UNIVERSITÀ DEGLI STUDI
DI SALERNO

Analysis and Improvement of a Dynamic Web Project's Dependability

Student Report

Presented by:

Stefano Nicolò Zito

Dominykas Kruminis

Alisher Khaireden

Università di Salerno, Dipartimento di Informatica

January, 2026

Abstract

This report presents an analysis and enhancement of *Shapedibles*, a servlet-based e-commerce web application developed as a case study in software engineering practices. The application follows a classic multi-tier architecture, using Java servlets for request handling, Data Access Object (DAO) classes for database interaction and JavaBeans for data transfer, with persistence provided by a relational SQLite database.

The project focuses on improving confidence in the correctness, reliability and deployability of the application through a combination of formal specification, automated testing and containerisation. Core components of the system, particularly the DAO layer and selected business logic, are formally specified using the Java Modeling Language (JML) and verified with OpenJML. The effectiveness of the test suite is evaluated through code coverage analysis, mutation testing and performance benchmarking of critical components.

In addition, the application is containerised using Docker and deployed as a reusable image, enabling consistent execution across different environments. The deployment process is documented and designed to support orchestration through Docker Compose. Finally, the project integrates security analysis tools into the continuous integration pipeline to identify vulnerabilities and assess the overall security posture of the application.

The results demonstrate that the application can be reliably built, tested, deployed and analysed using modern software engineering techniques, providing a structured foundation for future development and improvement.

Table of Content

Introduction

The project analysed in this report is *Shapedibles*, a simple dynamic e-commerce web application designed to support online transactions.The application served as our case study for analysing software quality, correctness and deployability through a range of engineering techniques.

The system was made using a classic servlet-based architecture. In this model, Java objects named servlets are responsible for handling HTTP requests (HTTPS in the deployed configuration) and dynamically generating HTML responses, building pages.

The application data used is stored in a relational database (SQLite) and accessed through a set of Data Access Object (DAO) classes. These classes encapsulate all CRUD (Create, Read, Update, Delete) operations and transfer data using JavaBean objects, which the servlets can then use and consume.

While functional, the project is currently under development and is missing key features, such as link between pages and having a precarious implementation of other ones.

This report documents the process of analysing and enhancing the application through continuous integration, formal verification using JML, automated testing, container-based deployment, performance benchmarking and security assessment.

1 The application is buildable in CI/CD and locally.

https://github.com/stefzito8/SwD-Shapedibles

2 The core methods of the application have a formal specification in JML, verified using OpenJML.

2.1 Choosing core methods

The core methods chosen for this task are those related to the DAO classes. As these methods reside at the very bottom of our architecture, verifying them allows us to prevent contract errors that, if unchecked, could propagate through the rest of the application. In addition to the DAOs, basic JML specifications have been added to the Bean classes' methods, since these are used by the DAO classes and are also checked by OpenJML when invoked. The Cart class also has methods specified using JML, as it manages a unique object bound to the user session that manipulates bean objects.

2.2 Implementing and checking JML

The DAO classes are divided into two directories within the project:

- Dao: This directory stores the interfaces that define the methods.
- DaoDataSources: This directory contains the actual DAO classes which implement the aforementioned interfaces.

The files in both directories have been annotated with JML. Since the system supports inheritance in specifications, a specification declared in an interface method is automatically applied to all its implementations. Therefore, pre-conditions and post-conditions are applied to the method declarations in the interfaces, while assert statements, loop specifications and variable specifications (such as nullable) are applied within the DaoDataSource classes.

As previously mentioned, the Bean classes received the minimum specifications required to enable the DAO classes to invoke their methods: getter methods were marked as pure and setter methods were marked with assignable statements. The Cart class received specifications for the addProduct and deleteProduct

methods, ensuring that the operations for adding and deleting units from the cart are correctly implemented.

After being added to the classes, the specifications were statically analyzed with OpenJML first locally and then in CI/CD by integrating OpenJML check operations into the Maven Package GitHub Action.

