POLITECNICO DI TORINO
Master Degree Course in Computer Engineering

Master Degree Thesis

A Groovy-based Domain-Specific Language For Digital Payment System End-To-End Testing

Relatore
Prof. Riccardo Coppola

Vito Ruggirello
matricola: 289902

Supervisor
Pay Reply
Dott. Mattia Ognibene

A. Y. 2024-2025

Summary

Automated software testing remains one of the most overlooked processes in software engineering, primarily due to its costly implementation and the limited understanding of its benefits outside the field of testing expertise. This research aims to develop a tool that supports the automated execution of integration tests for an application in the digital payments domain while demonstrating the advantages of automated testing approaches. The digital payments domain serves as an ideal testing environment for this purpose due to its *business-critical* nature and the *data-driven*, *browser-based* testing processes required by its business requirements.

The first part of this thesis presents the challenges posed by domain and business requirements. In the second part, a `Java` solution to execute a set of user-defined test is proposed. In particularly, a *Domain Specific Language* has been designed to dynamically extract tests definitions and parameters from an input file and run them inside `JUnit`. `Groovy` has been employed to define the DSL and directly parse the input file into Java classes. Compared to other parsing solution, Groovy provided maximum flexibility and easy of implementation. As output, the program produces an HTML report using the new `opentest4j` standard maintained by the JUnit team.

The developed solution offers several significant advantages over traditional testing approaches. First, it provides a more accessible interface for non-technical stakeholders to define and comprehend test cases through the custom DSL. Second, the integration with JUnit leverages an established testing framework while extending its capabilities for specialized payment processing scenarios. Third, the HTML reporting functionality enables clear visualization of test results and facilitates communication between technical and business teams.

Future research directions include extending the DSL to support more complex testing scenarios and integration of additional technologies; for example by developing a web application interface and including Large Language Models in DSL definition and utilization pipelines. Additionally, the approach could be adapted for other domains requiring similar data-driven, browser-based testing workflows.

Keywords: Software Engineering, Automation Testing, Integration Testing, Digital Payments, JUnit, Domain-Specific Language, Groovy, Selenium

Contents

Chapter 1

Introduction

1.1 Context and Motivation

Every engineering field has developed, over the years of its own history, some methods and processes to test and validate the proposed solutions. For instance, aerospace engineering provides a set of standards and rigorous mechanical tests to assess the airplane reliability. The software engineering field is no exception, and since the birth of modern software development, programmers needed a way to validate their products against a set of formal requirements and to ensure that the quality of what they produced would meet customer expectations. Yet, software engineering is unique in the nature of the solutions to be tested and in the way these tests are executed. Software possesses no *physical* properties that can be measured or subjected to critical scenarios, but is instead abstract and primarily involves information manipulation.

Testing has been demonstrated to be the most reliable method for assessing product maturity, thereby becoming the *common factor* across all software engineering development models.

In addition to showing the presence of *bugs*, automated software testing can be of great help when well integrated into the development process as it provides a fixed set of constraints preventing regressions and allowing for frictionless code modification and refactoring.

1.2 Problem Statement

The testing phase is typically assigned either to developers or to dedicated Quality Assurance teams responsible for performing manual testing of the delivered application. Developer-led testing may be susceptible to domain misconceptions, potentially resulting in defective products, whereas QA-team testing presents significant resource requirements and necessitates execution with each release to prevent

regressions.

While the domain misconceptions can be addressed adopting Domain-Driven Design methodologies[10], several tools, such as Cucumber [17], have been proposed to bridge the technical gap between software testing and less technical stakeholders. These tools introduce a common language—a Domain-Specific Language—purposefully designed for test requirements definition. Since the DSL is written in common english, it can be accessed by both stakeholders and developers to enable a collaborative effort in writing business requirements, reducing the gap between formal requirements and techical solution. In this way, stakeholders and developers can share the same requirements representation with the added clarity due to formality. This approach presents significant implementation challenges, requiring both highly coordinated teams and well-defined human processes. Furthermore, it is optimally implemented at project inception, as post-launch integration may not provide sufficient return on investment. Additionally, Cucumber test data are fixed by test definitions, and the tool is not designed to execute tests with varying user-defined input parameters.

The digital payments domain, however, necessitates a highly customizable dynamic test definition framework that supports repeated execution of identical tests with variable parameters. For example, this would enable testing a payment process initially with a MasterCard debit card and subsequently with a Visa card. While tests framework usually provide out-of-the-box support for these data-driven tests scenarios, they actually did not fit the SUT requirements: *the data point are still defined and embedded in the test case and cannot be changed at run-time based on the user input data.*

The testing challenges addressed by this research work are the following:

- Test definitions and descriptions that align closely with business domain terminology and remain accessible to stakeholders with limited technical expertise.
- Highly parameterized test frameworks that support execution with variable user-defined parameters.
- Visual reporting mechanisms that facilitate inspection of test results and enable effective sharing among diverse stakeholders.

1.3 Objectives

Having identified these key challenges in the testing domain, this research aims to develop a Domain-Specific tool that facilitates system testing of domain applications.

As a *business-critical* sector, digital payments necessitate rigorous correctness testing to verify system functionality and integration between components. The thesis work focuses specifically on *testing* the client's business processes, verifying

that the integration between various payment chain actors is correctly aligned. In particular, the objective is to **automate** the execution of such tests to reduce costs and prevent human errors.

Although test implementation incurs initial development costs, empirical evidence suggests that long-term benefits—including time savings for operators and accelerated integration processes—typically exceed these initial investments [23][14].

The final objective is to integrate a Domain-Specific Language in an automation testing framework to allow the user to execute and configure a test suite with run-time test parameters.

1.4 Digital Payments as Testing Domain

Since the implementation of the first Payment Service Directive (PSD), the digital payments market has experienced substantial growth attributable to the establishment of a well-defined legal framework. The directive established a highly regulated environment comprising distinct legal entities with clearly delineated responsibilities. This framework defined several new roles within the digital payments ecosystem, classified as Third Party Providers [11], including PISP (Payment Initiation Service Providers), AISP (Account Information Service Providers), CISP (Card Issuers Service Providers). The integration of these services facilitates a comprehensive digital payments experience for consumers.

Digital payments represent an appropriate domain for this research for several reasons:

- The business-critical nature of digital payment systems necessitates comprehensive testing coverage;
- The target application represents the integration of multiple interconnected services;
- Domain-specific business processes can be effectively formalized through a specialized language while reducing manual testing interventions, thereby creating opportunities for automation and performance optimization.

1.4.1 Thesis Structure

This introductory chapter has established the context, challenges, and objectives of the present research, with specific focus on the digital payments domain as an appropriate testing domain.

The following chapters will elaborate on the test automation state-of-the-art and current trends, domain analysis, proposed solution architecture, implementation details, and evaluation methodology.

In particular, Chapter 2 presents the current background and state-of-the-art of automation testing, describing the various methodologies and their implementations.

Chapter 3 is dedicated to a general analysis of the system to be tested and the digital payments domain as a whole.

Chapter 4 presents the solution implementation, including a detailed analysis of the tools employed.

Chapter 5 provides an empirical validation of the concepts presented in this research and an evaluation of their impact.

Chapter 2

State Of The Art

Software engineering is the discipline of engineering concerned with the systematic development of software systems[22]. The necessity for rigorous analysis and formal models of the software development life-cycle arises from the inherent complexity of these systems. Just to cite some examples, the Chromium Web Browser contains more than 32 million lines of code [6] and the GNU/Linux operating System just about some million more [21]. To manage such complexity without rigorous methodologies and practices is indeed impossible and overwhelming.

Beyond code complexity, software engineers must address the application domain to which the software is applied; this aspect will be examined in greater detail in Section 2.4.2. This includes managing new knowledge not strictly related to programming itself but to the ecosystem of ideas and information that the software must manipulate. The domain requirements are in fact a crucial aspect of software engineering and overlooking them is the primary cause of failing projects and high evolution costs[10]. For this reason, software engineering has developed a set of theories and practices to minimize the complexity of developing large systems and to limit the number of errors that could lead to software failure. This chapter examines these techniques with particular emphasis on the testing phase.

2.1 Software Development Models

In order to provide a methodological and tractable representation of the software development problem, a set of common frameworks has been developed in the field.

These models attempt to formalize the process of building software products by breaking it down into smaller problems. The degree of model complexity and sub-processes involved mainly depends by the software complexity and the *risks* involved. For bigger products that could lead to catastrophic outcomes, more time is spent on planning its evolution and carefully validating the solution.

The main tasks usually defined by software development models includes:

Requirements Elicitation Focused on collecting and extracting any hidden domain knowledge and business requirements, both functional and non-functional.

System Design Involve creating a detailed design of the system that would meet the elicited requirements.

Testing Is responsible of validating the solution to assess that requirements are fully respected.

Maintenance Once a solution has been developed and concluded, it enters maintenance mode, which includes both corrective operations or new evolution of the products.

2.1.1 V&V Model

The Verification & Validation model (V&V)[5] is a structured approach to ensure that software systems meet specified requirements and fulfill their intended purpose. This model establishes distinct phases for verification (building the system correctly) and validation (building the correct system).

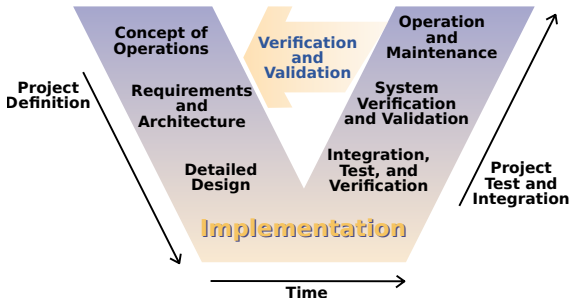

Figure 2.1. V-Model process represantion. frey Brummond, Robert Hart, Mohsen (Moe) Zarean Ph.D., P.E, Steven Conger; Image extracted from Clarus Concept of Operations. Publication No. FHWA-JPO-05-072, Federal Highway Administration (FHWA), 2005, Public Domain

2.2 Software Testing

> Code without tests is broken by design.
>
> – Jacob Kaplan-Moss

Software testing is the phase, in the software development life-cycle, where the developed product is assessed for requirements compliance. This generally means that the software is executed with predetermined input data for which expected behavior has been defined and subsequently verified.

2.2.1 Testing Methodologies

The software product can be seen at many different layers of abstraction: at the highest one it is a black box providing output for input but as additional levels of detail are introduced, the software reveals its composition of integrated modules, each containing its own set of defined functional units. Just as we can see the software at different levels of abstraction, in the same way the testing process can be applied at different levels, so that, for example, we could test units independently from one another and then at a higher level of integration to see how they behave together. This perspective provides significant value for testing methodologies as it enables considerable flexibility in both test scope and implementation. For example, when a defect is identified during black box testing, it may be isolated through iterative application of additional testing layers until the source is located. It is important to observe that the defect's origin is not necessarily confined to a single unit but may extend across the integration of multiple subsystems. This section presents an overview of the primary testing levels and examines their respective advantages and disadvantages.

White Box Testing

With the White Box Testing approach the system is transparent and its internals are exposed to the test cases.

While this allows deeper testing it is more complicated to carry out as it requires a great understanding of the system. As explained before, once we can access the internals of the system, we can test its behaviour in many ways.

Unit testing Represents the most fundamental level of testing, focusing on individual functional components of the system in isolation;

integration Integration testing focuses on the dependencies among subsystems and their inner-working as a whole. Depending on the dependency tree, there are different ways to conduct an integration testing: big-bang to test them all together, bottom-up to start from the lowest one and top-down to do the opposite,

In addition to provide passing results for test cases, white box testing can provide insightful metrics about the system:

Code coverage the percentage of source code lines executed during the testing phase. Higher percentages correlate with increased confidence regarding system behavior verification;

Path coverage provides more details than the code coverage as it gives numerical insights about the percentage of logical path followed by system during the testing.

Black Box Testing

In the Black Box testing the system is instead totally opaque and accessible only by its input and output interfaces.

In this scenario we can provide a set of test cases as a pair of expected input and expected output. The test cases are passes if the system outputs correspond to the expected ones for every input provided.

While this approach presents certain limitations, it demonstrates high implementation efficiency and practical applicability.

Figure 2.2. Black Box Testing

Under the umbrella of Black Box testing falls also the **End-to-End testing** approach, particularly relevant for web applications, as it simulates a requirements scenario from start to finish including any user interactions with the web application. The system is tested as a whole and from the perspective of a user interacting with it, and so without knowledge about its implementation. This is the approach that we are going to apply to test our digital payment system.

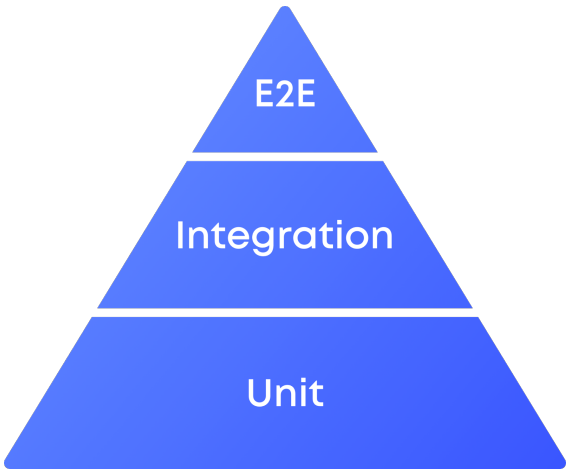

Figure 2.3. The hierarchy of testing levels

2.3 Automation Testing Tools

As the main contribution of this thesis is a tool for software automation, we will now focus on this subject and the available tools to deal with it.

Test Automation is the process of running the testing phase with little human intervention and supervision. A dedicated software infrastructure and tools are employed to run a predefined set of test cases and execute them against the running application. Although automation is not essential for end-to-end testing and requires initial configuration time, it ultimately provides significant benefits through accelerated test execution, increased testing frequency, and reduced operational costs.

Automation testing tools need to provide ways to define tests, run them and visualize the results. Depending on the programming language chosen for the system implementation there are different testing frameworks available. Since the language adopted for our system under test is Java, we kept it not to break continuity with company standards and maintain homogeneity in the project It should be noted, however, that as end-to-end testing operates externally to the system, alternative programming languages could have been selected for test implementation.

2.3.1 Testing Framework

Testing framework are software libraries designed to help in the task of writing and executing tests. For Java, the most important and common implementations are JUnit and TestNG. For our task we decided to adopt JUnit as it is the most commonly used one and provided a set of API and features which matched our requirements, discussed in more details in the methodology section related to JUnit.

Spock is another testing library worth mentioning as it exploits the power of Groovy DSL for elegant test definitions. Cucumber also deserves its mention as it proposes a new approach to test definitions by combining a test oriented DSL to let stakeholders write scenarios to be used by developers to implement tests. This is a gap-reducing approach limiting the risk of requirements misunderstanding.

In general, every programming language has its own ecosystem of framework to define tests and verify results. A useful framework should allow developers great flexibility in the testing definition and provide useful reporting infrastructure. JUnit completely reach this target by providing a powerful assertions library and many different ways to define tests, from standard one, to data driven tests, extension points, test template and totally dynamic tests. We will explore its features in more details in the following chapters.

2.3.2 Project Management Tools

A typical application relies on many external dependencies to work properly, and managing them manually becomes increasingly challenging as their number grows. Dependency management represents one of the primary functions addressed by project management tools. These tools function as extensions of compilers, expanding their operational scope to address additional aspects of application development.

Project management tools serve multiple purposes beyond dependency management and compilation, including test execution and result reporting. Maven Surefire, for example, operates as a plugin for the Maven management system that executes tests and generates HTML reports of the results. These tools typically implement specific support for testing frameworks to facilitate automatic test discovery and execution.

The evolution of build systems in software development represented a significant progression in managing increasingly complex projects. In the early days of Java development, build processes were often managed through shell scripts or proprietary solutions, which lacked standardization and portability. This situation changed significantly with the introduction of formal build systems.

The first build tool developed for the Java platform was Apache Ant, released in 2000 [8]. At that time, Java applications' dependencies were manually managed by developers. The solution proposed by Ant was to employ a declarative XML-based approach to define build processes, test tasks, configure dependencies, and so on. This represented a significant improvement over shell scripts. Ant provided a task-based model that allowed developers to specify build operations such as compilation, testing, and packaging in a platform-independent manner, since Ant was built with Java itself and did not rely on operating systems' utilities. Ant did not come with a set of convention, and every task had to be explicitly defined by the developers using Ant's specific XML model. However, this flexibility often resulted in verbose build files that were difficult to maintain across large projects.

Maven [27], introduced in 2004 by the Apache Software Foundation, proposed a different approach [28]: unlike Ant's procedural approach, Maven defined a set of convention and *lifecycles* that standardized project structures and main tasks. Differently from Ant, the users could not define new project lifecycles (the Ant's tasks substitutes), which were fixed and could only be extended by plugins. Maven's innovation included also dependency management, introducing the concept of centralized repositories for sharing Java artifacts which led to the birth of the Maven repository[1]. This development significantly simplified the management of external

[1]What nowadays is given for essential - package repositories such as npm for the Javascript environment and crates.io for the Rust ecosystem - actually traces back to the ideas of Maven developers.

dependencies, allowing developers to specify required libraries declaratively rather than managing them manually.

Gradle [16], first released in 2007, can be seen as a synthesis of Maven's convention-based approach and Ant's flexibility. Its distinctive feature was the adoption of Groovy, a dynamic JVM language, for build script definition. A domain-specific language implemented in Groovy substituted the verbose XML files used by both Maven and Ant, and provided developers with greater expressivity and reduced verbosity in build specifications. Gradle also implemented incremental build capability and performance optimizations to address long compilation time for large-scale projects.

The currently adopted structure of modern Java applications derives from Maven's defined conventions, which separate test code (located in the src/test directory) from primary application code (located in the src/main directory). This organizational approach enhances code clarity and allows for the distinction between test dependencies - such as the JUnit library - and primary dependencies. This separation ensures that test dependencies are referenced only during the testing phase and not embedded in the final application artifacts.

A common characteristic shared by these three Java build management tools is their explicit support for popular testing frameworks such as JUnit and TestNG, enabling test discovery and execution within dedicated build phases.

Build tools	Lifecycle support	Configuration
Ant	Non opinionated, must be defined by users	XML
Maven	Opinionated and pre-defined	XML
Gradle	Easily extensible	Groovy DSL

Table 2.1. Comparison of the most popular Java build systems

2.3.3 CI/CD Servers

The preceding analysis has examined test execution through build management tools and supported test frameworks. Although this process automates test execution, it nevertheless requires human intervention to initiate, as the command to run the tests must be given to the build management tool. This limitation highlights the need for a mechanism to ensure that code deployed to production environments undergoes mandatory testing prior to release.

