System design document for Shedbolaget

Authors: Emil Svensson, Pouya Shirin and Daniel Rygaard

TDA367

Chalmers tekniska högskola

2021

Version 4.0

Contents

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 liqour store in Sweden.
- **Shedbolaget** - the project/application name.
- **APK** - (alkohol 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
- When searching with an empty search query in the Drink Generator it shows an empty *Ingredient* object list.

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

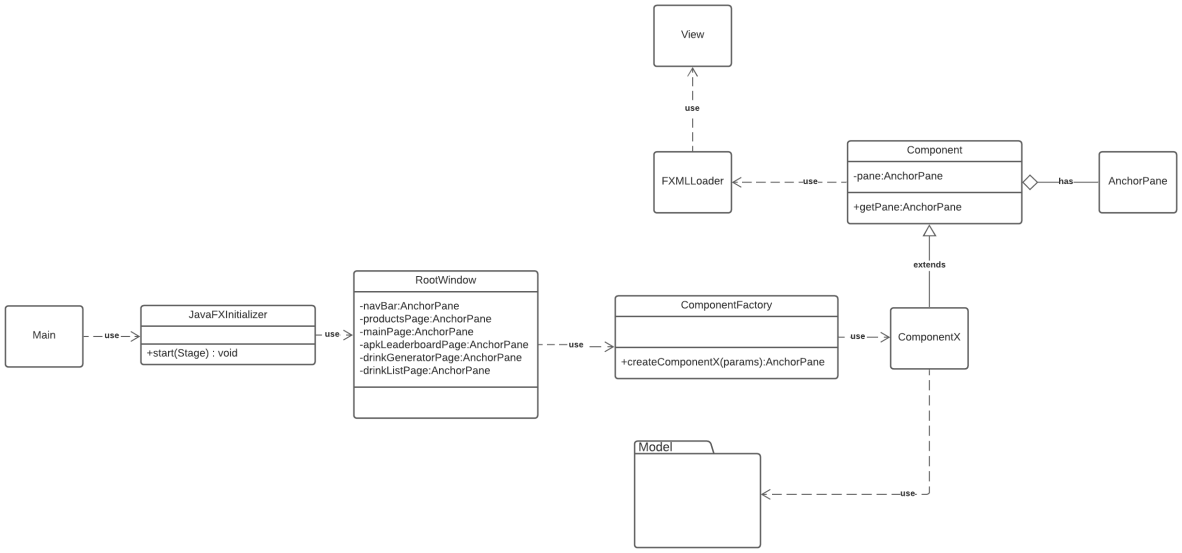


