



# Requirements and analysis document for Shedbolaget

Authors: Samuel Kajava, Emil Svensson, Pouya Shirin and  
Daniel Rygaard

TDA367

Chalmers tekniska högskola

2021

Version 4.3

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Purpose of the application . . . . .	1
1.2	Scope of application . . . . .	1
1.3	Objectives and success criterias of the project . . . . .	1
1.4	Definition, acronyms and abbreviations . . . . .	1
<b>2</b>	<b>Requirements</b>	<b>2</b>
2.1	Functional requirements . . . . .	2
2.2	Non-functional requirements . . . . .	2
2.2.1	Usability . . . . .	2
2.2.2	Reliability . . . . .	2
2.2.3	Performance . . . . .	2
2.2.4	Supportability . . . . .	2
2.2.5	Packaging and installation . . . . .	2
2.3	Application models . . . . .	2
2.3.1	User case: Show specific products . . . . .	3
2.3.2	User case: Show all beverages . . . . .	3
2.3.3	User case: Popular beverages . . . . .	4
2.3.4	User case: Search for a beverage . . . . .	4
2.3.5	User case: Detailed beverage information . . . . .	4
2.3.6	User case: APK information . . . . .	5
2.3.7	User case: Custom product . . . . .	5
2.3.8	User case: Drink generator . . . . .	6
2.3.9	User case: Favourites . . . . .	6
2.4	Definition of Done . . . . .	7
2.5	User interfaces . . . . .	8
<b>3</b>	<b>Domain Model</b>	<b>15</b>
3.1	Class Responsibilities . . . . .	15
<b>4</b>	<b>References</b>	<b>16</b>

# 1 Introduction

## 1.1 Purpose of the application

The purpose of this project is to recreate a similar version to Systembolaget.se with additional functionality. The user is able to mark products as favourite, sort them with custom sorting methods and product variables. It is designed to extend the functionality of Systembolaget in some aspects. The user is also able to create custom products in case they do not find the product they're searching for.

## 1.2 Scope of application

Although the purpose of the project is to recreate Systembolaget, the goal is not to copy all of Systembolaget's functionality, but rather add some extra functionality on top of its basic design.

## 1.3 Objectives and success criterias of the project

- List all of Systembolagets products and display its information.
- Save favouritized products locally.
- Display products according to the user's needs.
- Custom products works like any other normal product.

## 1.4 Definition, acronyms and abbreviations

- **Systembolaget** - a government-owned chain liquor store in Sweden.
- **Shedbolaget** - the project/application name.
- **APK** - (alkohol per krona)/alcohol per crown

## 2 Requirements

### 2.1 Functional requirements

The user should be able to:

- Browse through all of Systembolaget's products.
- Filter products by two level of categories.
- Sort products from highest to lowest and vice versa.
- Favourite products and check their favourites.
- Add their own custom product.

### 2.2 Non-functional requirements

#### 2.2.1 Usability

The application's primary language is Swedish and will be designed to target an adult audience. With a single navigation bar, the application is designed to be simple and reliable.

#### 2.2.2 Reliability

The project fetches its data from the original website - Systembolaget. The displayed data is not completely in-sync with Systembolaget's, as it was fetched on 2021-09-02.

#### 2.2.3 Performance

The application should be running with low loading times.

#### 2.2.4 Supportability

Application should run on Mac, Windows and Linux.

#### 2.2.5 Packaging and installation

To install and run the project, it's required to clone the project repository and build the application with Maven. The instructions are found in the README.md file inside the project root.

### 2.3 Application models

**High priority use cases:**

- Show specific products
- Show all beverages
- Search for a beverage
- Popular beverages

**Medium priority use cases:**

- Popular beverages
- Detailed beverage information

**Low priority use cases:**

- APK information
- Custom product
- Drink generator
- Favourites

### **2.3.1 User case: Show specific products**

As a user, I want to select specific beverage types to view so that I can see all beverages within that selection.

**Status:** Done.

**Acceptance:**

- Can I select a specific type and show all products which share that property?
- Can I select several types at once and view all relevant products?
- Can I go the other way and exclude products?

**Tasks:**

- Show options for what to select.
- Show what property is selected.
- Show products within the selection.

**Priority:** High.

### **2.3.2 User case: Show all beverages**

As a user, I want to be able to browse all beverages so that I can get an overview of what's available. **Status:** Somewhat done.

**Acceptance:**

- Can I view all products in the app?
- Can I see an image of each product?
- Can I see specific information about the product?

**Tasks:**

- Show all beverages.
- Show the price of each product.
- Show a thumbnail of each beverage.

**Priority:** High.

### **2.3.3 User case: Popular beverages**

As a user, I want to see new beverages as soon as I launch the application because I often want to find drinks when I'm unsure what to order at the bar.

**Status:** Done.

**Acceptance:**

- Can I see images of new beverages without much effort?
- Can I easily see how much a new beverage costs?

**Tasks:**

- Show new beverages.
- Show thumbnail image of the popular beverage.
- Show price of the popular beverage.

**Priority:** Medium.

### **2.3.4 User case: Search for a beverage**

As a user, I want to be able to search for a product and find information about it so that I can quickly look up specific beverages.

**Status:** Done.

**Acceptance:**

- Can I search for specific products in the range of products?
- Can I use the search function to filter out products in a specific category?
- Am I able to search for products anywhere in the application?

**Tasks:**

- Show search box.
- Allow the user to enter a search term in the search box.
- Show results based on the search term.

**Priority:** Medium.

### **2.3.5 User case: Detailed beverage information**

As a user, I want to see detailed information about a product, so that I can learn more about products that interest me.

**Status:** Done.

**Acceptance:**

- Can I view an image of the product?
- Can I see:

- How much it costs?
- Product APK?
- Product volume?
- Alcoholic percentage?
- Product description?
- Can I obtain this information from anywhere where there are products?

**Tasks:**

- Show search box.
- Allow the user to enter a search term in the search box.
- Show results based on the search term.

**Priority:** Medium.

### 2.3.6 User case: APK information

As a user, I want to be able to find the products with the highest APK in a leaderboard so that I can see which products are the most cost efficient in terms of alcohol contents.

**Status:** Done.

**Acceptance:**

- Can I view the products with the highest APK?
- Can I choose which product types to view when I'm looking at APK values?
- Can I access an item and get more information about it from the list?

**Tasks:**

- Show the leaderboard.
- Show the products with the highest APK ratio.
- Allow the user to choose which type of beverages are shown.
- Show which type of beverage each product is as well as its name, alcohol contents, volume, apk and price.

**Priority:** Low.

### 2.3.7 User case: Custom product

As a user, I want to be able to add a product that doesn't exist in the application so that I can use the custom product with the application's services.

**Status:** Done.

**Acceptance:**

- Can I add the the custom product?

- Can I use the custom product as any other product?

**Tasks:**

- Offer to add the product.
- Allow user to enter values for the custom product
- Display the custom product
- Use the custom product like a normal product

**Priority:** Low.

### **2.3.8 User case: Drink generator**

As a user, I want to be able to find drinks based on products I select so that I can find new drink recipes to try out.

**Status:** Done.

**Acceptance:**

- Can I choose some ingredients and get new drinks with added ingredients?
- If I'm not satisfied with my result, can I retry?

**Tasks:**

- Allow users to select some known ingredients.
- Generate new drinks based on the known ingredients.
- Show the generated drinks on screen.

**Priority:** Low.

### **2.3.9 User case: Favourites**

As a customer, I want to be able to mark some products as favourites because I want to remember to keep buying those.

**Status:** Done.

**Acceptance:**

- Can I mark a beverage as a favourite?
- If I change my mind, can I unmark favourited beverages?

**Tasks:**

- Allow user to mark beverages as favourite.
- Allow the user to view all marked beverages.
- Allow the user to unmark beverages in case they change their mind.

**Priority:** Low.



## 2.4 Definition of Done

- All acceptance criteria are met
- All public methods has unit tests.
- All Travis checks pass.
- Completed Java doc (if applicable)
  - All new files need to have an author.
  - All public methods except getters and setters has Javadoc.
- Pull request is accepted by everyone in the group

## 2.5 User interfaces

# Välkommen till Shedbolaget.

Sök och sortera bland 22170 produkter.

## Nya produkter

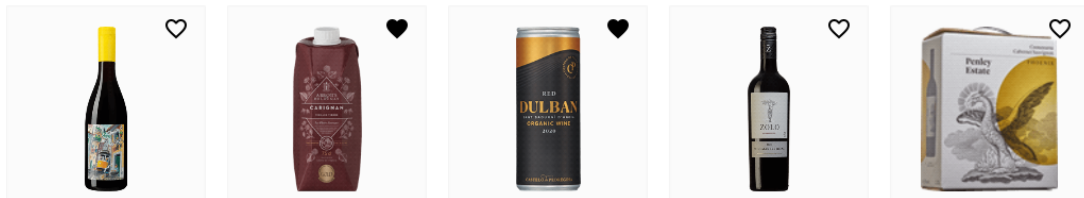


Figure 1: This is the starting page for the application. A welcoming message with products is displayed at the bottom part of the application.

## Nya produkter

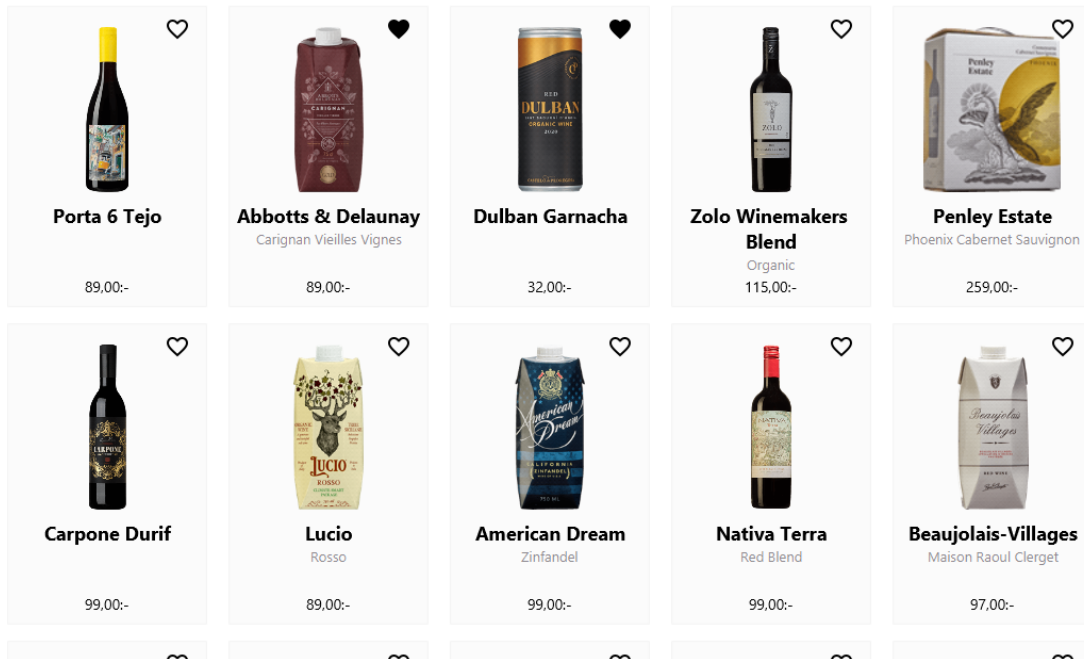


Figure 2: Further down in the start page is a grid of products under the label "Nya produkter".

