PARCEL TRACKING SYSTEM: DESIGN, IMPLEMENTATION, TESTING AND REVIEW

LEARMONTH Anders
l018160a@student.staffs.ac.uk

Contents

Introduction

This report details the design, build, testing and review of a parcel tracking system which allows users of multiple pre-assigned roles to perform differing actions against parcels, orders, and their current stage in the delivery process. This could be used for an online shop looking to migrate to a single enterprise system with all data accessible in one location.

The application uses Java EE to allow interaction with a Java database and includes JUnit to enable automated testing at all stages of development and to safely add features to the system in the future without the risk of unknown breakages elsewhere.

Design and Build

The design and build of this application are wholly consistent with the design document (Appendix A) which includes consideration for use cases from the specification, activity diagrams, the analysis model, class diagram which also accurately represents the later explained design patterns used, sequence diagrams and the entity relationship diagram. This document was produced early in the development process to ensure consistency within the application and to avoid deviating from the requirements.

Choices made

Revision control

GitHub has been used for source control on this project to help mitigate risk of loss of work and to allow rollbacks of un-successful implementation as well as clarity on progress made over time to ensure timely completion of milestones.

Database

The database design is complex but manageable and highly scalable allowing future additions to complement the overall system without negatively impacting the existing areas. Making use of 7 tables with 4 one-to-many relationships and 10 foreign keys set up, all the required data can be stored, acting as the central storage of the application, accessible by a series of SQL statements.

Model View Controller (MVC)

The application has been split into its functional components being the view, model, and controller. Where the view is the web tier including all xhtml files and templates including CSS and other visual components. The model contains the business logic to ensure business level rules are enforced throughout the application along with being responsible for communicating with and manipulating the database(s) attached to the system. The controller in this application was handled by Java Server Faces where the action requested by the client is automatically processed through the model and redirects the user to the specified view without developer interaction.

Filter

A filter was added to handle user logins and restricts access by reviewing a white list of resources that can be accessed by users that do not satisfy the “is logged in” requirement, including the logo, CSS files, the xhtml pages and templates but to leave the remaining, un-specified files as restricted to keep unauthorised users outside of the system until they have logged in with valid credentials.

Design pattern 1: Data Transfer Object (DTO) pattern

This serializable object handles the transfer of data for this specific object type within the system and enforces its validity of data and completeness (Fowler, Data Transfer Object (EAA Catalog), 2003).

Design pattern 2: Table/view data gateway pattern

Acting as an interface between the application and the database through use of CRUD methods and usually a series of additional find methods specific to the application. Typically, one table data gateway class exists per table or one view gateway class per one or more tables in the database and all SQL for the application should reside in these classes (Fowler, Table Data Gateway (EAA Catalog), n.d.). A view data gateway pattern was used in this application.

Design pattern 3: Factory pattern

A factory is useful when a client can make a variety of requests and the developer requires a method of directing the correct resources to fulfil the request (Design Pattern - Factory Pattern, n.d.). It is possible to use a single factory for the entire application but here the commands have been split into one factory per user type to improve readability and scalability for future development, at the cost of more dependency injection where some beans may require access to multiple command factory files to complete their tasks.

Design pattern 4: Command pattern

The command pattern gives developers the ability to append the command factory to insert new functionality into the system without affecting other, pre-existing features, helping to keep the system modular and highly scalable (Design Patterns - Command Pattern, n.d.).

Design pattern 5: Façade pattern

Providing a single point of entry for requests that will be directed to its required action on the user's behalf, this will help to hide complexity when there are many classes involved. In this application the Command factory was used as the Façade (Design Patterns - Facade Pattern, n.d.).

Alternative design strategies

The main problem during implementation turned out to be the time restriction, where the application turned out to be much larger than originally expected which took up a lot of the allocated project time which meant the following two strategies could not be implemented.

Session beans

Enterprise java beans (EJB) allow the developer to separate concerns in the sense of business logic being stored separately to the application, enabling the possibility of running the back end on a separate server, meaning the client is only responsible for client-intended tasks. It should be carefully considered whether to incorporate session beans into an application due to its added complexity and time commitment to set up (Oracle, n.d.). Session beans were not used in this application due to the time constraint and to keep complexity low.

Java persistence API (JPA)

Acts as an interface between application and database which can be mapped to the system entities through an Entity manager and allows data transfer using DTOs. This can be especially useful when switching database provider as the API can be re-pointed with the new settings and application queries can remain unchanged (Biswas & Ort, 2006). However, due to time constraint JPA was not included in this project.

Testing & Results

The black box testing approach was applied to this project, where the tester knows the inputs that will be provided and the expected outputs but does not care too much about what happens in-between. This is useful when unit testing to ensure functions are performing their intended action, while abstracting away from the inner workings of the system since this level of complexity would not provide added value (Black box testing, n.d.).

Automated testing should be used wherever possible and from as early in the project as is feasible to allow finding problems earlier, resulting in easier and usually much cheaper adjustments to correct. Using JUnit for this allows the initial developer(s) of an application to start a test suite which can be later appended by any successors in a way to ensure future additions or modifications to the system do not break any existing components (Guru99, n.d.).

