that helps the development of tests. Selenium Grid is a tool that simplifies the running of tests on multiple browsers and operating systems.

Selenium WebDriver is available for several programming languages, including Java, Kotlin and JavaScript and can be used to run Cucumber tests.

Using the Selenium WebDriver is done in three steps: create a new WebDriver object and instantiate it for the type of browser it must start and control, open the

web page to test in the browser and then find HTML elements on the page and execute commands on them, like clicking on a button or entering text in a field. The Listing 6.1 shows these steps.

Listing 6.1: How to create and use the Selenium WebDriver

```
WebDriver driver = new FirefoxDriver();
driver.get("https://www.selenium.dev");
driver.findElement(By.id("navbarDropdown")).click();
```

Gecko Driver

A browser engine is a software that interprets HTML, CSS, JavaScript, and images and renders web pages in a browser. Gecko is a browser engine that supports internet standards and is used in browsers and email clients. Mozilla developed it, and it is the standard engine of the Firefox browsers.

The GeckoDriver is a native software that acts as a proxy between the Selenium WebDriver tests and the browser. The Selenium WebDriver cannot start the browser without it. Older versions of Selenium were able to run tests without the GeckoDriver, but the native implementation of Firefox has been removed in Selenium 3 to improve compatibility.

WebDriverManager

Manually downloading and setting up the environment variable for the GeckoDriver is not ideal, as this process must be done on each machine where the tests are executed. Furthermore, each operating system requires its native version of the GeckoDriver and a different way of setting the environment variable. A better idea is to have the GeckoDriver be automatically managed, like how maven manages the java packages. That solution is called WebDriverManager, an open-source library that does exactly that.

Setting up the GeckoDriver using the WebDriverManager is done with one line of code (Listing 6.2) that has to be executed before all tests.

Listing 6.2: Setup the GeckoDriver with WebDriverManager

```
WebDriverManager.firefoxdriver().setup();
```

Shadow DOM (Document Object Model)

ServiceNow uses its JavaScript framework to create the applications. That framework is called Now Experience UI Framework, based on web components standards. The World Wide Web Consortium introduced web components as a framework that simplifies the Web development with reusable components. Web components provide a standard component model for the Web and allow developers to create custom

elements and use them in web apps. The key features of web components are encapsulation (mechanism of restricting access to data from outside) and interoperability (the ability for a software to communicate with other software). There are four main specifications for the web components: custom elements (JavaScript API for creating user-defined elements), Shadow DOM (JavaScript API for changing DOM elements), ES modules (for reusing JavaScript documents) and HTML template (for markup templates that will not be displayed on the page).

The ServiceNow applications extensively use the Shadow DOM. A DOM represents a web document as nodes and objects, and programs can use it to change the document's content, style, or structure and are displayed in a browser window. Shadow DOM is a further encapsulation level of a DOM that completely isolates a portion of the HTML document and improves reusability and portability. The architecture of the Shadow DOM poses challenges for the test automation with Selenium, as the current Selenium implementation does not have support for Shadow DOM. Selenium WebDriver cannot locate elements in a Shadow DOM as the Shadow DOM allows the Web document to have hidden DOMs attached to other elements. Figure 6.1 illustrates the structure of the Shadow DOM.

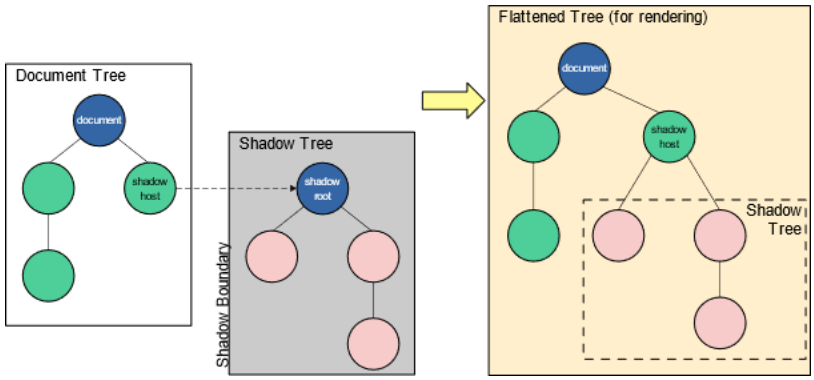

Figure 6.1: ShadowDOM structure(mozilla.org n.d.)

The main Document Tree (main DOM) contains Shadow Host nodes. These are normal DOM nodes that Selenium can correctly locate. The Shadow Tree is the actual DOM located in the Shadow DOM, and it has a Shadow Root as the root node and its own Shadow Boundary that isolates it from the rest of the regular DOM. A developer can manipulate a Shadow DOM, appending, deleting children, or modifying CSS styles like any regular DOM.

There are two ways for a developer to make Selenium be able to work with Shadow DOM: extending Selenium to support the ShadowDOM functionality or using an existing library that already does that.

I chose to use an existing open-source library called Shadow Automation Selenium for this work. It allows Selenium to work with Shadow DOM by wrapping the WebDriver in a new Shadow class. The new class has methods that allow locating Web

elements using CSS selectors. The Listing 6.3 shows how to create and use the library to locate a text field by title and get its contents.

Listing 6.3: How to find a WebElement using Shadow

```
Shadow shadow = new Shadow(webDriver);
WebElement element = shadow.findElement("input[title='User']");
String username = element.getText();
```

6.2 Best Practices for Writing Test Code

BDD scenarios are defined in a domain-specific language like Gherkin, and they serve as a specification for the application to be developed. They will be then implemented as automated tests using any BDD framework like Cucumber or Behave. The implementation requires a developer/tester to write the code for the test. Principles of clean code (Martin 2013: 124-127) and software engineering apply to the automated tests too. Such principles include: easy to understand code, no duplicates, explanatory variables or class names, easy to change and extend classes, small functions and generally following the rule of keeping it simple.

Some additional guidelines, practices, and test design patterns can be used to improve the test code, like the AAA (Arrange-Act-Assert) Pattern or Page Object Pattern.

6.2.1 Arrange-Act Assert (AAA) Pattern

The AAA (Khorikov 2020: 42) is a general pattern used for unit tests. It structures the test code in three parts, as its name suggests: arrange, act, and assert. The arranging part deals with preconditions needed to execute the test, like preparing the test data, creating variables, or opening a browser window. The acting part is the execution part of the test. The assert part verifies that the software behaved as expected. The parts of the AAA pattern are like the given/when/then steps of a BDD scenario, and the implementation of a BDD test follow the AAA pattern. The Listing 6.4 shows a simple unit test that is written following the AAA pattern.

Listing 6.4: A test method implementation using the AAA pattern

```
@Test
public void shouldAddNumbers() {
  // arrange = given
  int a = 2;
  int b = 3;
  // act = when
  int c = summ(a, b);
  // assert = then
  assertEquals(5, c);
}
```

6.2.2 Page Object Pattern

The Page Object is a design pattern used to reduce code duplication and improve the test maintenance in Web/UI test automation. It was first described by Martin Fowler in 2004 under the name Window Driver (martinfowler.com 2004) as a pattern that provides a programmatic API to drive and interrogate a UI window. The name Page Object became popular together with the Selenium library and remained in use, replacing the original Window Driver name. UI tests need to reference elements in an application to set data in a field, click on a button or verify the contents of a field. Accessing the UI elements directly from the tests required will make the tests easier to break when UI changes, as the functionalities to access the UI are spread across several tests. Furthermore, the code will be duplicated as many tests access the same fields. The effort for the maintenance of the test code will increase over time. A page object is used to wrap an HTML page or part of it into a reusable code element. Tests can use page objects without knowing how a specific HTML page is internally built and if the UI changes, only the page object must be updated.