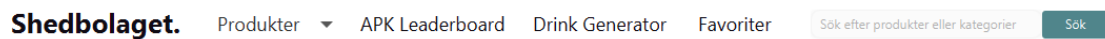



Figure 3: This navigation bar is shown at the top of the application. This bar is used for almost all of the navigation paths. The Escape Hatch included in the top left corner and the Search bar in the opposite corner. Between these, the user finds the necessary links to the main pages of the application.

**Shedbolaget.**
Produkter
APK Leaderboard
Drink Generator
Favoriter

**Sortera efter:**  
Pris


**Drycker:**  
☒ Alkoholfritt  
☐ Öl  
☐ Mousseerande  
☐ Must  
☐ Cider & Blanddryck  
☐ Glögg & andra juldrycker  
☐ Vitt  
☐ Aperitif & Bitter  
☐ Rosé  
☐ Rött

## Alkoholfritt




**Einbecker Brauherren**  
Alkoholfrei Pils  
Alkoholhalt: 0,5 % | 330 ml | APK: 0,21  
Öl | Tyskland

7,90:-




**Maisel's Weisse**  
Alkoholfrei  
Alkoholhalt: 0,0 % | 330 ml | APK: 0,00  
Öl | Tyskland

8,90:-




**Störtebeker 1402**  
Alkoholhalt: 0,5 % | 330 ml | APK: 0,19  
Öl | Tyskland

8,90:-



**Kiviks**  
Skånsk Äppelcider Alkoholfri  
Alkoholhalt: 0,5 % | 330 ml | APK: 0,17  
Cider & Blanddryck | Sverige

9,90:-



**Pistonhead**  
Flat Tire Non Alcoholic  
Alkoholhalt: 0,5 % | 330 ml | APK: 0,15

10,90:-

Figure 4: The products page shows a list of products on the right, and a choice of secondary categories to the left.

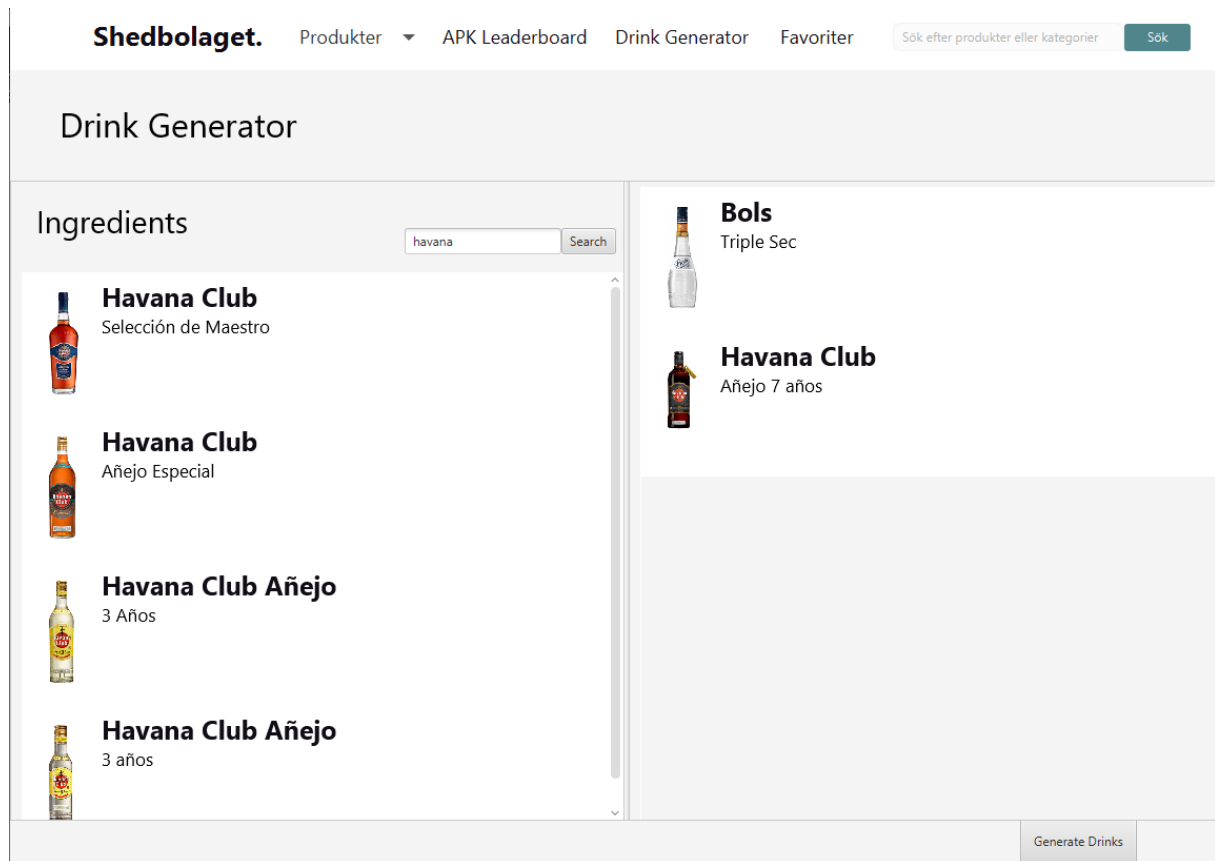


Figure 5: The drink generator page is split into two panels where the left panel displays a list of liquor that is used in drinks, and the right side is the selected liquors.

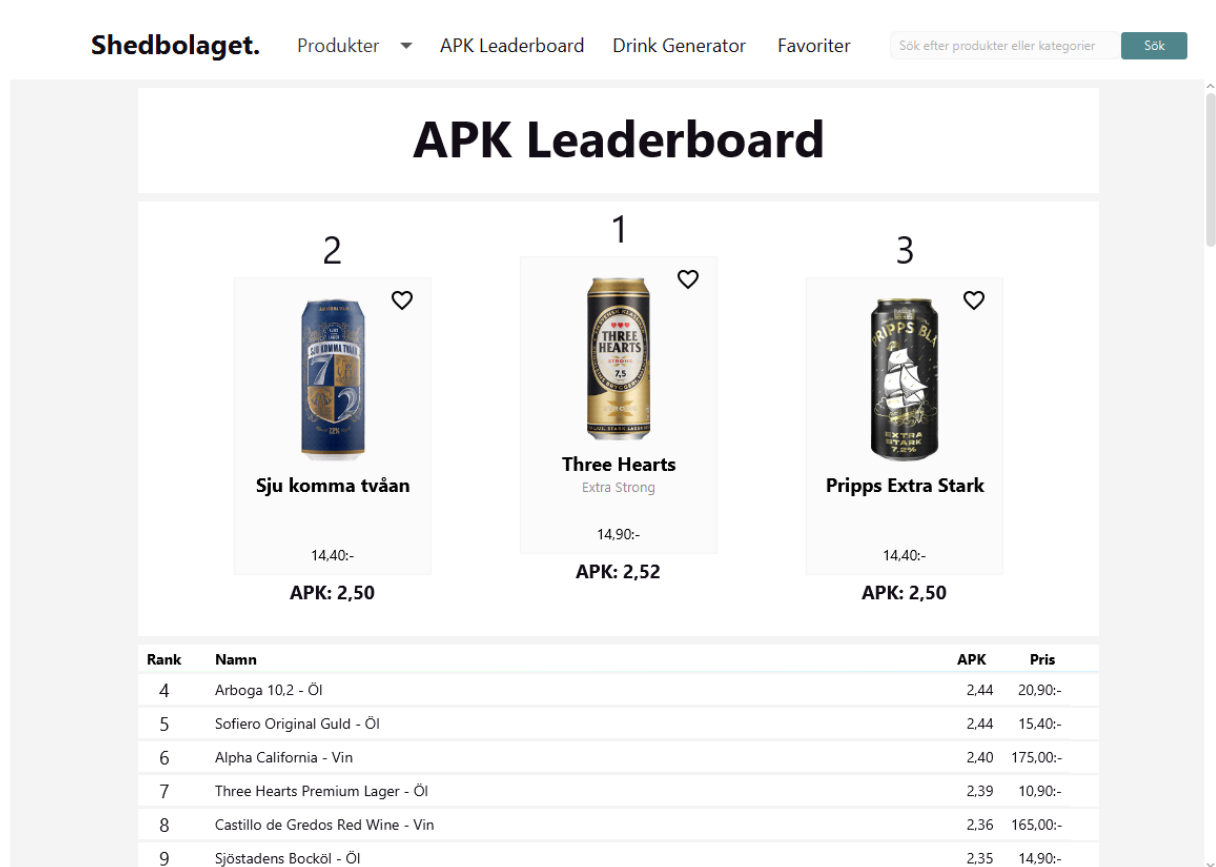


Figure 6: APK leaderboard.

The APK leaderboard shows the top 100 products with the highest APK in descending order.




**Shedbolaget.** Produkter ▾ APK Leaderboard Drink Generator Favoriter


Sök efter produkter eller kategorier **Sök**

**Hittar du inte din produkt?**  
Lägg till den!


## Favoriter

**Faustino VII**  
Rosado  
Alkoholhalt: 12,5 % | 187 ml | APK: 1,17  
Rosé | Spanien

20,00:-  
[Ta bort favorit](#)

**Wildberries Seend**  
Alkoholhalt: 4,5 % | 330 ml | APK: 1,31  
Blanddryck | Sverige

11,30:-  
[Ta bort favorit](#)

**Fem komma tvåan**  
Alkoholhalt: 5,2 % | 330 ml | APK: 2,00  
Ljus lager | Sverige

8,60:-  
[Ta bort favorit](#)

< 1/1 >

Figure 8: List of favourite products.