Figure 1: Overview of the MVC design pattern

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

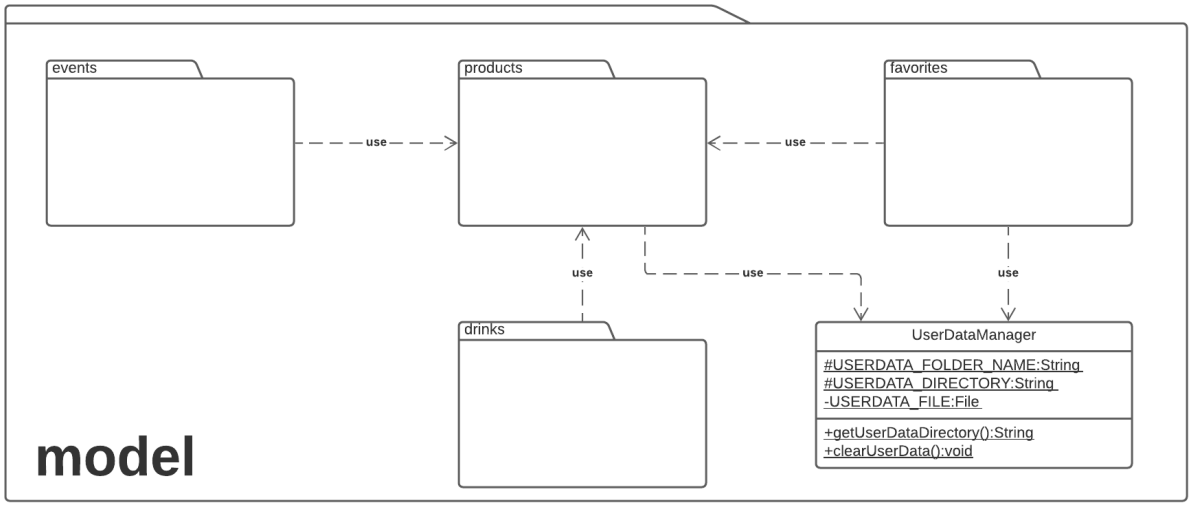


Figure 2: *model* package

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

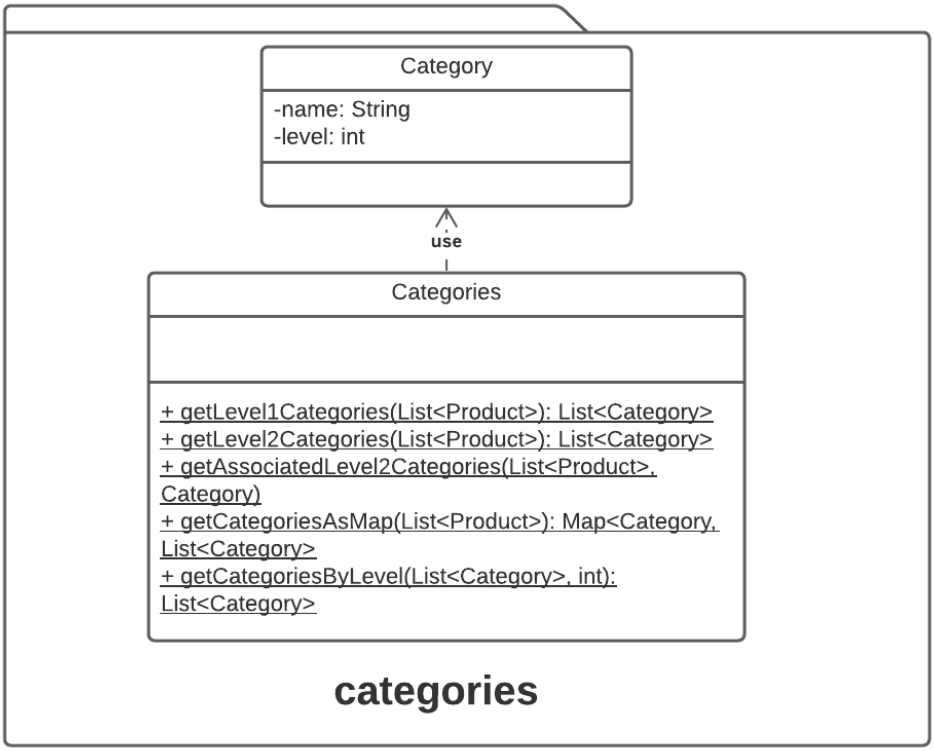


Figure 3: *categories* package

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

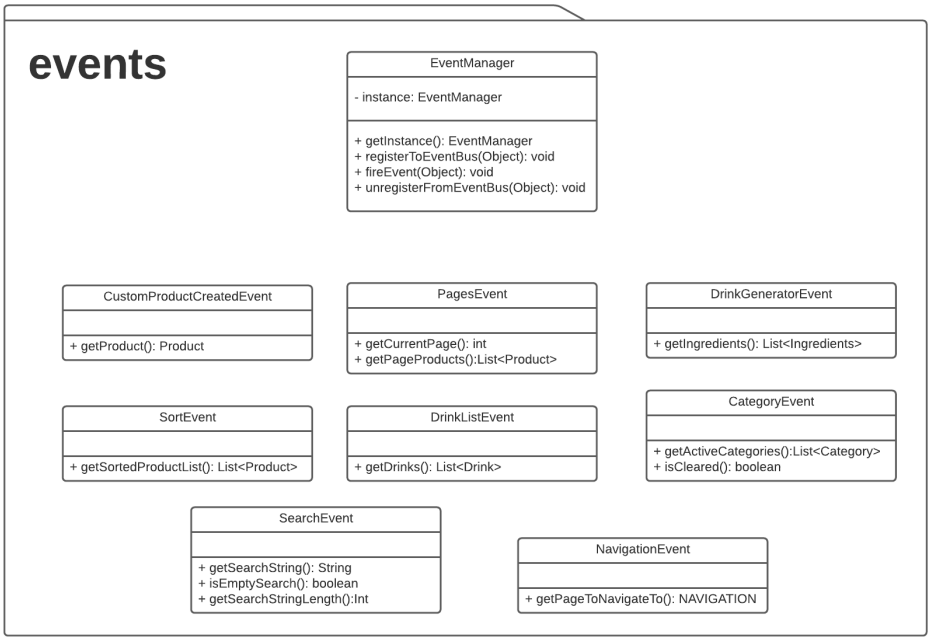


Figure 4: *events* package

The *events* package contains events that gets sent to update the view. The observers subscribes 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