A typical automated test that does not use the Page Object pattern looks like Listing 6.5.

Listing 6.5: Implementation of a test method without using Page Objects

```
@Test
public void testLogin() {
  // enter user name
  driver.findElement(By.id("user_name")).sendKeys("user");
  // enter password
  driver.findElement(By.id("user_password")).sendKeys("password");
  // login
  driver.findElement(By.id("login")).click();
  // verify expected behaviour
  ...
}
```

The above code uses Selenium WebDriver to log in a user in the application by entering the username and password in their fields and then clicking on the login button. It then verifies the expected behaviour using assertions. This approach works well if only one method executes a login, but that will not be the case as probably lots of tests will have to log in the user before using the application. Furthermore, there is no separation of concerns between the test method and the locators/finders for the UI elements.

The UI functionalities can be extracted in a separate class by using the page object pattern as in Listing 6.6.

Listing 6.6: Login Page Object

```
public class LoginPage {
  protected WebDriver driver;
```

```
  private By usernameBy = By.name("user_name");
  private By passwordBy = By.name("user_password");
  private By signinBy = By.name("login");

  public LoginPage(WebDriver driver){
    this.driver = driver;
  }

  public LoginPage loginUser(String userName, String password) {
    driver.findElement(usernameBy).sendKeys(userName);
    driver.findElement(passwordBy).sendKeys(password);
    driver.findElement(signinBy).click();
    return new LoginPage(driver);
  }
}
```

All the UI related code is now encapsulated in a class that can be reused in different tests. The test in Listing 6.5 can be now rewritten as in Listing 6.7.

Listing 6.7: Implementation of a test method using Page Objects

```
@Test
public void testLogin() {
  LoginPage loginPage = new LoginPage(driver);
  loginPage.loginUser("user", "password");
  // verify expected behaviour
  ...
}
```

Page objects do not always model a complete web page but parts of a page that can be extracted as components. An example would be a page object for a search field that can be part of several pages. The search field can be viewed as a modelled component in its page object to avoid code duplication.

Page objects generally do not contain verification code, as that is the purpose of the test code. However, they can check whether the UI elements they use have been loaded or displayed on the page. Without those required UI elements, the test code will fail.

6.2.3 Locators and Finders

A usual problem of UI tests is locating elements inside a web page. That is a fundamental aspect that frameworks like Selenium try to solve by providing locators and finders. Locators are ways to identify elements of a page by parsing the DOM. Finders use the locators to locate the elements.

Selenium has several types of built-in locator strategies: class name, CSS selector, id, name, link text, partial link text, tag name, and XPath. The Listing 6.8 shows examples for each locator strategy.

Listing 6.8: Locators in Selenium

```
// locates elements whose class name contains button-class
By classNameLocator = By.className("button-class");
// locates elements using the given CSS selector
// that means elements of type div who have the specific
// style display:block
By cssSelectorLocator =
By.cssSelector("div[style='display:block']");
// locates the element with the id username
By idLocator = By.id("username");
// locates the element named password
By nameLocator = By.name("password");
// locates elements of type link whose displayed text
// matches my-link
By linkTextLocator = By.linkText("my-link");
// locates elements of type link whose displayed text
// matches partial-link and returns the first one found
By partialLinkTextLocator = By.partialLinkText("partial-link");
// locates elements base on the xpath expression
// an image whose alternative text is Bunny
By xpathLocator = By.xpath("//img[@alt='Bunny");
```

Most of the locators are self-explanatory. There are, however, two that deserve special attention: the CSS selector locator and the XPath locator.

Cascading Style Sheet (CSS) selectors are patterns used to locate elements to style in an HTML page. They are highly versatile, and almost any HTML element can be located using a CSS selector. There are several CSS selectors: basic selectors, grouping selectors, combinators, and pseudo. Some basic selectors are the universal selector for selecting all elements or restricting them to a specific namespace, the type selector for selecting elements with a given node name, the class selector for selecting all elements with a given class, the ID selector for selecting all elements with a given ID or attribute selector for selecting all elements with a given attribute. Grouping selectors are a grouping method for different selectors. Combinators select elements based on the parent/child/sibling relationship of elements inside the DOM. Pseudo allows selecting elements not in the HTML or based on their state information, like whether that element was visited/accessed by the user. Selenium offers direct locators for IDs, names, or class names so that the user does not need to write her CSS selectors for them.

XPath (XML path language) is a syntax that defines parts of an XML document. XPath uses a path very similar to how a computer file system looks, as displayed in Figure 6.2.

XPath uses the concept of nodes to navigate through a tree type document. There are several types of nodes: element, text, attribute, namespace, comment, document, and processing-instruction. Each node has relations with one or more nodes. The relation types are parent (each element has only one parent), children (each element may have zero or more children), siblings (elements with the same parent), ancestors

Figure 6.2: Folder structure looks like XPath structure (w3schools.com n.d.)

(the parents of the parent element) and descendants (the children of children elements). Atomic values are nodes without children or parents. Additionally, XPath uses operators that can add, subtract, or multiply values or check whether an element is equal, less than, or greater than a given value. XPath contains over 200 built-in functions that can be used and is the most complex and extensive way of locating elements.

Selenium 4 introduced the concept of relative locators that can be used when building a locator for the desired element is complex. It is easier to locate another element whose position relative to the original element is easy to identify. The relative locator strategies are: above (for an element located above a reference element), below (for an element located below the reference element), left of (for an element located left of the reference element), right of (for an element located right of the reference element) and near (whenever the relative position is not obvious or cannot be identified by any of the other strategies). The near strategy uses a maximum of 50 pixels to the reference element as the rectangle for its locator.

The Listing 6.9 shows how to identify the password field for an element of type input (text field) located above the login button.

Listing 6.9: Relative locator for a field above another

```
By passwordLocator = RelativeLocator
  .with(By.tagName("input")).above(By.id("login"));
```

Similarly, the login button can be identified based on the password field because it is an element of the type button, located below the password field, as in Listing 6.10.

Listing 6.10: Relative locator for a field below another

```
By loginLocator = RelativeLocator
  .with(By.tagName("button")).below(By.id("password"));
```

It is also possible to chain relative locators for more complex needs. The Listing 6.11 finds an element of type button that is located above the about element and to the left of the submit button:

Listing 6.11: Complex relative locator

```
By cancelLocator = RelativeLocator
  .with(By.tagName("button")).above(By.id("about"))
  .toLeftOf(By.id("submit"));
```

The same element can be located with Selenium by using different locator strategies. Even if a locator strategy works, it does not mean that it is best suited for a particular element. For example, the XPath strategy can locate any HTML element. However, that flexibility comes at a cost, the performance of the XPath locators is not optimal, and its syntax is complicated to debug. In general, searching by IDs is the fastest and most predictable way to locate an element on a web page, as little to no DOM traversal is done. Most applications do not have IDs for all elements, and in that case, CSS selectors are preferred. Finally, the XPath strategy should come into play if there is no way to identify the element using a CSS selector. The link text and partial link text locators are restricted only to links. The tag name locator is not a good way to locate elements as there may be several elements having the same tag inside the page. As a rule of thumb, the locators should be as compact and precise as possible, and DOM traversals should be avoided for performance reasons. The performance is not a problem for a small application with only a handful of UI tests but can be a significant bottleneck for an extensive application with thousands of UI tests.