2.3 Results

The results of these tasks are successful, as demonstrated by the completion of the Maven Package action in GitHub. The static analysis revealed no contradictions between the specifications of the chosen methods and their implementations.

3 Dockerised deployment of the web application

A Docker image of the web application has been created and pushed on Docker Hub. The image is designed to be easily accessible, readily orchestrated and executed in a containerised environment, ensuring portability and reproducibility across different operating systems and host configurations.

Before I go on, note that you may review the image here, download the zip file with the files used to build the image here and find the .yml file on the GitHub page of the project, please.

3.1 Understanding the stack

'*Minimising headaches.*'

Before containerising the application, the underlying technology stack was analysed. As a developer primarily experienced with *TypeScript* and *Preact*, transitioning to a more configuration-heavy Java/XML ecosystem configuration was a significant shift.

The use of *Fedora 43 KDE Plasma* was a turning point; its reliable environment and streamlined package management made installing the JDK and managing Java

environment variables much more approachable for a first-timer. To ensure full control over the deployment and to avoid the configuration choices made by other team members, a custom `server.xml` was authored from scratch. This ensured the Tomcat instance was tuned specifically for this container's requirements rather than relying on external defaults.

The application is strictly configured to serve HTTPS traffic. Special attention was given to understanding TLS certificates and Java .keystore files, which are necessary for Tomcat to handle secure handshakes.

3.2 The iterative dockerisation process

'*Commands used.*'

The application was containerised by defining a Dockerfile that builds upon an official Apache Tomcat base image. The Dockerfile installs the required runtime dependencies, configures the server, and deploys the application WAR file into the appropriate Tomcat directory.

3.2.1 Build, test, fail, learn, repeat

The containerisation process was an iterative cycle of trial, error and refinement:

- Initial attempt: created a basic Dockerfile using default Tomcat configurations, which resulted in a failure.
    - Reason: standard Tomcat defaults were insufficient for our specific `.war` requirements.
- Custom configuration: authored a server.xml via simply uncommenting the `SSL/TLS HTTP/1.1 Connector on port 8443` block of the file and specifying a specific certificate position, password, keystore type, encryption type and alias, which also resulted in a failure.
    - Reason: while a step in the right direction, the application also required secure HTTPS communication, which the container lacked.
- Manual security: Added a pre-generated .keystore file from the local Fedora environment, which finally resulted in an initial success.
    - Nuance: while the app worked, it created a massive privacy breach vulnerability.

3.2.2 The privacy breach and the pivot

During the testing phase, it became clear that including a local .keystore file in the build context was a critical security risk. If pushed to a public repository like Docker Hub, the private credentials would be compromised. To resolve this, the Dockerfile was completely rewritten to automate security. Instead of copying a sensitive file, the keytool command was integrated directly into the `RUN` layer:

```Dockerfile
# generating a new .keystore file to prevent a security leak
RUN keytool -genkeypair -alias tomcat -keyalg RSA -keysize 2048 \
    -storetype PKCS12 -keystore /root/.keystore -validity 3650 \
    -storepass changeit -dname "CN=localhost, OU=Student, O=Demo,
L=Salerno, S=Italy, C=IT"
```

This shift ensured that each build generated its own fresh credentials, keeping the developer's local environment separate from the public image.

3.2.3 Final deployment and pushing to Docker Hub

Once the "build-test-fail" loops were resolved and the Java XML configurations were understood and stabilised, the final image was built using `docker build -t mia-applicazione:vN.N .`, then – after logging into docker via `docker login` – the generated image, tagged as justalish/mia-applicazione:v1.3, was pushed to Docker Hub. This final version encapsulates the custom server.xml, the automated keystore generation, and the .war application, providing a "plug-and-play" deployment experience that abstracts the underlying implementation details from the end user.