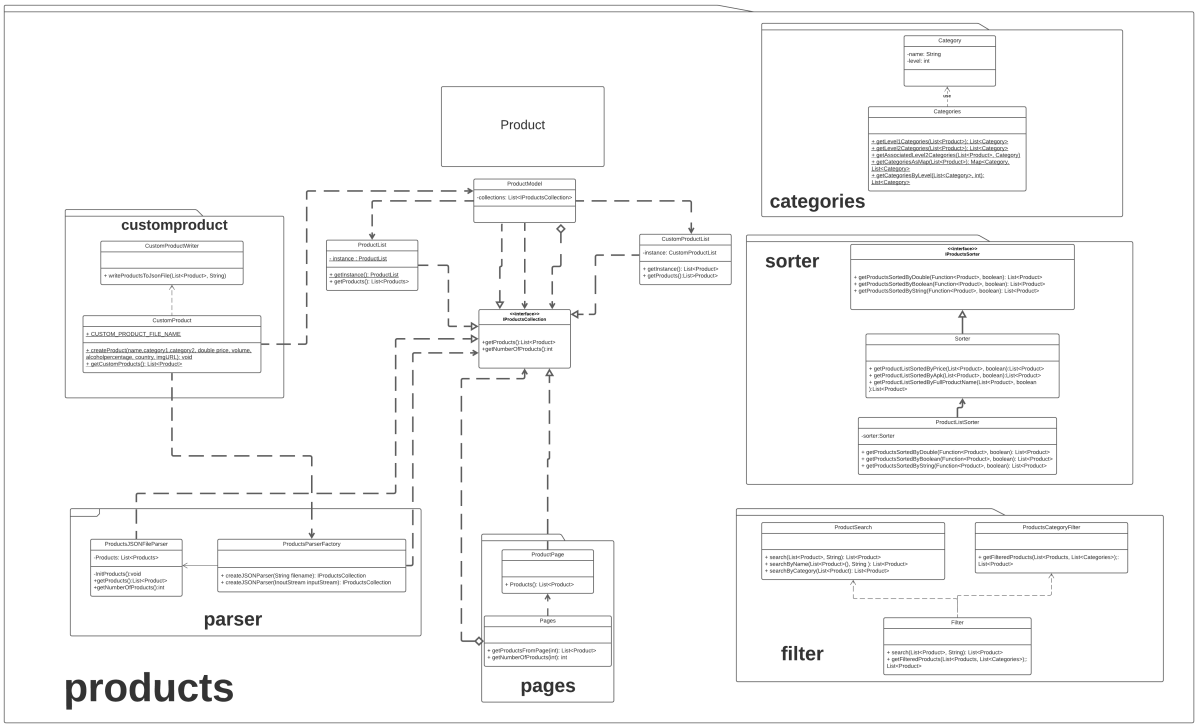


Figure 5: *products* package

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*.

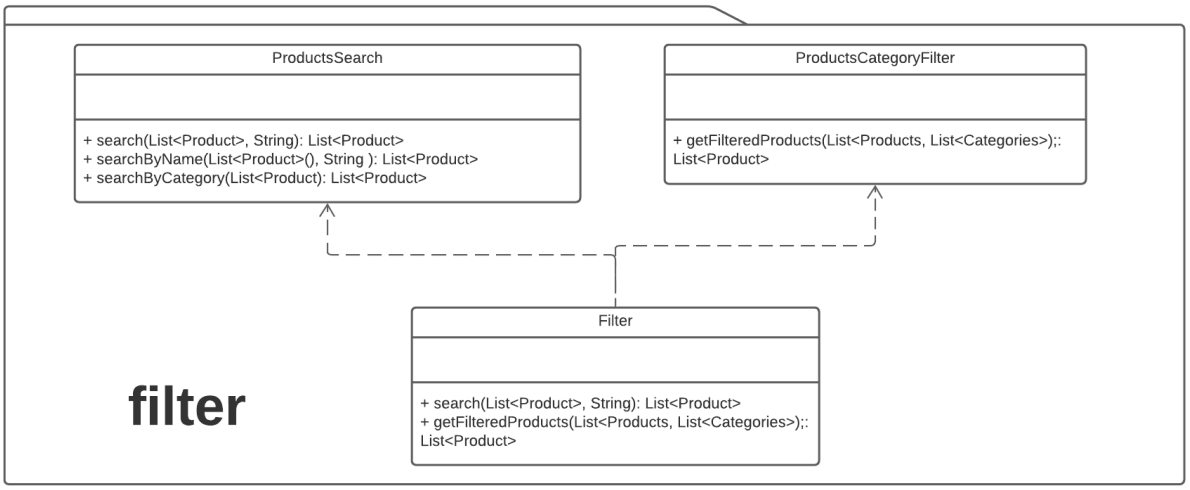


Figure 6: *filter* package

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.

