1DV600
Software Technology
Assignment 1

Jonas Sjöberg
860224-xxxx
Linnaeus University
js224eh@student.lnu.se
https://github.com/jonasjberg
http://www.jonasjberg.com

Written during: February 1, 2017 – February 5, 2017
Teacher responsible: Jesper Andersson

Abstract

Assignment in the course *1DV600 – Software Technology*, distance given by Linnaeus University during the spring of 2017. The assignment covers basic planning and documentation of a software project, with the main purpose to provide an insight into the process of creating computer software. The assignment is focused on the process itself rather than the code, this includes documentation, planning, testing, estimations of requirements, etc.

Contents

List of Tables

List of listings

0 Assignment overview

This is the report for the first assignment in the course `1DV600` – "Software Technology" given at Linneaus University during the spring of 2017.

0.1 Background

The assignment covers development of a web application for storing books in a library or database. The course and assignment is primarily focused on the "workflow" itself, as compared to the production of source code.

0.2 Purpose

Main emphasis is on the planning and documentation of the project as to simulate developing an actual product in the real word, more often than not in a team of other developers. The purpose of this is to practice a range of engineering skills; analysis, planning, documentation, estimating required time and work, etc.

1 Task 1 – Personal Planning

Instructions from the course Wiki[1]:

> When the client requests a list of books to present for the user it does the call `http://localhost:9090/api/books/` to the server and it expects the answer as a JSON object (an associative array). We are going to split the functionality into three tasks, but it is your task to plan these tasks. Take into account the time for learning and understanding of the problem when you plan the time. Make your planning with 15 minutes as the minimum unit. Repeat the following pattern for all subtasks (A, B, C):
>
> - Plan
> - Implement
> - Reflect
>
> Each subtask should be documented with at least 100 words.

1.1 Subtask A – Books

Instructions from the course Wiki[1]:

> The main objective of the subtask is to create a list of books and a function or method to get them. There are slight differences depending on which implementation you are using, either Java or Node.js but those differences will be clearly noted. Common for both is that they should handle books, and for each book we need the information id, title, author, genre, publish date, price and description.
>
> **Java** Create a class in the package "models" that represents a book. After that, create a short list of fictive (or real) objects in the function getBooks that is available in GetBooksResource. When calling the URL http://localhost:9090/api/books the list of books should be outputted with System.out.println. The subtask is done when you see the objects in the terminal (where vagrant is run).

1.1.1 Plan

The `book` class is a critical datatype that will be used throughout the software. As such, it would probably be wise to design the class in a way as to make it flexible and open for future modifications, as stated by the well known "open/closed principle"[2] – in brief[3]:

> Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification.

In practical terms, this would mean avoiding primitive data types and instead creating ones own custom data types for the metadata fields; author, title, etc.

The class `book` should meet the following requirements:

- Store information needed to describe books handled by the system. We are to include the following set of attributes:
  - Id
  - Title
  - Author
  - genre
  - Date of publication
  - Price
  - Description
- Provide suitable interfaces for accessing and possibly modifying the data encapsulated in the class.

The initial plan is to implement a basic class with suitable private fields for storing data about a certain book, as well as traditional mutators for accessing private fields.

This first implementation will use primitive data types for its fields, which often means future expansion and modifications will require refactoring. This design choice goes against the previously mentioned "open/closed principle", but future refactoring seems a reasonable trade-off in order to get a prototype working as soon as possible.

In other words: this implementation should use the least amount of code possible to meet the requirements.

1.1.2 Implement

The class was implemented in less than 10 minutes. What followed was investigating bugs in the underlying framework "dropwizard"[4]. The problems seems to be caused by printing a combination of newlines and other characters to "stdout", which is hooked by either "dropwizard" or the build system "gradle"[5], in order to add additional information and timestamps to the message. An error occurs somewhere along this chain.

1.1.3 Reflect

This simple implementation matches the criteria which seems vague at this point, the `book` class presents its state in `JSON`-format, and so deciding on implementation details becomes very difficult. These details are for example the internal data types in the "book" class and the overall program structure, including communication between different parts of the program

which again uses some kind of data types. These details grow to reach ever larger parts of the source code as the software grows in size and complexity. I know for a fact that this simple implementation will need serious rework in order to be extended. But making the design more involved and modular at this stage, with no information to go on, could possibly be a waste of time.