3.3 Making it composable

'C*ommands not used, on purpose.*'

To simplify execution and orchestration, a docker-compose.yml file was provided to minimise the need for long, verbose Docker commands. Docker Compose is used to define how the container should be run, including port mappings and restart policies, without embedding runtime configuration into the image itself.

```yml
services:
  web-app:
    image: justalish/mia-applicazione:v1.3
    ports:
```

```
      - "8443:8443"
      - "8080:8080"
    restart: unless-stopped
```

Notably, the Compose configuration is intentionally limited to runtime concerns and does not include build instructions. This design choice ensures a clear separation between image creation and container execution, allowing the same image to be reused consistently across different environments.

3.4 Documentation

'*Making sure we're equal.*'

We think that a good piece of open-source software is effectively useless if no one can understand it, which is especially apparent in complex Java-based deployment. Recognising that, effort was put into documentation.

3.4.1 Dockerfile internal documentation

Every layer of the Dockerfile was commented to explain the why behind the keytool parameters and the file paths, making it accessible for future maintainers.

3.4.2 The README strategy

A comprehensive README was written to bridge the gap between source code and production. It serves three roles:

1. Repo Guide: Explaining the GitHub project structure.
2. Brief Tomcat Manual: Detailing how .war containers operate within Apache 9.0.
3. Deployment Guide: Providing a "one-command" startup guide using Docker Compose.

3.5 Issues, acknowledged yet undealt with

'*Compromises necessary.*'

While the final Dockerised deployment achieves its primary goal of providing a secure, reproducible, and portable execution environment, several known limitations remain. These issues were consciously identified during development

but left unresolved due to scope, time constraints, or because they fell outside the immediate requirements of the project.

3.5.1 Keystore password handling

Although the keystore is generated dynamically at build time to avoid credential leakage, the keystore password is currently hard-coded within the Dockerfile. In a production-grade deployment, this would be considered suboptimal. A more robust approach would involve injecting sensitive values via environment variables or Docker secrets at runtime, allowing credentials to be rotated without rebuilding the image. For the purposes of this project, the chosen approach prioritises clarity and reproducibility over operational flexibility.

3.5.2 Self-signed TLS certificates

The deployment currently relies on a self-generated TLS certificate, which is sufficient for enabling encrypted transport but does not provide trusted identity verification and results in browser warnings. During development, it was recognised that a more appropriate solution would involve the use of TLS certificates issued or approved by the University of Salerno, aligning the deployment with institutional security policies and trusted certificate chains.

Obtaining and integrating such certificates would require coordination with the university's IT or security services, including validation of domain ownership and compliance with internal infrastructure requirements. As this process extends beyond the technical scope of the project and depends on external administrative procedures, it was acknowledged but not pursued during implementation. Should the application be deployed within an official university-managed environment, replacing the self-generated certificate with institutionally issued credentials would be a necessary step.

3.5.3 Lack of container health checks

The Docker image does not define an explicit HEALTHCHECK. Container health was instead assessed through Tomcat startup logs and manual testing. While adequate during development, the absence of a health check limits automated orchestration capabilities and would reduce observability in more complex deployment scenarios.

3.5.4 Image size and optimisation

The deployment relies on the official Apache Tomcat base image, prioritising stability and documentation over minimal footprint. No image size optimisation techniques (such as multi-stage builds or distroless base images) were applied. This decision was intentional, favouring transparency and maintainability over aggressive optimisation.

4 The web application has a significant number of test cases

The SwD-Shapedibles application is supported by a substantial test suite comprising around 1000 test cases. This volume is not arbitrary but rather the result of applying distinct testing methodologies tailored to the specific responsibilities of each architectural layer. To effectively manage this scale and support efficient regression strategies, the project implements custom JUnit 5 tagging through categories.UnitTest and categories.IntegrationTest. This notable organizational feature enables selective test execution, allowing the team to differentiate between rapid, frequent unit tests and slower, periodic integration sweeps.