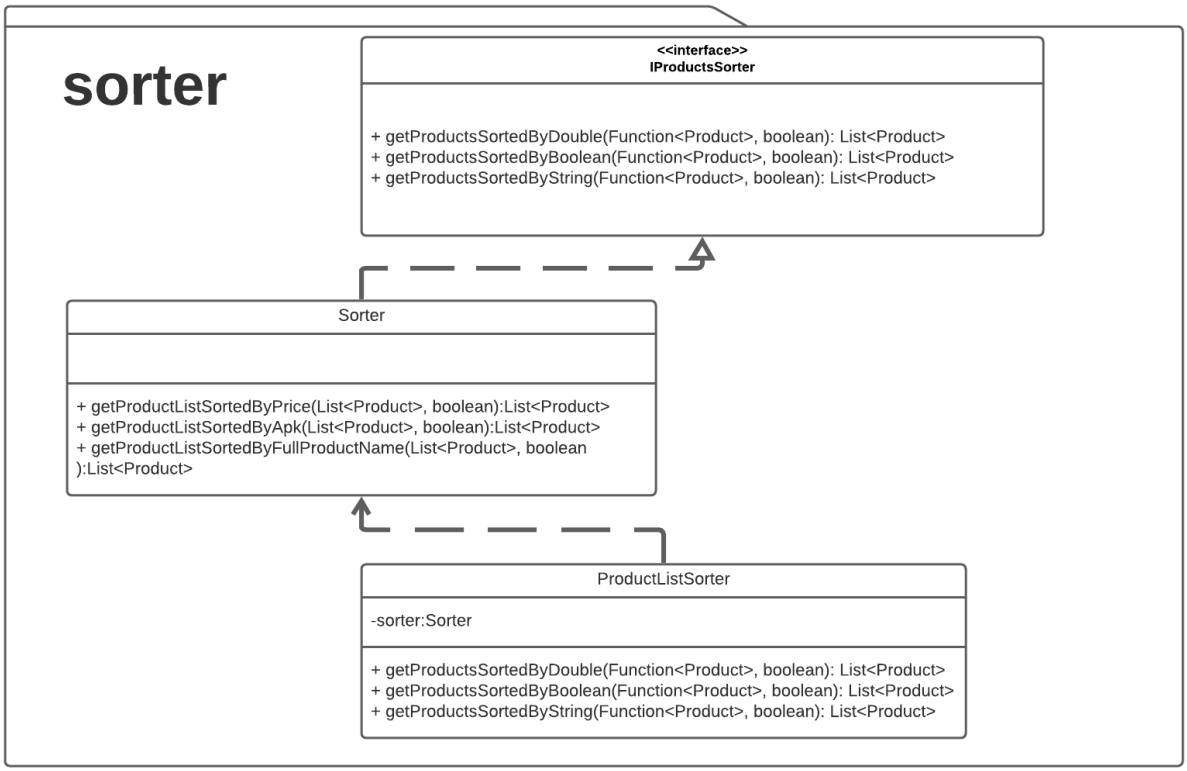


Figure 7: *sorter* package

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).

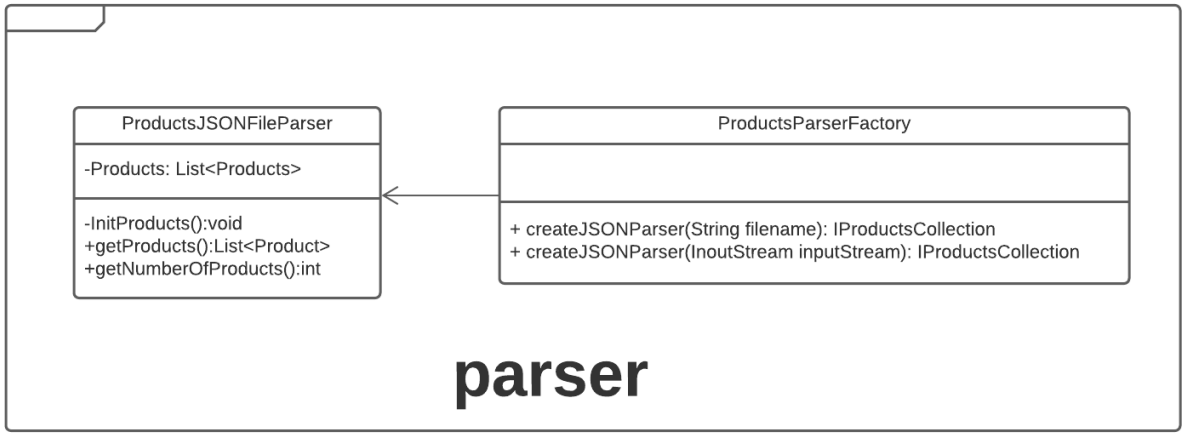


Figure 8: *parser* package

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.