### 3 Domain Model

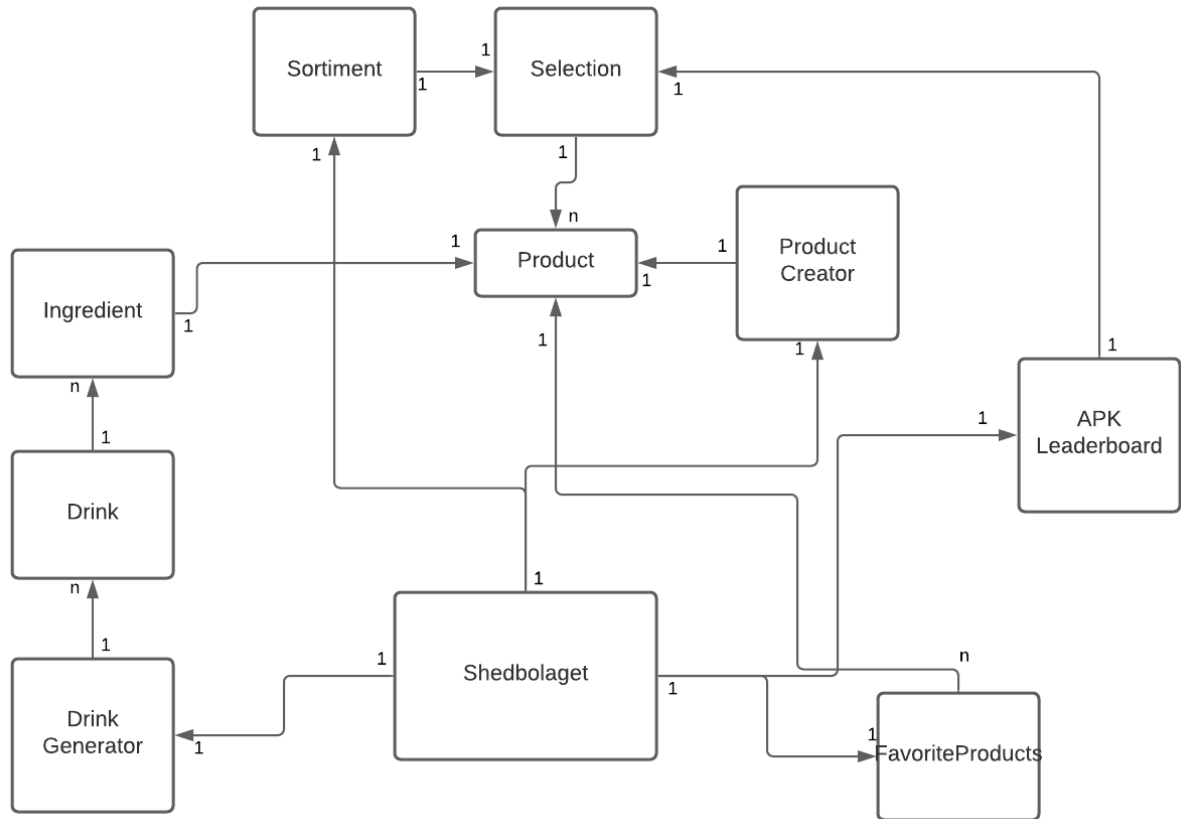


Figure 9: Domain Model

#### 3.1 Class Responsibilities

As observed in the domain model image, all class modules work independently with the exception of the references to the Product class. All modules are bound to the product class in some way, since the whole program is built upon the product data. Each of the modules has their own area of responsibility and the model 'Shedbolaget' ties together them to create the designed application.

## 4 References

- Systembolaget
- Maven



# System design document for Shedbolaget

Authors: Emil Svensson, Pouya Shirin and Daniel Rygaard

TDA367

Chalmers tekniska högskola

2021

Version 4.1

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Overview . . . . .	1
1.2	Purpose of this document . . . . .	1
1.3	Word book . . . . .	1
<b>2</b>	<b>System architecture</b>	<b>2</b>
2.1	Application flow . . . . .	2
<b>3</b>	<b>System design</b>	<b>3</b>
3.1	Software decomposition . . . . .	3
3.1.1	Model . . . . .	3
3.1.2	Controller . . . . .	4
3.1.3	View . . . . .	4
3.2	Model package . . . . .	5
3.2.1	Categories . . . . .	6
3.2.2	Events . . . . .	7
3.2.3	Products . . . . .	8
3.2.4	Favorites . . . . .	14
3.2.5	Drinks . . . . .	15
3.3	Domain model . . . . .	16
3.3.1	Drink . . . . .	16
3.3.2	Product . . . . .	16
3.3.3	FavoriteProducts . . . . .	17
3.3.4	Product Creator . . . . .	17
<b>4</b>	<b>Persistent data management</b>	<b>18</b>
<b>5</b>	<b>Quality</b>	<b>19</b>
5.1	Build automation . . . . .	19
5.2	Tests . . . . .	19
5.3	Continuous integration . . . . .	19
5.4	Analytical tools . . . . .	20
5.4.1	PMD . . . . .	20
5.4.2	Dependency matrix . . . . .	20
5.5	Known issues . . . . .	21
<b>6</b>	<b>References</b>	<b>22</b>

# 1 Introduction

## 1.1 Overview

Shedbolaget is an application used for browsing through products collected from Systembolaget. It mimics the basic functionality of Systembolaget's website, but also has additional features that's not available on the original website. It is for example possible to create custom products, mark products as favorites and get suggestions on drinks to make based on selected products.

## 1.2 Purpose of this document

The SDD document contains all relevant information required to get a clear picture of the system architecture, system design and level of quality. It can be used as a resource for the development team to assist in further development of the application, but also aims to give the non-initiated an understandable overview of the system design.

## 1.3 Word book

- **Systembolaget** - a government-owned chain liquor store in Sweden.
- **Shedbolaget** - the project/application name.
- **APK** - (alcohol per krona)/alcohol per crown
- **Drink** - alcoholic drink
- **MVC** - Model-View-Controller
- **FXML** - an user interface markup language used by JavaFX
- **JSON** - a lightweight data-interchange format easy for both computers and humans to read
- **Event Bus** - component used to send messages between different parts of the system

## 2 System architecture

The architectural design pattern Model-view-controller is used to structure the application. The views are responsible for the presentation of the application, displaying what the user sees. For every view, there's a separate controller class that communicates with the view while the application main logic is handled by the model. The controllers contain minimal logic (user input logic) and utilizes the model to update the state of the application. Each view gets updated accordingly to reflect the model's state. Furthermore, the model is implemented to work separately without the views and the controllers, which is why they communicate through events. The model sends out an event as soon as its state gets changed, and the controllers observe these events to update the views.

The application retrieves its data from multiple static JSON files containing products and drinks and does not depend on any database or external source for storing or retrieving information. In addition to this, a file saving system is implemented and utilized to save favorited and custom products locally.

### 2.1 Application flow

: The following steps occurs when running the application:

1. The main method in *Main* gets called, which in turn calls for the JavaFX initialization process. This instantiates the *RootWindow* class, which creates views and controllers for each page of the program. Every page uses *ProductModel* - a Singleton class that loads the products into a List (via the parser package) at startup. The *Favorites* class - another Singleton class will also instantiate itself at startup which parses the text file of saved product ID:s into Product objects.
2. The application is idle at this stage, waiting for user input. Whenever user input occurs, an event gets fired which will update the views via the Event Manager in the model.
3. When closing the application, a shutdown hook in *Favorites* gets called that writes all favorited product ID:s in a file before the application terminates itself.

## 3 System design

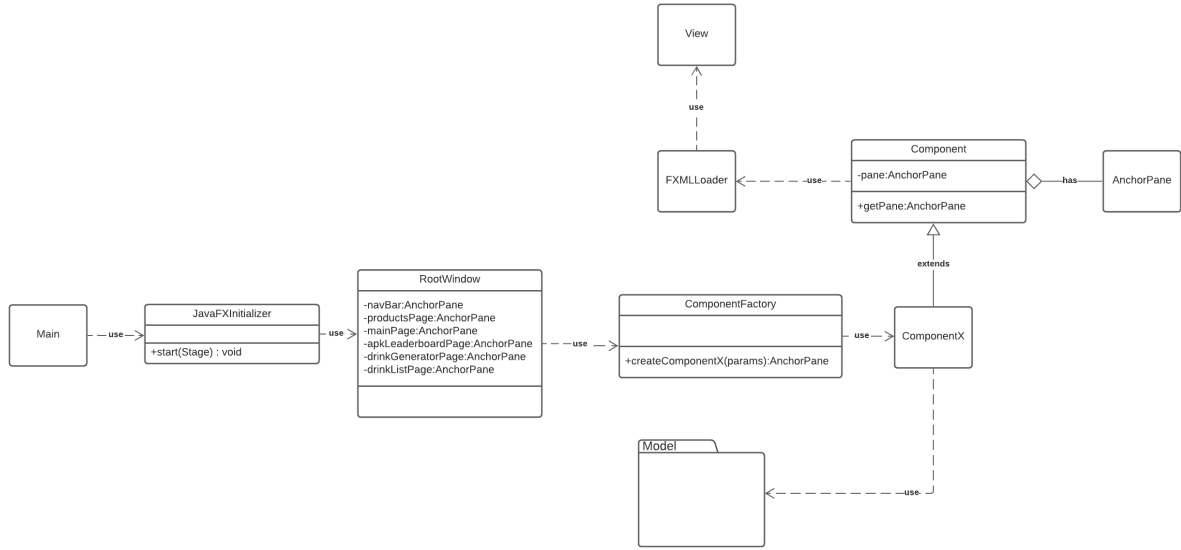


Figure 1: Overview of the MVC design pattern

The source code is divided into three main packages with the intention to follow Separation of Concern: model, view and controller.

Figure 1 shows how the MVC pattern is implemented in the application. Note that JavaFX, together with SceneBuilder is used for the views which uses the FXML format for its views. This differs a bit from the classic MVC design, as the views do not contain any code. Instead, the views depend on a controller to manipulate its state and the controllers depend on the views. This connection between the view and controller is handled by the abstract class *Component*, which maps a view to a controller and loads the view using *FXMLLoader*.