Continuous Integration/Continuous Deployment (CI/CD) practices, originating from the *DevOps* methodology, address this requirement effectively. CI/CD represents a paradigm shift in software delivery by establishing automated pipelines that integrate testing directly into the deployment workflow [13]. These practices create a systematic approach wherein software deployment proceeds only after successful

validation through automated test suites.

The primary function of CI/CD is to enforce testing procedures in the deployment processes. By constructing this relationship, organizations ensure that only code that meets predefined quality standards reaches production environments. When tests fail, the deployment process terminates automatically, preventing potentially defective code from affecting end users. This integration significantly reduces the risk of deploying faulty software and consequently minimizes system downtime and user-reported defects [18].

Beyond basic test execution, CI/CD infrastructure supports sophisticated quality policies. For example, pipelines can be configured with conditional progression rules based on code quality metrics. A common implementation involves establishing minimum code coverage thresholds (e.g., 80

CI/CD servers operate by interfacing with Version Control Systems (VCS) through event-driven mechanisms. When developers commit code to the repository, the VCS triggers a webhook notification to the CI/CD server, initiating the execution of predefined pipeline workflows. This automation eliminates the need for manual intervention in the testing and deployment process, establishing a consistent and reliable software delivery mechanism.

A CI/CD pipeline comprises a sequence of defined stages, typically including:

1. **Code checkout**: Retrieval of the latest source code from the repository
2. **Compilation**: Transformation of source code into executable artifacts
3. **Unit testing**: Validation of individual code components in isolation
4. **Integration testing**: Verification of component interactions
5. **Static code analysis**: Evaluation of code quality without execution
6. **Security scanning**: Detection of potential security vulnerabilities
7. **Artifact generation**: Creation of deployable software packages
8. **Artifact publication**: Distribution to artifact repositories
9. **Deployment**: Installation in target environments
10. **Post-deployment testing**: Validation of the deployed application

Each stage in this sequence depends on build management tools for execution, demonstrating the hierarchical relationship between technologies in the automation ecosystem. This layered approach enables the construction of increasingly sophisticated software delivery mechanisms while maintaining modularity and separation of concerns. Figure 2.4 illustrates this hierarchical relationship.

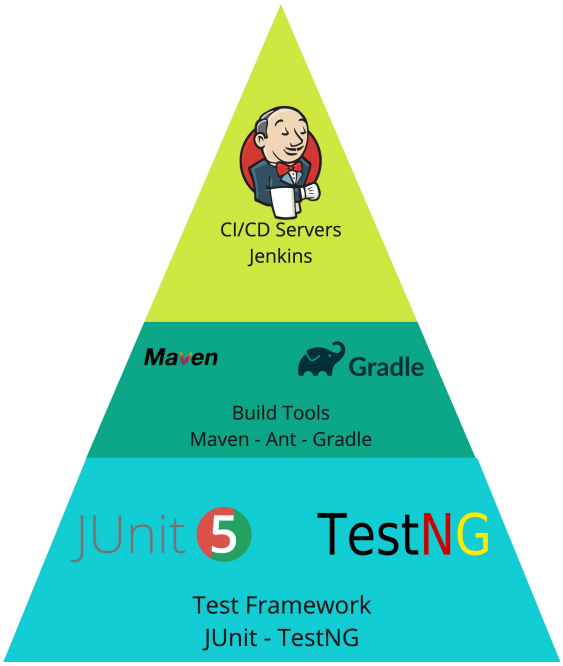

Figure 2.4. Hierarchy of the automation testing tools

Jenkins has emerged as the de facto standard for CI/CD implementation in enterprise environments, commanding significant market share due to several distinguishing characteristics [19]. Its architecture supports extensive customization through a vast ecosystem of plugins (over 1,800 as of 2023), enabling integration with virtually any development tool or technology. Jenkins' pipeline definition language, implemented as a Groovy-based Domain-Specific Language, provides both flexibility and readability, allowing complex deployment workflows to be expressed in a concise, code-based format.

Alternative CI/CD platforms have emerged to address specific market requirements:

GitLab CI/CD Offers tight integration with GitLab repositories, emphasizing a unified user experience

GitHub Actions Provides workflow automation directly within GitHub repositories

CircleCI Focuses on containerized execution environments for consistent build environments

Travis CI Originally specialized in open-source project integration

Azure DevOps Integrates with Microsoft's comprehensive development ecosystem

AWS CodePipeline Optimized for deployment to Amazon Web Services infrastructure

The implementation of CI/CD practices yields quantifiable benefits for organizations, including reduced deployment frequency (from months to days or hours), decreased time to market, improved software quality, and enhanced developer productivity [12]. By automating the repetitive aspects of software testing and deployment, CI/CD enables development teams to focus on value-adding activities such as feature development and innovation rather than manual integration and deployment procedures.

2.4 Domain-Specific Languages

> The limits of my language mean the limits of my world.
>
> – Ludwig Wittgenstein, *Tractatus Logico-Philosophicus* (1921)

2.4.1 Programming Languages

As demonstrated in previous sections, Domain-Specific Languages (DSLs) are increasingly integrated into software applications to provide specialized functionality, as evidenced in tools such as Spock, Gradle, and Jenkins. Before examining Domain-Specific Languages in detail, it is essential to establish a foundational understanding of programming languages.

Programming languages are formal, artificial languages designed to control and describe operations executed by computing systems. From a linguistic perspective, they comprise a *sequence of symbols* with an *associated semantics*. The *syntax* of a language determines the validity of symbol sequences within its framework. For execution on computing hardware, programs must undergo translation to a machine-interpretable representation, typically *binary code*. This translation process, known as *compilation*, is inherently machine-dependent; each target architecture requires a specific compiler to generate appropriate binary or assembly code from the source program.

Languages such as C, C++, Java, and Python are classified as *high-level* programming languages because they abstract away the underlying hardware details, such as register allocation and memory management. This abstraction allows programmers to focus on solution development rather than hardware-specific implementation concerns. These languages provide a higher conceptual level of operation, liberating developers from addressing low-level computational details and enabling concentration on the solution domain.

This abstraction represented a significant advancement in the evolution of computing, as direct hardware manipulation requires cognitive effort and is susceptible to errors that prove difficult to diagnose and correct. The fundamental principle of high-level language design was to establish a set of constructs sufficiently generic to express diverse problem domains. These languages introduced abstract concepts (such as classes and structures) that made natural domain representation easier.

2.4.2 Problem Domain

For effective application to real-world problems, a programming language must facilitate the formalization of domain-specific concepts, information structures, and constraints. In software engineering terminology, the *domain* refers to the contextual environment in which software operates, encompassing specialized knowledge, operational constraints, conceptual vocabulary, and related elements.

The domain-agnostic architecture of high-level languages enables them to model diverse domains using their generic constructs while maintaining hardware abstraction. This flexibility allows, for instance, the representation of a financial transaction as a Java class with appropriate fields containing domain-relevant information.

Table 2.5 delineates the comparative characteristics of the languages discussed in this analysis.

Language Type	Abstraction	Expressivity
Assembly	• Low-level • Hardware-specific • Machine instructions	• Hardware operations • Memory management • Architecture-specific • System-level only
General-Purpose	• High-level • Platform-independent • Reusable components	• Generic constructs • Standard algorithms • Multi-domain • Broad applicability
Domain-Specific	• Domain-focused • Business-level • Specialized abstractions	• Domain terminology • Specialized operations • Single domain • Deep coverage

Table 2.2. Comparison of Programming Language Types

Despite the significant advantages offered by high-level programming languages, domain definition and modeling remain among the most challenging aspects of software development [10].

Domain-Specific Languages constitute a category of programming languages designed for particular application contexts. Unlike general-purpose programming

languages such as Java or C, they are not intended for expressing generic computational operations and algorithms. Instead, DSLs are engineered to provide enhanced expressivity within their target domains, necessarily sacrificing generality to achieve this specialized functionality.

For illustrative purposes, while the C language permits the formalization of diverse algorithmic solutions, these solutions are expressed using C's general-purpose constructs, which maintain distance from the semantic structures of the specific problem domain. If generality and domain expressivity could be quantified as language attributes, they would demonstrate an inverse relationship; an increase in one characteristic necessarily results in a corresponding decrease in the other. Consequently, DSLs represent a deliberate optimization strategy that exchanges generality for enhanced domain expressivity.

Figure 2.5 provides a visual representation of the relationship between domain expressivity and generality across different programming language categories.

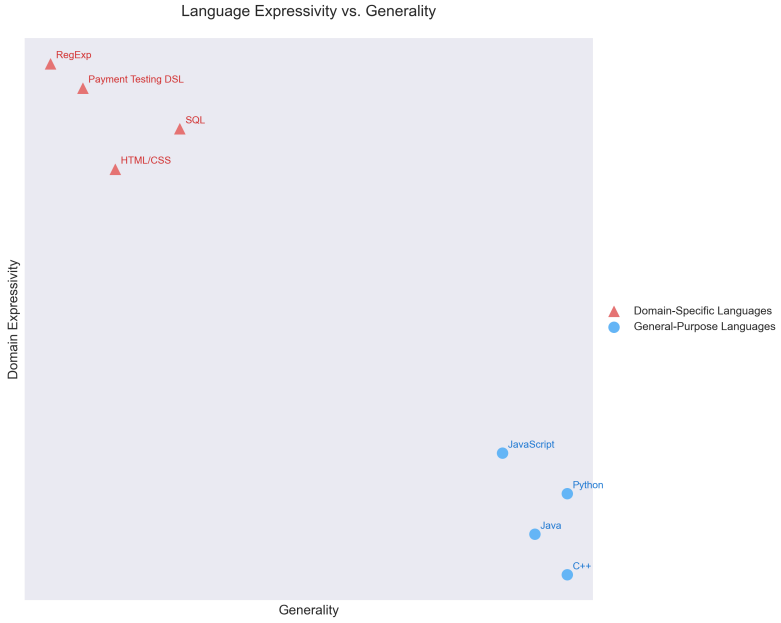

Figure 2.5. This visualization illustrates the trade-off between domain expressivity and generality across different types of programming languages. Domain-Specific Languages (triangles) excel in expressivity within their domains but have limited general applicability, while General-Purpose Languages (circles) offer broader applicability at the cost of domain-specific expressiveness.

Chapter 3

Domain Analysis and Challenges

3.1 Pay Reply

Figure 3.1. Pay Reply Logo

This thesis work was conducted within Pay Reply Srl, an Information Technology consulting firm with specialized expertise in digital payment technologies.

Pay Reply has been founded in 2012 as a component of the Reply network of companies, to provide support to banking institutions and major retail organizations in the integration of digital payment technologies. Beyond the integration of digital payment services, Pay Reply has developed specialized in Smart Point of Sale (POS) solutions. These advanced transaction terminals extend functionality beyond conventional payment processing to embed business-specific applications. Smart POS systems can manage different aspects of the business, including human resources management, loyalty program administration, inventory control and synchronization across distributed retails.

3.2 Analysis of the Case Study

This section presents an introduction to the real-world payment, which is the domain of the test framework developed in this work.

Figure 3.2. Smart POS Terminal (Source: it.mobiletransaction.org)

3.2.1 Digital Payments

Digital payments, alternatively called *electronic payments*, are value transfers exchanged through interconnected banking systems via telecommunication networks. They are distinguished from conventional cash transactions by the absence of physical currency exchange.

The term refers to transactions executed through electronic instruments, including payment cards, digital wallets, and direct bank account debits utilized for the acquisition of goods and services. This modality of transaction has become a fundamental component of contemporary economic systems.

3.2.2 Digital Payment Systems

The introduction of digital payments has fundamentally transformed the landscape of modern financial ecosystem. These systems represent the technological convergence of traditional financial services with contemporary digital infrastructure, facilitating secure, instantaneous, and traceable transactions through electronic channels. As organizational entities increasingly adopt digital transformation strategies, payment systems have evolved beyond simple transaction processing to encompass comprehensive financial management solutions, integrating seamlessly with enterprise resource planning frameworks, accounting applications, and banking infrastructure.

Digital payment adoption has demonstrated accelerated growth in recent years, driven by technological innovation, evolving consumer preferences, and organizational requirements for operational efficiency. Current market research and statistical analyses underline the significant growth and economic impact of digital payment systems. According to the McKinsey Global Payments Report (2023),

digital payments account for 12% of credit transfer inside the Single Euro Payments Area (SEPA) [26], and is forecasted to double by 2027. The World Bank Global Findex Database (2021) indicates that 64% of adults worldwide made or received at least one digital payment in 2021 [15]. In the commercial sector, Juniper Research (2024) forecasts that business-to-business digital payments will reach $124 Trillion Globally by 2028, highlighting these systems' critical function in contemporary commerce [2].

This transition is particularly evident in organizational financial management processes, where the value obtained by combining automation and cash flow management is greater. For commercial enterprises, digital payment systems deliver invaluable operational and strategic benefits through multiple mechanisms. First, payment processing and reconciliation automation significantly reduces operational expenses, with organizations typically reporting 40-50% cost reductions compared to traditional methodologies. Then, real-time transaction visibility enables detailed cash flow management, allowing organizations to optimize capital utilization and implement data-driven financial decision processes.

In summary, the evolution and adoption of digital payment systems is expected to keep growing, especially in SEPA where the anticipated PSD3, expected for 2026, promises to bring increased fraud protection for customers [4].

3.2.3 The General Payment Process

This section presents the main actors and steps involved in the processing of a digital payments. In particular, the following list describes the sequence of operations performed during a credit card payment transaction using a POS terminal:

1. The **Customer** (Cardholder) initiates a purchase transaction with a Merchant and selects card payment as the settlement method.

2. The **Merchant** (the product or service provider) processes the payment through a POS terminal.

3. The **POS Terminal** transmits the payment authorization request (containing the card data) to the Acquirer bank (the financial institution maintaining the merchant's account).

4. The **Acquirer**, operating under contract with the merchant, routes the transaction request to the appropriate Card Network associated with the presented card.

5. The **Card Network** directs the request to the Issuer Bank (the financial institution maintaining the customer's account).

6. The **Issuer**, which provided the payment card to the customer, verifies fund availability and communicates transaction authorization or denial to the Card Network.

7. The Card Network transmits the response to the Acquirer Bank.

8. The Acquirer Bank communicates the transaction authorization or rejection to the merchant.

9. The Acquirer Bank advances the transaction amount to the merchant through account credit.

10. The Acquirer Bank transmits to the Card Network a data stream containing confirmed transaction information, which the Card Network utilizes for transaction reconciliation and determination of the bank's credit/debit position.

11. The Card Network provides the calculated balance of amounts to be credited, adjusted for applicable fees and commissions.

This sequence of operations can be categorized into three distinct phases:
1. The **Authorization Phase** includes steps 1 through 8, focusing on transaction verification and approval. 2. The **Clearing Phase** span across steps 9 and 10, defined as the process of transmitting and reconciling payment transfer instructions prior to settlement. 3. The **Settlement Phase** is represented by step 11, constituting the transaction completion through the execution of inter-bank fund transfers to fulfill the obligations of all involved parties.

3.2.4 PaymentHub Solutions

A payment hub functions as a centralized platform for the management and processing of diverse payment transactions across multiple channels, currencies, and payment methodologies. It serves as a unified interface between varied payment systems, banking institutions, and financial organizations, effectively operating as a sophisticated transaction management system for financial operations.

The payment hub concept represents a platform where all organizational payment flows are transformed into a single, structured, and centralized information repository, generating unified inbound and outbound data streams. Payment information may originate from various Enterprise Resource Planning (ERP) systems or alternative payment initiation platforms utilized by organizational departments including accounting, treasury, finance, and human resources.

Figure 3.3 illustrates a representative payment hub architecture in which transaction flows from diverse channels converge within a centralized processing component.

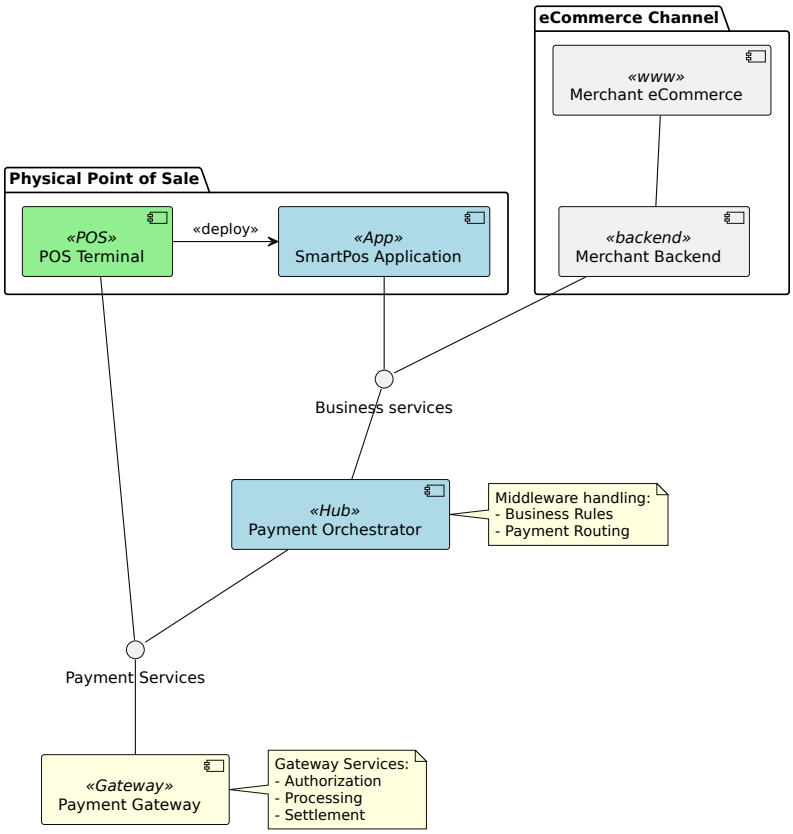

Figure 3.3. Representative Payment Hub Architecture

The architectural framework comprises two principal components: the *Payment Orchestrator* and the *Payment Gateway*. The former represents an internally developed system implementing business logic and routing payment requests to the gateway. The Payment Gateway functions as a Software-as-a-Service (SaaS) application providing payment functionality through HTTP Application Programming Interfaces (APIs). Both components are examined in greater detail in subsequent sections.

3.2.5 Payment Orchestrator

The payment orchestrator represents the core solution developed by Pay Reply, responsible for managing the business aspects of the payment hub operation. As previously described, it leverages the API functionality provided by the payment gateway to execute transaction operations, augmenting these capabilities with business-specific features.