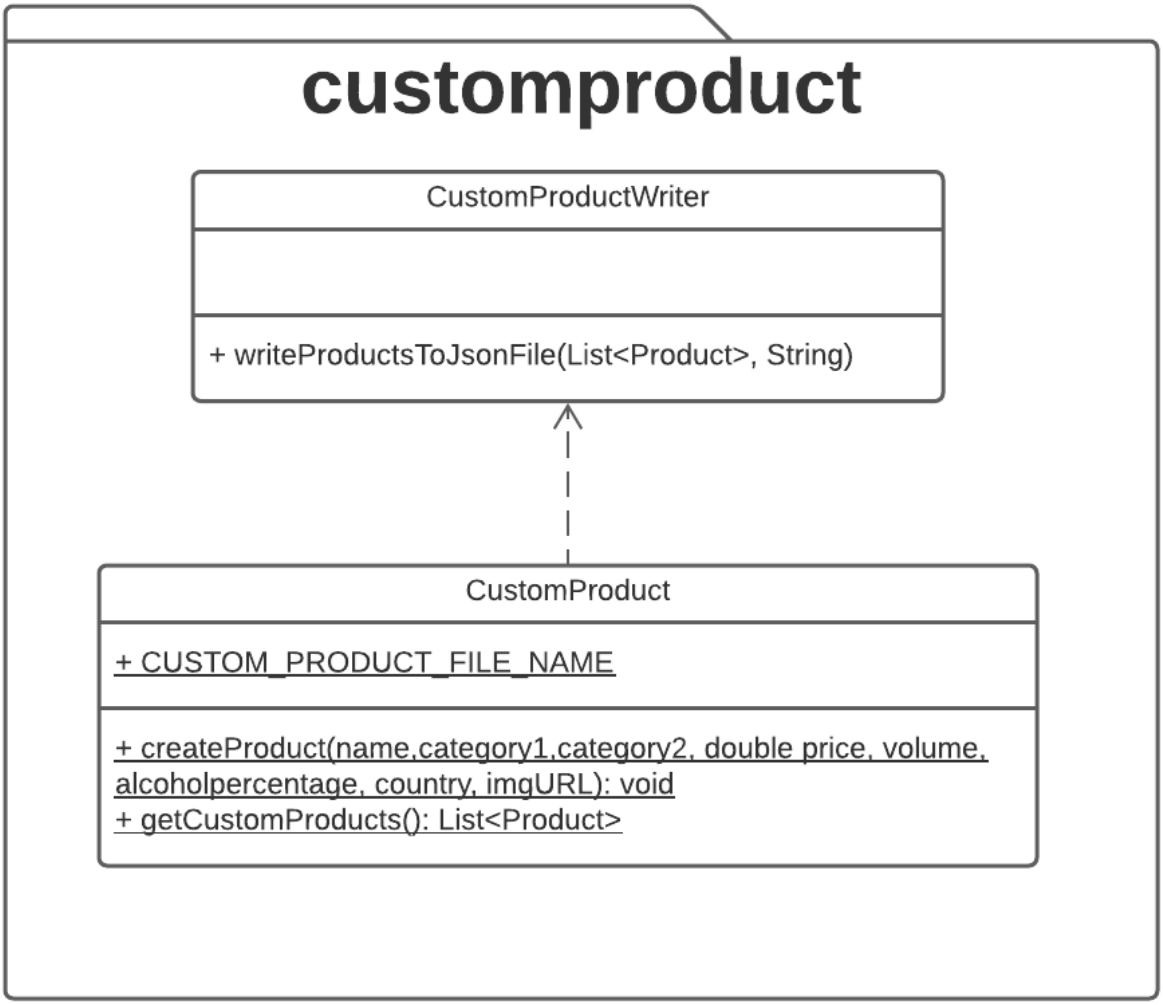
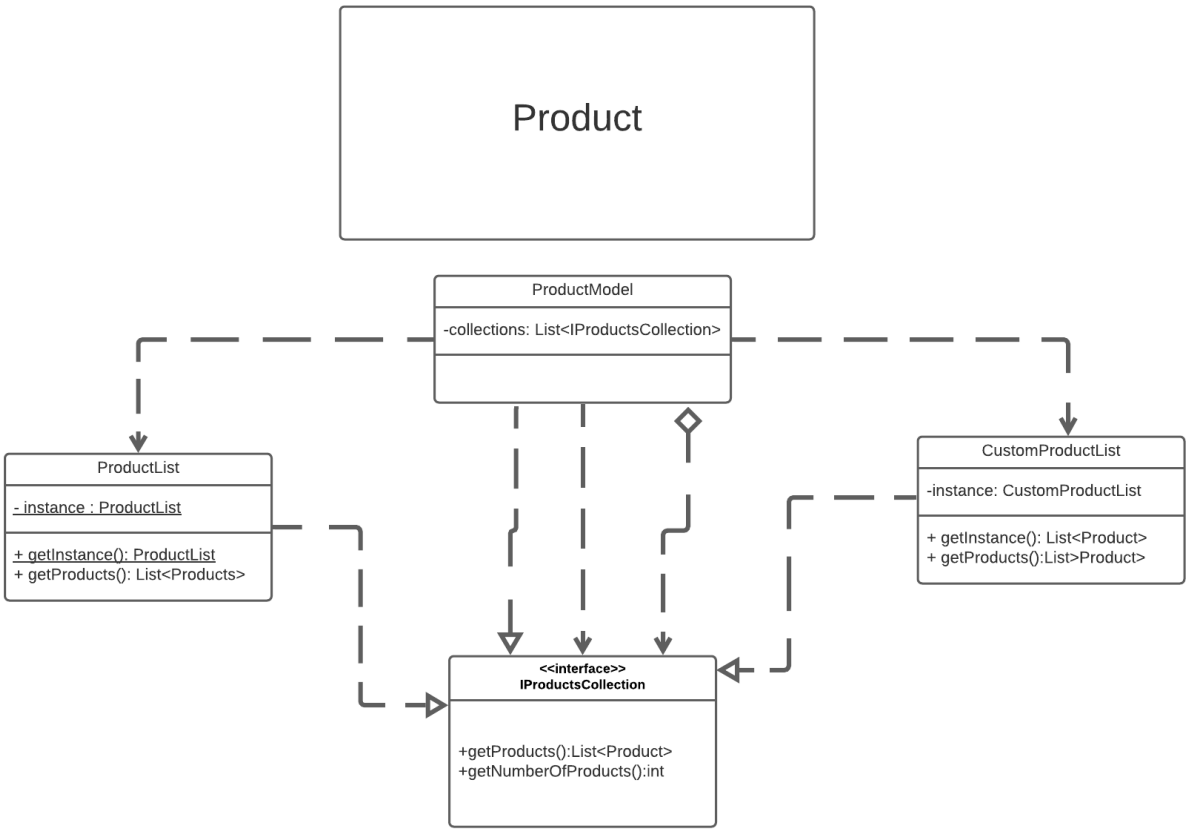