A finder receives the argument returned by a locator as input and uses it to locate elements. A finder can either return on more elements if it is successful or throw an exception if the element could not be found. Selenium finders can find the first matching element or all matching elements. The results can be further refined to identify the elements of an element or the active element. A finder can either evaluate the whole DOM and search the element inside it or just evaluate a subset of the DOM by using a specified element as the root of the subset. The Listing 6.12 shows how to use finders in Selenium.

Listing 6.12: Finders in Selenium

```
// finds the element with class name user-class by
// evaluating the entire DOM
WebElement userElement =
  driver.findElement(By.className("user-class"));
// finds the element with id name by evaluating the DOM
// subset for the userElement
WebElement nameElement = userElement.findElement(By.id("name"));
// finds a collection of elements of type button
List<WebElement> buttonElements =
  driver.findElements(By.tagName("button"));
```

The Selenium WebDriver throws an exception when the element identified by the provided locator could not be found. One cause for this error is that the element does not exist on the page. Or the element exists, but the user is not allowed to interact with it. That happens when the element is found inside the DOM but is not visible.

6.2.4 Other Considerations

Each test has to be independent, which means it does not depend on the execution of other tests, and the order in which tests are executed is not relevant.

Each test starts with a clean and defined state. Starting a new browser (instantiate a new WebDriver) for each test ensures that no cookies or temporary data are shared between the tests.

Sometimes the same test must be executed with different input data. A wrong way of doing this would be to duplicate the test. Instead, data-driven tests must be used. Data-driven testing is a technique where the test data is stored outside a test, usually in a table-like structure, and allows the same test to be executed for each data entry in that table. Cucumber scenarios come with built-in functionalities for data-driven testing.

UI tests are executed in a browser, and the automated tests interact with HTML elements. Each action of the test (clicking on an element or setting data in an element) can be associated with wait times where the test must wait until the page or some specific part of the page is loaded. One of the most common examples is logging into a page. The user enters her username and password and clicks on the login button. A new page will be displayed after a while on a successful login. Human users use their eyes and brains to check that the new page has been loaded. An automated does not know that a new page is expected, and it must be programmed to do that. The easiest way is to tell the test to wait for a predefined time before subsequent operations. Java offers the Thread.sleep() method that can wait for several milliseconds. That approach is to be avoided as it blocks the test for the whole time if the sleep method is over. Selenium offers a much better way of dealing with the wait times with its WebDriverWait class, as shown in Listing 6.13.

Listing 6.13: WebDriverWait in Selenium

```
// creates a new WebDriverWait with a maximum duration of
// 10 seconds that waits till either the URL of the current
// page contains /user or 10 seconds have passed
// if the condition is not met after 10 seconds
// then an exception will be thrown
new WebDriverWait(driver, Duration.ofSeconds(10))
  .until(ExpectedConditions.urlContains("/user"));
```

7 Low-Code Application Implementation using BDD

This chapter showcases the development of an example ServiceNow application for managing a company's employees' vacations. First, it defines the business goals and the corresponding epics.

The application development is done in iterations, with each iteration covering a user story. Only two iterations (user stories) are presented, as a complete application implementation is out of the scope for this work. Each iteration describes the user story and its acceptance criteria. A version of the Vacation application without BDD is initially developed based on those. Afterwards, the Cucumber scenarios/features are defined in an analysis session. The initial implementation of the Vacation application is verified against the Cucumber scenarios and updated if needed. Finally, the automated tests based on these scenarios are written using Selenium.

7.1 Define the business goals

In order to determine the business goals for an application, the problem to be solved has to be identified. Then the project vision, goals and capabilities are specified. Features, stories, acceptance criteria and examples are then defined before starting the coding of the application. Figure 7.1 illustrates how to understand the business in the form of a pyramid. The pyramid is traversed from top to bottom, starting with a vision and finishing with delivering the code of the resulting application.

Problem

The first step in defining the business goals is to identify the problem to be solved.

The current process of requesting and approving time-off from work, either vacation or illness days, is done mainly via email and manually administrated through spreadsheets. This approach has several disadvantages:

1. It results in a lack of an overview, like who is on vacation and for how long.
2. Not all time-off requests are processed in time or even processed.
3. The overall process lacks transparency, and all parties involved (employees and managers) are not happy.

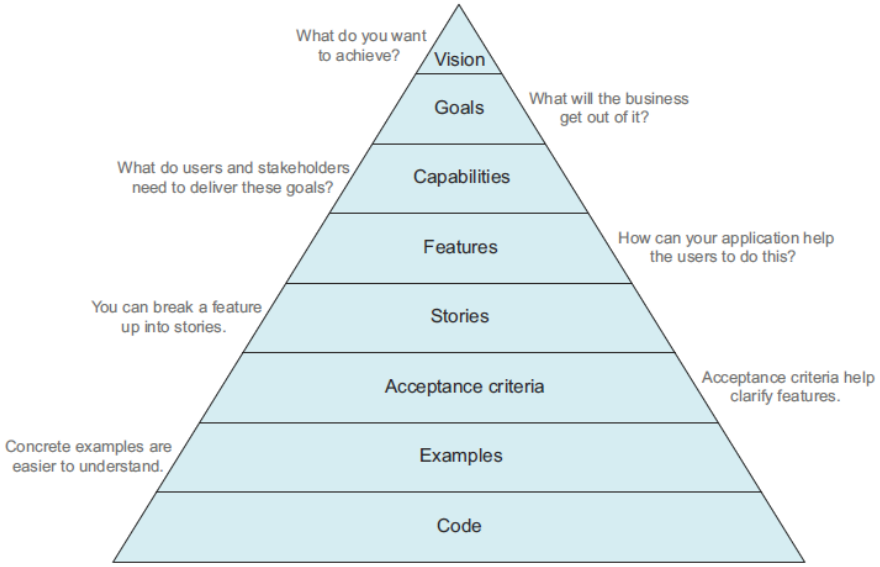

Figure 7.1: Features and code map the business goals and vision (Smart 2015: 66)

Project Vision

The next step is to define a project vision based on the above problem.

The project vision is a short statement describing what the project wants to achieve. The vision is generally written using a FOR/WHO/THE/IS/WHAT template:

```
FOR <customers>
WHO <need something>
THE <product> IS <something>
THAT <benefits customers>
```

The project vision statement for the time-off project is:

```
FOR employees
WHO want to optimally manage their time off/vacation time
THE Vacation Application IS an automation process
THAT lets employees quickly and transparently manage their
  vacation plans
```

A project vision can also contain additional information that defines what makes the product different from the competition so that the customers choose it. Considering that internal users will use the application in the discussion, people outside the organisation cannot use it. Market competition is of no importance in this case.

Solution

There is a need for a solution that solves the described problem. The solution is an automated system for scheduling and managing employee vacations and time off.

Business Goals

Automation of the vacation requests workflow will speed up the handling of requests. It also improves transparency and reporting by allowing managers to search and display time off data.

The users of the system and their benefits of using the system have to be identified. These benefits will define the business goals. Afterwards, epics can be derived from the business goals.

The application has two types of users: employees and managers, each with their own benefits.