A comprehensive test plan including Unit, Integration and manual tests was derived directly from the use cases outlines in the specification document. By creating a series of test cases including normal, invalid, boundary and special values, it was possible to create an extensive test plan with almost 100 percent coverage of the application using 153 JUnit and 3 manual tests which were required to test the application filter, all of which passed and are evidenced in the Test Plan document (Appendix B).

All boundary and invalid test cases are tested in isolation to ensure one error does not mask another and that all situations are accounted for prior to release to production. Each test is independent of all other tests without dependency to allow to be run out of sequence or in part, both saving time and reducing complexity, allowing the developer to isolate problem areas and fix bug much more easily.

It was expected that there would be problems when testing the Managed Beans that had dependency injection of other classes due to a known issue with JUnit (Lopez, 2019). However, this was not the case and all tests were able to run without third-party tools.

Future development opportunities

Improvements to existing components

Although not required in the original specification, there should be consideration to include more checks using validators prior to actions taking place, for example, if a user or parcel already exists in the DB then it should warn the requestor to prevent duplicate records being added. To accompany this, there should be full implementation of validator message and messages outputs in the case of incorrect data entry/failed validation checks.

The use of a third-party testing tool may or may not be necessary but enhanced testing should be investigated using tools including JSFUnit or Mockito which will allow automated testing of classes which have injections by processing the files as if they are mocks of the real object instead of its literal representation within the system. This will allow more coverage of the entire system, increasing confidence that it functions correctly when initially releasing or applying updates, although the current test set up did not see the expected issues so this may not be required.

New features

The order details page could be modified to perform a calculation of sub-total weight per line item and a total weight of the order to add a delivery charge based on overall weight using a database lookup with pre-set prices per weight range.

A converter class could be created to handle weight conversion from grams to pounds and a similar approach applied for converting from GBP to EUR and other currencies to enable a more pleasant user experience with future additions to the application.

Design alterations

With more time to work on the project, the simple JSF application could be migrated to use session beans and the java persistence API to increase portability with options to change database provider and to further separate business logic to a dedicated server to increase both performance and maintainability. With the server side of the application externally hosted, there should be a discussion on whether to develop a native front-end to replace the web-based faces front end which can provide a native feel for the user.

Conclusion

The application in its current state is fit for purpose and fully meets the original specification including performing for all 22 use cases across the 4 user types outlined in the specification, successfully performing all the required tasks consistently and without issues. This is backed up by the comprehensive set of tests both automated and manual, with the results providing confidence that every aspect of the application functions as expected with both valid and erroneous data input attempts.

The sheer number of requirements derived from the original specification document should be limited for a project of this complexity requirement and timeframe restriction as it was quickly discovered that the work required to implement and test these increased exponentially. If this project were to be undertaken again, 12 use cases will be used rather than the 22 found in this solution.

Overall, the solution presents a database of 7 tables with 4 one-to-many relationships which is used in a Java EE application using JSF, managed beans and 5 design patterns including DTO, data gateway with CRUD operations, factory, command, and façade. With a full test suite of JUnit, manual tests and integrations tests based on the use cases, which fully satisfies the outlined specification requirements and is wholly consistent with the professional design document (Appendix A).

References

Biswas, R., & Ort, E. (2006, May). *The Java Persistence API - A simpler programming model for entity persistence*. Retrieved from Oracle: https://www.oracle.com/technical-resources/articles/java/jpa.html

Black box testing. (n.d.). Retrieved from Guru99: https://www.guru99.com/black-box-testing.html

Design Pattern - Factory Pattern. (n.d.). Retrieved from Tutorials Point: https://www.tutorialspoint.com/design_pattern/factory_pattern.htm

Design Patterns - Command Pattern. (n.d.). Retrieved from Tutorials Point: https://www.tutorialspoint.com/design_pattern/command_pattern.htm

Design Patterns - Facade Pattern. (n.d.). Retrieved from Tutorials Point: https://www.tutorialspoint.com/design_pattern/facade_pattern.htm

Fowler, M. (2003, January). *Data Transfer Object (EAA Catalog)*. Retrieved from Martin Fowler: https://martinfowler.com/eaaCatalog/dataTransferObject.html

Fowler, M. (n.d.). *Table Data Gateway (EAA Catalog)*. Retrieved from Martin Fowler: https://martinfowler.com/eaaCatalog/tableDataGateway.html

Guru99. (n.d.). *JUnit*. Retrieved from Guru99: https://www.guru99.com/junit-tutorial.html

Lopez, D. A. (2019, October 7). *Unit testing and dependency injection*. Retrieved from The Coders Tower: https://coderstower.com/2019/10/07/unit-testing-and-dependency-injection/

Oracle. (n.d.). *The Java EE 6 Tutorial - Session Beans*. Retrieved from Oracle: https://docs.oracle.com/javaee/6/tutorial/doc/gipjg.html

Appendices

Appendix A Design documents

ParcelTracker
DesignDiagrams.pd

Appendix B Test plan

ParcelTracker
TestPlan.pdf