Figure 9: *customproduct* package

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 *ProductList*s. 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

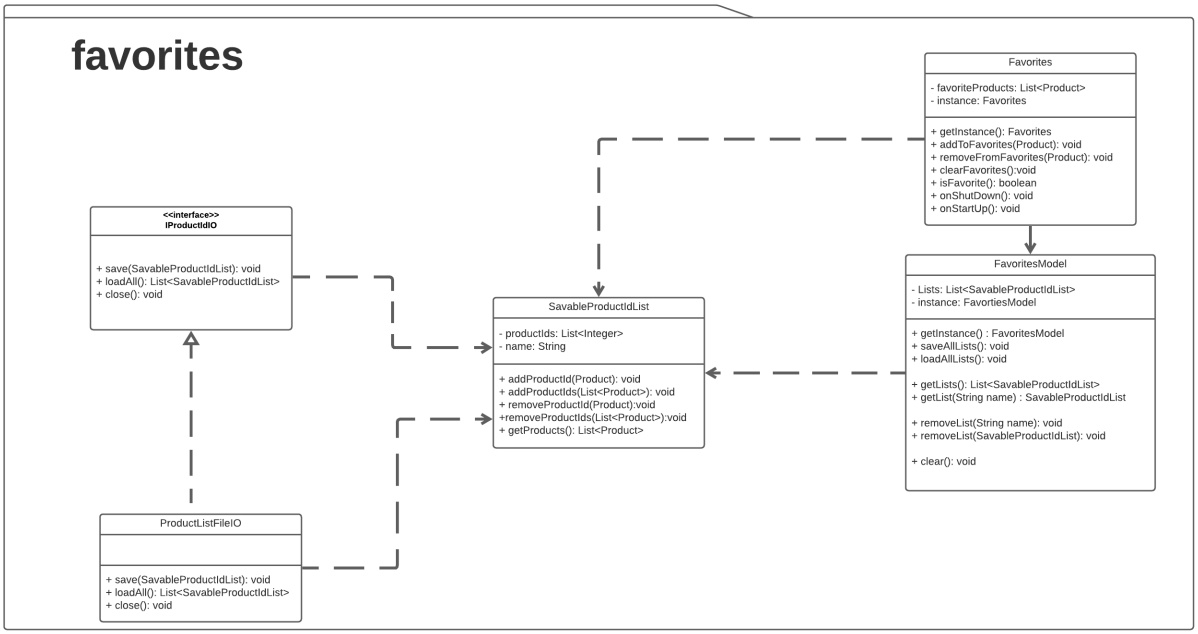


Figure 10: *favorites* package

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