The orchestrator's business responsibilities include installment management,

which is executed automatically on scheduled payment dates through MIT operations utilizing previously tokenized payment instruments.

Additionally, the orchestrator exposes APIs for registered merchants to initiate payment processes, access transaction records, configure installment parameters, and perform related operations. It functions as a middleware layer to the payment gateway, maintaining transaction records and enabling new transaction execution. This architecture provides a unified access point for financial services integration with business applications.

3.2.6 Payment Gateway

As previously indicated, the payment gateway exposes a set of APIs to facilitate payment operations. This section provides a concise description of these operations for contextual understanding.

Payment transactions are classified into two distinct categories: *Customer Initiated Transactions* (CIT) and *Merchant Initiated Transactions* (MIT). These operation types differ fundamentally in their execution processes. The former, as the designation implies, is initiated by the customer and requires direct interaction for the submission of payment card information. The latter operates without cardholder intervention and can be executed automatically by a system or administrative personnel.

An eCommerce purchase is an example a CIT, while an automated installment payment falls under the MIT category.

3.2.7 Security Considerations

The regulatory framework established by the European Union's Payment Services Directive 2 (PSD2) imposes strong privacy and security standard regarding payment card data manipulation. Since this information is critically sensitive, such data cannot be transmitted or processed over standard interfaces. Only certified PSD2 entities, such as the payment gateway, are authorized to manage card information.

For CIT operations, since only the payment gateway can access unencrypted card data, eCommerce platforms redirect customers to a PSD2-compliant web interface managed by the gateway, where payment information can be securely entered. The payment orchestrator requests the gateway to initiate a transaction and receives a redirection URL to which the customer is redirected. The customer then enters card information within the gateway's secure web pages, and once finished, the gateway notifies the orchestrator through a configured webhook callback.

For MIT operations, the system must be able to identify the payment card in order to issue a payment with it. In order to do so without directly storing it, a mechanism is required to reference the card across the two systems. The gateway

addresses this requirement through card tokenization, generating a character string identifier for each payment card. This token contains no actual card information and maintains complete data opacity while providing a consistent reference mechanism. This way, if a payment must be issued to a certain card, only the card token is stored and sent to payment gateway.

3.2.8 Context Diagram

The context diagram presented in Figure 3.4 illustrates the principal actors and systems interacting with the payment orchestrator, which constitutes the system under test in this analysis.

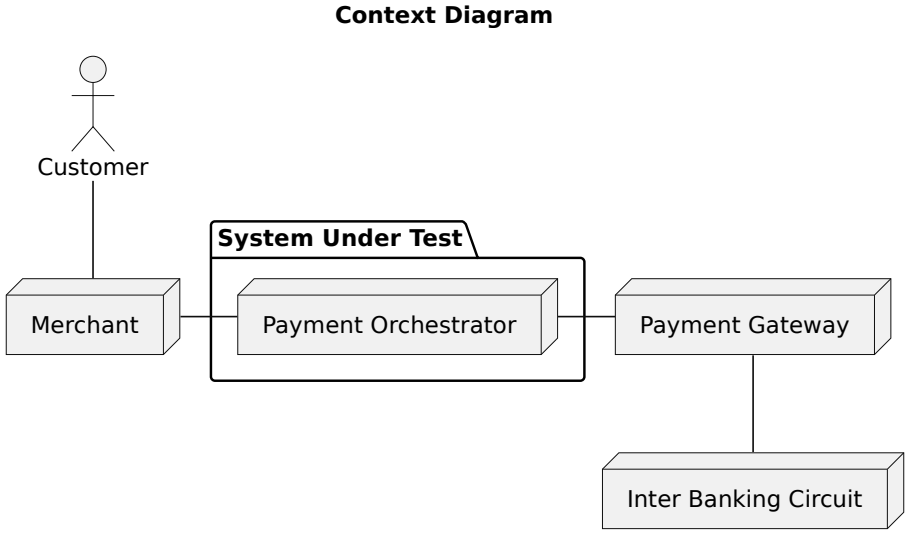

Figure 3.4. Context Diagram of the Payment System

3.2.9 Glossary

Figure 3.5 presents the principal concepts introduced in the preceding analysis.

3.3 Scenarios to be Tested

In this section are presented the use case scenarios that will be tested. A scenario is a narrative description of the use case with added details including user interaction with the system and expected behaviors.

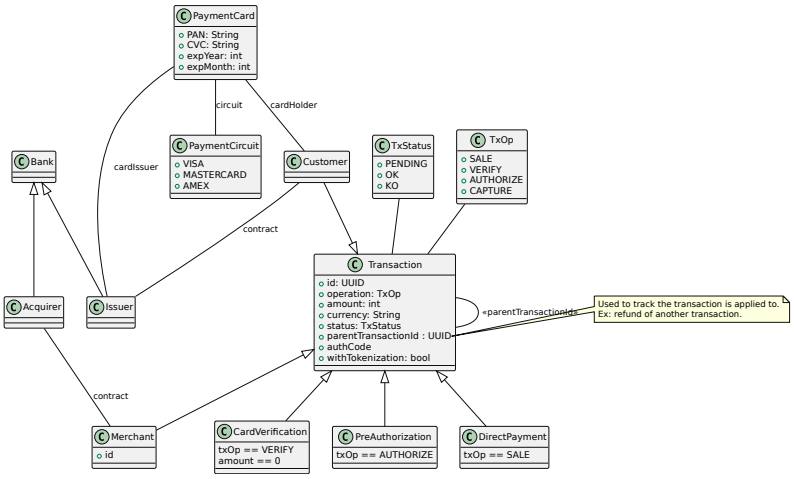

Figure 3.5. Glossary of Key Payment System Concepts

3.3.1 Direct Payment

The *direct payment* scenario represents one of the most critical use cases within the system. It constitutes the primary payment flow enabling value transfer from a merchant's customer in exchange for products or services. The principal scenario is described in Table 3.3.1.

3.3.2 Card Tokenization

This scenario represents an extension of the direct payment process. In addition to the primary transaction flow, the customer may select an option to "remember the payment method" for future transactions. When this option is activated, the payment card undergoes tokenization, and the resulting token is returned to the orchestrator for utilization in subsequent automated payment processes, eliminating the need for repeated card information entry.

3.3.3 Pre-Authorization

Pre-Authorization represents a common methodology in payment processing when the final transaction amount cannot be determined prior to payment card processing. This approach is frequently implemented in contexts such as fuel dispensing stations to verify fund availability and prevent fraudulent transactions.

In this scenario, a specified amount is pre-authorized from the customer's available funds, rendering it temporarily unavailable for alternative uses. Once the actual transaction value is determined (e.g., after fuel dispensation), only the corresponding amount is transferred, with the remaining pre-authorized funds released

Table 3.1. Action «Direct Payment»

Action	**Direct Payment**
Precondition	The customer initiates a transaction completion process on the merchant's eCommerce platform.
Postcondition	The verify-transaction API indicates that the transaction status is "OK".
Actors	1. Customer accessing the eCommerce platform 2. Payment Orchestrator 3. Payment Gateway
Main path	1. The orchestrator receives a start-payment request 2. The orchestrator preserves the transaction data and forwards the payment request to the payment gateway 3. The gateway establishes a new transaction and returns the redirect URL to the payment interface 4. The customer is redirected to the payment interface, enters payment card information, and confirms the transaction 5. The orchestrator receives a verify-payment request, which it forwards to the gateway, obtaining confirmation that the transaction status is "OK"

back to the customer's account.

The pre-authorization process follows a sequence identical to the direct payment flow, the only difference being in the transaction type. Rather than immediate fund transfer, pre-authorization reserves funds and enables subsequent "capture" of a potentially lower amount at a later time.

3.3.4 Merchant Initiated Transaction

As previously described, merchant initiated transactions proceed without customer interaction. The sole prerequisite for this transaction type is the availability of a previously tokenized payment card. Table 3.3.4 outlines the MIT process.

Table 3.2. Action «MIT»

Action	**MIT**
Precondition	The customer's payment card has been previously tokenized.
Postcondition	The transaction status is "OK".
Actors	1. Payment Orchestrator 2. Payment Gateway
Main path	1. The orchestrator receives a MIT request specifying the payment card token to be utilized. It forwards this request to the gateway. 2. The gateway processes the transaction and returns the status information. 3. The orchestrator verifies that the transaction status is "OK".

3.4 Testing Challenges and Solutions

The preceding analysis has demonstrated that the payment orchestrator supports multiple payment scenarios and thus requires extensive testing. The principal challenges in testing methodology relate to the labor-intensive operations necessary for end-to-end payment verification, including manual API invocation, browser navigation for test card data entry, and transaction verification procedures.

Unfortunately, the diversity of payment scenarios exponentially increases the number of required tests, rendering manual execution increasingly time-consuming and resource-intensive.

The subsequent chapters will present a solution for automated test execution utilizing contemporary testing frameworks and web automation tools. Additionally, the research will introduce a Domain-Specific Language specifically designed for test definition and parameter specification, including payment card data, operation types, and related testing variables.

Chapter 4

Methodology

This chapter presents the methodological contribution developed for the testing framework. The analysis begins with an examination of the JUnit framework and its testing capabilities, followed by an identification of the domain-specific limitations encountered in the standard implementation. Subsequently, the chapter introduces a novel approach to overcome these constraints while preserving the framework's core advantages, including comprehensive report generation and concurrent test execution capabilities. The methodology demonstrates the implementation of a Domain-Specific Language (DSL) designed to enhance test suite definition through improved descriptive capabilities.

4.1 JUnit 5

JUnit represents one of many libraries available for testing Java-based applications. It maintains the position of the most frequently downloaded library in the Maven repository under the "Testing Frameworks and Tools" category and has maintained continuous development since 1997, establishing it as the longest-maintained testing framework in the Java ecosystem. Its intuitive Application Programming Interface (API) has established foundational patterns for subsequent libraries that extend its functionality, most notably TestNG and Spock. JUnit has become the de facto standard for unit testing and general test execution within the Java ecosystem, receiving first-class support from Integrated Development Environments (IDEs) and project management tools including Maven, Ant, and Gradle. The decision to implement JUnit for this research was based on its widespread adoption and extensible API, complemented by its report generation capabilities, concurrent test execution support and dynamic test definition functionality.

JUnit 5.0.0, released on September 10, 2017, approximately 11 years after the 4.0 release, is designed with a modular architecture composed by three distinct components [24]. These modules include:

- **junit-platform**: A common infrastructure module providing test suite execution capabilities, supporting both jupiter API implementations and legacy implementations through the junit-vintage compatibility layer.
- **junit-vintage**: A compatibility module maintaining support for version 4 and version 3 test engines through segregation within a separate module. Tests implemented with junit-vintage maintain compatibility with legacy features while supporting execution through the current junit-platform architecture. This backward compatibility has been crucial for many legacy Java projects that have not migrated to the jupiter implementation.
- **junit-jupiter**: The contemporary, feature-enhanced test engine released independently from its predecessor.

Figure 4.1 illustrates the dependency relationships between these modules and their respective artifacts.

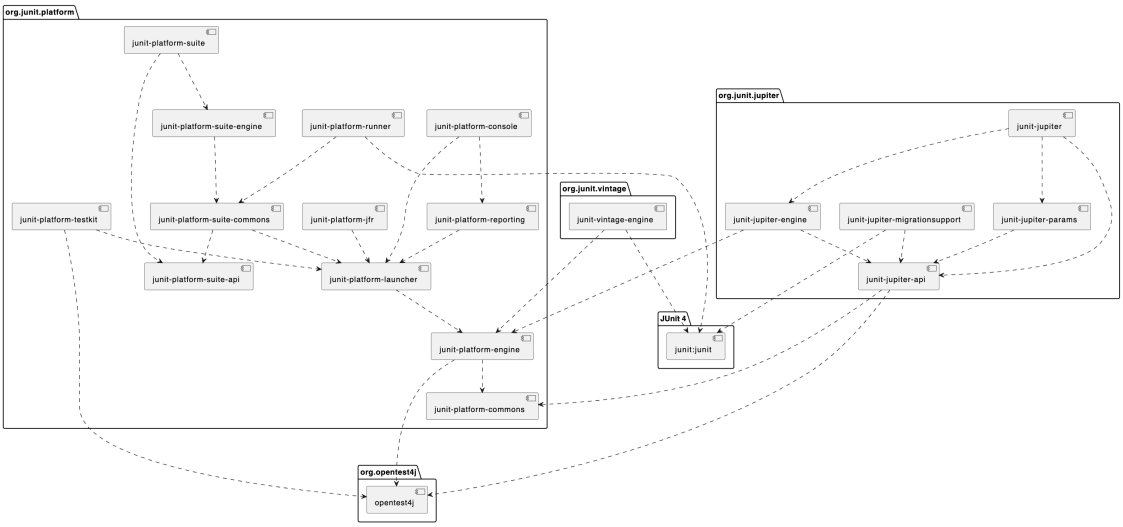

Figure 4.1. Dependency Diagram of JUnit Architecture

This research employed the 5.12.0[1] released in February 2025, which was under development during the research period. This version introduces enhanced reporting capabilities essential for the current implementation. These features are examined in detail in Section 4.2.5.

JUnit provides multiple test definition approaches to accommodate diverse testing scenarios. The subsequent sections present both fundamental and advanced

[1]Release note available at: https://junit.org/junit5/docs/5.12.0/release-notes/

testing methodologies supported by the framework. Prior to examining the specific implementation methodology, it is necessary to evaluate the framework's core capabilities and identify areas requiring extension to support the objectives of this research. For the proposed solution to function as designed, the following capabilities are required:

- Programmatic test execution, as detailed in Section 4.2.3.
- Parameterized test execution with customizable inputs, for which TestTemplate and DynamicTests present viable implementation options, albeit with certain operational constraints.
- Test report generation with possibility to add attachments such as captured in-browser screenshot.

To comprehend JUnit's operational architecture, it is essential to first understand fundamental Java annotation concepts and their implementation advantages.

4.2 Java Annotations

Java annotations represent one of the language's most widely implemented features, extensively utilized by frameworks including Spring for configuration purposes. JUnit similarly employs annotations extensively for test configuration, parameter provision through extensions, and related functionality.

At their core, Java annotations enable metadata attachment to annotated elements, which annotation processors can subsequently utilize to modify the behavior of these elements based on the specified annotation parameters.

Annotations can be categorized according to the following criteria:

- Retention type: Defines the annotation management policy. Compile-time annotations are removed during compilation and absent from generated code, while runtime annotations persist in the compiled output and remain accessible to runtime processors. Typically, code-generating annotations such as Lombok's @Data or @Sl4j operate exclusively at compile time, as their utility terminates once the annotation processor completes its operations (which includes enriching the annotated class with methods and additional fields).
- Target specification: Defines the elements to which the annotation may be applied, including classes, methods, fields, or other annotations.

Listing 4.1 demonstrates the implementation of JUnit's fundamental @Test annotation:

```
1  package org.junit.jupiter.api;
2
3  import static org.apiguardian.api.API.Status.STABLE;
4
5  import java.lang.annotation.Documented;
6  import java.lang.annotation.ElementType;
7  import java.lang.annotation.Retention;
8  import java.lang.annotation.RetentionPolicy;
9  import java.lang.annotation.Target;
10
11 import org.apiguardian.api.API;
12 import org.junit.platform.commons.annotation.Testable;
13
14
15 @Target({ ElementType.ANNOTATION_TYPE, ElementType.METHOD })
16 @Retention(RetentionPolicy.RUNTIME)
17 @Documented
18 @API(status = STABLE, since = "5.0")
19 @Testable
20 public @interface Test {
21 }
```

Listing 4.1. JUnit Test annotation

The @Target annotation on line 15 indicates that this annotation may be applied exclusively to methods and other annotations. Line 16 specifies a runtime retention policy through the @Retention annotation.

Annotations have gained widespread adoption because they enable declarative programming paradigms, allowing developers to specify desired outcomes rather than implementation mechanisms. As the following section illustrates, annotations constitute a fundamental component of JUnit's API architecture.

4.2.1 Writing Tests with junit-jupiter

Standard Tests with @Test

In JUnit, fundamental test cases can be defined using the @Test annotation as demonstrated below:

```
1  import static org.junit.jupiter.api.Assertions.assertEquals;
2
3  import example.util.Calculator;
4
5  import org.junit.jupiter.api.Test;
6
7  class MyFirstJUnitJupiterTests {
```

```
8
9       private final Calculator calculator = new Calculator();
10
11      @Test
12      void addition() {
13          assertEquals(2, calculator.add(1, 1));
14      }
15
16  }
```

Listing 4.2. JUnit Test Definition

The @Test annotation identifies a method as a test case. When the JUnit test launcher scans the test class and identifies methods annotated with @Test, these methods are processed as test sources.

4.2.2 Dynamic Tests with @TestFactory

A significant limitation of basic test implementations is their predefined nature—there exists no mechanism to modify test behavior or input data at runtime. Each test executes with fixed parameters and produces success or failure outcomes accordingly.

As stated in the JUnit documentation:

> These test cases are static in the sense that they are fully specified at compile time, and their behavior cannot be changed by anything happening at runtime.

While this approach satisfies many real-world testing requirements, it presents limitations for the objectives of this research. The primary goal is to develop a framework where test definitions and parameters can be dynamically modified at runtime based on external test specification files.

The JUnit documentation further describes:

> In addition to these standard tests a completely new kind of test programming model has been introduced in JUnit Jupiter. This new kind of test is a dynamic test which is generated at runtime by a factory method that is annotated with @TestFactory.
>
> [...]
>
> A DynamicTest is a test case generated at runtime. It is composed of a display name and an Executable. Executable is a @FunctionalInterface which means that the implementations of dynamic tests can be provided as lambda expressions or method references.

Dynamic tests provide maximum flexibility in test definition, requiring only an implementation of a functional interface to derive the test. This enables runtime test generation rather than static specification.

Dynamic tests proved ideal for addressing the requirements of this implementation, as they allow for runtime test definition through the return of a Collection, Stream, or Iterable of `DynamicNode` objects. When such a collection is provided to the JUnit engine, these objects are processed as valid tests.

