Object Oriented Design

Checkout App

Adomavicius Tomas
Backx Rafael
Vlaeyen Thijs

2019-2020 2TI-2SO

18/12/2019

University Colleges Leuven-Limburg
Campus Proximus
Geldenaaksebaan 335
3001 Heverlee
Tel. +32 (0) 16 375 700

General remark

This report is about the assignment 'Chackout App', which will count for 5 points of your total score of this course. This report is the explanation of your code.

You are expected to fully follow this template for the report. Apart from this 'general remark', all paragraphs are mandatory parts of the report. You print this report (double-sided) during the retake 000

You create a zip file of all your source code (.java files, not .class files) and of all files relevant to this OOO command. You also add the latest version of your report (Word document) to this zip file.

You upload this zip file via Toledo no later than Monday 23 December 2019 - 23.59 hrs. Name of the zip file: sequence no_family name1_family name2_family name3_Kassa_OOO2019. You get the sequence number from the lecturer.

It goes without saying that the code on Toledo matches the code in your repository on GitHub.

URL GITHUB repository

Copy/paste here the URL of your Github repository with your self-evaluation app project

URL

https://github.com/ThijsVlaeyen/13_Adomavicius_Backx_Vlaeyen_Kassa_OOO2019

Requirements

Indicate for the entire project which requirements (possibly further elaborated/divided on the basis of the assignment) have successfully implemented you, and which topics have not been successful. If you didn't finish some of the requirements provided, indicate why not... The reason may be "lack of time", it may be an issue "didn't know how, it crashed", or it may be that you had a very good reason not to implement it....

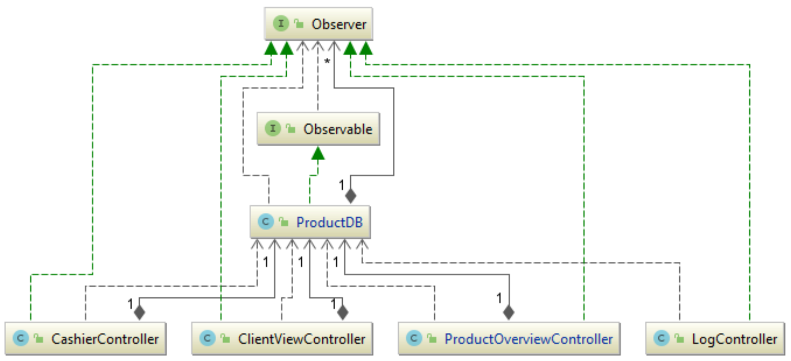
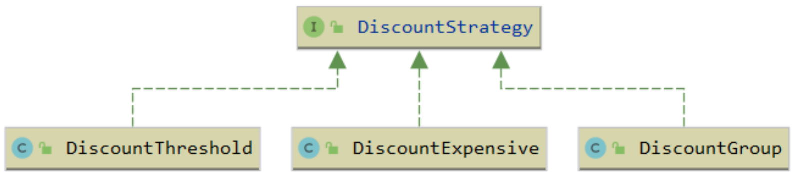
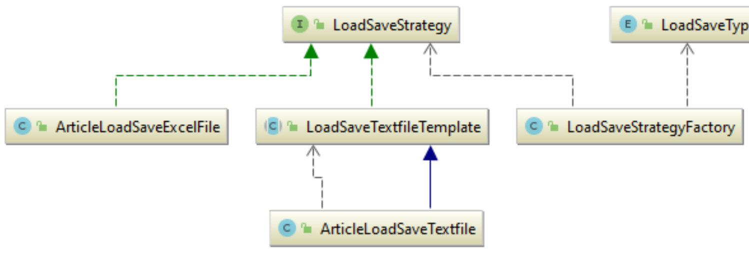
Translated with www.DeepL.com/Translator (free version) Add the final generated class diagram of your code, as a separate image, as an attachment to this report.