The employees will have the following benefits when using the new system:

1. can request time off or a vacation
2. can visualize and modify time off requests
3. can see the history and status of time off request (approved/declined)

The managers will also have their benefits:

1. are automatically notified of time off requests
2. can see a list of open time off requests
3. can approve/decline a request
4. can generate additional reports

Automation of the vacation requests workflow will speed up the handling of requests. It also improves transparency and reporting by allowing managers to search and display time off data.

Features (Epics)

The name Feature is a synonym for Epic in this context and should not be confused with the Cucumber feature files, which contain scenarios for specific uses stories.

The following epics define the application:

1. Vacation Portal. This Epic defines the application employees use to request time off and track their vacation times.

2. Vacation Workspace. This Epic defines the application managers use to manage employees' vacation times and generate different reports.

3. Users and Roles. This Epic is needed to set up the portal and workspace applications infrastructure.

7.2 Iteration: Users and Roles

The first iteration deals with creating initial versions of the Portal and Workspace landing pages and defines both pages' user/roles (security).

User Story: Setup employee and manager roles (Epic: Users and Roles)

As an administrator

I want to set up roles for employees and managers

So that they can log in to the portal and workspace

Acceptance Criteria

The acceptance criteria are as follows:

1. Employees can log in to the portal
2. Managers role can log in to the workspace
3. Other users cannot log in to the portal or the workspace

Implementation without BDD

The implementation of the application and the corresponding tests can now be done based on the user story, acceptance criteria and test scenarios. The implementation can be further broken into concrete tasks:

1. Create an employee role
2. Create a test employee user that receives the employee role
3. Create a manager role
4. Create a manager test user that receives the manager role
5. Create a new ServiceNow application
6. Create a new experience of type portal
7. Allow the employee role to access the portal
8. Create a new experience of type workspace
9. Allow the manager role to access the workspace

Administrators manage users and roles on the ServiceNow Service Management page under System Security -> Users and Groups. The button New on the Roles page allows the creation of new roles for an employee (x_790251_vacation.employee) and manager (x_790251_vacation.manager). The naming conventions of ServiceNow impose that user-defined roles start with 'x_', followed by the server ID (790251 in this case), the application name (vacation in this case) and then the name of the role. The button New on the Users page allows the creation of test users vacation_employee and vacation_manager with their corresponding roles.

A new application can be created in the ServiceNow AppEngineStudio by using the "Create App" button and providing the application name (ex: Vacation), description, and an optional logo image. The AppEngineStudio also provides templates that help set up some standard applications faster.

Landing pages are created as Experiences. A ServiceNow Experience is a graphical interface for users to interact with the application. A new Experience can be created in the application dashboard by clicking on the Experience -> "Add" button. ServiceNow provides predefined experiences for the most used cases, like record producer (for inserting/updating/deleting data from the database), mobile Experience (for optimized interfaces for mobile), workspace or portal. For this work, I create an experience of type portal and one of type workspace. The Experience requires a name, a description, and a list of roles that are allowed to access it: the employee role for the portal and the manager role for the workspace. A link to access the application inside the personal developer instance is also created, ex:

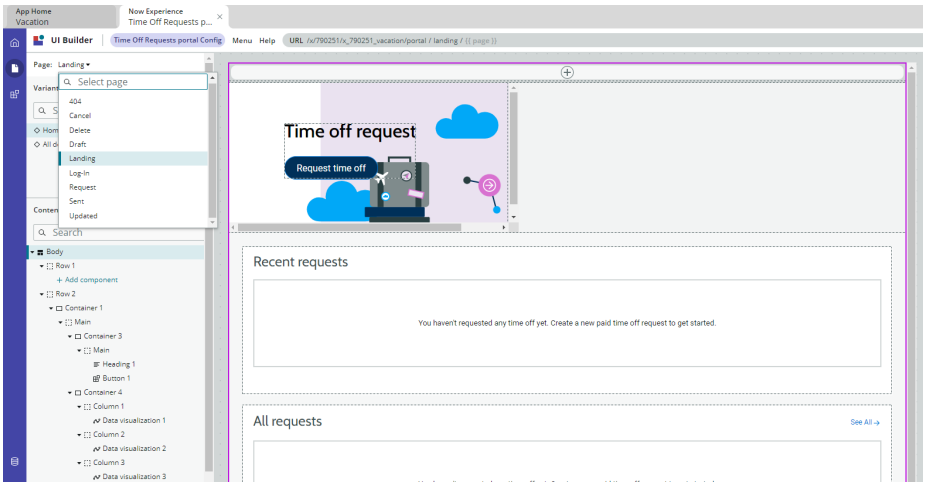

Figure 7.2: Vacation portal user experience: landing page (self-made)

/x/790251/x_790251_vacation/portal is the link for the portal inside the application with ID 790251. The newly created Experience comprises several pages, including the landing page (page displayed after the user logs in as shown in Figure 7.2) and a 404 page (displayed when the users try to access an inexistent link inside the portal). The workspace experience also has a landing page called dashboard in this context.

Adding UI elements to any of the pages is done visually by clicking on a component (Figure 7.3) and then using the menu in its header to either edit it or add components before or after it.

The implementation without BDD is now finished.

Implementation with BDD

An additional analysis session is needed to create the BDD scenarios. The first Scenario is named: "An employee can access the Portal application", as shown in Listing 7.1.

Listing 7.1: Feature: Check User Roles

```
Scenario: An employee can access the Portal application
  Given I am an "employee"
  When I login into "portal"
  Then I can see the "portal" main page
```

A second scenario is named "A manager can access the Workspace application" and looks very similar to Listing 7.1. A third scenario is named "A manager can access the Portal application", as managers are employees.

There are now three scenarios that look similar and can be grouped into a single Scenario Outline, a data-driven scenario. Additionally, Employees are users having the "employee" role, while managers are users having the "manager" and "employee"

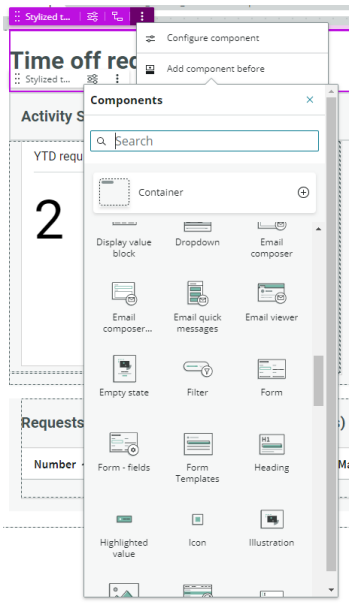

Figure 7.3: Add a new UI element on the portal page (self-made)

roles. The Scenario's name can then be changed to "User with certain roles can access applications".

There are two more additional scenarios: "User without manager role cannot access the Workspace" and "User without employee role cannot access the Portal". The final BDD feature file is shown in Listing 7.2.

Listing 7.2: Feature: Check User Roles

```
Check Portal and Workspace security by role

Scenario Outline: User with certain roles can access applications
  Given I am a user with "<role>" role
  When I login into "<application>"
  Then I can see the "<application>" main page
  Examples:
    | role      | application   |
    | employee  | portal        |
    | manager   | workspace     |
    | manager   | portal        |

Scenario: User without manager role cannot access the Workspace
  Given I am a user without manager role
  When I login into "workspace"
  Then I cannot see the "workspace" main page

Scenario: User without employee role cannot access the Portal
  Given I am a user without employee role
  When I login into "portal"
  Then I cannot see the "portal" main page
```

The automated tests based on the feature file can now be written using Cucumber and Java. The code for the implemented tests is listed in Appendix C.