### 3.1 Software decomposition

#### 3.1.1 Model

The *model* package is divided into several modules for each main feature and submodule for every responsibility and concern. The *model* package is also responsible for the data that the application uses.

The application's main module is the *products* package. This module is used by almost every other submodule in some way. The *products* package contains the *Product* class which the whole application depend on. Inside the *products* package, a *ProductModel* Singleton is used as a facade for the underlying process of parsing the data. Every module that is used directly with the *Product* class is inside the *products* package. Inside the *model* package is another subpackage that takes care of serialization/de-serialization of product data.

### 3.1.2 Controller

The *controller* package consists of controller classes with the exception of a Factory class that instantiates the controllers. Controller classes are 'stupid' in the way that they do not contain any logic. They are used as a middle-man between the view and the model. The controller observes the model and updates the view to reflect the model's state, and the controller also mutates the model accordingly to the user's actions. The controller handles the input from the view and acts appropriately.

### 3.1.3 View

The view is the collection of FXML files in the resources folder that displays the model data. Each view is represented by their respective FXML file and has their own unique controller they communicate with. This is what the user of the application sees. Like the model, the view is somewhat independent. However, without a controller class, the view is unable to do anything.



### 3.2 Model package

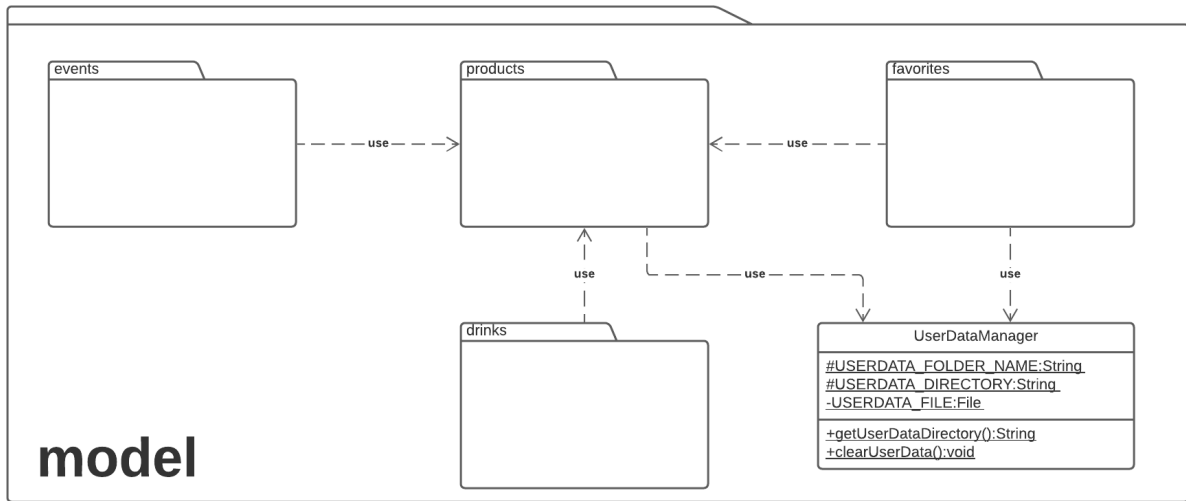


Figure 2: *model* package

As observed in Figure 2, the *model* package is divided into smaller sub-packages, reflecting their functionality. All packages rely on the *products* package in some way since the whole application is built on it.

The *UserDataManager* class is a static class which handles the user data folder and returns the path directory.

### 3.2.1 Categories

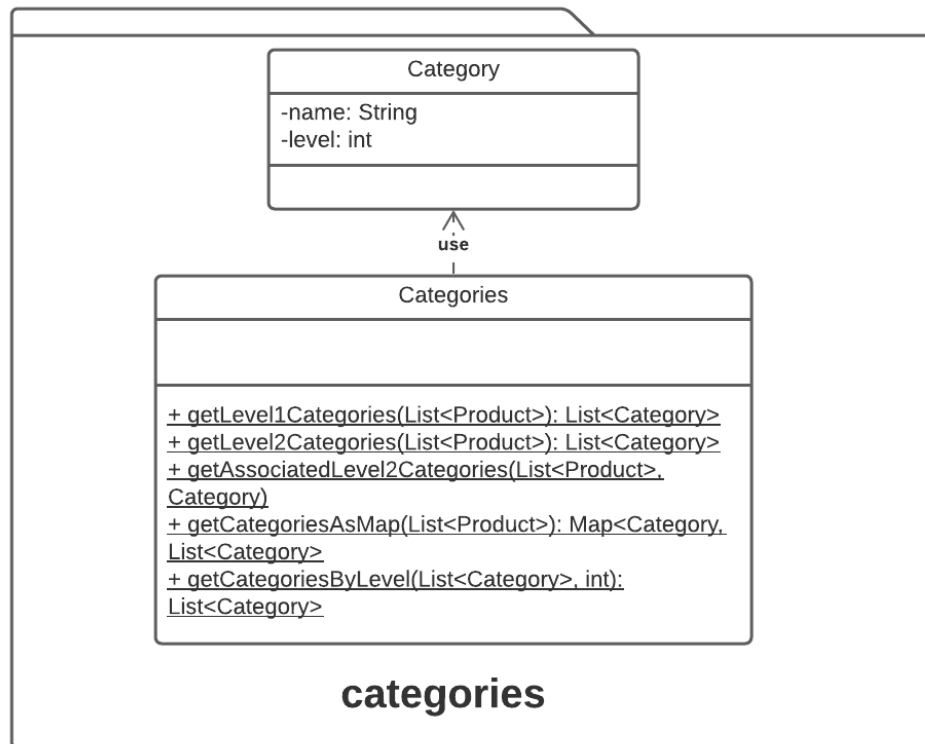


Figure 3: *categories* package

This package contains two classes: *Category* and *Categories*. *Category* is used by the *Product* class and *Categories* class. The *Categories* class mainly creates lists out of the *Category* class with a *List* of products as input.

### 3.2.2 Events

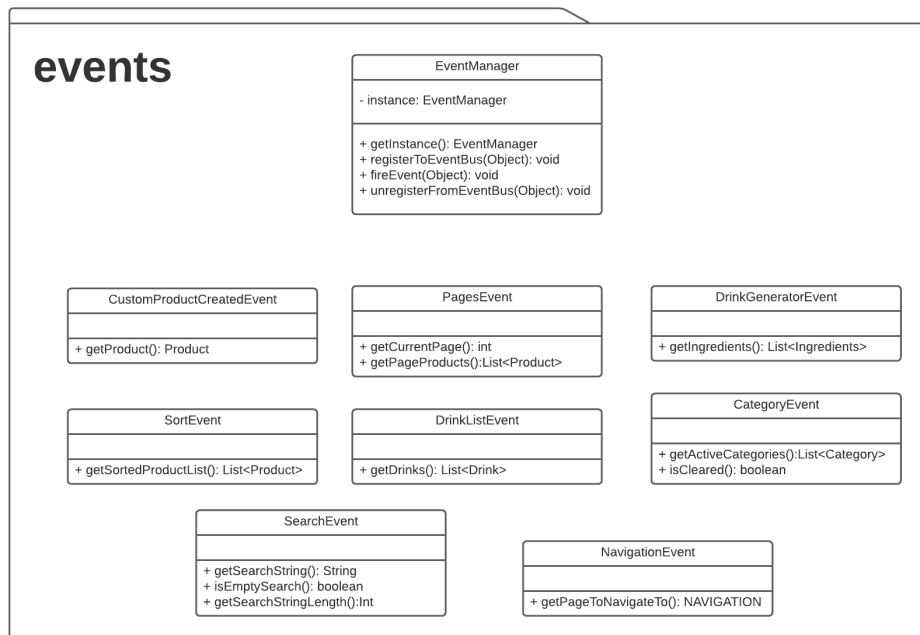


Figure 4: *events* package

The *events* package contains events that get sent to update the view. The observers subscribe to the event bus via the *EventManager* class, a Singleton instance that makes use of the Google Guava *EventBus*. The model then fires the events correspondingly. This makes the model independent and separated from the controllers, reducing coupling. This way, we can use the same model for another application.

### 3.2.3 Products

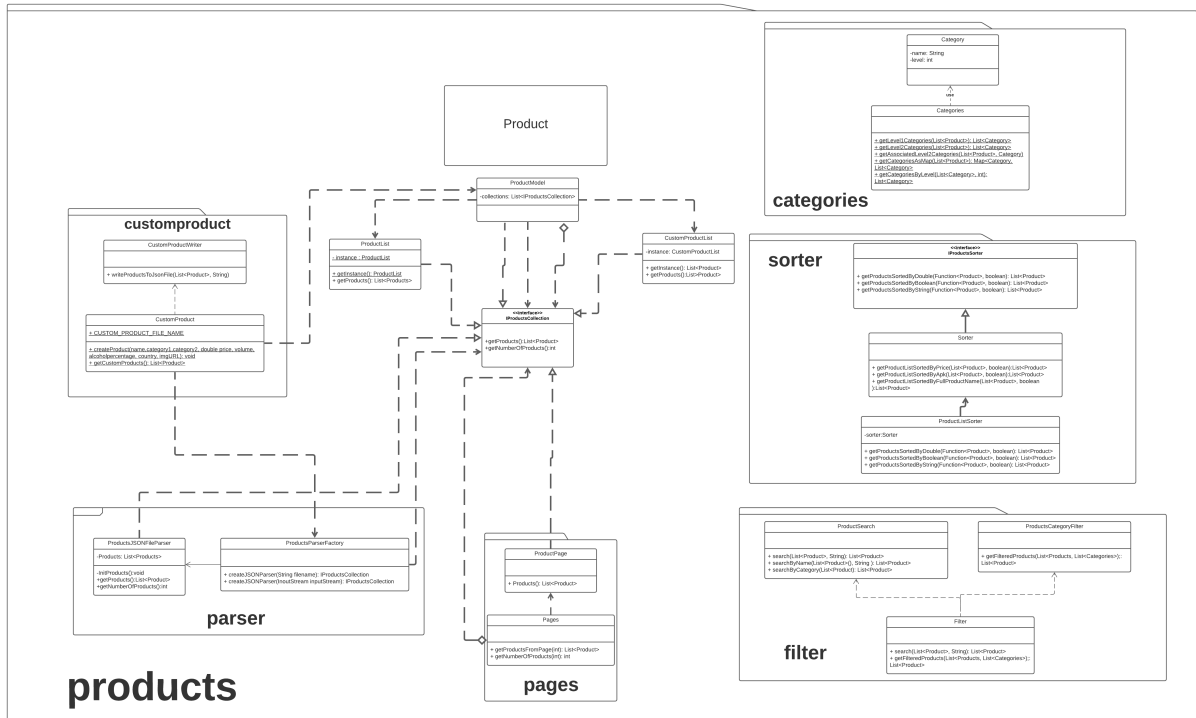


Figure 5: *products* package

The *products* package contains classes and modules that directly relies on the *Product* class. This diagram does not include arrows pointing to *Product*, as all packages depend on this class in one way or another.