Listing 4.2.2 demonstrates the implementation of dynamic tests:

```
1  import static example.util.StringUtils.isPalindrome;
2  import static org.junit.jupiter.api.Assertions.assertEquals;
3  import static org.junit.jupiter.api.Assertions.assertTrue;
4  import static org.junit.jupiter.api.DynamicTest.dynamicTest;
5
6  import java.util.Arrays;
7  import java.util.Collection;
8
9  import example.util.Calculator;
10
11 import org.junit.jupiter.api.DynamicTest;
12 import org.junit.jupiter.api.TestFactory;
13
14 class DynamicTestsDemo {
15
16     private final Calculator calculator = new Calculator();
17
18     @TestFactory
19     Collection<DynamicTest> dynamicTestsFromCollection() {
20         return Arrays.asList(
21             dynamicTest("1st dynamic test", () -> assertTrue(
                   isPalindrome("madam"))),
22             dynamicTest("2nd dynamic test", () ->
                   assertEquals(4, calculator.multiply(2, 2)))
23         );
24     }
25 }
```

Listing 4.3. JUnit Dynamic Test

While the API exhibits straightforward implementation, certain operational constraints must be considered, as documented in the JUnit specifications. Dynamic tests do not adhere to the standard test lifecycle of JUnit tests. For example, the BeforeEach callback, which normally executes before each test, is invoked only once before the test container initiates execution of all dynamic tests.

This constraint applies similarly to injected parameters. JUnit allows to define parameters for test methods, which are subsequently injected by external extensions

or the framework itself using the `ParameterResolver` interface. This capability allows the addition of a `TestReporter` to a test, which is a special class used to report data about the test execution..

With dynamic tests, these parameters are associated with the top-level container and remain consistent across all tests. Consequently, an injected test reporter would report data as originating from the test container rather than individual tests. This consideration is particularly relevant when implementing data reporting for dynamic tests, where a solution must be developed to overcome this limitation (more on this later).

4.2.3 Running Tests Programmatically

While test execution is typically managed by project management tools such as Gradle or Maven, the proposed solution requires test execution as a *component of the main application*. Conventionally, JUnit is defined as a *test-only* dependency to prevent its inclusion in compiled applications, restricting its use to the testing phase. Similarly, classes in the `src/test` directory are accessible only during the test phase.

To execute tests as part of the main application, the following requirements must be satisfied:

- JUnit must be available on the main application classpath
- The test classes must also reside on the main classpath

The first requirement can be fulfilled by specifying the appropriate dependency retention policy (dependency scope) using project management tool utilities. In Gradle, this is accomplished by labeling the dependency as `implementation` rather than `testOnly`, while Maven utilizes the `scope` property (test, compile, etc.) to define dependency scope.

Listing 4.2.3 demonstrates the configuration of JUnit dependencies in Gradle for availability as main application dependency:

```
1 dependencies {
2     // JUnit Platform
3     implementation platform('org.junit:junit-bom:5.12.0-
          SNAPSHOT')
4     implementation 'org.junit.platform:junit-platform-
          launcher'
5     implementation 'org.junit.platform:junit-platform-
          reporting'
6
7     // JUnit Jupiter
8     implementation 'org.junit.jupiter:junit-jupiter-engine'
9     implementation 'org.junit.jupiter:junit-jupiter'
```

```
}
```

Listing 4.4. Gradle JUnit Implementation Dependency

With junit-platform and junit-jupiter engine accessible from the main source code, the launcher API can be employed to invoke tests directly from the main application, as demonstrated in the following example:

```
package it.payreply.cli;

import it.payreply.tests.MyTestClass;
import lombok.extern.slf4j.Slf4j;
import org.junit.platform.engine.discovery.DiscoverySelectors;
import org.junit.platform.launcher.core.LauncherDiscoveryRequestBuilder;
import org.junit.platform.launcher.core.LauncherFactory;

import java.nio.file.Files;
import java.util.List;


@Slf4j
public class TestRunnerApplication {

    public static void main(String[] args) throws Exception {

        // create the JUnit test launcher
        var junitLauncher = LauncherFactory.create();

        // create a test execution request via the builder
        var junitTestRequest = LauncherDiscoveryRequestBuilder.request()
                // use DiscoverySelectors to select which test class to run
                .selectors(
                        DiscoverySelectors.selectClass(RunGroovyTests.class.getName())
                )
                // eventually add some parameters with configurationParameter(), skip for now
                .build();
        junitLauncher.execute(junitTestRequest);
    }
}
```

Listing 4.5. Launching Tests Programmatically

4.2.4 Test Report Generation and TestExecutionListener

An essential component of the testing process is the generation of comprehensive reports containing relevant test results information. These reports typically include test status (pass/fail), exception details for failed tests, and potentially log entries written to standard output and standard error during test execution.

JUnit enables report generation through the registration of one or more TestExecutionListener implementations, which intercept lifecycle events during test execution. The TestExecutionListener interface is illustrated in Figure 4.2.

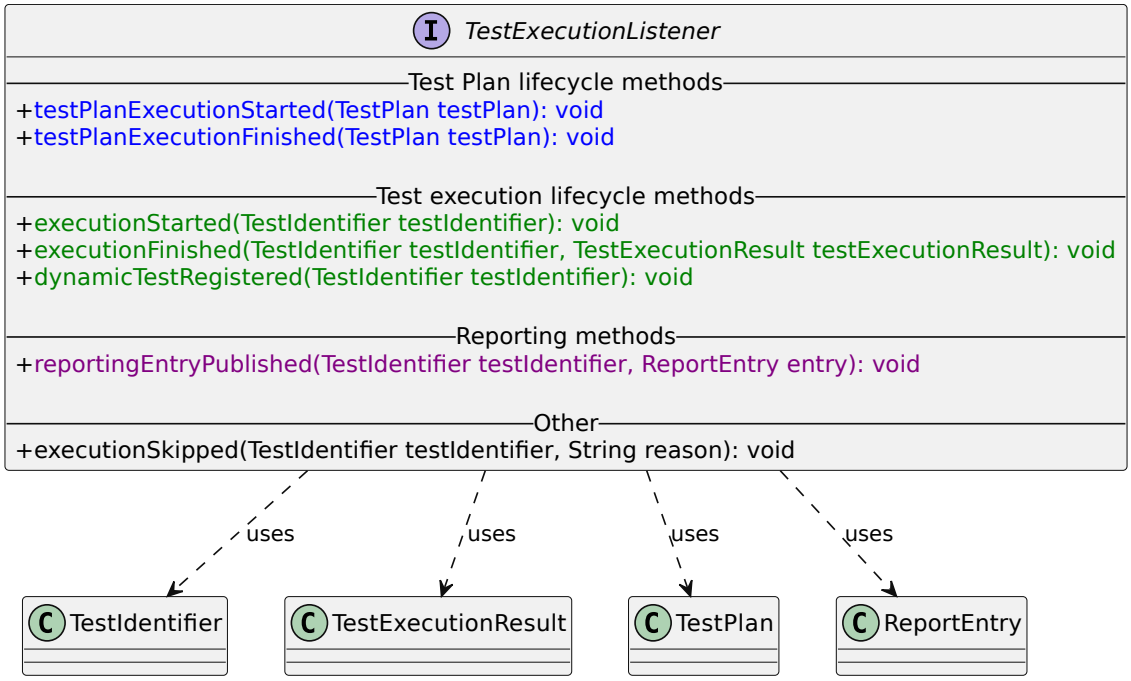

Figure 4.2. JUnit 5 TestExecutionListener Interface

The methods of TestExecutionListener implementations are invoked upon specific test execution lifecycle events. These events may relate to the entire test execution (represented by the TestPlan) or individual tests (identified by TestIdentifier objects).

TestIdentifier objects are "Immutable data transfer objects that represent tests or containers which are usually part of a TestPlan." TestIdentifiers are categorized as either `containers` or actual `tests`. Containers do not represent actual tests but serve as sources for the tests they contain. For example, in a test class containing multiple methods annotated with @Test, the class functions as a TestIdentifier container for the test-type TestIdentifiers associated with its methods.

The entire test execution can be conceptualized as a tree structure where the root TestIdentifier represents the JUnit engine container, which contains additional

TestIdentifier containers. The leaves of this tree structure are the actual tests, represented by test-type TestIdentifiers.

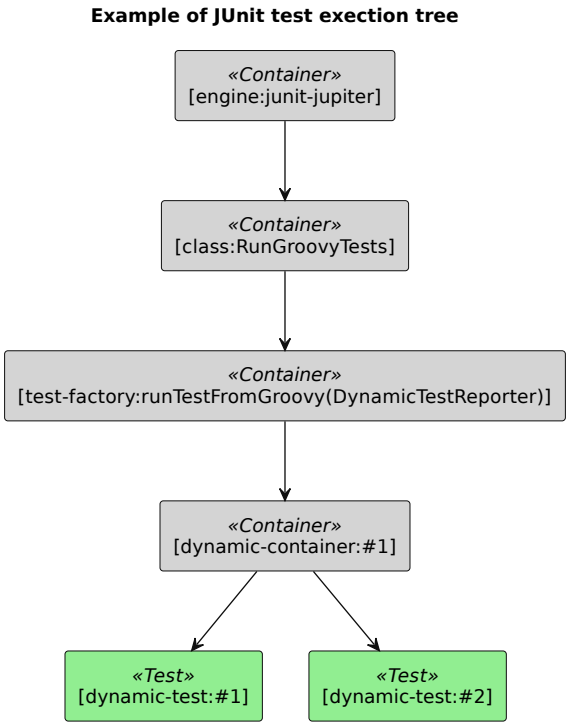

Figure 4.3. JUnit Test Execution Tree Example

The `reportEntryPublished` method of the TestExecutionListener is invoked whenever a test calls the `publishEntry` method on the auto-wired `TestReporter`, as demonstrated in Listing 4.6. Reporting entries enables the attachment of contextual data to tests for inclusion in the reporting infrastructure.

The TestReporter is automatically injected by JUnit and associated with the current TestIdentifier. However, due to the distinctive behavior of dynamic tests mentioned earlier, the TestReporter is not linked to individual dynamic tests but rather to the parent container. To address this limitation, the JUnit maintenance team developed a workaround through the implementation of a DynamicTestReporter extension, as documented in the GitHub discussion.

```
1 class TestReporterTest {
2
3     @Test
4     void reporterExample(TestReporter testReporter) {
5         var actual = 1 + 1;
6         assertEquals(2, actual);
7         testReporter.publishEntry("expected", "2");
```

```
8           testReporter.publishEntry("actual", String.valueOf(
               actual));
9           var myMapEntry = new HashMap<String, String>();
10          myMapEntry.put("1", "1");
11          myMapyEntry.put("2", "2");
12          // published togheter
13          testReporter.publishEntry(myMapEntry);
14      }
15
16  }
```

Listing 4.6. TestReporter Example

4.2.5 JUnit 5.12

This research employed JUnit 5.12.0 due to its enhanced reporting infrastructure, which, in addition to previously described capabilities, includes functionality for file and directory publication. This feature is required for publishing screenshots captured during Selenium web browser sessions. Specifically, the following methods were added to the TestReporter (with corresponding implementations in the TestExecutionListener interface):

- `void publishFile(Path file, MediaType mediaType)`
- `void publishDirectory(Path directory)`

4.3 Open Test Reporting

Open Test Reporting [25] represents an initiative by the JUnit team aimed at standardizing the Java testing ecosystem. Its primary objective is to establish a common foundation for Java testing procedures applicable across widely used frameworks.

Historically, testing frameworks have implemented proprietary approaches to represent test session execution and generate associated reports. The Open Test Reporting initiative seeks to standardize XML-based test reporting, simplifying the development of common reporting infrastructure. This standardization benefits both testing frameworks, by eliminating the need to develop proprietary reporting systems, and reporting infrastructure components such as project management tools and IDEs, which can now support a single reporting protocol.

Specifically, Open Test Reporting provides a standardized event-based XML output format and an HTML report generator for human-readable visualization of test results. An example of report page is available in Figure 4.4.

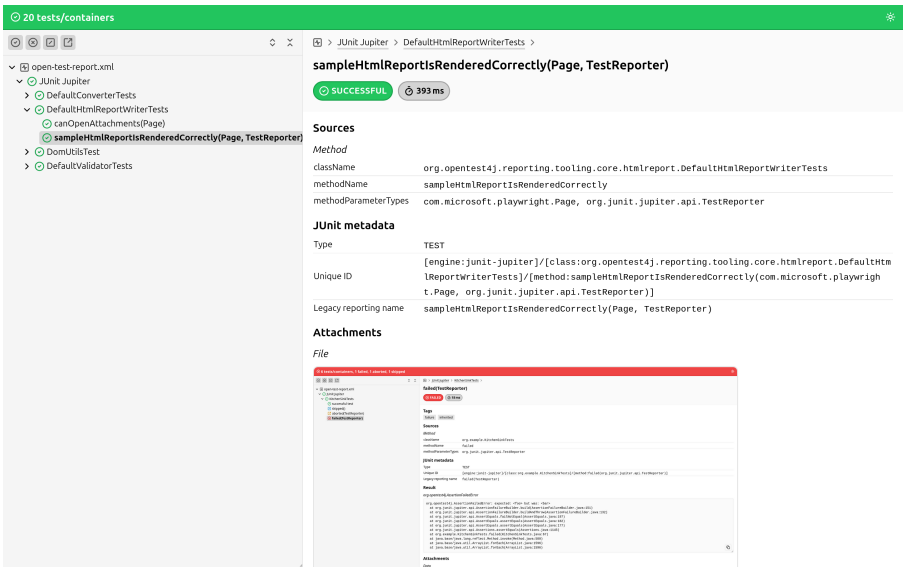

Figure 4.4. An example of produced HTML report using open-test-reporting.

To implement Open Test Reporting, the LauncherDiscoveryRequest must be configured to utilize the new reporting platform through the provision of additional configuration parameters enabling the Open XML reporting format.

4.4 Replacing Manual Postman HTTP Requests

A critical component of the testing framework involves issuing HTTP requests to the system under test. In manual testing scenarios, Postman represents the preferred tool, providing a graphical user interface for HTTP request configuration, including host URL specification, request method and body definition, authentication header configuration, and related parameters.

Since the objective of this research is to replace manual operations with programmatic testing, Postman functionality must be replicated through a Java-accessible HTTP client. While numerous HTTP client implementations exist, specific requirements must be considered. Due to security constraints, the HTTP client must support proxy configuration (as explained in Section 4.4.2) and implement request signing using a cryptographic algorithm based on pre-shared keys.

The signing algorithm requires the addition of specific headers to each request, including host, date, and digest (the SHA-256 hash of the request body, when present), and their subsequent signature using the HMAC-SHA256 algorithm [2] to

[2] https://datatracker.ietf.org/doc/html/rfc2104

ensure message authentication. The resulting signature is incorporated into the Authentication header before request transmission. On the server side, the pre-shared key is used to verify the message integrity and authentication.

4.4.1 OkHttp and Retrofit

OkHttp[3] and Retrofit[4] are Java libraries developed by Square Inc.[5]. OkHttp implements the HTTP client API, while Retrofit builds upon this foundation to create comfortable service interfaces.

These libraries provide the necessary capabilities to implement the specified requirements through an intuitive programming model. The OkHttpClient can be customized with `Interceptor` implementations to modify requests before transmission. This architecture enables implementation of the previously described signature algorithm through the creation of a custom interceptor added to the processing chain. Additionally, the HttpLoggingInterceptor provides request logging to standard output, which are valuable for test analysis purposes.

Listing 4.7 demonstrates the configuration of these interceptors:

```
1 var okHttpClient = new OkHttpClient.Builder()
2     .addInterceptor(new SigningRequestInterceptor(merchant.
          getKeyId(), merchant.getKeySecret()))
3     .addInterceptor(new HttpLoggingInterceptor().setLevel(
          HttpLoggingInterceptor.Level.BODY)) //new
          OkHttpCallReporter(testReporter)))
4    .proxy(testEnvironmentSpec.getProxyJump().newProxy())
5    .build();
```

Listing 4.7. OkHttpClient Configuration

This example illustrates the client configuration with a specific proxy for request transmission, as detailed in subsequent sections.

Once an OkHttpClient is configured, it can be utilized to construct a `Retrofit` interface to the target service. Retrofit enables the definition of HTTP services as Java interfaces to invoke HTTP services through standard Java method calls. The interfaces are instantiated through a builder object using an OkHttpClient pre-configured with appropriate interceptors. Listing 4.8 provides an example of Retrofit interface instantiation.

```
1 package it.payreply.paytests.domain.services;
2
3 public interface PaymentApi {
```

[3] https://square.github.io/okhttp/

[4] https://square.github.io/retrofit/

[5] https://squareup.com/us/en

```
4
5       @POST("start-payment")
6       Call<StartPaymentResponse> startPayment(@Body
            PaymentRequestDTO paymentRequestDTO);
7
8       @GET("verify-payment/{transactionId}")
9       Call<VerifyPaymentResponse> verifyPayment(@Path("
            transactionId") String transactionId);
10  }
11
12  // interface instantiation
13  Retrofit retrofit = new Retrofit.Builder()
14      .client(okHttpClient)
15      .baseUrl(testEnvironmentSpec.getFullBaseUrl())
16      .addConverterFactory(
17          GsonConverterFactory.create()
18      )
19      .build();
20
21  PaymentApi paymentApi = retrofit.create(PaymentApi.class);
```

Listing 4.8. HTTP Service Interface Built with Retrofit

4.4.2 Connection to the Test Environments via SSH Proxies

Security protocols restrict direct internet exposure of servers hosting the APIs accessed by the HTTP client. Instead, these servers are accessible only through a *bastion host* located within the same Local Area Network (LAN), which itself is accessible through a Virtual Private Network (VPN) connection. This architecture implements two security levels: the VPN protocol securing access to the bastion host, and a secondary security layer controlling access to test servers through the bastion host. This network topology is illustrated in Figure 4.5.

The bastion host exposes the SSH protocol on port 22, which can be utilized to establish a SOCKS5 service for dynamic port forwarding. This is typically accomplished using the `-D port` option of the `ssh` command when establishing a connection. With this option, the bastion host functions as a SOCKS5 server, creating an SSH tunnel from the specified local port to the bastion host port providing the SOCKS5 service.

SOCKS5 is a network protocol implementing *dynamic port forwarding*, which processes network packets containing metadata about the intended destination IP address and port, forwarding these packets accordingly. When responses are received, they are returned to the original sender.