The Cucumber test execution is done using the command "mvn test". All tests must run without errors. Otherwise, there are bugs in either application or test implementation. The iteration is considered successful (finished) when all the implemented tests run without errors.

A possible error in the test will look like Listing 7.3.

Listing 7.3: Cucumber error message

```
[ERROR] Errors:
[ERROR]   //span[.='The page you are looking for could not be found']
For documentation on this error, please visit:
https://selenium.dev/exceptions/#no_such_element
```

The BDD approach implementation improved the functional completeness, functional correctness and modifiability of the developed Low-Code application.

The described BDD scenarios ensure the functional completion for the current iteration as the feature file captures all user requirements. The implementation without BDD did not consider the scenario in which a manager logs in to the Portal application.

The automated tests ensure functional correctness. They are coded based on the BDD scenarios and verify that the developed software is correct. The missing scenario from the implementation without BDD generated a bug discovered by the automated tests.

The modifiability of the application has been improved through the automated tests, as they become regression tests and find defects introduced by changes to the application.

7.3 Iteration: Create Vacation Request

The second iteration deals with creating a simple vacation request for an employee.

User Story: Create vacation request (Epic: Vacation Portal)
As an employee
I want to be able to create a vacation request
So that I can enjoy my free time

Acceptance Criteria

The acceptance criteria are:

1. Employees can submit a vacation request
2. The vacation request has a start date, an end date, and an optional reason for the request

3. A reference number can identify the vacation request

Implementation without BDD

The implementation in ServiceNow can be split into the following tasks:

1. Create the data model (table) for the vacation request. Figure 7.4 displays a few of the columns in the newly created table.

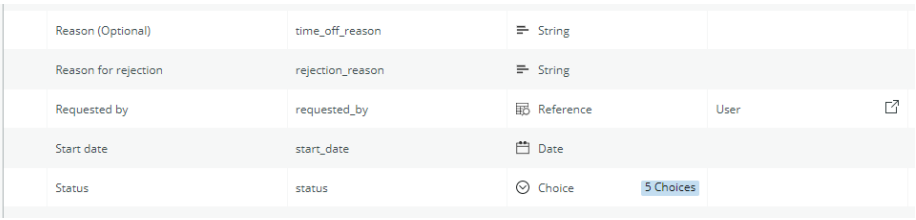

Figure 7.4: Column examples from the data table (self-made)

2. Create an experience (UI) containing a form with the required fields for creating a request. The form contains the fields: Start date (mandatory), End date (mandatory) and Reason (optional). Figure 7.5 shows the newly created form. The form is then integrated in the application using the UI Designer, as illustrated in Figure 7.6.

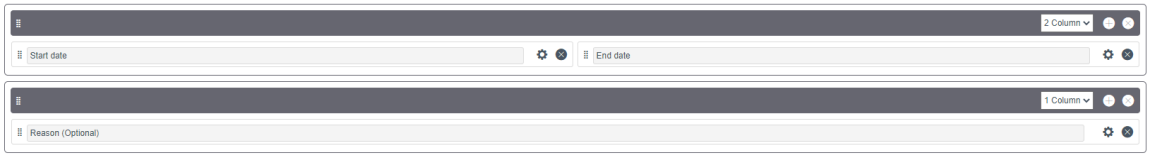

Figure 7.5: Input form containing the needed fields (self-made)

3. Add the logic and automation workflow: a Request vacation button loads the form, and a Send button in the form creates the request.

4. The Send button will execute a save action when clicked and redirect to a popup that displays the reference number.

Implementation with BDD

The acceptance criteria will lead to the following scenario examples, shown as a Cucumber feature file in Listing 7.4.

Listing 7.4: Feature: Create vacation request

```
Creates a vacation request for an employee

Scenario Outline: Cannot submit vacation request when
    start or end date missing
  Given I am a user with "employee" role
```

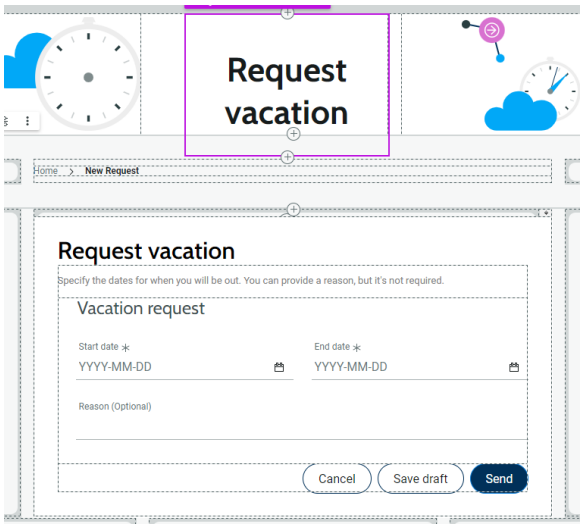

Figure 7.6: Input form integrated in the application using the UI Designer (self-made)

```
  When I successfully login into "portal"
  And I request a vacation with start date "<start_date>"
    and end date "<end_date>"
  Then I cannnot submit a vacation request

  Examples:
    | start_date      | end_date      |
    |                 |               |
    |                 | 2030-05-31    |
    | 2030-05-01      |               |

Scenario: Cannot submit vacation request when end date
    before start date
  Given I am a user with "employee" role
  When I successfully login into "portal"
  And I request a vacation with start date "2030-05-31" and
    end date "2030-05-01"
  Then I cannnot submit a vacation request
  And I see an error message that end date may not be
    before start date

Scenario: Cannot submit vacation request in the past
  Given I am a user with "employee" role
  When I successfully login into "portal"
  And I request a vacation with start date "2019-05-01" and
    end date "2019-05-31"
  Then I cannnot submit a vacation request
  And I see an error message that start date may not be
    before today

Scenario Outline: Create a vacation request
```

```
Given I am a user with "employee" role
When I successfully login into "portal"
And I request a vacation with start date "<start_date>" and
  end date "<end_date>" and reason "<reason>"
Then I can submit a vacation request
And a reference number for the vacation request is generated

Examples:
  | start_date     | end_date     | reason        |
  | 2030-05-01     | 2030-05-01   |               |
  | 2030-05-02     | 2030-05-03   |               |
  | 2030-05-04     | 2030-05-08   | football game |
```

The code for this section is listed in Appendix D.

Similar to 7.2, the implementation using the BDD approach improved the functional completeness, correctness and modifiability of the developed Low-Code application. The feature file contains some concrete scenarios not covered by the implementation without BDD. The result of not implementing the functionalities described by these scenarios is an incomplete application delivered with several bugs. "Cannot submit vacation request in the past" is an example of a scenario first discovered during the feature file creation.

8 Results

This chapter analyzes the results of the implementation iterations from the previous chapter.

All forms and navigation are created in ServiceNow using the UI Designer (Figure 7.3) without writing code. That is not the case when saving data in the database, like submitting a vacation request to the portal (Section 7.3). The client script for this functionality is written in JavaScript and has two steps: it sets the vacation status to 'requested' and saves the vacation to the database using an internal ServiceNow API (Appendix D).

The automated tests are implemented using Java, Selenium and Cucumber. It is hard to write them for a ServiceNow application as the application has a lot of generated fields that cannot be easily identified by only using an ID or a name. Some of the input forms have names for their fields, and these names are generated based on the column they map in the database. However, for most of the applications, the developer of the automated tests must have good XPath or CSS knowledge to identify the fields in the DOM. That makes the development of automated tests cumbersome and time-consuming. The class CreateVacationPage in Appendix D contains such examples of XPath and CSS finders.