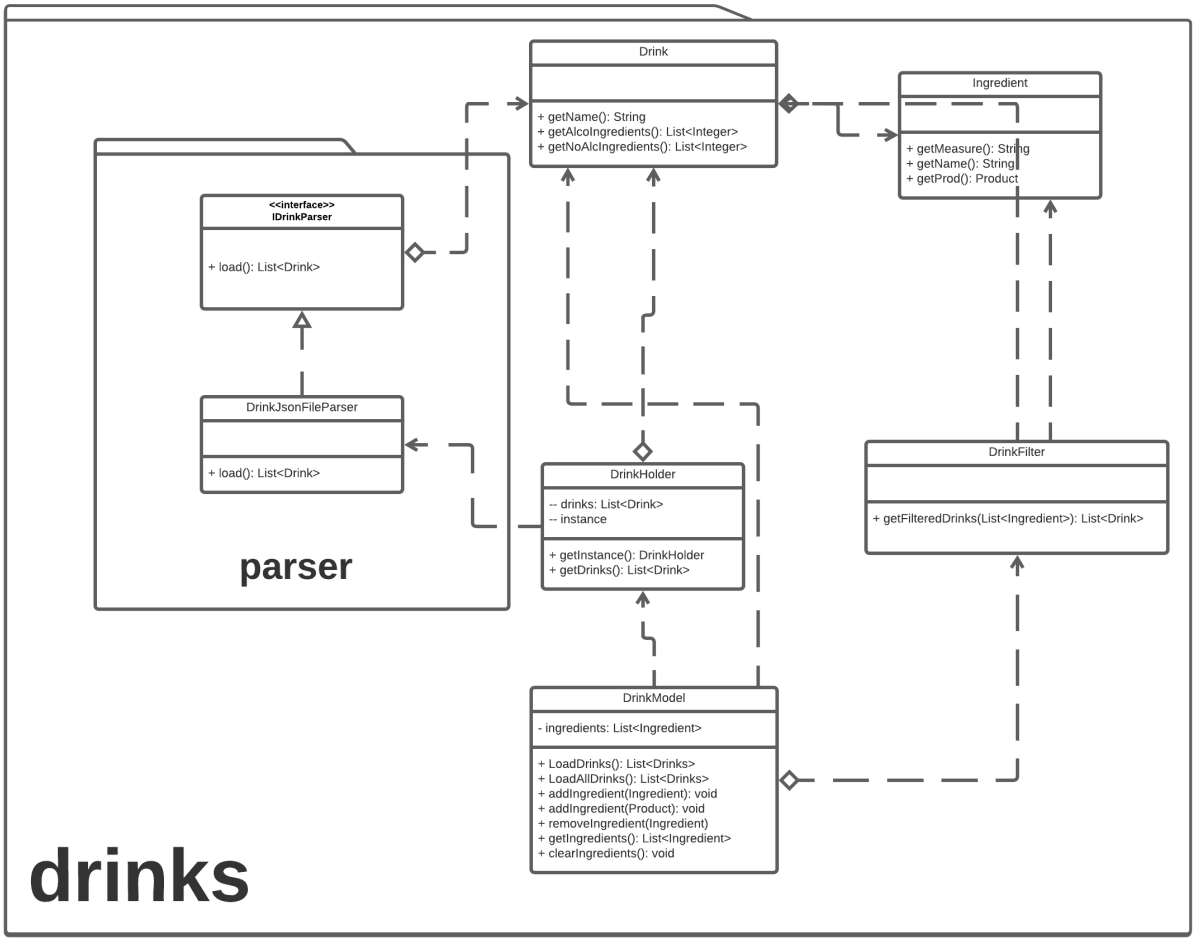


Figure 11: *drinks* package

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

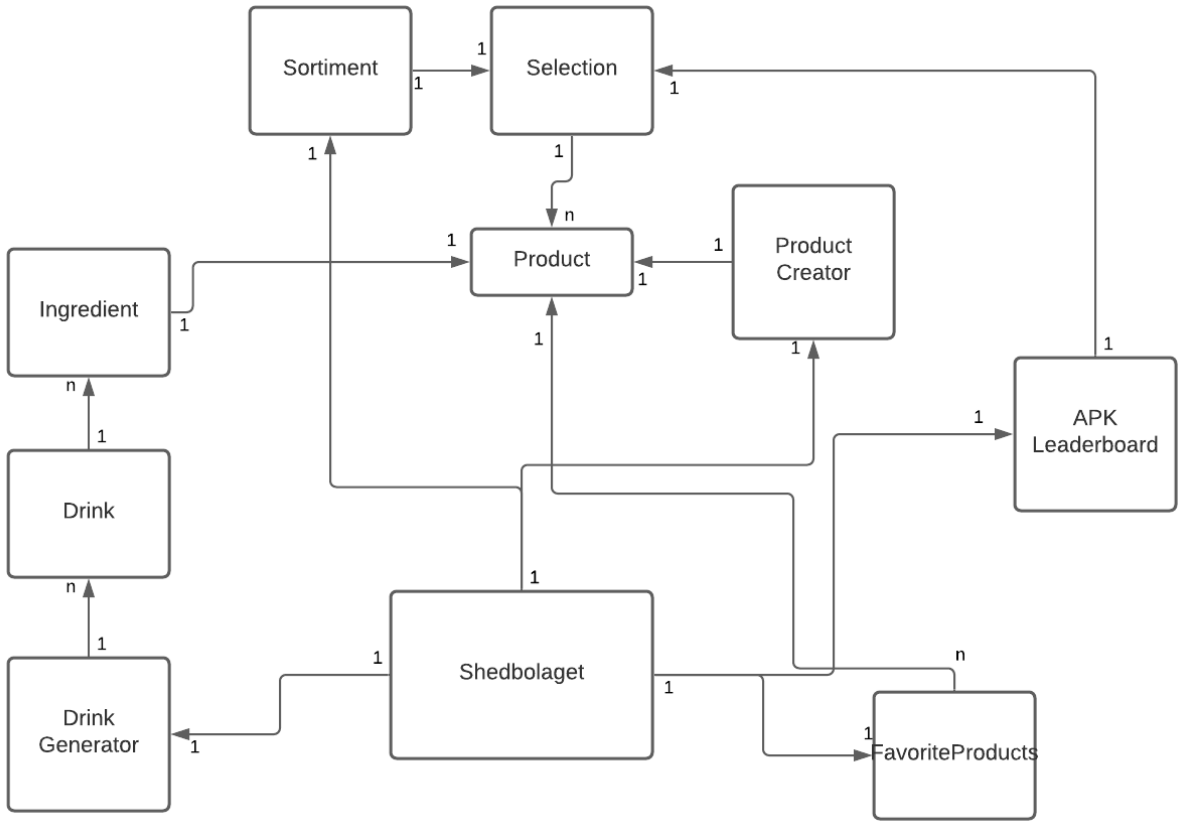


Figure 12: Domain Model

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 trough *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 resulsts:

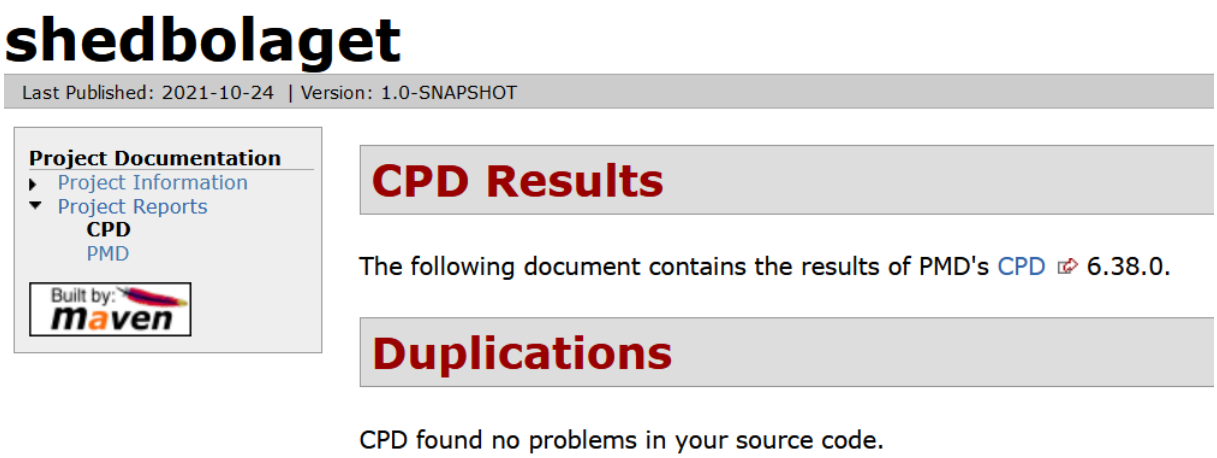


Figure 14: CPD source code results

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:

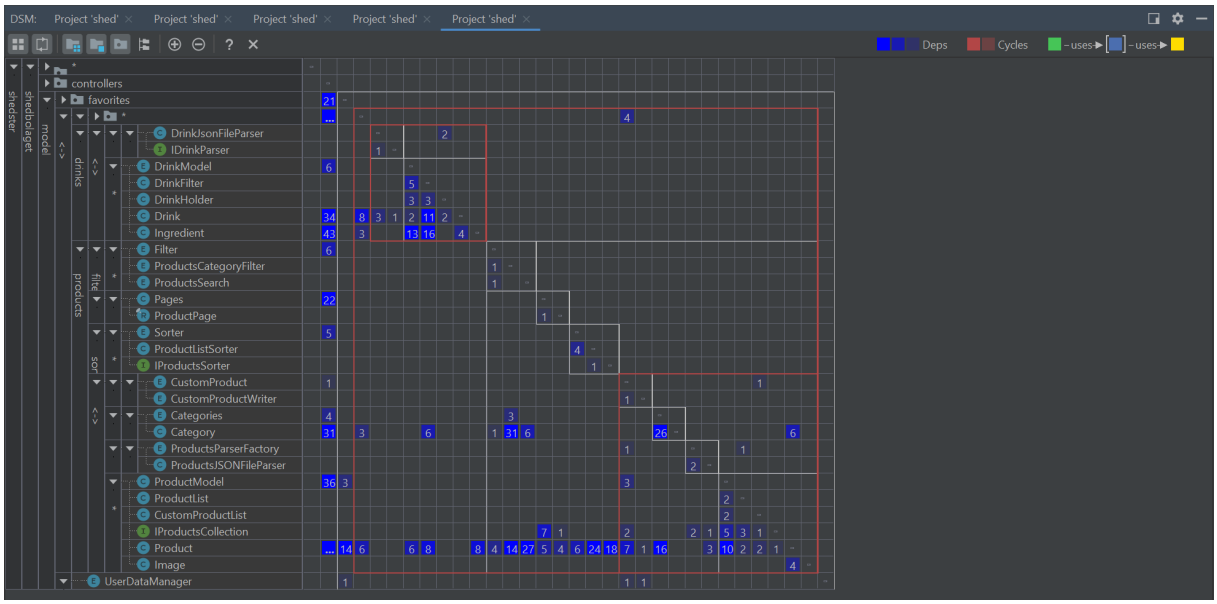


Figure 16: IntelliJ dependency matrix

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