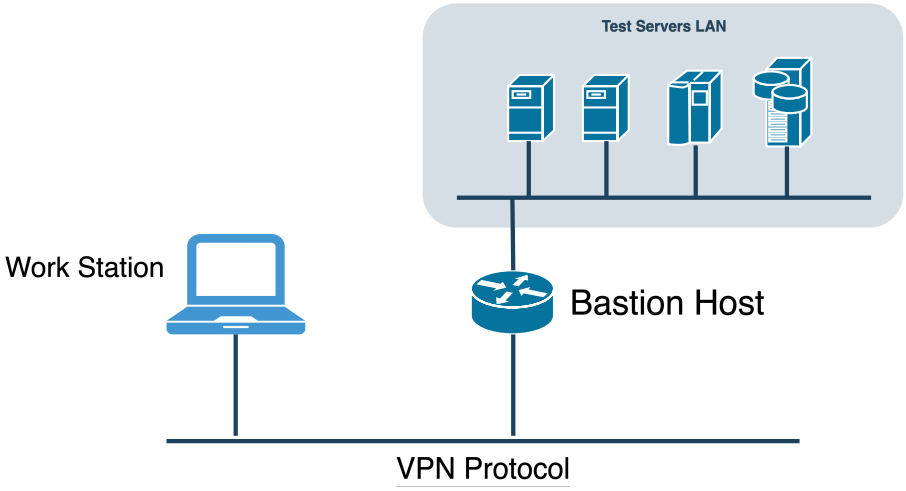

Figure 4.5. Network Layout of the Test Environment

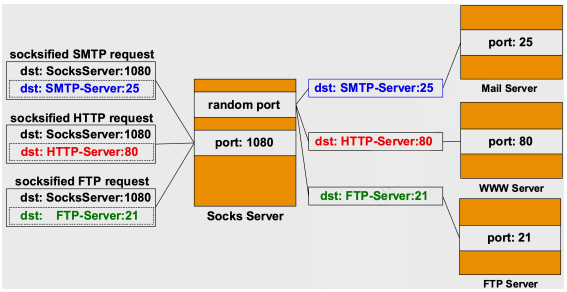

Figure 4.6. SOCKS5 Implementation Details. IP Packets are Enhanced with SOCKS5 Application-Level Data Containing Information About the Intended Destination.

4.4.3 Apache Mina SSHD

While SOCKS5 proxies over SSH can be established using the ssh command, the developed solution employs the Apache Mina library and its SSH package to configure the proxy programmatically through Java APIs.

Apache Mina[6] is a Java networking library supporting diverse network protocols in a programmatically accessible manner. It includes a specialized package for SSH connection management, Apache Mina SSHD[7].

Listing 4.9 demonstrates the use of `org.apache.sshd` classes to establish dynamic port forwarding (SOCKS5) over an SSH connection. This implementation

[6] https://mina.apache.org/

[7] https://mina.apache.org/sshd-project/

requires only the 'org.apache.sshd:sshd-core' dependency. The resulting proxy can be utilized to configure the OkHttpClient for proxy-based request transmission.

```
import org.apache.sshd.client.SshClient;
import org.apache.sshd.client.future.ConnectFuture;
import org.apache.sshd.client.session.ClientSession;
import org.apache.sshd.common.util.net.SshdSocketAddress;

import java.io.IOException;
import java.net.InetSocketAddress;
import java.net.Proxy;
import java.util.concurrent.TimeUnit;


public class ProxyJump {
    // ...

    public Proxy getProxy() {
        log.info("Starting ssh proxy");
        var sshClient = SshClient.setUpDefaultClient();
        try {
            // Open the client
            sshClient.start();

            // Connect to the server
            ConnectFuture cf = sshClient.connect(username,
               host, port);
            ClientSession session = cf.verify().getSession();
            session.addPasswordIdentity(password);
            session.auth().verify(TimeUnit.SECONDS.toMillis
               (3000));

           log.info("Connected to the ssh host");

            // setup the socks proxy using the established
               ssh session
            session.startDynamicPortForwarding(new
               SshdSocketAddress("localhost", localPort));
            return new Proxy(Proxy.Type.SOCKS, new
               InetSocketAddress("localhost", localPort));
        } catch (IOException e) {
            throw new RuntimeException(e);
        }

    }
```

Listing 4.9. Apache Mina SOCKS5-SSH Proxy Configuration

4.5 Selenium Web Driver

Selenium WebDriver[8] is a browser automation framework extensively utilized for end-to-end testing of web applications. It enables the simulation of user interactions with web pages, including navigation, element identification, button activation, and form input.

The architecture comprises three principal components:

User Agent The actual browser binary being controlled, such as Chrome, Firefox, or Safari.

WebDriver The component issuing commands to the user agent. It functions as a server, translating HTTP requests into user agent commands. The interface between the WebDriver and user agent is browser-specific, necessitating distinct WebDriver implementations for each supported browser.

Selenium Bindings The programming language-specific API interfacing with the WebDriver. These APIs fundamentally transmit HTTP requests to the WebDriver, which subsequently issues corresponding commands to the user agent.

Figure 4.7 illustrates this architectural relationship:

The WebDriver represents a specification that each browser implements independently. Two specifications currently exist: the JsonWire Protocol [9] and the W3C WebDriver [10]. While JsonWire represents the original open-source specification developed by the Selenium team, it is no longer actively maintained. The W3C specification represents the current standard and the focus of ongoing development.

According to the WebDriver protocol documentation[11]:

> WebDriver is a remote control interface that enables introspection and control of user agents. It provides a platform- and language-neutral wire protocol as a way for out-of-process programs to remotely instruct the behavior of web browsers.

4.5.1 Selenium WebDriver Manager

While the general architecture of the Selenium/WebDriver stack functions in theory, it presents a practical challenge: effective communication between the WebDriver and browser requires version compatibility. Since the WebDriver is typically

[8] https://www.selenium.dev/documentation/webdriver/

[9] https://www.selenium.dev/documentation/legacy/json_wire_protocol/

[10] https://w3c.github.io/webdriver/

[11] https://www.w3.org/TR/webdriver2/

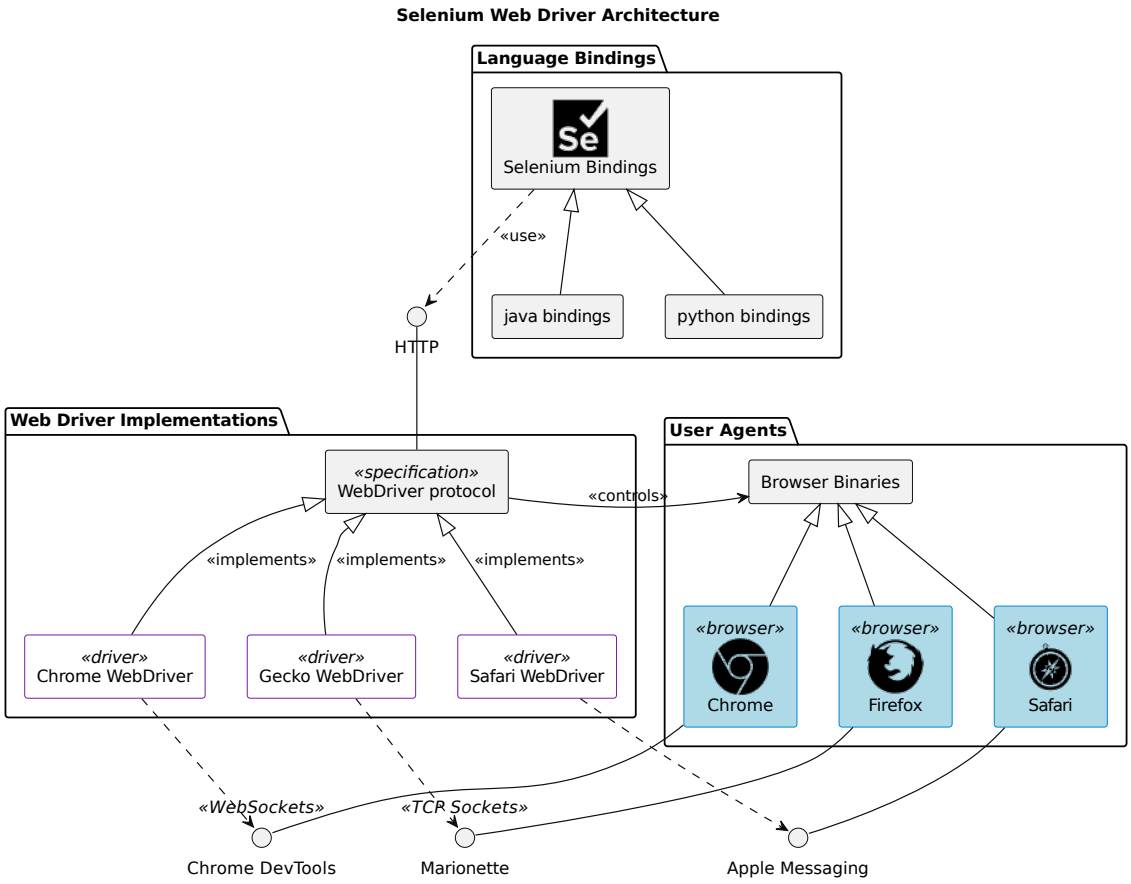

Figure 4.7. Component Diagram of the Selenium Architecture

downloaded manually as a standalone executable, it requires proper management to maintain compatibility with the browser version.

Browsers implement aggressive update strategies to address security vulnerabilities, often updating automatically without user intervention. While this practice enhances security, it frequently causes runtime failures in WebDriver-browser communication due to version incompatibilities.

To address this issue, various solutions have been developed to automate driver management and prevent version conflicts. These solutions typically involve selecting a target browser (Chrome, Firefox, Safari) and automatically configuring the corresponding WebDriver version without manual intervention.

This research employs WebDriverManager[12] developed by Professor Boni Garcia[13] of Universidad Carlos III de Madrid. In addition to automated driver configuration, WebDriverManager supports browser execution in Docker containers and provides a JUnit extension, Selenium Jupiter[14], which facilitates the integration of Selenium into JUnit test lifecycles through the JUnit extension mechanism. While Selenium Jupiter was not required for this implementation, WebDriverManager significantly reduced potential Selenium failures.

4.5.2 The Page Object Model

In the Java ecosystem, Selenium implementation frequently employs the Page Object Model pattern[15]. This pattern represents each user-accessible page with a corresponding class implementing methods for web page interaction. A primary advantage of this approach is the decoupling of test flow logic from page interaction implementation. When user interface changes necessitate interaction modifications (e.g., selector name changes), only the page class requires updating, while the test flow logic remains unmodified.

Figure 4.8 presents the Page Object Model classes designed for the application testing framework. The diagram includes classes from the selenium-java module used for driver interaction (package org.openqa.selenium). The By classes locate elements within web pages, while the WebElement and Select classes represent Java models of DOM elements used for interaction with input fields, dropdown menus, and related components.

4.5.3 Waiting Strategies

When automating web applications with Selenium, waiting strategies represent essential mechanisms for synchronizing test execution with the application's dynamic state. In complex web applications, particularly digital payment platforms, page elements load asynchronously, undergo state changes, and respond to various AJAX calls. Without proper waiting strategies, automated tests become unreliable and generate false negatives when attempting to interact with elements that have not yet loaded or become interactive.

The fundamental challenge addressed by waiting strategies is the temporal gap between browser automation commands and the actual state of the web application.

---

[12] https://bonigarcia.dev/webdrivermanager/

[13] https://bonigarcia.dev/

[14] https://github.com/bonigarcia/selenium-jupiter

[15] https://www.selenium.dev/documentation/test_practices/encouraged/page_object_models/

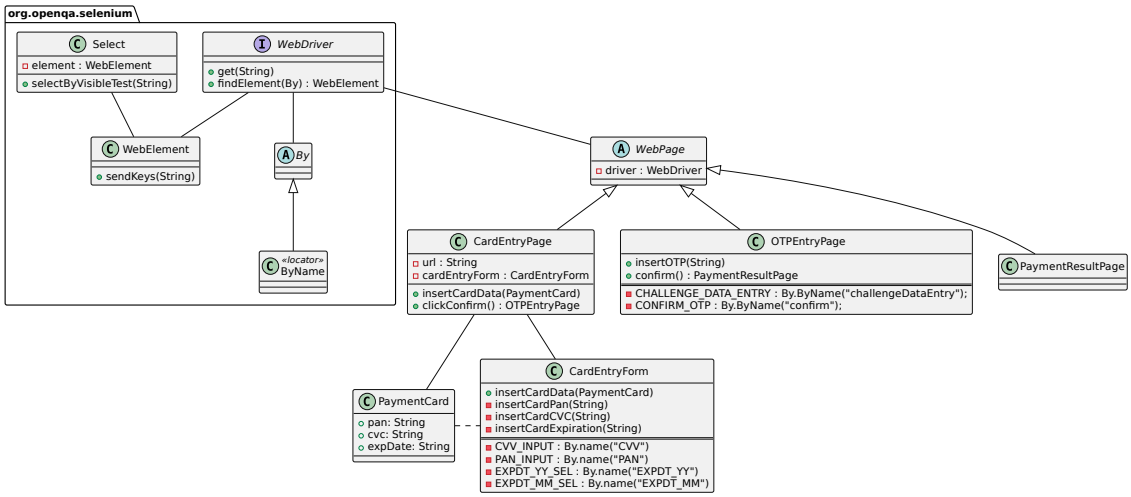

Figure 4.8. Page Object Model for Payment Flow Testing

For example, when testing a payment form, attempting to activate a "Confirm" button immediately after page loading might fail for several reasons:

- The button might not yet exist in the Document Object Model (DOM)
- The button might be present but obscured by a loading overlay
- The button might exist in the DOM but remain disabled until form validation completes

Traditional static waiting mechanisms (Thread.sleep()) are inadequate for several reasons:

1. They unnecessarily slow tests when elements become ready earlier than the specified sleep duration
2. They cannot adapt to variable network conditions and browser loading times
3. They remain susceptible to unpredictable failures when elements require longer loading times than the specified wait period
4. They reduce test maintainability by introducing arbitrary wait times throughout the code

Selenium's dynamic waiting strategies address these limitations by providing intelligent synchronization mechanisms:

Implicit Waits

This mechanism instructs the driver to wait for a specified duration before raising exceptions. This represents a global setting applied to the entire test session. Importantly, when the target element becomes available, the driver immediately returns the element reference and continues execution. Consequently, a longer implicit wait value does not necessarily increase the session duration (unlike the Thread.sleep() approach).

Explicit Waits

Explicit waits enable the configuration of custom conditions to await, such as the visibility of an element identified by a specific locator. These are implemented as explicit polling loops that continuously verify condition fulfillment. If the condition remains unsatisfied within the specified timeout period, an error is generated.

Fluent Waits

Fluent waits represent an evolution of explicit waits, providing more granular control over the polling mechanism. They enable specification of polling intervals and exceptions to ignore (e.g., ElementNotInteractableException.class). Unlike explicit waits, fluent waits accept a callback executed during each polling iteration, containing the element interaction code. If the callback executes without errors and returns true, the wait terminates and test execution continues. If the callback generates an error configured to be ignored, polling continues until either the timeout expires, a non-ignored exception occurs, or the operation succeeds.

4.6 Groovy

The preceding sections have described the utilization of JUnit and Selenium to define and execute end-to-end tests requiring user agent interaction and report generation. This section demonstrates the integration of Groovy with Java to parse script files defining a Domain-Specific Language (DSL) for dynamic test configuration.

Describing Groovy [1] as a library would be reductive, as it represents a comprehensive programming language powering modern technologies [20] including the Gradle project management tool, Jenkins CI/CD server DSL, and Grails web framework, with robust scripting capabilities. The official documentation [7] describes Groovy as follows:

> Apache Groovy is a powerful, optionally typed and dynamic language,

Figure 4.9. Groovy Logo

> with static-typing and static compilation capabilities, for the Java platform aimed at improving developer productivity thanks to a concise, familiar and easy to learn syntax. It integrates smoothly with any Java program, and immediately delivers to your application powerful features, including scripting capabilities, Domain-Specific Language authoring, runtime and compile-time meta-programming and functional programming.

This section presents Groovy integration into a Java application for Domain-Specific Language development. The discussion begins with the fundamental Groovy concept of Closures, then examines the Builder pattern in Groovy, with emphasis on DSL implementation using custom script classes and closure delegation strategies.

4.6.1 Closures

Functional programming languages trace their origins to the 1940s and the development of lambda calculus by Alonzo Church[9]. In the functional programming paradigm, computation is expressed through functions applied to arguments. Functions are considered first-class citizens, serving as the primary computational unit and capable of being passed and returned as parameters.

Functional programming support has become commonplace in modern programming languages, including Python, JavaScript, and, since version 8, Java with its lambda expression support.

From an implementation perspective, functions are created through function literals—expressions representing function values that can be referenced. An anonymous function (or lambda expression) represents a function defined through a function literal, creating a reference-able function value.

When an anonymous function references variables defined in its lexical scope, it is called a closure. A closure "closes over" the environment in which is defined,

maintaining access to lexical scope variables even after the outer function returns. This behavior allow the closure to capture such variables and reuse them at a later time. This mechanism enables previously difficult programming patterns and is particularly adopted for event handling and callback mechanisms.

Groovy extends this concept with a more sophisticated closure implementation. Groovy closures go beyond anonymous functions to become full-fledged objects of the Closure class, offering enhanced capabilities compared to similar constructs in other languages. While Java lambdas and Python lambda functions remain relatively constrained, Groovy closures provide extensive features particularly suited for Domain-Specific Language creation. One of these being the *delegation strategy*.

4.6.2 Delegation Strategies

The effectiveness of Groovy closures in DSL creation derives from their delegation mechanism. Unlike standard anonymous functions, Groovy closures can modify their resolution strategy and delegate object, enabling execution in different contexts. This feature is fundamental for creating expressive DSLs that closely resemble natural language while maintaining programmatic structure and safety.

Delegation in this context refers to the method call resolution process within closures. When a method is invoked inside a closure, it must be resolved and executed. By default, closures search for the method within the owner class. By specifying a custom delegate and modifying the delegation strategy, developers can override the default behavior and configure a custom object for method resolution and invocation.

The Closure class supports the following primary delegation strategies:

OWNER_FIRST With this resolve strategy, the closure attempts to resolve property references and methods first to the owner, then to the delegate (this represents the default strategy).

OWNER_ONLY With this resolve strategy, the closure resolves property references and methods exclusively to the owner without invoking the delegate.

DELEGATE_FIRST With this resolve strategy, the closure attempts to resolve property references and methods first to the delegate, then to the owner.

DELEGATE_ONLY With this resolveStrategy, the closure resolves property references and methods exclusively to the delegate, bypassing the owner entirely.

The delegation strategy proves invaluable for DSL implementation as it allows for fine-grained customization of the method resolution while permitting user-defined side-effects, such as appending created elements to a list. Next sections will provide additional implementation details.