ServiceNow recommends starting the application development by defining the data layer and then continuing with the experience layer as all forms map database fields. The next step is implementing the logic and automation layer. BDD can take a different path. For example, scenarios can mock the backend or the database if the BDD requires it. Unfortunately, that is only partially possible within the ServiceNow platform, and pure BDD might not always be applicable.

The automated tests are written code (Appendices C and D). The code needs an IDE for proper development and is stored and runs outside the platform.

The Cucumber feature files and the reports are also written and stored outside the platform (Sections 7.2 and 7.3). Furthermore, feature files will pose small challenges to some stakeholders as they have a specific format (including formatting of whitespaces) and cannot be executed if that format is violated. Current IDEs do not provide enough information on whether the format is violated or whether the test steps associated with a feature file are implemented or not.

Current BDD frameworks and supporting tools that help create BDD scenarios are still developer-oriented. That makes these tools less accessible or entirely inaccessible to non-technical stakeholders, discouraging them from participating in the development. The current tools also violate the principles of the Low-Code platforms because much code is needed to write automated tests. A solution would be integrat-

ing the BDD development in the Low-Code platform. Automated tests and feature files are then created in a drag-and-drop style. Test management of the BDD feature files and generating test reports could also be done within the Low-Code platform.

Integrating the BDD development in the Low-Code platform could allow all stakeholders to participate in the application development actively. It will also help decrease the additional development effort while providing Citizen Developers with what they need to create better applications.

9 Conclusion

This paper has verified two presumptions while applying the BDD methodology in developing a Low-Code application: better quality of the delivered application and a change in development time.

The quality of the application is improved when the BDD process is used compared to the development without BDD. The main reasons for the improved quality are the analysis sessions required for creating the BDD scenarios and implementing the automated tests based on these scenarios. The analysis sessions allow the creation of concrete and detailed examples of how the software behaves. The generated feature files ensure missing details in the user stories are found, and all user requirements are taken into account, improving the functional completeness. The automated tests based on BDD scenarios find bugs in the implementation, improving functional correctness. These tests serve later as regression tests and find defects introduced by changes to the application, improving modifiability.

The development time increases when using a BDD approach compared to the standard Low-Code approach. The reason is the effort needed for both the BDD analysis sessions and the implementation of the automated tests based on the BDD scenarios. This is counterbalanced by an improved quality standard of the application. The result is better software with fewer bugs, as most bugs are found and fixed earlier in the development process.

This paper concludes that a BDD approach increases the quality of a Low-Code developed application. Especially functional suitability and maintainability of the product are increased. Additional development effort is needed, but maintenance costs are decreased in the long run as the delivered product has fewer bugs and mirrors what the users expect.

List of Figures

Listings

Bibliography

(agiledata.org, 2003) Test-Driven Development flow Scott W. Ambler http://agiledata.org/essays/tdd.html, 2003-2006, last accessed: 30th April. 2022

(agility.im, n.d.) Product development with BDD Duncan Evans https://agility.im/insights/product-development-bdd/, 27 March 2015, last accessed: 29th April 2022

(Beck, 2003) Beck Kent: *Test-driven Development: By Example.*, AddisonWesley Professional, 2003

(Bik et al., 2017) Bik N. , Lucassen G. , and Brinkkemper S.: *A reference method for user story requirements in agile systems development*, IEEE 25th International Requirements Engineering Conference Workshops (REW), 2017

(Black, 2009) Black Rex: *Managing the Testing Process: Practical Tools and Techniques for Managing Hardware and Software Testing, 3rd Edition*, Wiley Computer Publishing, 2009

(Braams, 2017) Braams, S.: *Developing a Software Quality Framework for Low-Code Model Driven Development Platforms Based on Behaviour Driven Development Methodology*, Twente Student Conference on IT, University of Twente, Faculty of Electrical Engineering, Mathematics and Computer Science, The Netherlands, 2017

(Clark, 2004) Clark Mike: *Pragmatic Project Automation: How to Build, Deploy and Monitor Java Applications*, The Pragmatic Programmers, 2004

(cucumber.io, n.d.) Autocompletion feature of Cucumber Studio: https://cucumber.io/tools/cucumberstudio/, last accessed: 10th May 2022

(Dimensional Research, 2019) Digital Disconnect: A Study of Business and IT Alignment in 2019: https://www.mendix.com/wp-content/uploads/IT-Business-Alignment-Study-Global.pdf, September 2019, last accessed: 03rd June 2022

(docs.servicenow.com, 2022) Test Management dashboard: https://docs.servicenow.com/en-US/bundle/sandiego-it-business-management/page/product/test-management/reference/r_TestManagementDashboard.html, last accessed: 10th May 2022

(Dustin et al., 2009) Dustin Elfriede, Garett Tom, Gauf Bernie: *ImplementingAutomatedSoftware Testing, How to Save Time and Lower Costs While Raising Quality*, The Pragmatic Programmers, 2009

(Everts, 2016) Everts, Tammy, Mobile Load Time and User Abandonment https://developer.akamai.com/blog/2016/09/14/mobile-load-time-user-abandonment, September 9, 2016, last accessed: 22nd May. 2022

(Fewester, 1999) Fewster M. & Graham D., : *Software Test Automation: effective use of test execution tools* , Addison-Wesley, 1999

(Fowler, 2004) Window Driver: https://martinfowler.com/eaaDev/WindowDriver.html, last accessed: 22nd April 2022

(Gamma et al., 1995) Gamma E., Helm R., Johnson R, and Vlissides J.: *Design Patterns - Elements of Reusable Object Oriented Software*, Addison-Wesley, 1995

(Gartner, 2021) Gartner Report https://gartner.com/en/newsroom/press-releases/2021-02-15-gartner-forecasts-worldwide-low-code-development-technologies-market-to-grow-23-percent-in-2021, February 16, 2021, last accessed: 26th April. 2022

(Gartner ITSM, 2021) Gartner ITSM Magic Quadrant 2021 https://www.servicenow.de/company/media/press-room/8-time-gartner-itsm-mq-leader.html, last accessed: 28th April 2022

(github.io, n.d.) Example of a living documentation screen generated by Serenity : https://developer.servicenow.com/connect.do#!/legal_agreement, August 2021, last accessed: 6th May 2022

(Glenford, 2004) Glenford J.M.: *The Art of Software Testing*, Wiley Computer Publishing, 2004

(Gupta, 2017) Gupta Sagar: *ServiceNow Application Development*, Packt Publishing, 2017

(ISO 25010 n.d.) ISO/IEC 25010 https://iso25000.com/index.php/en/iso-25000-standards/iso-25010, last accessed: 17th June 2022

(jacoco, n.d.) Example of a coverage report generated by the JaCoCo tool https://www.jacoco.org/jacoco/trunk/coverage/, last accessed: 19th April 2022

(Kaner et al., 2002) Cem Kaner, James Bach, Bret Pettichord: *Lessons learned in Software Testing: a Context driven Approach*,Wiley Computer Publishing, 2002

(Khorikov, 2020) Khorikov, Vladimir: *Unit Testing Principles, Practices, and Patterns*, Greenwich, Manning, 2020