The *products* package is a connection between four packages: *parser*, *sorter*, *customproduct* and *categories*.

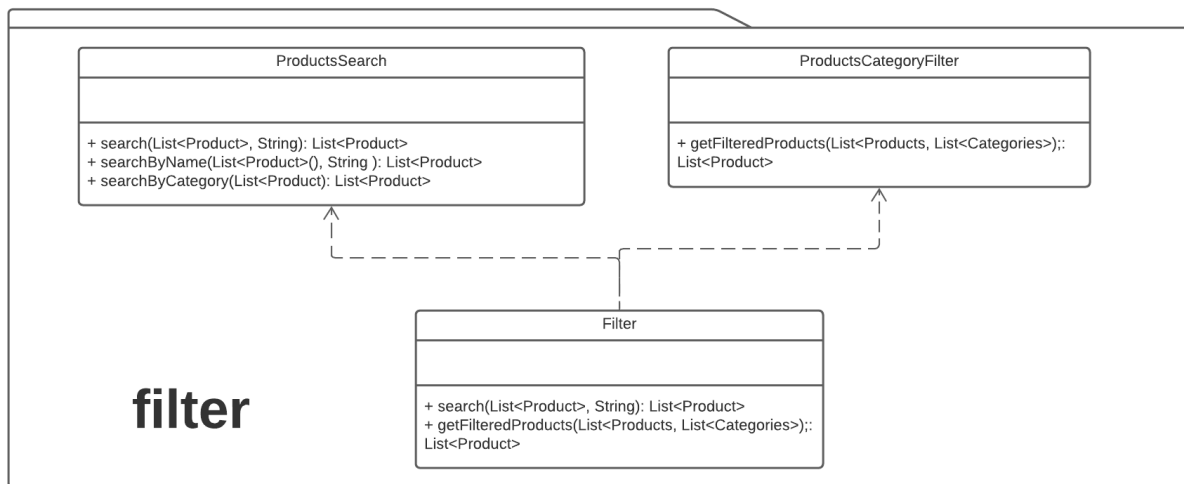


Figure 6: *filter* package

This package filters *Product* objects based on categories and search queries. The *Filter* class is a facade for *ProductsSearch* and *ProductsCategoryFilter* - two classes that are package-private to hide the internal implementation of the filtering functionality.

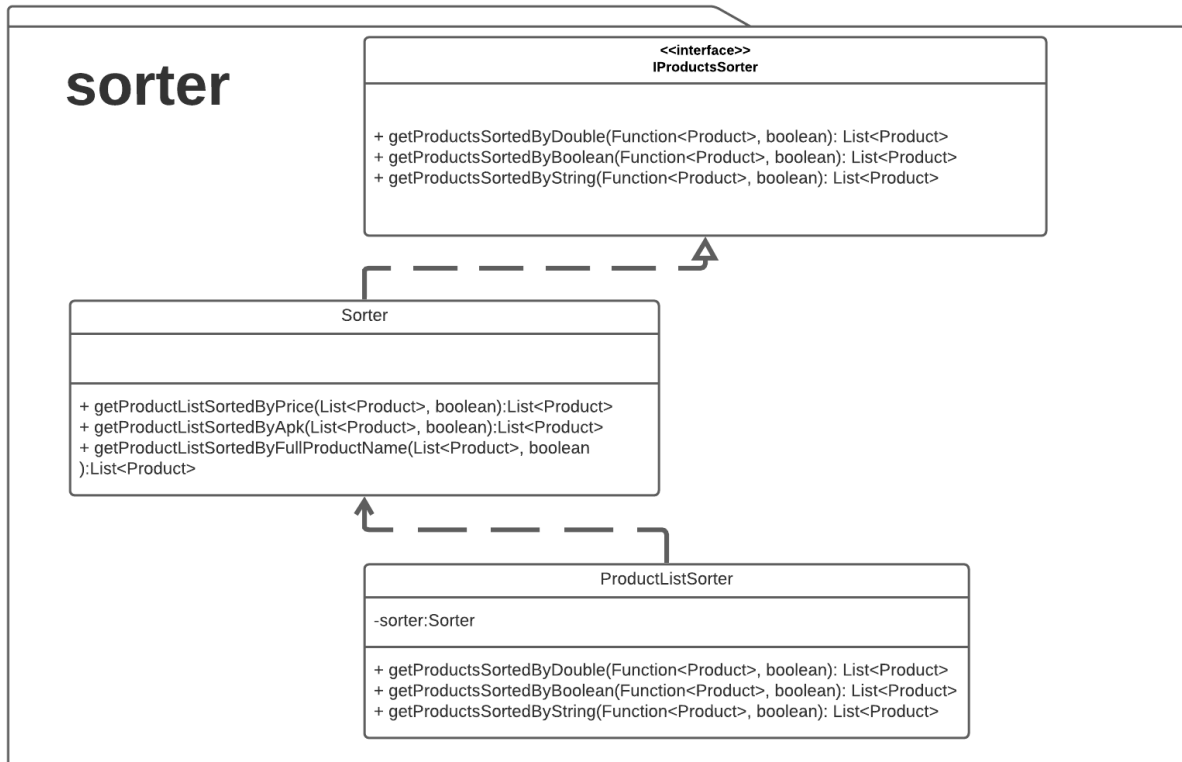


Figure 7: *sorter* package

The *sorter* package responsibility is to sort products based on certain attributes that the *Product* object has. The same concept is utilized in this package as in *filter*: *sorter* is a facade for the internal implementation (which utilizes the Java *BiFunction* classes).

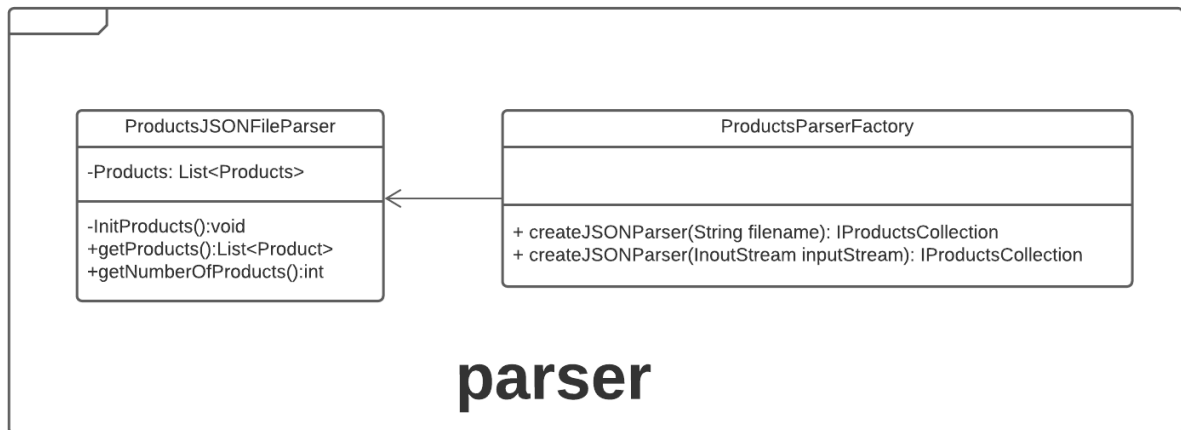


Figure 8: *parser* package

The *parser* package reads data from JSON files and converts it into a *List* of *Product* objects. *ProductsJSONFileParser* implements *IProductsCollection*, and the factory pattern is utilized to create JSON parser objects.