4.1 Methodological selection: Black-Box vs. White-Box testing

To balance thoroughness with efficiency, a hybrid testing strategy was adopted.
For the Model Layer, specifically the Beans and Data Access Objects (DAOs), the team employed a Black-Box testing approach. These components act primarily as data carriers and validation enforcement points. Consequently, the internal structure of a ProductBean or UserBean is less critical than its adherence to external specifications. By treating these classes as opaque entities, we utilized Equivalence Partitioning and Boundary Value Analysis to rigorously test input handling. This ensured that the domain constraints, such as preventing negative prices or enforcing string limits defined in the JSP forms, were validated strictly against requirements, regardless of the underlying implementation.
In contrast, the Control Layer required a White-Box strategy. The controllers in this architecture manage complex decision trees, including authentication flows, session handling, and conditional routing. A black-box approach would likely miss

subtle execution paths, such as specific error-handling blocks or edge cases in AJAX detection logic. Therefore, we designed tests based on the internal logic of the code, explicitly targeting Statement Coverage and Branch Coverage. This ensured that every logical branch, including exception handling for database failures, was exercised, verifying the robustness of the application's "nervous system".

4.2 Integration strategy: The Modified Sandwich approach

For Integration Testing, we selected a "Modified Sandwich" strategy. This approach was favored over a "Big Bang" integration because unit tests for the bottom layer (Model) were already robust. The strategy involved integrating the middle layer (Controllers) with the lower layer (Model/Database) while effectively simulating the upper layer (View/Filters). This allowed us to validate the critical path of data flow - from the H2 test database through the controller logic - without the overhead and complexity of a full UI deployment.

4.3 System testing limitations

It is important to acknowledge the limitations regarding System Testing. While end-to-end functional testing was considered, it was deemed infeasible for this phase due to the lack of an embedded servlet container in the test infrastructure. Without a running container to compile and render JSP pages, we focused our efforts on verifying the logic immediately preceding the view layer, ensuring that the correct attributes and forwarding paths were set, even if the final HTML rendering was not explicitly tested.

5 Code coverage is analysed using JaCoCo

To provide a quantitative measure of the test suite's thoroughness, we employed JaCoCo. This tool is great for its ability to generate granular metrics regarding bytecode execution, which served as a primary indicator of our White-Box testing effectiveness.

5.1 Selection of metrics: TER1 and TER2

The analysis prioritized two specific metrics derived from IEEE standards:

1. TER1 (Instruction Coverage): This measures the percentage of executable code instructions run during testing.
2. TER2 (Branch Coverage): This measures the percentage of control flow branches executed.

Tracking TER2 was particularly critical for this project. In web applications handling sensitive logic like user authentication or cart management, high instruction coverage can be deceptive. It is entirely possible to execute a method's lines without triggering every decision outcome (e.g., executing an if block but never the else). By enforcing a target of roughly 70% for Branch Coverage, we ensured that the decision logic itself, not just the linear execution, was solid. It's worth mentioning that we've also set a target of around 80% for Instruction Coverage.

5.2 Analysis of results and limitations

The project successfully achieved instruction coverage of 95% and branch coverage of 80%, surpassing the initial quality targets. However, the analysis revealed areas where coverage metrics must be interpreted with context. For instance, the config package exhibited 0% coverage. This was a deliberate exclusion rather than an oversight. Because this package handles the application's startup context and listener initialization, testing it would require a fully running servlet container. Mocking the entire servlet context to achieve artificial coverage here would have resulted in brittle tests that offered little value regarding production behavior. This underscores that while coverage is a necessary metric for identifying untested logic, it is not a standalone guarantee of quality.

6 A mutation testing campaign is conducted to analyse the test cases using PiTest

We wanted to ensure that the tests were sensitive to even minimal changes in business logic, such as a subtle error in a price calculation or a bypassed authorization check.

6.1 Execution and findings

The campaign yielded a strong Mutation Coverage score of 83% and a Test Strength of 89%. The analysis of the "surviving mutants" provided valuable insights into the suite's limitations. A notable portion of these survivors were identified as "equivalent mutants" - changes that do not alter the program's observable behavior, such as removing a logging statement or modifying a toString() method. Since these changes do not impact functional logic, the tests correctly continued to pass.
However, the campaign also uncovered valid gaps, particularly within the Controller layer's exception-handling logic. Certain mutations within try-catch blocks survived, indicating that while the tests triggered the exception paths (satisfying JaCoCo coverage), they did not always rigorously assert the system's state after the exception was caught. This finding validates the use of mutation testing, as it exposed weaknesses in error-handling verification that standard coverage metrics had missed.