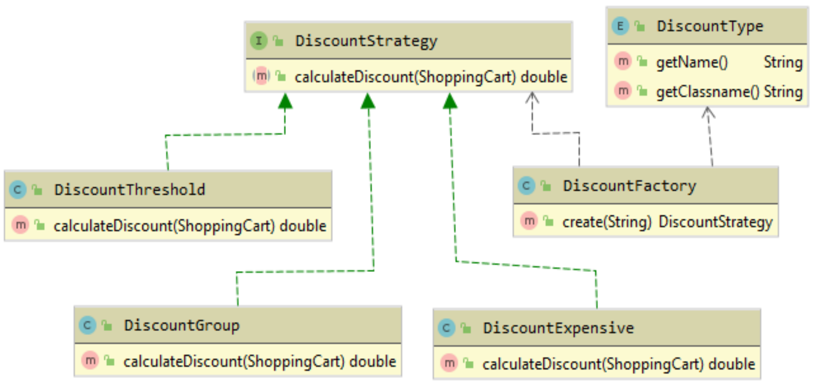
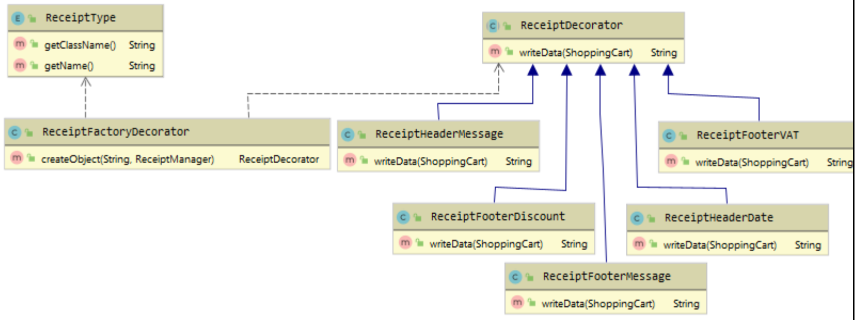
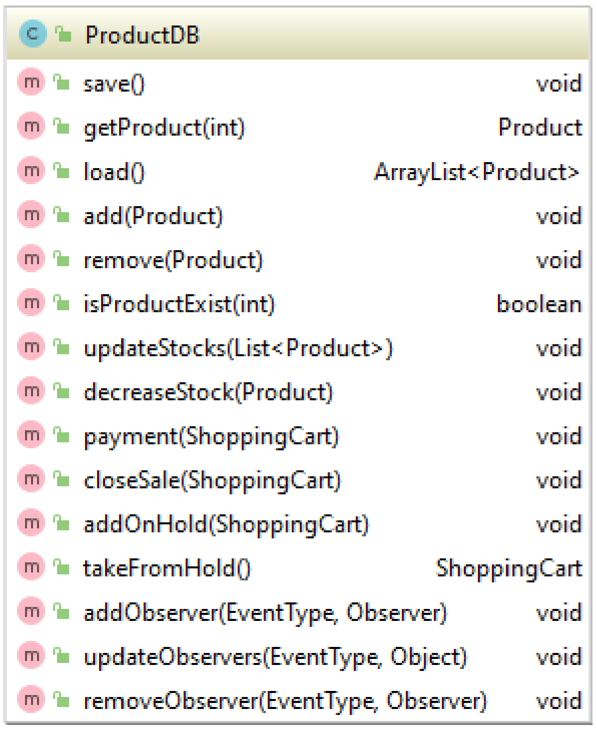
If there are things from the assignment that you have not been able to work out or that you would like to work out better, then you should also add them here (in the last row of the table).