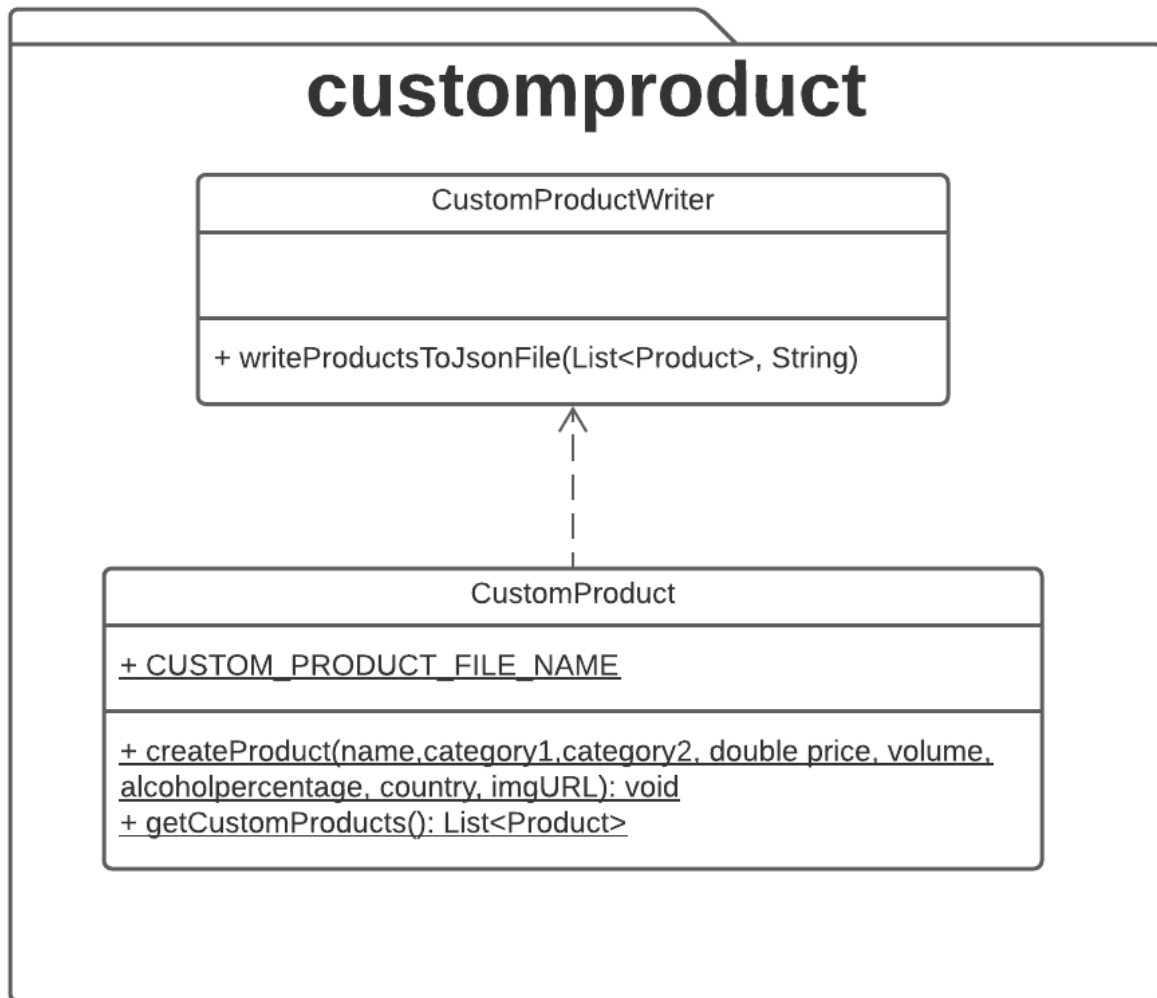
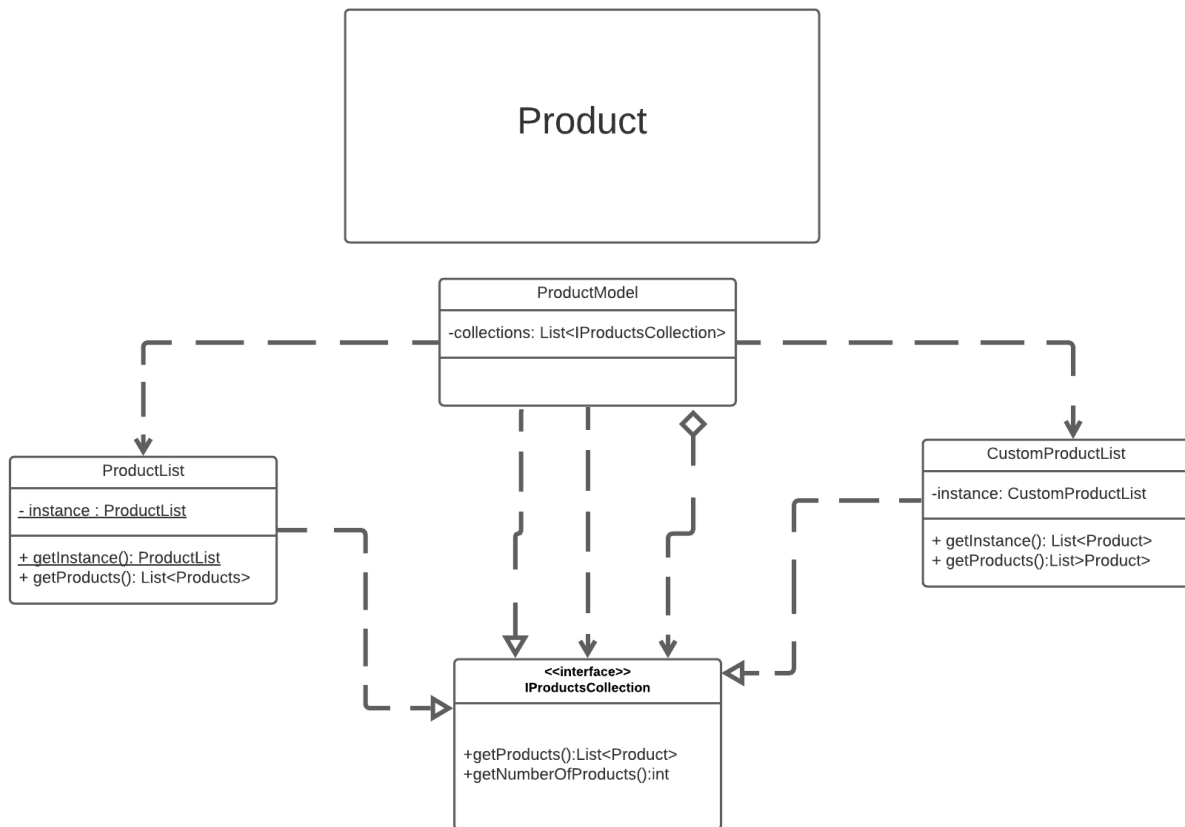


Figure 9: *customproduct* package

The *customproduct* package has the functionality to create new *Product* objects and save them into a JSON file. This opens up the feature to add new *Product* objects to the program. The *customproduct* class is mostly made up of static methods.



## Classes



These classes are responsible for coupling together all the packages. Here we have a interface called *IProductsCollection* which is implemented by *ProductList* and *CustomProductList*. These are then created to separate the already existing *Product* objects and the *Product* objects that are created by the user while maintaining the same functionality for both. The *ProductModel* holds all the *Product* objects that are available outside and in combination with the *sorter* and *filter* packages creates the functionality that is needed.

## Patterns

- Composite pattern - This class 'module' uses a form of Composite pattern. *IProductsCollection* is our component, *ProductModel* is our composite class and the other *Productlist* classes are the 'leaves'. This would make it possible for *ProductModel* to hold several other *ProductLists* and *ProductModels* which in turn hold another layer of *ProductModel* or *ProductLists*. However, in this case, the *ProductModel* class is a singleton and thus breaks the composite pattern.
- Singleton - *ProductModel* makes use of the singleton pattern.

### 3.2.4 Favorites

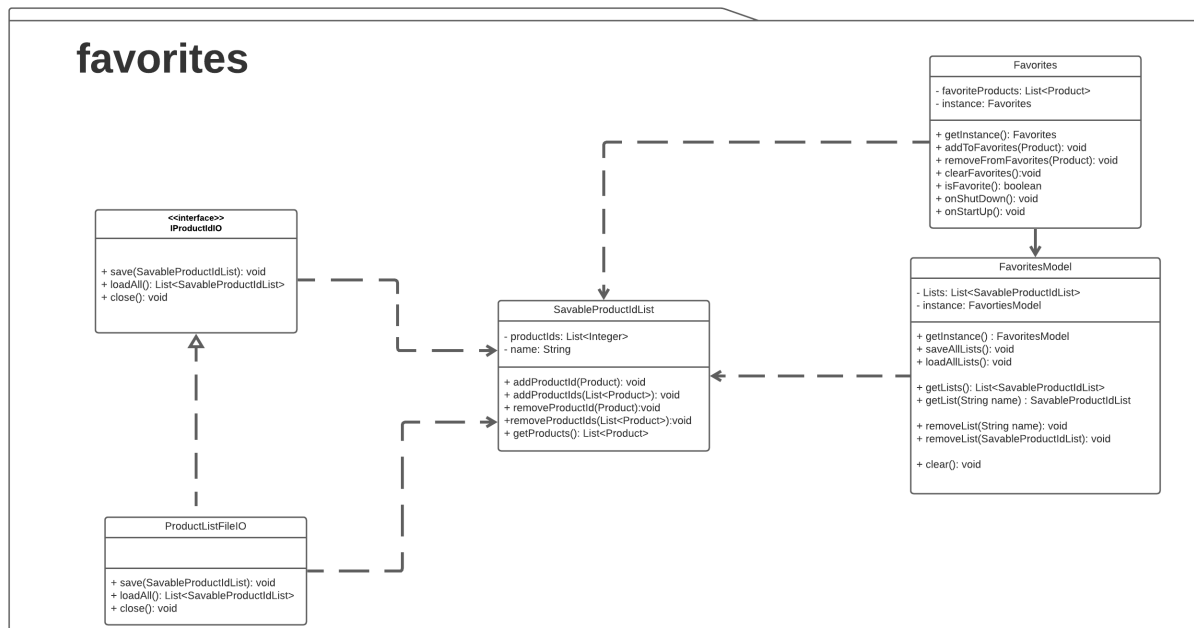


Figure 10: *favorites* package

The *favorites* package is used for saving and loading certain *Product* object ID:s from and to a data source. In this program the *favorites* package saves to a .txt file. These ID:s are for the *Product* objects that has been marked as a favorite. The *favorites* module is made in such a way that it is easy to create and save multiple lists of products ids if it is needed in the future. It is also easy to switch between data sources using the *IProductListIO* interface. As a result, this makes the module modular and extensible.

#### Patterns

- Facade - The *Favorites* class is used as a facade that represents all the functionality of the package

