LEARMONTH Anders

l018160a@student.staffs.ac.uk

Parcel tracking system: Design, implementation, testing and Review

Contents

[Introduction 1](#_Toc59643709)

[Design and Build (include references to back up my descriptions of tools) 1](#_Toc59643710)

[Choices made 1](#_Toc59643711)

[Filter to handle user logins 1](#_Toc59643712)

[Design patterns used 1](#_Toc59643713)

[Alternative design strategies 2](#_Toc59643714)

[Session beans (enterprise java beans (EJB) here? ) 2](#_Toc59643715)

[Java persistence API 2](#_Toc59643716)

[Java Messaging Service 2](#_Toc59643717)

[Active record? 2](#_Toc59643718)

[Data Mapper pattern? 2](#_Toc59643719)

[Testing & Results 2](#_Toc59643720)

[JUnit tests 3](#_Toc59643721)

[Integration tests 3](#_Toc59643722)

[Future development opportunities 3](#_Toc59643723)

[Conclusion 4](#_Toc59643724)

[Appendices 4](#_Toc59643725)

[References 4](#_Toc59643726)

Write a report (up to 1,000 words) in which you critically evaluate your application. You should consider:

• the design, describing alternatives and justifying your selections;

• the results of testing, making recommendations for further development.

Critically reflect on and draw conclusions about the fitness for purpose of an application

You should ensure that your report presents evidence to support the awarding of marks in each criterion of the rubric

# 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 (include references to back up my descriptions of tools)

## Choices made

Justification for the choices/selections of tools/things I used/did with this project

Db is complex but manageable and highly scalable allowing future additions to complement the overall system without negatively impacting the existing areas.

**Model View Controller (MVC)**

View is the web tier with xhtml files and templates including CSS and other visual components.

Model contains the business logic to ensure business level rules are enforced within the application along with being responsible for communicating with and manipulating the database(s) attached to the system.

Controller is handles by Java Server Faces in this case where the action requested by the client will be handled automatically, processing the model actions and redirecting the user to the specified view without developer interaction. (Talk about why the front controller isn’t/cannot be used in this application here)

### Filter to handle user logins

Providing a white list of resources that can be accessed by users that do not satisfy the 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.

### Design patterns used

Which 5 design patterns did I use, why did I use them and how did I implement them?

**DTO pattern, data transfer object** – This object handles the transfer of data for this specific object type within the system and enforces its validity of data and completeness.

**Table/view data gateway pattern** – acts 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.

**Factory pattern** – what is command factory? Although its possible to use a single command factory for the entire application, the commands have been split into one factory per user type to allow greater readability and scalability for future development to remove irrelevant commands from the list being worked on, further reducing the chance of breakages elsewhere in the system. Although this requires more dependency injection as a trade off where some beans may require access to multiple command factory files to complete their required tasks.

**Command pattern** – Developers can append to the command factory to insert new functionality into the system without affecting other, pre-existing features, keeping the system modular and highly scalable.

**Façade pattern** –

Any troubles when building? Design patterns, libraries, dependency injection? …

## Alternative design strategies

Describe alternative methods available that could have been used in this project, including their pros and cons. And why I did not use them in this project, could they later be implemented to help solve a specific problem or improve scalability/maintainability/performance/reliability?

### Session beans (enterprise java beans (EJB) here? )

### Java persistence API

Handles DB interactions

### Java Messaging Service

### Active record?

### Data Mapper pattern?

# Testing & Results

I have used black box testing with a combination of automated unit tests and integration tests, some of which must be carried out manually due to required dependency injection.

Include results from testing, using JUnit tests, and why? (allow modular improvements while ensuring the overall system still functions as expected against the already outlines unit tests, nothing breaks)

My test plan… including edge cases in my testing, checking expected pass and fail results

Using black box testing with known inputs, do we get the expected output(s)?

Automated testing of classes with dependency injection have been omitted for this project for simplicity as they would have failed or been much more complex to prepare and set up.

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.

## JUnit tests

Automated testing should be used wherever possible and from as early in the project as is feasible, finding problems earlier is always easier and cheaper to correct than to find problems later in development. Benefits of automated testing, how did I use this, why did I use this, alternative(s) to this are manual testing and… pros/cons

## Integration tests

With my use cases

## Manual tests

For tests that must be carried out manually due to dependency injection of some beans.

# Future development opportunities

What features would be nice to add?

Much better validation, including checks for if a user/parcel already exists in the DB to prevent duplicate records being added.

Make use of session beans + java persistence API, why??

Consider developing a native front end to replace the web based faces front end, why??

Any features that can be optimised/re-designed to provide more functionality or be better performance/more scalable or reliable?

Full implementation of validator message and messages for incorrect data entry/failed validation checks. Currently an application error is thrown in erroneous input, it would be more professional to catch these and display meaningful error messages to guide the user.

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.

A parcels.timesSold column could be added to keep a total number of times a parcel has been added to an order, this will enable metrics of most popular parcels and their customers within the system.

Perform calculation of sub-total weight per line item and total weight of order to add a delivery charge based on overall weight using a database lookup with pre-set prices per weight range.

Converter classes for weight from grams to pounds and converting from GBP to EUR and other currencies.

# Conclusion

Critically evaluate the application

Does the application meet the original specification, does it perform all required tasks?

What were the results from testing?

Are all of the original 12 use cases satisfied?

What could be improved or done differently if this project were to be completed again?

# Appendices

**No table of figures entries found.**

# References

**There are no sources in the current document.**