7 JMH micro-benchmarks test the performance of the most demanding components of the web application

Functional correctness is insufficient if the system cannot scale. To address non-functional requirements, we employed the Java Microbenchmark Harness to evaluate the performance of the application's most critical components.

7.1 Component selection and justification

Rather than attempting to benchmark the entire application, we targeted specific components based on their execution frequency and computational complexity:

1. Authentication Filter: This component intercepts every single HTTP request. Any latency here scales linearly with traffic, making it a potential global bottleneck.
2. Cart Operations: The shopping cart logic involves iterating over collections of items, presenting a risk of *O(n)* complexity issues as cart sizes increase.
3. Database Access (DAO): As I/O operations are typically the slowest part of any web application, measuring the overhead of connection acquisition and query execution was essential.

7.2 Methodology: Throughput vs. Response Time

We selected different measurement modes depending on the operational context. Throughput was the primary metric for the Authentication Filter and Controller Routing. In a high-traffic e-commerce environment, the server's capacity to handle concurrent requests is the defining metric for scalability.
Conversely, for Cart Operations and Database Searches, we focused on Average Time (latency). These operations directly impact the user experience - a user perceives the delay when adding an item to the cart or searching for a product, making latency the more relevant metric for satisfaction.

7.3 Isolation strategy and tool limitations

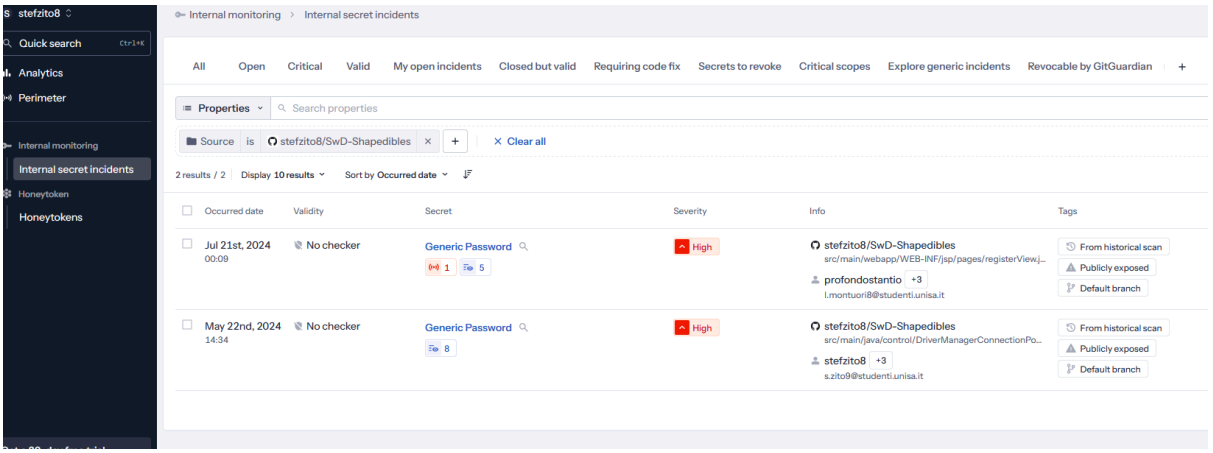
To benchmark the Data Access Object (DAO) layer without the noise of network latency or disk I/O, we utilized an H2 in-memory database as a test double.
We acknowledge the limitation that H2 does not perfectly replicate the performance characteristics of the production SQLite database. However, this trade-off was acceptable because our goal was to identify algorithmic bottlenecks rather than absolute production latency. By removing I/O variables, the benchmarks successfully highlighted code-level inefficiencies, such as the recalculation of cart totals on every read operation, providing a clear path for optimization that functional testing could not have revealed.