Listing 4.10 demonstrates closure utilization for class instance construction:

```groovy
class PaymentCard {
    public String pan
    public String cvc
    public String expDate

    void pan(String panNumber) {
        this.pan = panNumber
    }

    void cvc(String securityCode) {
        this.cvc = securityCode
    }

    void expDate(String expirationDate) {
        this.expDate = expirationDate
    }
}

// Define a closure that will call the methods on whatever object is set as its delegate
def closure = {
    pan '4111111111111111'
    cvc '123'
    expDate '12/25'
}

def newCard = new PaymentCard()

// Set the delegate of the closure to the new PaymentCard instance
closure.delegate = newCard

// Execute the closure - it will call methods on the delegate (the PaymentCard)
closure()
assert newCard.pan == '4111111111111111'
assert newCard.cvc == '123'
assert newCard.expDate == '12/25'
```

Listing 4.10. Closure Delegation Example

4.6.3 The Builder Pattern

While the builder pattern can be implemented in various programming languages, its combination with Groovy Closures significantly enhances expressiveness and

readability.

Listing 4.11 provides an illustrative example:

```
def writer = new FileWriter('markup.html')
def html   = new groovy.xml.MarkupBuilder(writer)
html.html { // start the html tag
    head { // start head tag
        title 'Constructed by MarkupBuilder' // add title tag
            with content
    }      // close head tag
    body { // start body tag
        'Groovy XML builder example'
        form (action:'example') {
            for (line in ['Produce HTML','Produce XML','Have
               some fun']){
                input(type:'checkbox',checked:'checked', id:
                   line, '')
                label(for:line, line)
                br()
            }
        }
    } // body tag closed
} // html tag closed
```

Listing 4.11. XML Builder in Groovy

This example demonstrates declarative XML construction without explicit `addChild` method invocations for each node. Instead, through the closure mechanism, nested elements can be defined declaratively by invoking the corresponding method (e.g., form()) and passing a closure defining nested elements.

Groovy provides built-in builder support for common data structures such as XML, JSON, directory trees, and other formats. Beyond these pre-defined builders, Groovy offers dedicated classes for creating new builders using some conventional patterns[16]. While these classes offer utility in numerous scenarios, they were not adopted for this work as complexity increased during their evaluation. Instead, this work relies only on closure delegation and custom script classes, examined in the following section.

4.6.4 Integrating Groovy into a Java Application

Groovy functions as a complete independent language executable as a standalone application or invokable from a JVM process. The latter approach enables integration with Java applications, providing access to shared variables and methods.

[16] https://docs.groovy-lang.org/docs/latest/html/gapi/groovy/util/FactoryBuilderSupport.html

The `GroovyShell` class manages Groovy script parsing, compilation, and execution. This class provides both fundamental methods for parsing String or File objects as Groovy scripts and advanced customization options for the parsing process. The `CompilerConfiguration` can define the `scriptBaseClass`, adding behaviors to the executed script by defining methods callable from the script but implemented in the base class. Listing 4.6.4 demonstrates this approach in Groovy (chosen for clarity, though identical functionality is available through Java). This example illustrates how a script can be parsed within a context that implicitly enhances its functionality.

```
import org.codehaus.groovy.control.CompilerConfiguration

class PaymentTest {
    String name;
    public PaymentTest(String name) {
        this.name = name;
    }

    String toString() {
        "test: ${name}"
    }
}

abstract class PaymentTestScript extends Script {
    List<PaymentTest> paymentTests = new ArrayList();

    public void addTest(String name) {
        this.paymentTests.add(new PaymentTest(name));
    }
}

def compilerConfig = new CompilerConfiguration();
compilerConfig.setScriptBaseClass(PaymentTestScript.class.
   getName());

def shell = new GroovyShell(this.class.classLoader, new
   Binding(), compilerConfig);

def parsedScript = shell.parse('addTest("1st Test")');

assert parsedScript instanceof PaymentTestScript
parsedScript.run();
assert parsedScript.paymentTests.size() == 1
println parsedScript.paymentTests // will print: [test: 1st
   Test]
```

Listing 4.12. Parsing Groovy Script with Custom Base Class

This mechanism enables users to define executable test sets. The following section demonstrates how this approach can be enhanced using closures to implement the DSL builder pattern.

4.6.5 Combining Custom Script Class With Closures Delegation

Listing 4.11 demonstrated how closures enable declarative object construction through the delegation mechanism illustrated in Listing 4.10. Listing 4.6.4 illustrated script enhancement with domain-specific functionality. This section demonstrates the integration of these approaches to implement the payment test DSL.

For clarity, this presentation examines a subset of the DSL—specifically, a single test instance instantiation—as additional components introduce only minor variations using identical methodology.

After multiple design iterations, the final DSL implementation appears in Listing 4.13. This design maximizes flexibility regarding target environments, merchant credential configuration, and connection parameters, all configurable independently for each test.

```
1  def jumpHost = "1.2.3.4"
2
3  def validationEnv = testEnv {
4      hostname "hidden"
5      port 8081
6      baseUrlPayment "example/api/payment"
7      proxyJump {
8          via jumpHost
9          username "tester"
10         password "tester"
11 }
12
13 def maestroCard = paymentCard {
14     pan "1234567890"
15     cvc "123"
16     expDate "04/2025"
17 }
18
19 def merchantOne = merchant {
20     keyId "test-keyId"
21     keySecret "test-keySecret"
22 }
23
```

```
24
25 directPayment("direct payment test example") {
26     withMerchant merchantOne
27     withPaymentCard maestroCard
28     toTestEnv validationEnv
29 }
30
31 preAuth("pre-auth test example") {
32     withMerchant merchantOne
33     withPaymentCard maestroCard
34     amount 200
35     toTestEnv validationEnv
36     thenCapture {
37         amount 100 // partial capture
38     }
39 }
40
41 MIT("mit test example") {
42     withMerchant merchantOne
43     withPaymentCard maestroCard
44     amount 200
45     toTestEnv validationEnv
46 }
```

Listing 4.13. Example of the Target DSL

4.6.6 Static Type Checking

The parsing process described thus far provides no mechanism for detecting potential type errors during compilation. Groovy compilation customizers enable type-checking implementation at compile time. Listing 4.14 demonstrates this approach, with reference to the code in Listing 4.6.4.

```
1 import groovy.transform.TypeChecked;
2 import org.codehaus.groovy.control.customizers.
     ASTTransformationCustomizer;
3
4 def compilerConfig = new CompilerConfiguration();
5
6 // add the type checking phase during compilation
7 config.addCompilationCustomizers(new
     ASTTransformationCustomizer(TypeChecked.class));
8
9 compilerConfig.setScriptBaseClass(PaymentTestScript.class.
     getName());
```

Listing 4.14. Forcing Type Checking at Compile Time with a *CompilationCustomizer*

With this enhancement, attempts to invoke the `addTest` method with incorrect parameter types (e.g., an integer instead of a String) will generate a compilation failure with a `MultipleCompilationErrorsException`:

```
org.codehaus.groovy.control.MultipleCompilationErrorsException:
startup failed:
Script1.groovy: 1: [Static type checking]
 - Cannot find matching method Script1#addTest(int).
Please check if the declared type is correct and if the method exists.
 @ line 1, column 1.
   addTest(0)
   ^

1 error
```

This enhancement significantly improves application robustness by enabling early detection of type errors before script execution, allowing users to correct issues proactively.

Section 4.6.5 demonstrated the integration of Closures and custom Script classes for DSL development using closure delegation on Spec classes. However, the type-checking mechanism described above does not adequately address closure body type-checking: while it enforces type checking on the script itself, the compiler cannot anticipate runtime closure delegation targets, as delegation occurs during execution.

Consequently, the compiler attempts to resolve methods within closures against the script class, generating errors even when runtime behavior would execute correctly:

```
1  import org.codehaus.groovy.control.CompilerConfiguration
2  import groovy.transform.TypeChecked
3  import org.codehaus.groovy.control.customizers.
       ASTTransformationCustomizer
4
5  class PaymentTest {
6      String name;
7
8      public PaymentTest() { name = null }
9
10     public void name(String name) {
11         this.name = name
12     }
13
14     String toString() {
15         "test: ${name}"
16     }
```

```
17 }
18
19 abstract class PaymentTestScript extends Script {
20     List<PaymentTest> paymentTests = new ArrayList();
21
22     public void createTest(Closure cl) {
23         def delegate = new PaymentTest()
24         cl.setDelegate(delegate)
25         cl.setResolveStrategy(Closure.DELEGATE_ONLY);
26         cl()
27         this.paymentTests.add(delegate)
28     }
29 }
30
31 def compilerConfig = new CompilerConfiguration();
32 // if the following line is uncommented it will raise an
   compile-time exception
33 // compilerConfig.addCompilationCustomizers(new
   ASTTransformationCustomizer(TypeChecked.class));
34 compilerConfig.setScriptBaseClass(PaymentTestScript.class.
   getName());
35
36 def shell = new GroovyShell(this.class.classLoader, new
   Binding(), compilerConfig);
37
38 def parsedScript = shell.parse("""
39     createTest {
40         name "a test created with closure delegation"
41     }
42     """);
43
44 assert parsedScript instanceof PaymentTestScript
45 parsedScript.run();
46 assert parsedScript.paymentTests.size() == 1
47 assert parsedScript.paymentTests[0].name == "a test created
   with closure delegation"
48 println parsedScript.paymentTests // will print: [test: a
   test created with closure delegation]
```

Listing 4.15. A Compile-Time Failing PaymentTestScript Due to Runtime Closure Delegation

To address this limitation, the Groovy compiler must be informed about method resolution against alternative classes. Groovy 2.1 introduced the `@DelegatesTo` annotation to specify delegate classes for Closure parameters and optionally define delegation strategies. According to Groovy documentation:

`@groovy.lang.DelegatesTo` is a documentation and compile-time annotation aimed at:

1. documenting APIs that use closures as arguments
2. providing type information for the static type checker and compiler

Modifying the createTest method signature as follows enables compatibility with static type checking:

```
public void createTest(
    @DelegatesTo(
     value=PaymentTest,
     strategy=Closure.DELEGATE_ONLY
    ) Closure cl)
```

4.7 DSL Implementation

The techniques described previously form the foundation of the DSL implementation detailed in this section. First, the target DSL is examined, including formal requirements and design guidelines. Subsequently, the section presents an architectural overview of the application, detailing its components and their respective responsibilities. Additional domain-specific mechanisms are also described.

4.7.1 DSL Requirements and Design

The application's core functionality enables users to define and execute test sets, generating execution reports for result analysis.

To ensure DSL completeness and utility, the following considerations were addressed:

- Available test templates for user implementation
- Parameter specifications for each test type
- User parameter definition mechanisms
- Environment configuration (validation, quality, etc.) with determination of whether configuration should be global or test-specific

These questions require opinionated resolution, as no universally optimal solution exists. The selected approach involved iterative DSL refinement until achieving the desired flexibility and usability.

Table 4.1 enumerates the tests incorporated into the payment test DSL.

Several tests share common parameters (payment card, amount, target environment), enabling reusability pattern implementation for test configurations.

Test method	Description	Parameters
directPaym	A generic transaction with configurable transaction type (verify, sale)	- paymentCard - amount - transactionType (verify, sale) - targetEnv
preAuth	A test involving amount pre-authorization. The user can also specify what should follow the authorization (capture, reversal)	- paymentCard - amount - then (capture or reversal) - targetEnv
MIT	A Merchant Initiated Transaction.	- paymentCard - amount - targetEnv

Table 4.1. Test Types Implemented in the DSL

```
def jumpHost = "1.2.3.4"

def validationEnv = testEnv {
    hostname "hidden"
    port 8081
    baseUrlPayment "example/api/payment"
    proxyJump {
        via jumpHost
        username "tester"
        password "tester"
}

def maestroCard = paymentCard {
    pan "1234567890"
    cvc "123"
    expDate "04/2025"
}

def merchantOne = merchant {
    keyId "test-keyId"
    keySecret "test-keySecret"
}


directPayment("direct payment test example") {
    withMerchant merchantOne
    withPaymentCard maestroCard
    toTestEnv validationEnv
```

```
29 }
30
31 preAuth("pre-auth test example") {
32     withMerchant merchantOne
33     withPaymentCard maestroCard
34     amount 200
35     toTestEnv validationEnv
36     thenCapture {
37         amount 100 // partial capture
38     }
39 }
40
41 MIT("mit test example") {
42     withMerchant merchantOne
43     withPaymentCard maestroCard
44     amount 200
45     toTestEnv validationEnv
46 }
```

Listing 4.16. Example of Final Test Configuration Script

4.7.2 Implementation Overview

Figure 4.10 presents the implementation architecture, depicting principal components contributing to the final system.

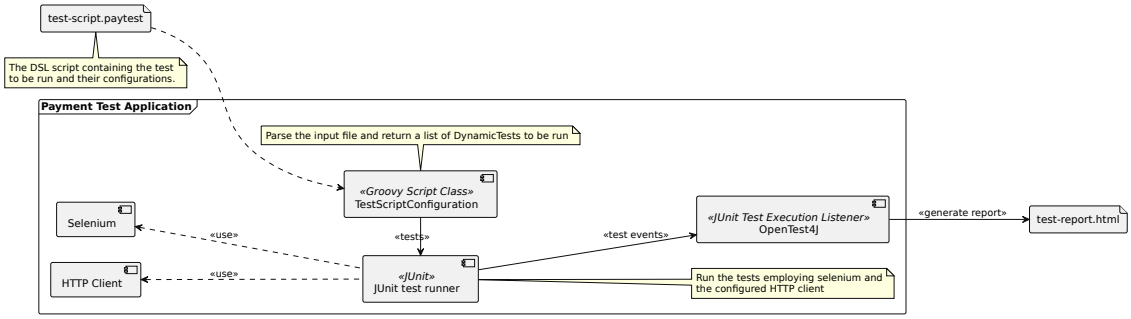

Figure 4.10. Component Diagram of the DSL Implementation

4.7.3 Defining Test Classes

Test implementation classes must address multiple requirements. First, they must implement the `Executable` interface for interpretation by JUnit as dynamic tests. This interface represents the most abstract execution interface in Java, consisting of a single method: `void execute() throws Throwable`.

To enhance the Executable interface with additional functionality shared across tests, a new interface, `PaymentTest extends Executable` has been defined. This interface introduces methods required for test configuration, such as `setReporter(TestReporter te` for configuring the reporter instance for each test, and `String getName()` to get the name identifier of the test.

Furthermore, this interface provides a default implementation of the execute method. For code reuse maximization, tests are designed as sequences (Lists) of common `TestStep` objects, where test execution entails sequential step execution. Listing 4.17 presents the PaymentTest interface, demonstrating its default execute method implementation, which reports test configuration via the testReporter and executes each step sequentially.

```
1  package it.payreply.paytests.dsl.testscenarios;
2
3  import it.payreply.paytests.dsl.domain.PaymentCard;
4  import it.payreply.paytests.dsl.testscenarios.steps.TestStep;
5  import org.junit.jupiter.api.TestReporter;
6  import org.junit.jupiter.api.function.Executable;
7
8  import java.util.List;
9
10 public interface PaymentTest extends Executable {
11
12     void reportTestConfiguration();
13
14     List<TestStep> getTestSteps();
15
16     String getName();
17
18     void setReporter(TestReporter testReporter);
19
20     default void execute() throws Exception {
21         reportTestConfiguration();
22         for (TestStep step: getTestSteps()) {
23             step.execute(getTestContext());
24         }
25     }
26
27     TestContext getTestContext();
28
29     default PaymentCard getPaymentCard() {
30        return getTestContext().getPaymentCard();
31     }
32 }
```

Listing 4.17. The PaymentTest Interface

The TestStep interface defines another execution method that accepts a TestContext parameter: `void execute(TestContext context) throws Throwable`. Step classes are stateless, utilizing the TestContext to access required data and store execution results. The TestContext maintains a TestReporter reference, enabling any step to report data, screenshots, and properties to the JUnit reporting infrastructure.

This architecture enables the `CompleteTransactionInBrowserStep` to report browser page screenshots to the JUnit reporting infrastructure. Each step employs assertions to verify expected conditions, such as confirming API calls return status 200 with expected response data. Assertion failures indicate test failures.

The implemented TestSteps include:

StartTransactionStep Initializes transactions and retrieves browser callback URLs

CompleteTransactionInBrowserStep Utilizes previously obtained URLs to navigate web pages, input card data, and proceed to the success page

VerifyTransactionStep After browser interaction completion, verifies transaction status via additional API calls

PerformMITStep Specific to MIT test scenarios, executes different API calls to complete transactions synchronously using payment tokens obtained in previous steps

4.7.4 Configuring Test Parameters

Having described test implementation through specialized classes, this section examines parameter configuration via the custom Groovy script class.

When the application executes, test classes are instantiated during Groovy script interpretation and execution, which happens within JUnit tests, as test classes must be returned as dynamic tests implementing the Executable interface. Figure 4.11 illustrates this process.

For the script to return a PaymentTest lists, it must track down every test creation internally. This is done by adding tests to the list as they are created, as demonstrated in Listing 4.6.5. The script class provides public methods for creating PaymentTest instances from Closures (and Strings for test naming). These Closures are delegated to test type specifiers that configure parameters and instantiate tests. Figure 4.12 illustrates this process.

4.7.5 Detailed Class Diagram

Figure 4.13 presents a more detailed class-level architectural diagram.

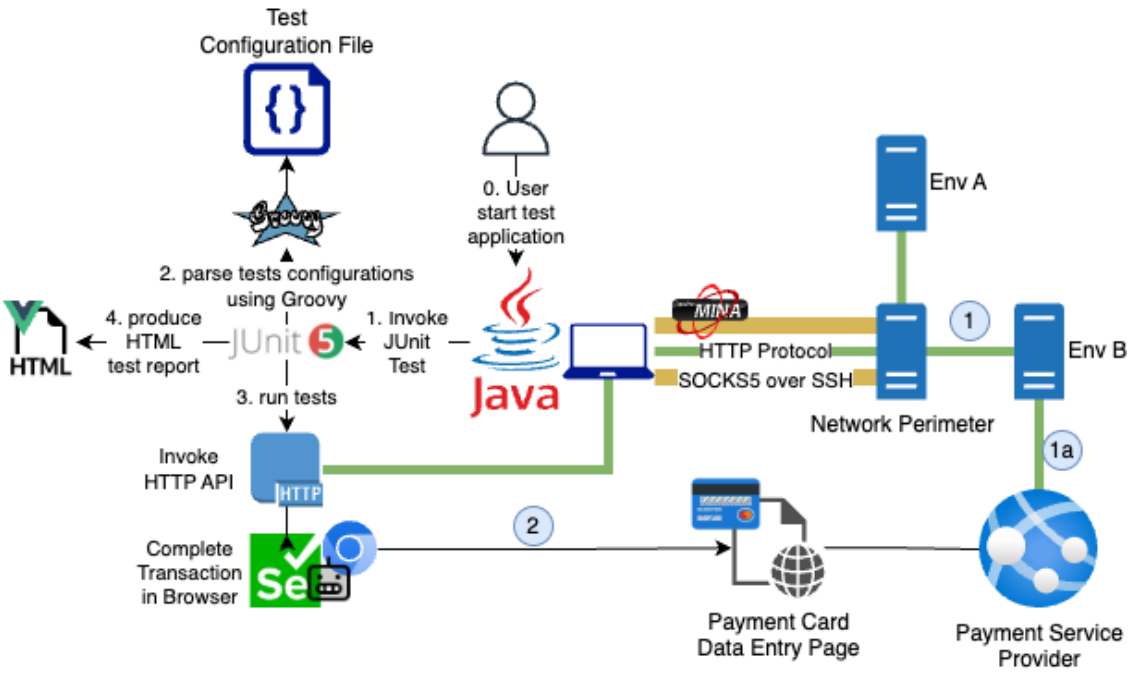

Figure 4.11. Application Lifecycle

4.8 Parallel Test Execution

The main advantage of a mature testing frameworks such as JUnit is its incredible features support, that most of the time only requires some additional configuration. Parallel test execution is one of these features: it can be easily configured to gain a non-negligible speed-up in test execution time. For test suites with many cases, parallel execution capability significantly enhances performance.