This kind of trade-off seems to reoccur and turn up in various forms. The trade-off is between the following two extremes:

A Try to predict future usage and development, design for maximum flexibility and modularity. Means more initial work is needed to reach a minimum working state, but can pay off in the long run, the design makes modification and extension easier. The risk is that the extra functionality and complexity might go complete unused, and the whole enterprise is thus a complete waste of time.

B Assume very little about future developments, use the simplest possible design that meets the requirements. Accept that the simple solution is more rigid and sensitive to changes — future changes in requirements or functionality will require major refactoring.

1.2 Subtask B – `JSON`

Instructions from the course Wiki[1]:

> Convert the objects created in subtask a into an `JSON` object and show it in the terminal using either System.out.println(Java) or console.log (Node.js).
>
> **Improvement Strategies** Choose two improvement strategies based on your reflections on subtask a and b. Describe what you have decided to improve and why. Implement your improvements in the next subtask.

1.2.1 Plan

As with the first subtask, the plan is to use the simplest possible solution that meets the requirements. The first implementation will use simple, error-prone and crude raw string manipulations to construct `JSON`-data. If this method performs as expected when integrated into the rest of the system, methods for converting from Java objects to `JSON`-data could be refactored into a separate utility package.

1.2.2 Implement

The code was written in less than 5 minutes. The basic method for constructing the strings was re-used from the previous subtask.

The method that returns the `JSON` data, "`toJSON()`" is shown in Listing 1.

```java
/**
 * Returns the state of this book as JSON data.
 *
 * @return A JSON representation of this book.
 */
public String toJSON()
{
    final String FORMAT = "\"%s\": \"%s\", ";

    StringBuilder sb = new StringBuilder("{");
    sb.append(String.format(FORMAT, "id", getId()));
    sb.append(String.format(FORMAT, "title", getTitle()));
    sb.append(String.format(FORMAT, "author", getAuthors()));
    sb.append(String.format(FORMAT, "genre", getGenre()));
    sb.append(String.format(FORMAT, "published", getDate()));
    sb.append(String.format(FORMAT, "price", getPrice()));
    sb.append(String.format(FORMAT, "description", getDescription()));
    return sb.append("}").toString();
}
```

Listing 1: Initial implementation of the `toJSON` method in the `Book` class.

1.2.3 Reflect

This should really be solved using the standard libraries native functions for handling `JSON`. At the very least, these methods should be extracted and refactored into utility functions stored in a separate package with other similar "utility" code. This enables re-use across the entire project. Many companies re-use utility code across many if not all projects.

Improvement Strategies Based on the reflections on Subtask A 1.1.3 and Subtask B 1.2.3, the two following improvements was chosen:

1. Improve `JSON` handling by leveraging the Java libraries.
2. Create classes to wrap primitive data types – add containers for the metadata fields such as author, title, etc.

1.3 Subtask C – Web

Instructions from the course Wiki[1]:

> In this subtask you are to answer the request in the web browser instead of printing it to the terminal. The subtask is done when you see the `JSON` object on screen. For inspiration, have a look at PingResource that you find in the same folder as the GetBooksResource.
>
> If you follow the API for the model (as seen in GET api/books) , you will be able to show the books in the list.

1.3.1 Plan

As per the instructions, the plan is to look for the existing code to hints on how to meet the requested functionality.

The chosen improvement strategies will be solved by using the suggested JSON library "jackson"[6].

1.3.2 Implement

The implementation was done as part of experimenting with the previous two subtasks and the strange behaviour that the framework "dropwizard" exhibited when new-lines was passed to the terminal.

The library "jackson" now handles `JSON` conversion, this revised method "`toJSON()`" is shown in Listing 2.

```
1  /**
2   * Returns the state of this book as JSON data.
3   *
4   * @return A JSON representation of this book.
5   */
6  public String toJSON()
7  {
8      String jsonString  = null;
9      ObjectMapper mapper = new ObjectMapper();
10
11     try {
12         jsonString = mapper.writeValueAsString(this);
13     } catch (JsonProcessingException e) {
14         e.printStackTrace();
15     }
16
17     return jsonString;
18 }
```

Listing 2: Revised implementation of the `toJSON` method in the `Book` class.

The book metadata field "genre" in the `Book` class was improved by using an `enum` instead of a string. This adds type checking as the compiler will catch any errors at compile-time – the genre must be set to one of the predetermined set of possible values. The `enum` type also opens up for future addition of other functionality.

1.3.3 Reflect

The modifications done to the class was minor but did improve the overall quality and robustness of the code.