8 Security mechanisms are implemented in CI/CD

To implement security measures in CI/CD we have created a custom security rule around the main branch of the project, this rules imposes that commits on the main branch of the project must pass a Snyk and Sonacloud inspections, (realized through actions, as explained in the next section) showing no new vulnerabilities have been created inside the project.
All of the github actions have also been realized following a minimal permission approach and with the correct management of secrets, making sure to leave no exposed hardcoded variables in them.

9 Security is analysed using GitGuardian, Snyk and Sonacube Cloud

The code in the repository has been analyzed with all three of the requested services. Snyk and Sonacube Cloud have been integrated in CD/CI with github actions.
The results of Snyk and Sonacube Cloud analysis are fully visible through their action's results, and in the next section we will talk about what vulnerabilities have emerged and how they were solved.
As for the GitGuardian analysis, it only showed two secrets in its original analysis and none in following analysises. as shown in the image:



As for the vulnerabilities, the ways these secrets were solved is relayed in the next section of this document.

10 The web application shows no vulnerabilities

10.1 Solving code Secrets

Both the code secrets revealed by GitGuardian, where relative to exposed generic Passwords.
The First one revealed itself to be a false alarm, GitGuardian was misunderstanding a piece of text in a HTML label posed before a password insertion prompt for a generic password being inserted. The secret was simply ignored.
The Second Secret was also not a real risk but helped make the project cleaner. Originally, the project did not use a SQLite database stored inside the project itself, but relied on a connection to a classic SQL database. Although the change from classic SQL to SQLite happened during Task1, to better assist the usage of CI/CD, a file called DriverManagerConnectionPool.java was still present in the control folder of the project and contained hardcoded credentials to the original database connection. Even if this wasn't a risk anymore, since the connection to that DB was eliminated, the file was still deleted from the project, resolving the secret.

10.2 Solving Vulnerabilities

In total, 71 vulnerabilities have been found by Snyk, of this 71, 70 revealed themself to be false alarms, they were all "Use of Hardcoded Credential" and were relative to the system signaling the use of hardcoded credentials in the testing code, which is not a real vulnerability since this code is not an actually reachable from an outside attacker.
The only real vulnerability found by Snyk was a "Trust Boundary Violation". The filter.java class was saving data provided by the user in a session, considered a secure boundary in the code, without checking if it was valid data or cleaning it from potentially dangerous characters, making the system vulnerable to *CRLF*

injection. The issue was solved by simply implementing the correct validation and cleaning mechanism for the user data inside of the filter.java class.

11 Role of Large Language Model in the study

Large Language Models (LLMs) were used as a supporting tool throughout the development and analysis of the project. Their role was limited to assistance in understanding unfamiliar technologies and in generating auxiliary artefacts, while all architectural decisions, implementations, and validations were performed and reviewed by the authors. The contribution of LLMs can be divided into two main areas: *conceptual understanding* and *artefact generation*.

11.1 Conceptual understanding

LLMs were primarily used to support the understanding of technologies and frameworks that were not part of the authors' prior background. In particular, they were employed to clarify the syntax and semantics of Docker Compose .yml files, the behaviour of Docker images and tags, and common Git workflows, including error recovery and retrying failed operations.

Additionally, LLMs were used to assist in navigating and understanding the structure of the Apache Tomcat 9.0 server, including its directory hierarchy and configuration files. This support was especially valuable given that the primary development background of some contributors was in TypeScript rather than Java. In this context, LLMs helped explain established Java enterprise patterns, such as the DAO pattern, the separation between interfaces and implementations, and the interaction between servlets, beans, and persistence layers.

In all cases, LLMs were used as an explanatory aid to accelerate learning and reduce onboarding time, rather than as an authoritative source. The information provided was cross-checked against official documentation and validated through experimentation.

11.2 Artefact generation

LLMs were also used to assist in the generation of auxiliary artefacts that support development and analysis. These included initial drafts of unit tests, Java

Modeling Language (JML) specifications for selected methods, and configuration files related to testing and verification workflows.