Without parallel execution, automation alone provides limited speed up in execution time: most of the gain relies in parallelization of the test executions, as this will allow to run more test cases at the same time. Compared to human operators which cannot reliably be multi-task without operational errors, computers are designed to do multiple things at the same time.

The implementation has thus taken into account parallel test execution while addressing domain-specific constraints. While JUnit enables parallel execution through simple environment variable configuration, a synchronization mechanism is required to prevent concurrent transactions using identical payment cards. This in fact could result in unspecified behaviors on testing environments.

Figure 4.12. Delegation Process in PaymentTest Instantiation

4.8.1 PaymentCard Synchronization

In order to prevent side effects from concurrent transactions using the same payment card, browser payment steps are synchronized relative to the payment card, ensuring that only one transaction per card executes simultaneously.

This constraint applies exclusively to browser payment steps (where transactions occur) and not to other steps, as broader synchronization would eliminate parallelization benefits. The implementation utilizes Java synchronization primitives, specifically synchronized blocks targeting payment card objects.

For effective implementation, each unique card configuration (pan, expiration date, CVV) must reference the same PaymentCard object, as synchronization occurs at the object level. Since the DSL enables PaymentCard object creation from multiple contexts (root context paymentCard method or closure delegation within withPaymentCard methods), a consistency mechanism is required.

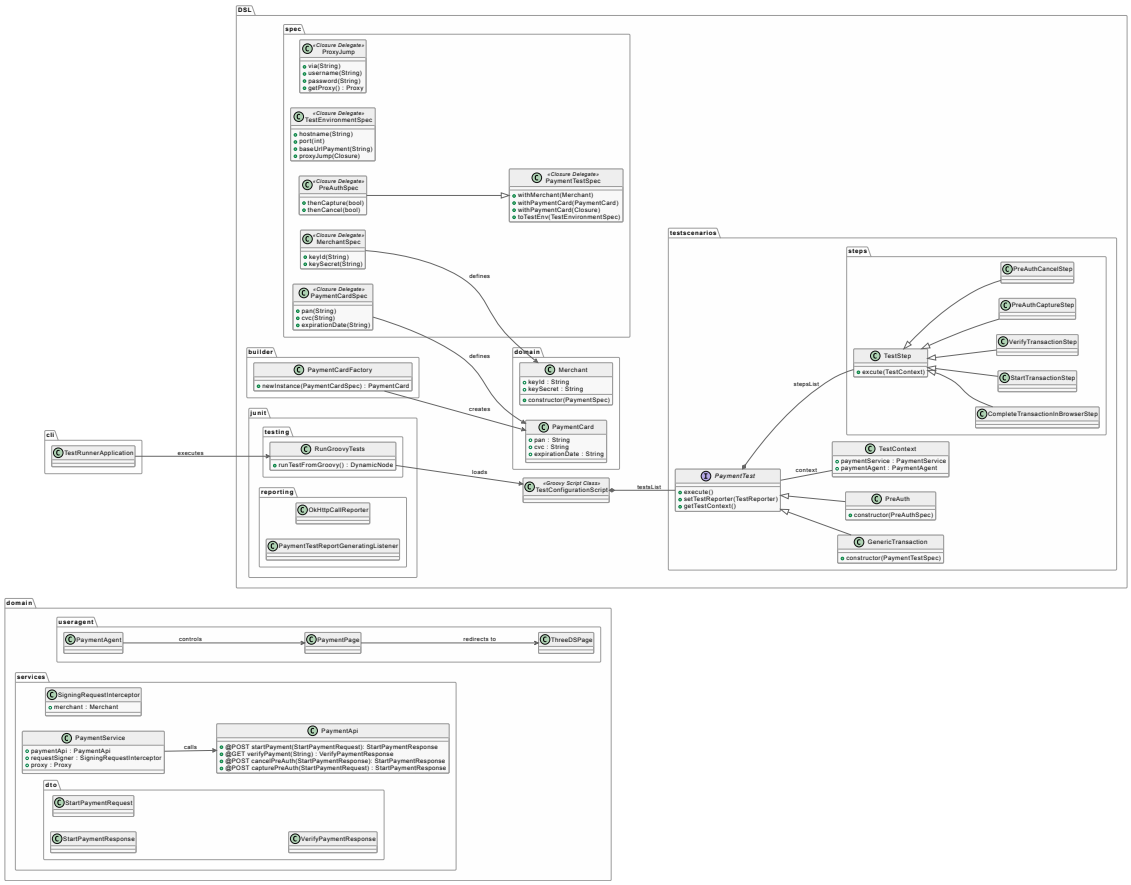

Figure 4.13. Class Diagram of the DSL Implementation

The implemented solution delegates PaymentCard creation to a specialized PaymentCardFactory that returns existing instances when requested to create PaymentCards with previously registered data. A `Map<PaymentCardSpec, PaymentCard>` associates PaymentCard instances with specifications containing card data. When creating a PaymentCard from a PaymentCardSpec, the system checks for existing objects, returning them when found or creating new instances when necessary. Listing 4.18 formally describes this behavior.

```
var card1 = paymentCard {
    pan "123456"
    cvc "123"
    expirationDate "04/2025"
}

def card2 = paymentCard {
    pan "123456"
    cvc "123"
```

```
10      expirationDate "04/2025"
11  }
12
13  def card3 = paymentCard {
14      pan "09876"
15      cvc "123"
16      expidationDate "05/2025"
17  }
18
19  assert card1 == card2
20  assert card3 != card1
```

Listing 4.18. Same Card Instantiation Behavior

4.8.2 Enabling Parallel Test Execution in JUnit

JUnit parallel test execution configuration requires three actions:

1. Add the `junit.jupiter.execution.parallel.enabled` configuration parameter with value `true` to the LauncherDiscoveryRequest created when launching the test class

2. Annotate the test method with `@Execution(ExecutionMode.CONCURRENT)`

3. For dynamic tests created with @TestFactory, wrap returned test lists in a `dynamicContainer` and modify the return type to `DynamicNode`

Listing 4.19 demonstrates this implementation.

```
1  // when creating the LaucherDiscoveryRequest:
2  var junitTestRequest = LauncherDiscoveryRequestBuilder.
       request().selectors(
3                          DiscoverySelectors.selectClass(
                               RunGroovyTests.class.getName())
4                  )
5                  .configurationParameter("junit.platform.
                       output.capture.stdout", "true")
6                  .configurationParameter("junit.platform.
                       output.capture.stderr", "true")
7                  .configurationParameter("junit.platform.
                       reporting.output.dir", OUTPUT_DIR)
8                  // enable parallel execution
9                  .configurationParameter(
10                         "junit.jupiter.execution.parallel.
                               enabled", "true"
11                 )
```

```
12                  .configurationParameter("junit.platform.
                       reporting.open.xml.enabled", "true")
13                  .build();
14
15
16 // on the test methods:
17     @TestFactory
18     @ExtendWith(DynamicTestReportingExtension.class)
19     @Execution(ExecutionMode.CONCURRENT) // new
20     public DynamicNode runTestFromGroovy(DynamicTestReporter
          testReporter) throws IOException {
21         // ... just as before
22         // wrap tests inside DynamicNode usign
              dynamicContainer
23         return dynamicContainer(SOURCE_SCRIPT, testList.
              stream().map(it -> {
24             it.setReporter(testReporter);
25             return dynamicTest(it.getName(), it);
26         }));
27     }
```

Listing 4.19. Enabling JUnit Parallel Execution

4.9 IntelliJ IDEA DSL Specification via GDSL

A remaining challenge involves assisting users in test script creation. While Section 4.6.6 demonstrated compile-time type checking for error prevention, IDE assistance for method availability, parameter types, and related guidance remains necessary. This capability resembles Jenkins steps definition support.

Jenkins steps derive from DSL files and would be challenging for DevOps developers to implement correctly without assistance. IntelliJ IDEA IDE provides mechanisms for DSL definition and application to Groovy files. This methodology is widely adopted in the Jenkins developer community and is detailed below.

4.9.1 GDSL Files

Without specification, Groovy files utilizing DSLs appear in IntelliJ as shown in Figure 4.14.

With DSL specification, the IDE immediately signal any type error or invalid method calls, while also providing code suggestion and method documentation during editing, as illustrated in Figure 4.15. This assistance significantly improve the script editing experience, especially for new users who might not be aware of method names and expected arguments.

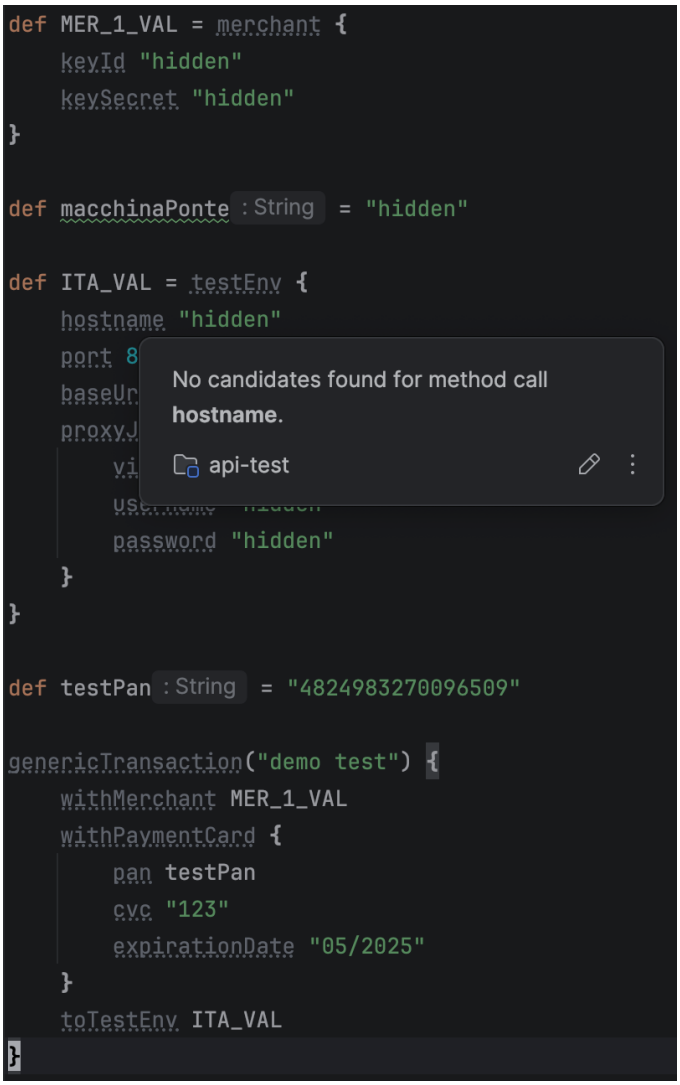

Figure 4.14. DSL Editing in IntelliJ Without Capability Information

Implementing this functionality requires informing IntelliJ about the DSL structure, which is composed of methods, parameters, and closure delegation. This is accomplished through GDSL files. GDSL files provide a way to explicitly describe what a DSL allows to do. For instance it allows to specify that a certain method is available and what kind of parameters it expects. By placing a `.gdsl` file in the `src/main/groovy` directory a new DSL is created and can be used (which is provided assistance) inside the code editor files.

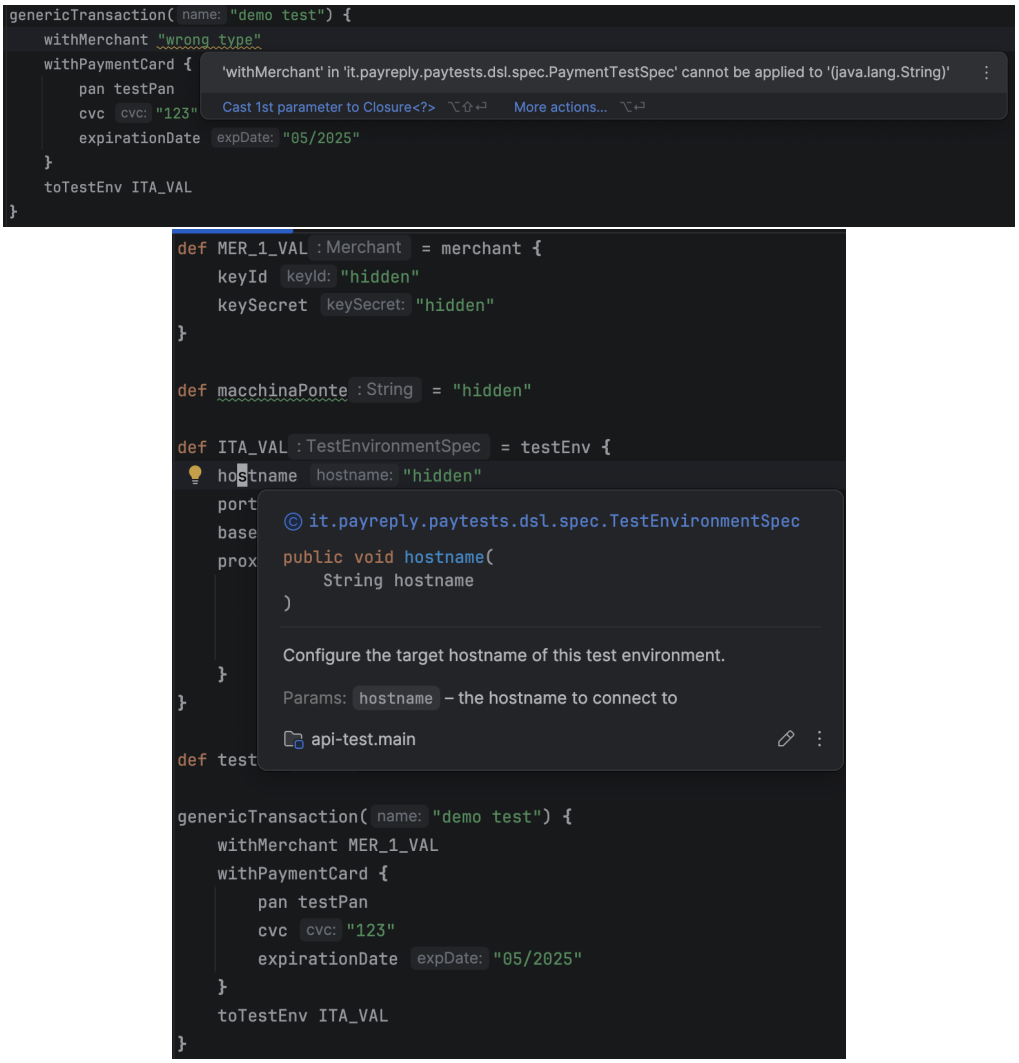

Figure 4.15. DSL Editing with Complete IDE Assistance, Including Method Documentation and Argument Type Resolution

This process is detailed in the IntelliJ documentation[17]. Yet, explicitly defining the DSL in this way has 3 main disadvantages: requires time and effort; is susceptible to error and misalignment with respect to the real DSL; and finally must be updated in accordance with every new modification of the DSL. Fortunately, there was an alternative: the GDSL can be used to infer the DSL knowledge directly from the Groovy custom script class, as shown in Listing 4.20. This has many benefit because ensure that the defined DSL is always based on the real script class.

[17] https://youtrack.jetbrains.com/articles/GROOVY-A-15796912

```
// PayTest.gdsl file for DSL specification
// place it in scr/main/groovy
// apply only to ".paytest" files
def rootCtx = context(scope: scriptScope(null), filetypes: ["paytest"])

contributor(rootCtx) {
    // delegates to our custom script class
    // it must be available on the classpath
    delegatesTo(findClass("it.payreply.paytests.dsl.TestConfigurationScript"))
}
```

Listing 4.20. The PayTest.gdsl File Implementation

Chapter 5

Results and Evaluation

This chapter presents the results of implementing the automated testing framework with a domain-specific language for digital payment systems. The evaluation of the developed solution is addressed both quantitative and qualitative, comparing the efficiency of the automated approach against manual testing procedures and analyzing the expressiveness of the DSL for test definition. Starting from a predefined list of tests, the results point out how the developed framework allows to formalize the test suite in an intuitive manner by employing Groovy language capabilities. The performance gain of the automated execution are evaluated for the whole test suite, and point out not negligible improvements.

5.1 Test Suite Definition

The test suite to be evaluated is extracted from a list provided by the customer, which contains the most important payment scenarios and consists of 82 tests, divided in 41 cases repeated twice for each of the two card type, MasterCard and Visa. The repetition of the same tests for the two card issuer attempts to validate the connection to the two circuits' networks. Of these 41 test scenarios, 26 requires the physical card to be inserted on the smart pos, and so are excluded from the evaluation test suite. The remaining 15 tests have been implemented using the methodologies described in the previous chapter. Since the tests are executed for each of the two payment card of MasterCard and Visa, the total number of tests is 30.

5.2 Qualitative Analysis: DSL Expressiveness and Usability

Beyond quantitative improvements in execution efficiency, the DSL-based approach offers significant qualitative advantages in test definition, maintenance, and extensibility. This section examines these aspects through illustrative examples and usability analysis.