### 3.2.5 Drinks

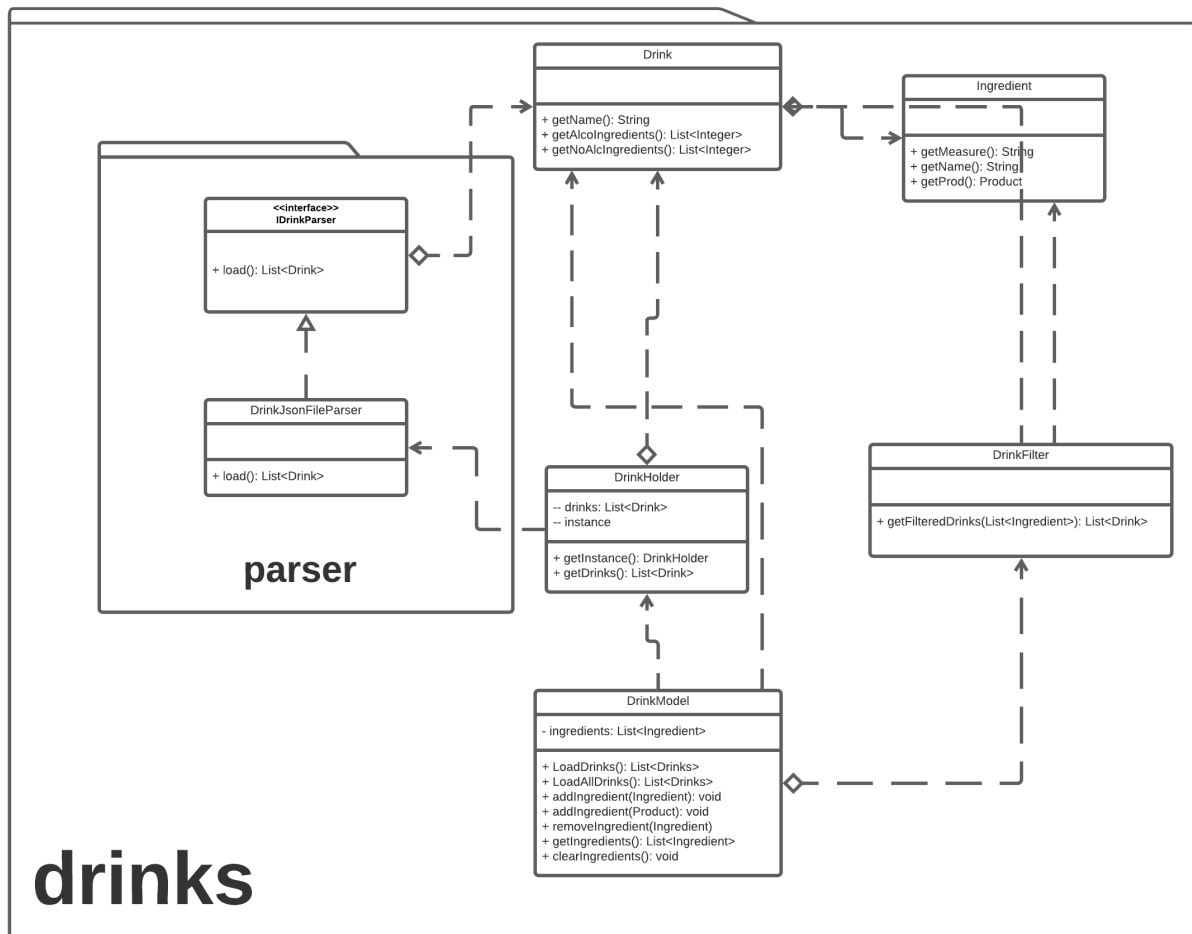


Figure 11: *drinks* package

The *drinks* package is responsible for all the functionality involving *Drink* objects in the program. This package represents one major functionality, Drink generation. The ability to select certain ingredients and from that get a list of drinks that contains these ingredients. We get the information by reading Drink recipes from a JSON file with the *DrinkParser*. The Drink parser then gives them to the *DrinkHolder* that is a common point where these *Drink* objects can be fetched from. *DrinkModel* keeps track of all the *Ingredients* objects that are selected. Drink filter uses these *Ingredients* objects and compares them to the Drink recipes in *DrinkHolder* and gives out a list of matching *Drink* objects.

## Patterns

- Facade - *DrinkModel* is used as a Facade to represent all the functionality of the package
- Wrapper - The *Ingredient* class is used as a wrapper for the *Product* class where the *Product* object is instead used as a *Ingredient* object.
- Singleton - The *ProductHolder* class makes use of the Singleton pattern

### 3.3 Domain model

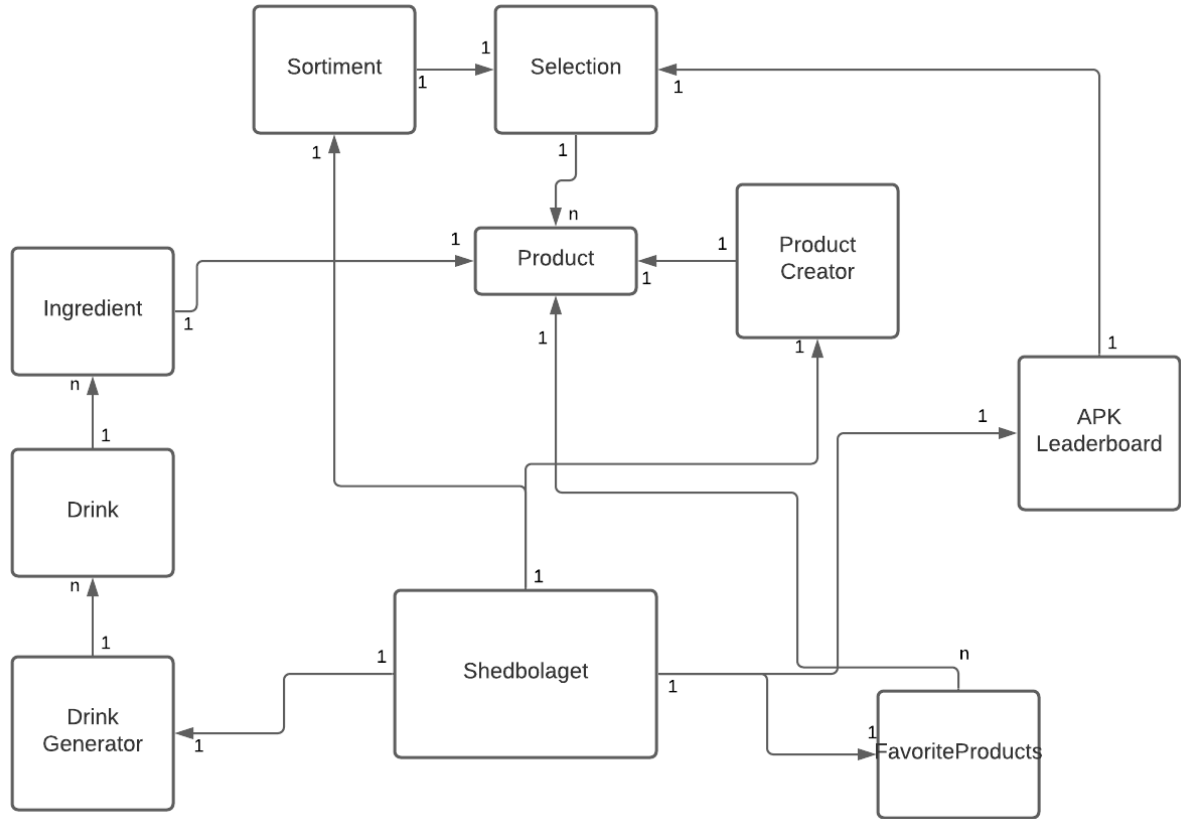


Figure 12: Domain Model

A domain model as seen in figure 12 is a descriptive model of how the application works. It includes all the different components that are in the program. The relation between the domain model and the design model is listed below:

#### 3.3.1 Drink

The *drinks* package and its functionality is represented in the domain model through the Drink, Ingredient and Drink Generator module in the domain model. A *Drink* object has a certain amount of *Ingredient* objects and the Drink Generator is the *DrinkHolder* and *DrinkFilter* class that has the functionality to select certain *Drink* objects based on a certain *Ingredient* objects.

#### 3.3.2 Product

In addition, the *products* package is represented through the Product, Selection, Sortiment and APK Leaderboard modules in the domain model. The Sortiment is equivalent to the *DrinkModel* that holds all the *Product* objects in the program. The Selection module in the domain model is represented through *filter* and *sorter* packages, and separates the *Product* objects into different selections based on certain attributes or specifications. However, one that is not so clear is the APK Leaderboard in the domain model. This

feature did not need its own separate module in the design model since it is possible to sort by APK through the *filter* and *sorter* packages.

### **3.3.3 FavoriteProducts**

Furthermore, the *favorites* package is presented through the FavoriteProducts module in the domain model. The FavoriteProducts feature is a certain specified list of products that the user has marked as favorite.

### **3.3.4 Product Creator**

The *customproduct* package is represented by the Product Creator module in the domain model. It allows the user of the application to create custom products.

## 4 Persistent data management

All static data is stored in a resources directory (in line with the Maven convention) and loaded using Java's 'ClassLoader.GetSystemResources' method. All dynamic data such as custom products and favorite products are stored in the user home directory in a folder called '.shedbolaget' which is managed by the UserDataManager class. This class makes sure that the user data directory exists, and provides a method to return the path to the directory.

The application serializes and de-serializes its JSON data with the help of the external library Jackson.

## 5 Quality

In order to maintain quality control of the application, a number of tools is utilized and described in detail below:

### 5.1 Build automation

Maven is used to build and maintain dependencies of the application.

### 5.2 Tests

JUnit is the unit testing framework used in the application. The goal is to have as high test coverage as possible in the model (controllers are not tested). This is measured by JaCoCo, a code coverage library that is integrated in the build automation. However, some classes were deemed as superfluous to test (such as the data class `Product` as well as events) as they do not contain any logic suitable for tests. The tests follows the Maven convention and can be found in `src/test/java/shedbolaget`. Below is a screenshot of the JaCoCo coverage results:

#### shedbolaget

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
shedbolaget.controllers.components		0%		0%	121	121	346	346	88	88	14	14
shedbolaget.controllers.components.DrinkGenerator		0%		0%	48	48	136	136	35	35	7	7
shedbolaget.model.events		10%		n/a	23	27	49	55	23	27	9	11
shedbolaget.model.products		69%		70%	20	56	34	99	15	45	2	6
shedbolaget.controllers		0%		0%	14	14	34	34	8	8	1	1
shedbolaget.controllers.components.customproducts		0%		n/a	9	9	28	28	9	9	2	2
shedbolaget.model.favorites		89%		87%	11	79	23	185	2	43	0	4
shedbolaget.model.drinks		83%		66%	13	43	14	74	9	34	0	5
shedbolaget		0%		n/a	5	5	11	11	5	5	2	2
shedbolaget.model.products.sorter		64%		33%	5	13	11	26	3	10	0	2
shedbolaget.model.products.filter		96%		81%	7	35	1	62	2	19	0	3
shedbolaget.model.products.customproduct		92%		n/a	0	8	4	26	0	8	0	2
shedbolaget.model.products.parser		90%		n/a	0	8	3	16	0	8	0	3
shedbolaget.model.drinks.parser		83%		n/a	0	3	3	9	0	3	0	2
shedbolaget.model		90%		100%	0	6	2	17	0	5	0	1
shedbolaget.model.products.pages		97%		100%	1	18	0	27	1	13	0	2
shedbolaget.model.products.categories		98%		92%	2	25	1	35	0	12	0	2
Total	2,858 of 5,074	43%	129 of 282	54%	279	518	700	1,186	200	372	37	69

Figure 13: JaCoco coverage results

With 68 tests in total, and a total line coverage of 75 % in the *model* package (which also contains the superfluous classes mentioned above) the application has tests for all relevant classes of the application.

### 5.3 Continuous integration

Travis CI is integrated into the GitHub repository and ensures that all tests pass and that the project successfully builds. This is used in the project workflow to verify that all tests passes before a branch is merged to main. The URL to the CI tests can be found at <https://app.travis-ci.com/github/emilsvennesson/shed>.