In particular, LLM-generated content was used as a starting point for writing JML annotations for core methods and for scaffolding unit test structures. All generated artefacts were manually reviewed, corrected, and refined to ensure consistency with the actual behaviour of the application and to meet the requirements of the analysis tools used, such as OpenJML.

Importantly, LLMs were not used to generate core application logic or architectural components. Their role in artefact generation was limited to reducing repetitive effort and providing structured templates, while correctness, completeness, and suitability remained the responsibility of the authors.

Conclusion

The work presented in this report confirms that the application of rigorous software engineering practices can significantly elevate the reliability, security and deployability of a legacy servlet-based application like Shapedibles. By integrating formal verification via OpenJML, the project established a proved foundation for the application's core data interactions, ensuring strict adherence to specifications within the DAO and Bean layers without revealing logical contradictions. This mathematical approach to correctness was complemented by a robust containerization strategy; the move to a Docker-based architecture not only resolved environment inconsistencies but also enforced security through automated keystore generation and reproducible deployments via Docker Compose.

Furthermore, the project's multi-layered testing strategy demonstrated the value of combining distinct methodologies. The hybrid approach of black-box and white-box testing resulted in a suite of approximately 1000 test cases, achieving 95% instruction coverage. Crucially, the inclusion of mutation testing with PiTest went beyond standard metrics to expose subtle weaknesses in the controller layer's exception handling.

Finally, the integration of performance benchmarking and automated security analysis into the CI/CD pipeline ensured the system was not only functional but

also efficient and secure. The resolution of critical vulnerabilities, such as Trust Boundary Violations, and the identification of algorithmic bottlenecks via JMH, transformed the application from a precarious case study into a hardened software artifact. Ultimately, this project successfully demonstrates that modern DevOps and verification techniques can be effectively applied to traditional Java architectures to create a reliable and maintainable system.

References

GITHUB. *How to generate unit tests with GitHub Copilot: Tips and examples*. [online]. 2024. [viewed 28 January 2026]. Available from: https://github.blog/ai-and-ml/github-copilot/how-to-generate-unit-tests-with-github-copilot-tips-and-examples/

APACHE SOFTWARE FOUNDATION. *Apache Tomcat 9.0: SSL/TLS Configuration How-To*. [online]. 2024. [viewed 28 January 2026]. Available from: https://tomcat.apache.org/tomcat-9.0-doc/ssl-howto.html

ORACLE CORPORATION. *keytool – Key and Certificate Management Tool*. [online]. 2021. [viewed 28 January 2026]. Available from: https://docs.oracle.com/en/java/javase/17/docs/specs/man/keytool.html

STACK OVERFLOW. *Does "Unit Testing" fall under white box or black box testing?*. [online]. 2012. [viewed 28 January 2026]. Available from: https://stackoverflow.com/questions/9892963/does-unit-testing-falls-under-white-box-or-black-box-testing

IEEE. *610.12-1990 - IEEE Standard Glossary of Software Engineering Terminology*. [online]. 1990. [viewed 28 January 2026]. Available from: https://ieeexplore.ieee.org/document/159342

JUNIT. *JUnit 5 User Guide: Writing Tests – Tagging and Filtering*. [online]. 2024. [viewed 28 January 2026]. Available from: https://junit.org/junit5/docs/current/user-guide/#writing-tests-tagging-and-filtering

GEEKSFORGEEKS. *Docker Compose YAML Explained: A Deep Dive into Configuration*. [online]. 2023. [viewed 28 January 2026]. Available from:

https://www.geeksforgeeks.org/devops/docker-compose-yaml-explained-a-deep-dive-into-configuration/

DOCKER. *Writing a Dockerfile*. [online]. 2024. [viewed 28 January 2026]. Available from: https://docs.docker.com/get-started/docker-concepts/building-images/writing-a-dockerfile/

COURSERA. *Introduction to Containers w/ Docker, Kubernetes & OpenShift*. [online]. 2024. [viewed 28 January 2026]. Available from: https://www.coursera.org/learn/ibm-containers-docker-kubernetes-openshift