(Martin, 2000) Martin, Robert C., Design Principles and Design Patterns: https://web.archive.org/web/20150906155800/http://www.objectmentor.com/resources/articles/Principles_and_Patterns.pdf, last accessed: 10th May 2022

(Martin, 2013) Martin, Rober C.: *Clean Code - A Handbook of Agile Software Craftsmanship*, Boston: Pearson Education, Inc., 2013

(mozilla.org, n.d.) ShadowDOM architecture: https://developer.mozilla.org/en-US/docs/Web/Web_Components/Using_shadow_DOM, last accessed: 6th May 2022

(Myint Myint Moe, 2019): „Comparative Study of Test-Driven Development (TDD), Behavior-Driven Development (BDD) and Acceptance Test-Driven Development (ATDD)“, *Published in International Journal of Trend in Scientific Research and Development (ijtsrd)* Volume-3 | Issue-4, June , 2019, last accessed: 4th May 2022

(Rayner, 2015) Leading by Design, blog on software design and process Paul Rayner http://thepaulrayner.com/bdd-is-a-centered-rather-than-a-bounded-community/, 27 March 2015, last accessed: 9th May 2022

(servicenow.com, 2020) ServiceNow® Website Terms of Use, Version 2.0: https://serenity-bdd.github.io/theserenitybook/latest/living-documentation.html, last accessed: 20.03.2022

(skillsmatter, 2009) Agile specifications, BDD and Testing eXchange. Daniel Terhorst-North https://skillsmatter.com/skillscasts/923-how-to-sell-bdd-to-the-business, 27 Nov. 2009, last accessed: 4th May 2022

(Smart, 2015) Smart Ferguson John: *BDD in Action*, New York: Manning Publications Co., 2015

(specflow.org, 2022) SpecFlow: https://specflow.org/using-specflow/the-retirement-of-specflow-runner, last accessed: 6th May 2022

(Spillner et al., 2007) Spillner Andreas, Tilo Linz, Hans Schaeffer: *Software Testing Foundations*, 2nd edition, Rocky Nook, 2007

(Unisphere Research, 2017) The Rise of the Empowered Citizen Developer: https://www.dbta.com/DBTA-Downloads/ResearchReports/THE-RISE-OF-THE-EMPOWERED-CITIZEN-DEVELOPER-7575.pdf, 2017, last accessed: 03rd June 2022

(w3schools.com, n.d.) Folder structure looks similar to XPath structure : https://www.w3schools.com/xml/xpath_intro.asp, last accessed: 6th May 2022

(Wynne et al., 2012) Wynne M., Hellesoy A., and Tooke S.: *The Cucumber Book: Behaviour-Driven Development for Testers and Developers*, The Pragmatic Programmers, 2012

A ServiceNow PDI License & Activation Instructions

License

According to the (servicenow.com 2020)(ServiceNow® Website Terms of Use, 2020):

"Developer Instances. ServiceNow hereby grants to Participant a limited, personal, revocable, non-sublicensable, non-transferable, non-exclusive right to access and use one or more Developer Instance(s), as provided to Participant by ServiceNow, and to modify Configurable Elements in the Developer Instances(s), all solely for Participant's own internal use to: (a) develop and test Participant Technology for use with the Subscription Service, including configurations, customizations, and integrations of the Developer Instance(s); (b) evaluate the Developer Instance(s); [...] Without limiting any license restrictions in this Agreement, Participant must not use the Developer Instances for any production or commercial use." (servicenow.com, 2020)

Activation Instructions

Obtaining a PDI requires two steps to be done. First, create a ServiceNow Platform account and request a Personal Developer Instance (PDI). The creation of the ServiceNow PDI for this work was done in April 2022.

A ServiceNow platform account is needed before requesting a ServiceNow personal developer instance. The account is free and can be created by opening the ServiceNow main page in a browser and then clicking on the "My Account" icon in the right upper corner of the page. A dialogue "Sign in with your ServiceNow ID" appears. Click on the "Get a ServiceNow ID" button to open the "Sign up for a ServiceNow ID" dialogue. Fill in the fields, accept the license agreement, and submit the form using the "Sign Up" button. A confirmation email will be sent; open it, and confirm the registration. The ServiceNow platform account is now created.

To obtain a PDI, first log in using the previously created account by signing in to the ServiceNow developer page. A redirect to the login page is done, entering the account data (email and password). A PDI can be requested by clicking on the "Request Instance" button and selecting the appropriate ServiceNow release version. The latest release is San Diego when writing this document (April 2022). The instance will be started and be ready to be used after a few minutes. The user will be notified in the browser when the instance is read. A dialogue showing the URL and credentials (username and password) for the created PDI is displayed. This information must

be stored in a safe place (ex: a password manager). The user can begin using the newly created PDI by clicking on the "Start Building" button and logging in using the provided credentials.

B Installation Instructions for BDD Software & Tools

Java

Java can be downloaded for free from Oracle, the only requirement being creating a free Oracle account. Java can be installed by selecting the appropriate version for the desired operating system (ex: x64 Installer for Windows), downloading the selected file and running it.

An environment variable called JAVA_HOME must be set up after a successful Java installation. The operating system uses this environment variable to find the Java executable programs. It is the path to the JDK installation folder. The setting up of this variable depends on the operating system. The steps on Windows are as follows:

1. Locate the installation directory, like:

```
C:\Program Files\Java\jdk1.8.0_202
```

2. Open the "Environment Variables" in the Control Panel and click "New" under "System Variables".

3. Enter the JAVA_HOME into the "Variable Name" and the installation directory into the "Variable Value" fields and then click "OK".

The Java installation can be verified by opening a terminal (or command line) and executing the "java -version" command. The displayed information should match the installed Java version.

Maven

Maven can be downloaded from the Maven Apache page by selecting the desired version. The downloaded ZIP file can be unpacked in any folder on the system, and the path to its executable must be added to the PATH variable of the operating system. On Windows, this can be done by opening the "Environment Variables" in the Control Panel, editing the Path variable under "System Variables", and adding a new value containing the folder with the maven executables, like:

```
C:\java\apache-maven-3.8.5\bin
```

The maven installation comes with script executables for both Unix and Windows system.

The maven installation can be verified by opening a terminal (or command line) and executing "mvn —version". The displayed information should match the installed maven version.

Eclipse Plugins

All installed Eclipse plugins are in an Eclipse folder called "plugins". The developer can install a new plugin by manually copying its files into the plugins folder or letting Eclipse automatically install the plugin. Most Eclipse plugins can be found in the Eclipse Marketplace, easily accessible from the Eclipse IDE menu Help -> Eclipse Marketplace.

Cucumber Project

The following command in a terminal will create the Cucumber project:

```
mvn archetype:generate \
  "-DarchetypeGroupId=io.cucumber" \
  "-DarchetypeArtifactId=cucumber-archetype" \
  "-DarchetypeVersion=7.0.0" \
  "-DgroupId=bachelor\_arbeit" \
  "-DartifactId=bdd" \
  "-Dpackage=bdd" \
  "-Dversion=1.0.0-SNAPSHOT" \
  "-DinteractiveMode=false"
```

"archetype-generate" instructs maven to create a new project from the provided archetype. The parameters archetypeGroupId, archetypeArtifactId, and archetypeVersion define which archetype and version will be used, in this case, the "cucumber-archetype". The "groupId" defines the id of the project group. The "artifactId" specifies the id of the generated project. The "package" defines all the Java classes will be found under which package. The "version" defines what version of the current software. The standard maven versioning scheme is as follows:

```
<major_version>.<minor_version>.<bugfix>[-SNAPSHOT]
```

Higher version numbers mean a newer version of the same software. The SNAPSHOT qualifier is optional and has a special meaning in maven. The respective software is considered a "not-yet-released" version, in other words, a version still in development. The "interactiveMode" means that the user must type in some answers in the terminal using the keyboard. It is set to "false" so that maven installs everything by assuming defaults as answers.