All the metadata fields should probably completely wrap their contained data. Possibly, an abstract class called "MetadataField" could be added. This class would have functionality for setting and getting the metadata value in a safe and controlled manner.

Specific fields, for example "author" could also be stored in a class "AuthorMeta" that inherits from the "MetadataField" class. These subclasses would then add their own behaviour, specific to their contained metadata.

This would allow future expansion and handling of special cases like multiple authors, etc.

2 Task 2 – Vision

Instructions from the course Wiki[1]:

> Create a vision document for the system. This should be a document covering about half an A4 page describing the system. The purpose of the document is to make sure that everyone involved in the project has the same vision of what is to be created. Use the "Assignment Overview" and previous subtasks as your source for what to write. In addition, write down your reflections on creating a vision document. This reflection should be about 100 words.

2.1 Vision Document

The project described is a web-application for managing a collection of books. This application is the foundation for a library system for books where you can add, modify and delete books.

2.1.1 Intended use

The user should be able to add books to the collection. Adding a book means that the book metadata should be provided. Books are added by manual entry – the user fills out a form with a predetermined set of fields, thereby populating the book metadata.

The application should be able to handle common user errors and missing data.

The program should be able to present the book collection to the user in a table-like format, with one book per row. Each column displays one metadata field and the columns are populated with metadata for the book at that row.

The user should be able to delete books from the database by clicking a suitably named button.

2.1.2 Design

Books are stored in a database using `JSON` serialization and basic key/value lookup.

Each book has an unique id. No two books can have the same id. The id can be used as a handle or reference to a certain book stored in the database.

Each book is described by and stored with a set of metadata fields. We are provided with a set of required fields which will be used as a starting point:

- Id
- Title
- Author
- genre
- Date of publication
- Price
- Description

The only field that must be present is the id.

2.1.3 Extended functionality

These are ideas on possible extra, "wishlist" functionality that could be added in future releases.

I have previously written a program in Python that automatically renames e-books from analysis and querying servers with `ISBN`-numbers to fill out missing fields. Software for managing e-books often have this kind of functionality, one example is the open source software package "calibre"[7].

This program could also be extended to query services for any missing metadata fields. I have some previous experience with automatic metadata extraction from various documents and related metadata.

It also possible to add functionality for the program to populate all fields automatically if the user simply enters a valid `ISBN` number.

2.2 Reflection on the creation of a Vision Document

I am used to documenting my own projects in this way. This includes both analog electronics, embedded systems and software. The first thing I do when starting a project is to sketch out what I imagine a working product could look like. I also like to pretend I'm using the product, this often makes it easier to see which features would be useful. This goes for both command-line programs and actual physical, analog signal processing units.

Sitting down and thinking hard about the product, its indented use and what it would be like to interact with it, can be very revealing. A lot of less good ideas can be weeded out quickly.

Also; at some level of complexity, there is simply no way to keep track of everything without continuously documenting the development process. After some time away from a project, it can be very difficult to remember what was going on.

The documentation also acts as a reminder as to which ideas sparked the initial interest. It is important to keep on track, especially when prone to "over-engineering"[8] over let "feature-creep"[8] hijack allocated time or budget.

Overall, I've found that documentation is key for a successful project. Especially when working in a team, but also when working with your future self.

3 Task 3 – Project Plan

Instructions from the course Wiki[1]:

> Write a project plan for the project. This project plan should show the way to the complete and finished application, something that you should be able to follow. Write as much as possible in the project plan, add the milestones from Assignment Overview, and update the document throughout the course when you know more in the later assignments. Again, as an addition, write down your reflections on creating a project plan. This reflection should be about 100 words.

3.1 Project Plan

3.1.1 Overarching Goals

The main overarching goal of the project is to produce a simple library system for managing a collection of books.

Starting points:

1. Fix bugs in the build system. Currently, I am unable to redeploy without restarting the virtual machine that runs the web server, which is completely unacceptable.
2. Research the frontend source code to better understand how exceptions seem to be thrown for simple errors in the `JSON` data, like stray commas and whitespace.
3. Develop classes to act as containers for all metadata fields, eliminating use of primitive data types to storage book metadata.
4. Continously adjust and re-evaluate during continous integration as problems occur when connecting the code being developed with the given front-end code.

I will initially focus mainly on the above issues. The project plan is thus expected to change drastically during development.