## 5.4 Analytical tools

### 5.4.1 PMD

PMD is used to monitor and analyze possible violations in the codebase. Below are screenshots of the CPD (copy/paste detector) and PMD results:

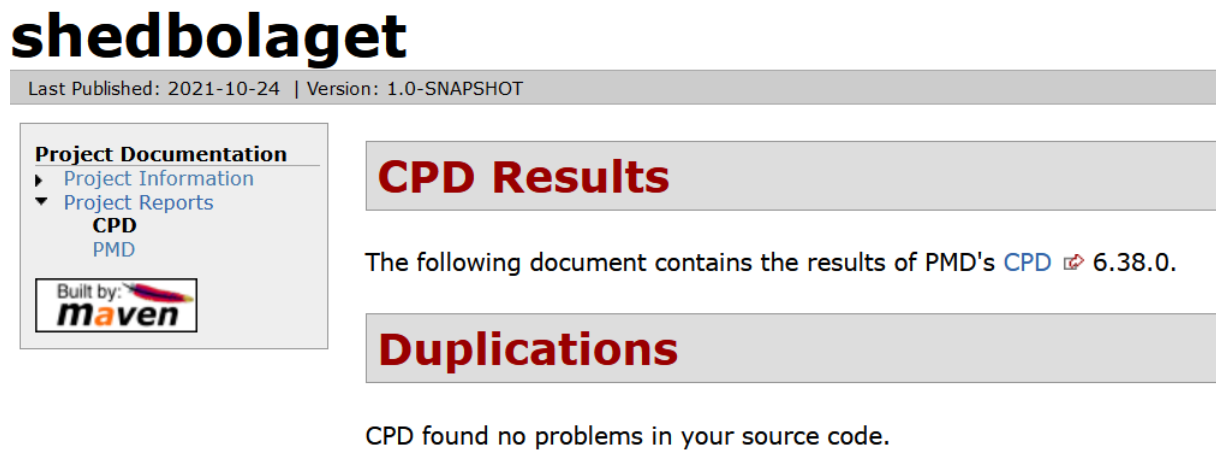


Figure 14: CPD source code results



Figure 15: PMD violations results

As seen in Figure 16, the codebase has one violation that we did not have the time to address.

### 5.4.2 Dependency matrix

Below is a screenshot of a dependency matrix generated by IntelliJ:



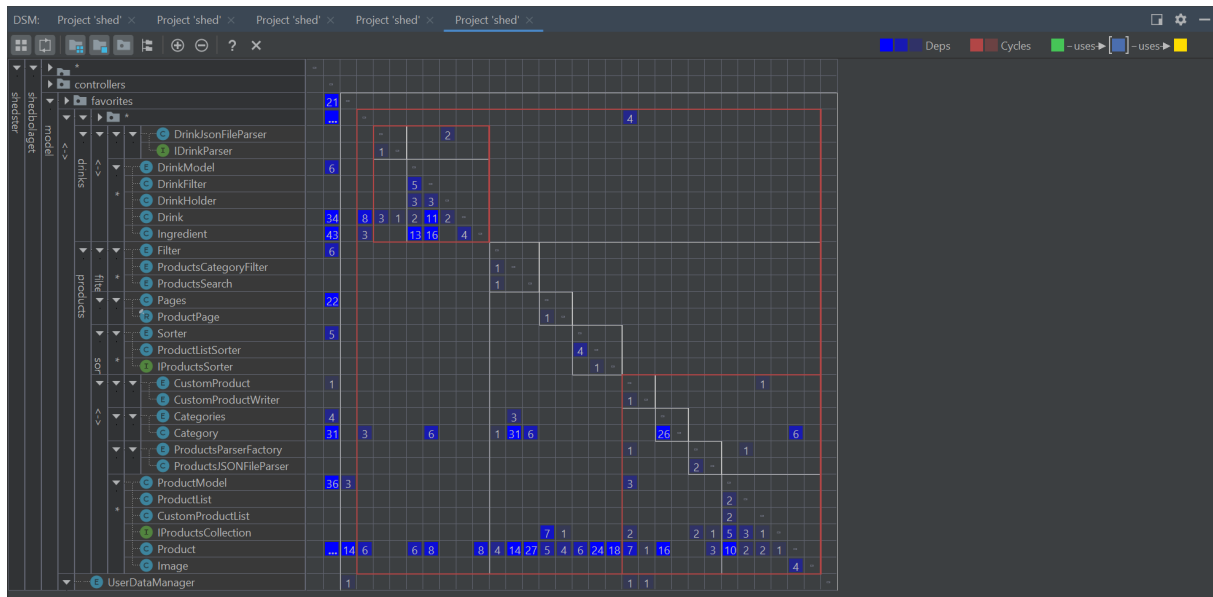


Figure 16: IntelliJ dependency matrix

## 5.5 Known issues

Below is a list of known issues that would have been addressed with more time:

- We are currently firing events in some of our controllers. Ideally, this should happen in the facade models that gets called by the controller.
- *favorites*, and *drinks* packages should be moved to the *products* package since they are directly related and dependent on the *Product* class.
- We should split up our components into its own packages with its own factory.
- Our *events* package needs a refactor and better thought out names.
- 'Nya produkter' in the main page does not actually show new products, but instead just lists the 20 first *Product* objects in the *Product* list.
- It is not possible to search for *Product* objects by using the enter key. You have to press the button to perform a search.
- The product categories in the navigation bar are displayed in random order.
- *Product* objects do not have a default image when there is no image available on the Systembolaget website.
- Sorting by name is broken. Currently it sorts by ASCII and not alphabetically.
- The *parser* package could be made more generic so it can be used for *products* as well as for *drinks*.
- boundary checks are needed everywhere in the frontend.

## 6 References

- Systembolaget
- JUnit
- FuzzySearch
- JavaFX
- Maven
- Google Guava
- PMD
- SceneBuilder
- IntelliJ
- GitHub

# Peer Review for Group OOPSIE

Samuel Kajava, Daniel Rygaard, Pouya Shirin, Emil Svensson

October 2021

## 1 Design - patterns and principles

### 1.1 Reusability

The code seems to be reusable in many aspects. A good example of this is the way the frontend has been implemented: the UI elements are split up to its own components, making it possible to reuse the same controller/UI element on different pages within the UI. The classes `ViewComponent` and `JavaFXViewInitializer` is a good boilerplate and could be used for other projects that wants to use components together with JavaFX. The same could be said for the class `EventList`, which is a good start for other projects that needs an event handler.

It could perhaps be possible to improve code reusability by making some of the classes/methods more generic (`FileHandler` for example). That would allow the codebase to reuse parts of the codebase for other types of file handling than attachments and images.

### 1.2 Maintainability

Maintainable code is code that is easy to modify or extend. In order to follow this they are using a significant amount of interfaces. For example, the interfaces `IDatabaseLoader`, `IDatabaseSaver` or `IFileHandler` are utilised in such a way that they make for a interchangeable save and load system. Therefore, if the data is transferred to another sort of medium, the program can be adapted with not much effort. However, there could be a support for fetching from multiple different databases.

### 1.3 Modularity

To judge if a code is modular or not, we have to define what modular programming is. A quick search on wikipedia tells us: "Modular programming is a software design technique that emphasizes separating the functionality of a program into independent, interchangeable modules, such that each contains everything necessary to execute only one aspect of the desired functionality".

Taking the model package as an example, we can see that the model classes are separated in their own area of responsibility. User appears to be the main

model, carrying references to the lower level modules. The lower level modules contains the logic and functionality. Then two separate packages, FileHandler and Database, being accessed with their respective Factory classes.

Which results in the project appears to be following modular programming quite well. However a suggestion for improvement would probably include separating each of these modules in their own packages.

## **1.4 Extensibility**

In regards to the Open-closed principle, modules should be open for extension and closed for modification in order to make it easier to extend the project in general. In the way you are using interfaces, I would say you are following this principle quite well. Let's look at IPageNavigator as an example, the interface is small, which is good, and if you wanted to extend the functionality and make another, more specific navigation tool, you could simply implement another interface (that implements IPageNavigator) with the extended functionality.

Another good implementation is the use of the abstract class ViewComponent and its factory. By simply extending the ViewComponent you can create new components without much hassle, which in my opinion is a good solution.

## **1.5 Design patterns**

To achieve the above-mentioned design principles, a number of design patterns have been utilised. The observer pattern has been used to properly implement the MVC structure and isolate the model from the views. The factory pattern is used in multiple packages (which helps out achieving the Open-Closed Principle).

# **2 Code quality**

## **2.1 Naming**

Classes and method names appears to be following the correct naming conventions well. Just by taking a quick glance at the class names gives you a general idea of the class' content, its function and area of responsibility. This results in an easier navigation through the project and the understanding of class interactions.

## **2.2 Documentation**

The code is very well documented with JavaDoc. The IFileHandler-interface is a good example, as I am not familiar with handling files. Despite this unfamiliarity, I am able to grasp what it means, and this is thanks to your documentation. The same can be said for JavaFXViewInitializer, I understand the code and agree with what you have written in the documentation, well done.

## 2.3 Tests

The test structure is very similar to the main structure, containing test classes for the Model and File Handler package. There's no test for the Database package. Code is well tested.

## 2.4 Understandability

The project has an easy identifiable MVC structure which you'll see by looking at the project files. The model package is split up in different model classes and the controllers each have a reference to them.

## 2.5 Improvements

A few things to consider, or maybe discuss:

- Is there any reason to not singleton most of the classes? For example, the Database class; will you be having more than one database? Right now, you can create a Database anywhere since the class is public. (Perhaps you want another Database if you want to load from two data types simultaneously, but that would mean loading two Users? And the Database Factory is made to only utilize one database)
- Same applies to the JSONDatabase classes. But will you ever need more than one JSONDatabaseLoader?
- You seem to be returning whole mutable objects in User (Eventlist, Contactlist and Taghandler). Normally I would be wondering if this was a case of breaking Law of Demeter, but it seems unavoidable?
- One thing I noticed though was the returning of lists. An example is in ContactList. The ContactList.java is a wrapper for contactList, however what use is a wrapper (except for the events) other than hiding the actual list object? Right now getList() is returning the whole list object (which we learnt in OOP is big no no). However the ContactList wrapper gives you the methods to mutate it anyways, so really what is the difference? One advantage that comes to mind is: preventing alias problem. You want to minimize the amount of unexpected issues in your program. Someone modifies the list outside of your wrapper and now notifyObservers won't be called.