Selenium WebDriver

Installing the Selenium WebDriver for Java requires updating the pom.xml file with an entry containing the WebDriver library information:

```
<dependency>
  <groupId>org.seleniumhq.selenium</groupId>
  <artifactId>selenium-java</artifactId>
  <version>4.1.2</version>
</dependency>
```

Maven will automatically resolve the Selenium library dependency and download it.

GeckoDriver

The GeckoDriver must be downloaded on the computer running the test and set up as an environment variable in code:

```
System.setProperty("webdriver.gecko.driver",
  "C:\\my\\gecko\\path\\GeckoDriver.exe");
```

WebDriverManager

Installing the WebDriverManager requires updating the pom.xml file with an entry containing the WebDriverManager library information:

```
<dependency>
  <groupId>io.github.bonigarcia</groupId>
  <artifactId>webdrivermanager</artifactId>
  <version>5.1.0</version>
</dependency>
```

Shadow

The corresponding entry in the pom.xml file for the Shadow library is:

```
<dependency>
  <groupId>io.github.sukgu</groupId>
  <artifactId>automation</artifactId>
  <version>0.1.4</version>
</dependency>
```

C Code Listings for the Iteration: User and Roles

```
public class LoginPage {
  private static final String GSFT_MAIN = "gsft_main";
  private static final String PDI_HOST
      = "https://dev57508.service-now.com/";

  protected WebDriver driver;

  private By passwordBy = By.id("user_password");
  private By signinBy = By.id("sysverb_login");
  private By usernameBy = By.id("user_name");

  public LoginPage(WebDriver driver) {
    this.driver = driver;
  }

  public LoginPage loginUser(String userName, String password) {
    // opens the personal developer instance page
    driver.get(PDI_HOST);
    // find Frame with id "gsft_main" and switch to it
    new WebDriverWait(driver, Duration.ofSeconds(10))
        .until(ExpectedConditions
        .frameToBeAvailableAndSwitchToIt(GSFT_MAIN));
    // login user
    driver.findElement(usernameBy).sendKeys(userName);
    driver.findElement(passwordBy).sendKeys(password);
    driver.findElement(signinBy).click();
    return new LoginPage(driver);
  }
}
```

```
public class CheckUserRoles {
  private String currentPassword;
  private String currentUser;

  private WebDriver driver;

  @Given("I am a user with {string} role")
  public void i_am_a_user_with_role(String role) {
    if (role.equals("employee")) {
      currentUser = EMPLOYEE_USER;
```

```
      currentPassword = EMPLOYEE_PASSWORD;
    } else {
      currentUser = MANAGER_USER;
      currentPassword = MANAGER_PASSWORD;
    }
  }

  @Given("I am a user without Manager role")
  public void i_am_a_user_without_manager_role() {
    currentUser = EMPLOYEE_USER;
    currentPassword = EMPLOYEE_PASSWORD;
  }

  @Given("I am a user without Employee role")
  public void i_am_a_user_withoutemployee_role() {
    currentUser = "xxx";
    currentPassword = "xxx";
  }

  @Then("I can see the {string} main page")
  public void i_can_see_main_page(String application) {
    if (application.equals("portal")) {
      PortalPage portalPage = new PortalPage(driver);
      portalPage.open();
      portalPage.assertPortalDisplayed();
    } else {
      WorkspacePage workspacePage = new WorkspacePage(driver);
      workspacePage.open();
      workspacePage.assertWorkspaceDisplayed();
    }
  }

  @Then("I cannot see the {string} main page")
  public void i_cannot_see_main_page(String application) {
    if (application.equals("portal")) {
      PortalPage portalPage = new PortalPage(driver);
      portalPage.open();
      portalPage.assertPageNotFound();
    } else {
      WorkspacePage workspacePage = new WorkspacePage(driver);
      workspacePage.open();
      workspacePage.assertPageNotFound();
    }
  }

  @When("I login into {string}")
  public void i_login_into(String application) {
    LoginPage loginPage = new LoginPage(driver);
    loginPage.loginUser(currentUser, currentPassword);
  }
}
```

D Code Listings for the Iteration: Create Vacation Request

```
public abstract class AbstractPageObject {
  protected WebDriver driver;
  protected Shadow shadow;

  protected AbstractPageObject(WebDriver driver) {
    this.driver = driver;
    this.shadow = new Shadow(driver);
  }
}
```

```
public class CreateVacationPage extends AbstractPageObject {

  private static final String END_DATE_CSS
    = "input[name='end_date-date']";
  private static final String REASON_CSS
    = "input[name='time_off_reason']";
  private static final String SEND_BUTON_XPATH
    = "/now-button//button[contains(text(), 'Send')]";
  private static final String START_DATE_CSS
    = "input[name='start_date-date']";

  public CreateVacationPage(WebDriver driver) {
    super(driver);
  }

  public CreateVacationPage createVacation(String startDate,
        String endDate) {
    return createVacation(startDate, endDate, "");
  }

  public CreateVacationPage createVacation(String startDate,
        String endDate, String reason) {
    shadow.findElement(START_DATE_CSS).sendKeys(startDate);
    shadow.findElement(END_DATE_CSS).sendKeys(endDate);
    shadow.findElement(REASON_CSS).sendKeys(reason);
    return this;
  }

  public CreateVacationPage send() {
    shadow.findElementByXPath(SEND_BUTON_XPATH).click();
```

```
    return this;
  }

}
```

```
  private static final String REQUEST_VACATION_BUTON_XPATH
    = "/now-button//button[contains(text(), 'Request vacation')]";

  public CreateVacationPage requestVacation() {
    shadow.findElementByXPath(REQUEST_VACATION_BUTON_XPATH)
        .click();
    return new CreateVacationPage(driver);
  }
```

```
  @When("I successfully login into {string}")
  public void i_successfully_login_into(String application) {
    i_login_into("portal");
    i_can_see_main_page("portal");
  }

  @And("I request a vacation with start date {string}
      and end date {string}")
  public void i_request_a_vacation_with_start_date_and_end_date(
        String startDate, String endDate) {
    PortalPage portalPage = new PortalPage(driver);
    CreateVacationPage createVacationPage
        = portalPage.requestVacation();
    createVacationPage.createVacation(startDate, endDate);
  }
```

```
/**
 * Client script for submitting a vacation request to the database
 * @param {params} params
 * @param {api} params.api
 * @param {any} params.event
 * @param {any} params.imports
 */
function handler({
    api,
    event,
    helpers,
    imports
}) {
    api.data.glide_form_1.setValue({
        fieldName: 'status',
        value: 'requested'
    });

    api.data.glide_form_1.save();
}
```

Ich versichere, die vorliegende Arbeit selbstständig ohne fremde Hilfe verfasst und keine anderen Quellen und Hilfsmittel als die angegebenen benutzt zu haben. Die aus anderen Werken wörtlich entnommenen Stellen oder dem Sinn nach entlehnten Passagen sind durch Quellenangaben eindeutig kenntlich gemacht.

Ort, Datum Magdalena Lucreteanu