3.1.2 Planning Strategy

I will initially focus primarily on these main points. More detailed planning currently barely seems worthwhile as the overarching starting points are very likely to shift or change rapidly. Additionally, I lack serious experience with web development in general. And in particular the build system `gradle` [5], the `VirtualBox` [9] wrapper `Vagrant` and the provided front-end based on the `dropwizard` [4] framework.

So I am not able to produce a valid long-term plan. The project has no choice but to adopt a more “agile” approach.

I will adopt an iterative, agile approach – short concise changes and constant re-evaluation of the planning.

Table 1: Time Log for Assignment 1

Date and time	Activity	Task performed
2017-02-05 19:14:30	Documentation	Add skeleton time log.
2017-02-05 18:56:00	Development	Update ..
2017-02-05 18:45:05	Documentation	Major fix-up, update and restructure.
2017-02-05 17:26:19	Documentation	Update report. Add first draft of Task 2.
2017-02-05 08:05:15	Documentation	Complete first draft of Task 1.
2017-02-05 07:29:13	Documentation	Update .. Add second JSON implementation.
2017-02-05 01:50:33	Documentation	Remove inline enumeration.
2017-02-05 01:45:39	Documentation	Add inline enumeration.
2017-02-05 01:20:14	Documentation	Various. Add code listing.
2017-02-05 01:15:28	Development	Various smaller fixes ..
2017-02-05 00:56:33	Development	Fix up Javadoc.
2017-02-04 20:47:33	Development	Start adding Javadoc ..
2017-02-04 20:47:16	Documentation	Update report ..
2017-02-04 20:24:52	Development	Update .. Completes subtask 1B.
2017-02-04 19:46:59	Development	Update ..
2017-02-04 19:20:56	Documentation	Update report ..
2017-02-04 18:36:17	Development	Work in progress ..
2017-02-04 18:14:22	Development	In progress update ..
2017-02-04 17:57:39	Development	Add in progress first implementation.
2017-02-04 16:47:24	Documentation	Initial commit.

4 Time Log

Instructions from the course Wiki[1]:

> Each assignment must be accompanied with a time log. This time log should contain the date, time and task to be performed. The reason for doing this is for you to get some experience in estimating your own time creating a time log is one of the best ways of doing this.

4.1 Generated time log

The time log for this assignment is shown in Table 1. I keep both the assignment report sources and the program source code in `Git` repositories. The logs over commits was extracted to produce the time log table.

The majority of time was spent on writing the documentation. My estimations of time needed to write the code were very good. However, the documentation took much longer than I had anticipated; which in turn, I actually *did* anticipate.

References

[1] T. Ohlsson, *Assignment 1 `tobias-dv-lnu/1dv600-lab` wiki, Revision commit hash `ac564f7`*, [Online; accessed 04-Feb-2017], 2017. [Online]. Available: `https://github.com/tobias-dv-lnu/1dv600-lab/wiki/Assignment-1/ac564f729dc829e8f83220338e65398e52723a3e`.

[2] R. C. Martin, "The open-closed principle [software maintenance]", *J-C-PLUS-PLUS-REPORT*, vol. 8, no. 1, pp. 37–43, Jan. 1996, ISSN: 1040-6042.

[3] B. Meyer, *Object-Oriented Software Construction (Prentice-Hall International series in computer science)*. Prentice Hall, 1994, ISBN: 0136290493.

[4] *Dropwizard, Production-ready, out of the box.* [Online; accessed 04-Feb-2017], 2016. [Online]. Available: `http://www.dropwizard.io/1.0.6/docs/`.

[5] *Gradle build tool, Modern open source automation*, [Online; accessed 04-Feb-2017], 2017. [Online]. Available: `https://gradle.org/`.

[6] Y. M. Kim, *Jackson 2 – convert java object to/from json*, [Online; accessed 05-Feb-2017], Oct. 2015. [Online]. Available: `http://www.mkyong.com/java/jackson-2-convert-java-object-to-from-json/`.

[7] K. Goyal, *Calibre – e-book management*, [Online; accessed 05-Feb-2017], Feb. 2017. [Online]. Available: `https://calibre-ebook.com/`.

[8] Wikipedia, *Overengineering — wikipedia, the free encyclopedia*, [Online; accessed 5-February-2017], 2016. [Online]. Available: `https://en.wikipedia.org/w/index.php?title=Overengineering&oldid=740045993`.

[9] *Oracle vm virtualbox, User manual*, version 5.0.10_Ubuntu, Mathworks, 2004 – 2015, p. 86. [Online]. Available: `http://www.virtualbox.org`.