5.2.1 Case Study: Multi-Card Test Definition

As explained above, the defined test suite involves executing identical test scenarios using two different payment cards from two issuers (Mastercard and Visa). The DSL design combined with Groovy language features intuitively support this pattern as allow to express it with minimal code reuse. This is demonstrated in Listing 5.1, where the test suite script employ the Groovy forEach to define the same tests for the different cards:

```
// Common test setup
def testMerchant = merchant {
    keyId "hidden"
    keySecret "hidden"
}

def targetTestEnv = testEnv {
    hostname "hidden"
    port 8081
    baseUrlPayment "/test/api/payment/"
    proxyJump {
        via "bastionHost's IP"
        username "hidden"
        password "hidden"
    }
}

def masterCard = paymentCard {
    pan "5111111111111111"
    cvc "000"
    expirationDate "12/30"
}

def visa = paymentCard {
    pan "4111111111111111"
    cvc "000"
    expirationDate "12/30"
}
```

```
29
30 def cardsToTest = [masterCard, visa];
31
32 cardsToTest.forEach { card ->
33     {
34         // Test Case 1: Direct Payment online
35         genericTransaction("${card.getIssuer()} Direct
               Payment online") {
36             withMerchant testMerchant
37             withPaymentCard card
38             amount 100
39             toTestEnv targetTestEnv
40         }
41
42         // Test Case 2: Direct Payment online with
               tokenization
43         genericTransaction("${card.getIssuer()} Direct
               Payment online with tokenization") {
44             withMerchant testMerchant
45             withPaymentCard card
46             amount 100
47             tokenize true
48             toTestEnv targetTestEnv
49         }
50
51         // Test Case 3: Card_verification & tokenization
               online
52         verifyCard("${card.getIssuer()} Card verification &
               tokenization online") {
53             withMerchant testMerchant
54             withPaymentCard card
55             tokenize true
56             toTestEnv targetTestEnv
57         }
58
59     // Test Case 4: Pre-Auth - online
60         preAuth("${card.getIssuer()} Pre-Auth - online") {
61             withMerchant testMerchant
62             withPaymentCard card
63             amount 100
64             toTestEnv targetTestEnv
65         }
66
67         // Test Case 5: Pre-Auth - online with token
68         preAuth("${card.getIssuer()} Pre-Auth - online with
               token") {
```

```
69              withMerchant testMerchant
70              withPaymentCard card
71              amount 100
72              tokenize true
73              toTestEnv targetTestEnv
74          }
75
76          // Test Case 6: Pre-authorization cancellation
77          preAuth("${card.getIssuer()} Pre-authorization
                cancellation") {
78              withMerchant testMerchant
79              withPaymentCard card
80              amount 100
81              then {
82                  cancel
83              }
84              toTestEnv targetTestEnv
85          }
86
87          // Test Case 7: Pre-authorization (with tokenization)
                cancellation
88          preAuth("${card.getIssuer()} Pre-authorization (with
                tokenization) cancellation") {
89              withMerchant testMerchant
90              withPaymentCard card
91              amount 100
92              tokenize true
93              then {
94                  cancel
95              }
96              toTestEnv targetTestEnv
97          }
98
99          // Test Case 8: Pre-authorization partial closure
100         preAuth("${card.getIssuer()} Pre-authorization
                partial closure") {
101             withMerchant testMerchant
102             withPaymentCard card
103             amount 100
104             then {
105                 capture 50
106             }
107             toTestEnv targetTestEnv
108         }
109
```

```
110         // Test Case 9: Pre-authorization partial closure
                with token
111         preAuth("${card.getIssuer()} Pre-authorization
                partial closure with token") {
112             withMerchant testMerchant
113             withPaymentCard card
114             amount 100
115             tokenize true
116             then {
117                 capture 50
118             }
119             toTestEnv targetTestEnv
120         }
121
122         // Test Case 10: Pre-authorization total closure
123         preAuth("${card.getIssuer()} Pre-authorization total
                closure") {
124             withMerchant testMerchant
125             withPaymentCard card
126             amount 100
127             then {
128                 capture 100
129             }
130             toTestEnv targetTestEnv
131         }
132
133         // Test Case 11: Pre-authorization total closure with
                token
134         preAuth("${card.getIssuer()} Pre-authorization total
                closure with token") {
135             withMerchant testMerchant
136             withPaymentCard card
137             amount 100
138             tokenize true
139             then {
140                 capture 100
141             }
142             toTestEnv targetTestEnv
143         }
144
145         // Test Case 12: Transaction refund intraday (
                Original transaction CNP)
146         genericTransaction("${card.getIssuer()} Transaction
                refund intraday (Original transaction CNP)") {
147             withMerchant testMerchant
148             withPaymentCard card
```

```
149             amount 100
150             then {
151                 refund 100
152             }
153             toTestEnv targetTestEnv
154         }
155
156         // Test Case 13: Transaction refund partial intraday
               (Original transaction CNP)
157         genericTransaction("${card.getIssuer()} Transaction
               refund partial intraday (Original transaction CNP)
               ") {
158             withMerchant testMerchant
159             withPaymentCard card
160             amount 100
161             then {
162                 refund 50
163             }
164             toTestEnv targetTestEnv
165         }
166
167         // Test Case 14: Transaction refund intraday (
               Original transaction Pre-Auth)
168         preAuth("${card.getIssuer()} Transaction refund
               intraday (Original transaction Pre-Auth)") {
169             withMerchant testMerchant
170             withPaymentCard card
171             amount 100
172             then {
173                 capture 100
174                 refund 100
175             }
176             toTestEnv targetTestEnv
177         }
178
179         // Test Case 15: Transaction refund partial intraday
               (Original transaction Pre-Auth)
180         preAuth("${card.getIssuer()} Transaction refund
               partial intraday (Original transaction Pre-Auth)")
               {
181             withMerchant testMerchant
182             withPaymentCard card
183             amount 100
184             then {
185                 capture 100
186                 refund 50
```

```
            }
            toTestEnv targetTestEnv
        }

    }
};
```

Listing 5.1. The test script containing the 30 tests to be executed. Note how the Groovy language allows to reuse code to configure the same tests for different cards.

The collection-based approach significantly reduces redundancy compared to defining each test case individually, and is only possible thanks to Groovy. Also, the test name is used to identify the value received by the test as it is parameterized with its value.

5.2.2 Environmental Configuration Flexibility

The DSL's environment configuration capabilities also allow tests to target different deployment environments without modifying test logic. Listing 5.2 illustrates this flexibility:

```
// Define environments with different configurations
def validation = testEnv {
    // ...
}

def quality = testEnv {
    // ...
}

// Define test that runs in both environments
[validation, quality].each { env ->
    genericTransaction("Cross-environment payment in ${env}")
        {
        withPaymentCard visaCard
        amount 1000
        tokenize true
        toTestEnv env
    }
}
```

Listing 5.2. Example of tests run across different environments

This is the same approach mentioned above but applied across the target test environment parameters. Allowing to target different deployments of the application can be useful to trace back introduced regression present in one environment but not the other.

5.2.3 Test Reporting Capabilities

The integration with JUnit 5's reporting infrastructure provides comprehensive test execution documentation. Figure 5.1 illustrates the generated HTML report for a test execution:

Figure 5.1. Example of generated HTML report visualized in a browser. The report contains all the test configurations parameters, captured standard output and standard error and screenshot of the web pages captured by the Selenium user agent.

The report includes:

- Test execution status (pass/fail) with clear visual indicators.
- Execution duration for each test case and the overall suite.
- Test parameters provided, including test environment properties, payment card details, and so on.
- Detailed failure information including exception stack traces and standard error.
- Screenshots captured during browser-based testing steps.
- Transaction data and API response details logged by the OkHttpClient.

This comprehensive reporting facilitates efficient debugging and provides detailed documentation of test coverage for compliance purposes.

5.3 Quantitative Analysis: Execution Time

One of the primary objectives of the automated testing framework was to reduce the time and effort required to execute comprehensive test suites for payment scenarios. This section presents a comparative analysis of execution times between manual testing procedures and the automated DSL-based approach.

5.3.1 Execution Time Comparison

To evaluate execution efficiency, the application has been run against the test suite defined in Listing 5.1. In order to compare it with manual execution, an estimation of the manual run was obtained by considering a constant time for a human times the number of tests. The constant is obtained empirically by measuring with a digital clock a single test manual execution time. Table 5.3.1 presents the execution

Execution Mode	**Number of Tests**	**Execution Time (s)**	**Average Time per Test (s)**
Manual (Estimated)	30	9,000	300
Automated (Measured)	30	600	40
Time Savings	–	**8,400**	**260**
Improvement (%)	–	**93.3%**	**86.7%**

Table 5.1. Test Suite Execution Time Comparison Note: Estimated manual execution time includes also manual filling of the test report. Automated execution can run 2 tests in parallel (one test for each card at a time), resulting in a total execution time of 600 seconds (15 test pairs × 40 seconds). Time savings calculated as the difference between manual and automated execution times. Improvement percentage calculated as: $\frac{ManualTime - AutomatedTime}{ManualTime} \times 100\%$.

time comparison between manual testing procedures and the DSL-based automated framework.

The results, as expected, demonstrate an evident improvement in execution time, with the automated framework reducing execution time by approximately **93.3%** for the whole test suites. This is partially due to the increased level of parallelism (two different card so two tests at a time), the faster speed at which the program can work, and that manual testing required also manual filling of the test report.

5.4 Challenges and Limitations

While the DSL-based automated testing framework delivers significant improvements, several challenges and limitations were identified during implementation and evaluation:

5.4.1 Test Stability

Since the test environment were reliant on external services provided by external parties, their availability was not under control and caused some non-reproducible instabilities during tests, making them fail. In order to prevent these kind of issues some retry strategies can be introduced in order to detect anomalies during tests and trigger a new run for involved tests.

5.4.2 Application Interfaces

While the implemented solutions currently utilizes a command-line interface, future work may focus on making it accessible from more intuitive interfaces, such as a dedicated web app with embedded test-script editor enriched with code-assistance functionalities. This extension would lower the entry barrier for non technical stakeholders and provide a centralized access to the testing infrastructure.

5.5 Future Works

5.5.1 Reporting Infrastructure

While the report generation fulfills the main requirements and expectations, there is still room for improvement by integrating more advanced infrastructure offering greater capabilities. Allure2 [3] would be a perfect candidate for this exploratory work as offers many useful features out-of-the-box, such as test history and success monitoring over time, including of course HTML test report output.

5.5.2 Solution Generality

The implemented solution has proven well for the target SUT. A possible extension of this work could focus on increasing its generality to support different target application in the domain of digital payments. This would require to define ad-hoc payment scenarios and API definition and calls. While this is indeed challenging, a feasibility study is required to analyze the costs of such a solution.

5.5.3 LLM Integration

The applications of Large Language Models (LLM) are ubiquitous and they have provided useful functionalities across several domains. Since the main point of LLM is language manipulation, they could be a good fit to generate or manipulate new DSL defintions as the one developed in this thesis. Future works could focus on their integration for DSL specification of test script generation from natural language interfaces, or event automatic information extraction from test reports.

5.6 Summary of Findings

The evaluation of the DSL-based automated testing framework has proven not negligible improvements both across qualitative and quantitative dimensions:

- **Execution Efficiency:** The automated framework reduced test execution time by 93.3% for the multi-card test suites through parallel execution and optimized test sequences.
- **Test Maintainability:** The DSL's expressive syntax and modular design enable concise test definition with minimal redundancy, facilitating maintenance and extension of test suites.
- **Coverage Scalability:** The framework's parallel execution allows to scale the test coverage across many parameters while maintaining low execution time. In other words: more configurations can be tested in the same time.
- **Reporting Quality:** Integration with JUnit 5's reporting infrastructure provides easily accessible and comprehensive test documentation with detailed diagnostic information.

These findings validated the effectiveness of the domain-specific testing approach for case study and demonstrated the value of investing in specialized testing solutions. Future works could focus on exploring new DSL design, implements new scenarios, integrate additional reporting framework or even new technologies such as LLM, which are adapt to manipulate the kind of language the DSL implements, which is simpler than a programming language but still can implements valuable semantics.

Chapter 6

Conclusion

This research has addressed the challenge of automating the testing process for digital payment systems through the development of a domain-specific language and testing framework. The work was motivated by the inherent complexity of payment system testing, which typically involves labor-intensive procedures spanning multiple interfaces, systems, and transaction types. By implementing a specialized testing approach, the developed solution successfully achieved its main goals of enhancing testing efficiency and supporting dynamically provided test parameters.

As described in detail in Chapter 3, the testing procedures observed for this domain require manual interaction with payment APIs, data insertion in browser, while also scaling poorly due to the dynamic range of possible configurations. The solution developed in this thesis effectively addresses both challenges, demonstrating that a DSL-based approach can significantly improve the testing process for complex financial systems with the same constraint as the one evaluated.

The tech stack ended up being a good fit for this project. JUnit gave us the mature testing framework we needed, especially since allowed for run-time test defitions. Selenium paired with WebDriverManager made browser automation manageable without the usual problems. The reports produced by open-test-reporting are a step forward standard Excel sheets, as are automatically created and can embed screenshots and many other information. The real advantage, though, was Groovy DSL implementation. Not only it delivered on the promise of concise, configurable test suites, but it also avoided the need of implementing different parsing solutions that would have required a higher degree of complexity without providing scripting capabilities.

The findings in Chapter 5 confirm the effectiveness of this approach on multiple fronts. Tests demonstrated measurable improvements in execution speed and required less manual monitoring. Equally important, the test scripts provided notably improved readability and maintainability. These results suggest similar domain-specific testing approaches could be valuable in other technical domains characterized by high complexity.

The DSL implementation demonstrates how Groovy's meta-programming capabilities can be leveraged to create intuitive, readable test definitions that closely align with the domain terminology of payment systems. By enabling non-technical stakeholders to understand and potentially contribute to test definitions, the DSL bridges the gap between technical implementation and business requirements.

While the current implementation successfully meets the project objectives, several paths for further development remain. Future work should focus on integration with enhanced test reporting infrastructure such as Allure2, incorporating LLM capabilities for DSL script definition, developing more accessible interfaces such as a dedicated web application, and increasing the generality of the solution to accommodate different applications within the digital payment domain.

Bibliography

[1] Laforge et al. *A flexible and extensible Java-like language for the JVM*. URL: https://groovy-lang.org/.

[2] *B2B Payments to Reach* $124 Trillion Globally by 2028$, as Instant Payment Rails Revolutionise Cr *border Payments*. URL: https://www.juniperresearch.com/press/b2b-payments-to-reach-124-trillion-globally-by-2028-as-instant-payment-rails-revolutionise-cross-border-payments/ (visited on 01/04/2025).

[3] Dmitry Baev. *Flexible, lightweight multi-language test reporting tool*. URL: https://allurereport.org/.

[4] Stripe Blog. *What platforms and marketplaces can expect from PSD3*. 2024. URL: https://stripe.com/gb/guides/what-platforms-and-marketplaces-can-expect-from-psd3 (visited on 01/04/2025).

[5] B.W. Boehm. «Verifying and Validating Software Requirements and Design Specifications». In: *IEEE Software* 1.1 (1984), pp. 75–88. DOI: 10.1109/MS.1984.233702.

[6] *Chromium lines of code number*. URL: https://openhub.net/p/chrome/analyses/latest/languages_summary.

[7] Groovy Community. *Groovy Official Documentation*. URL: https://groovy-lang.org/documentation.html.

[8] James Duncan Davidson. 2000. URL: https://ant.apache.org/ (visited on 03/23/2025).

[9] Harry Deutsch and Oliver Marshall. «Alonzo Church». In: *The Stanford Encyclopedia of Philosophy*. Ed. by Edward N. Zalta and Uri Nodelman. Spring 2025. Metaphysics Research Lab, Stanford University, 2025.

[10] Evans. *Domain-Driven Design: Tacking Complexity In the Heart of Software*. USA: Addison-Wesley Longman Publishing Co., Inc., 2003. ISBN: 0321125215.

[11] Fabrick. *PISP, AISP and CISP: the Third Party Providers (TPPs) introduced by the PSD2*. 2023. URL: https://www.fabrick.com/en-gb/insights/blog/pisp-aisp-cisp/ (visited on 01/04/2025).

[12] Nicole Forsgren, Jez Humble, and Gene Kim. *Accelerate: The Science of Lean Software and DevOps Building and Scaling High Performing Technology Organizations*. 1st. IT Revolution Press, 2018. ISBN: 1942788339.

[13] Martin Fowler. *Continuous Integration*. 2006. URL: https://martinfowler.com/articles/continuousIntegration.html (visited on 01/04/2025).

[14] Vahid Garousi and Mika Mäntylä. «When and what to automate in software testing? A multi-vocal literature review». In: *Information and Software Technology* 76 (Apr. 2016). DOI: 10.1016/j.infsof.2016.04.015.

[15] *Global Financial Inclusion (Global Findex) Database 2021*. URL: https://microdata.worldbank.org/index.php/catalog/4607 (visited on 01/04/2025).

[16] Adam Murdoch Hans Dockter. 2008. URL: https://gradle.com/our-story/ (visited on 03/23/2025).

[17] Aslak Hellesøy. *Open source tool for running plain-language automated acceptance tests.* URL: https://github.com/cucumber.

[18] Jez Humble and David Farley. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. 1st. Addison-Wesley Professional, 2010. ISBN: 0321601912.

[19] Kohsuke Kawaguchi. *The leading open source automation server, Jenkins provides hundreds of plugins to support building, deploying and automating any project.* URL: jenkins.io (visited on 01/04/2025).

[20] Paul King. «A history of the Groovy programming language». In: *Proc. ACM Program. Lang.* 4.HOPL (June 2020). DOI: 10.1145/3386326. URL: https://doi.org/10.1145/3386326.

[21] *Linux lines of code number*. URL: https://openhub.net/p/linux/analyses/latest/languages_summary.

[22] Ian Sommerville. *Software Engineering*. 10th. Pearson, 2015. ISBN: 0133943038.

[23] Ossi Taipale et al. «Trade-off between automated and manual software testing». In: *International Journal of System Assurance Engineering and Management* 2 (June 2011). DOI: 10.1007/s13198-011-0065-6.

[24] JUnit Team. *JUnit 5 User Guide*. 2024. URL: https://junit.org/junit5/docs/current/user-guide/ (visited on 01/04/2025).

[25] JUnit team. *open-test-reporting*. 2024. URL: https://github.com/ota4j-team/open-test-reporting.

[26] *The 2023 McKinsey Global Payments Report*. 2023. URL: https://www.mckinsey.com/industries/financial-services/our-insights/the-2023-mckinsey-global-payments-report (visited on 01/04/2025).

[27] Jason van Zyl. 2007. URL: https://maven.apache.org/ (visited on 03/23/2025).

[28] Jason van Zyl. 2023. URL: https://maven.apache.org/background/history-of-maven.html (visited on 03/23/2025).