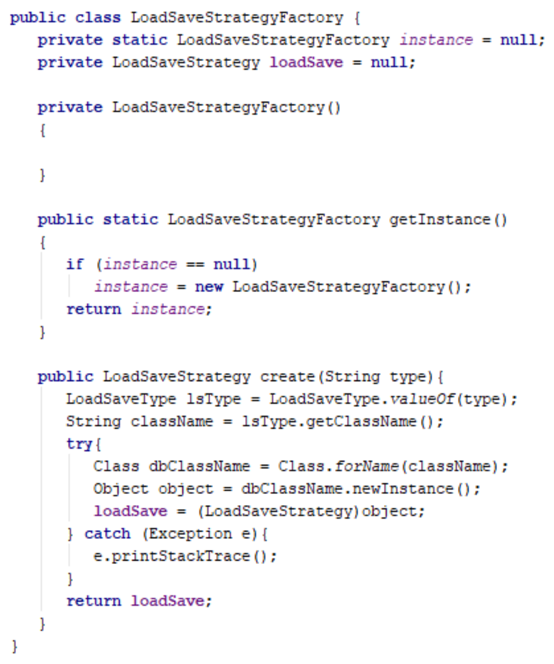
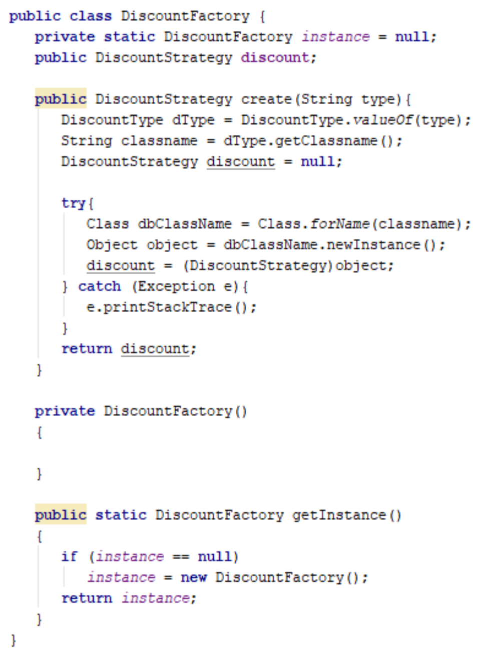
User story	OK?	if not ok - what does not work (see acceptance criteria) and why?
01. Show overview articles	OK	
02. Read Excel file	OK	
03. Register cash register	OK	
04. Show cash register sales to customer	OK	
05. Remove article from cash register sales	OK	
06. Cashier sales on hold	OK	
07. Apply discounts	OK	
08. Closing cash register	OK	
09. Pay cash register	OK	
10. Print receipt (on console)	OK	

Design patterns

For each pattern seen, please indicate where you used it (possibly more than once). Generate a class diagram for each situation in which you have used the pattern. Provide additional information (benefits / why / ...). If you didn't use a pattern, explain why not.

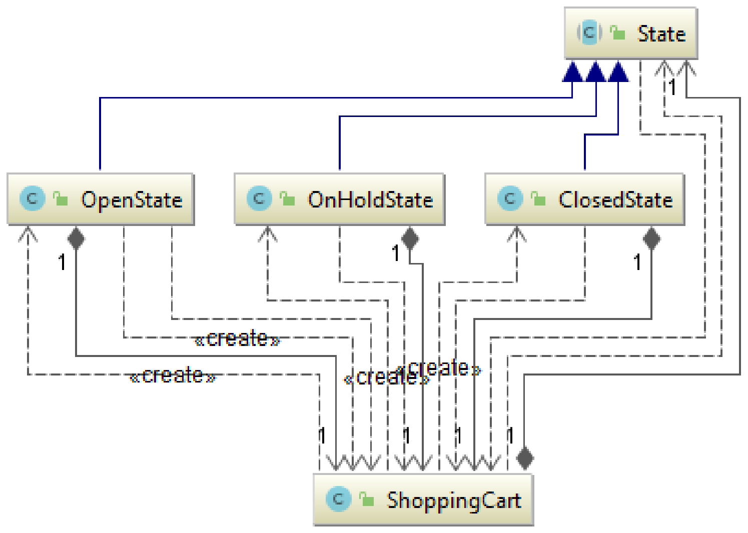
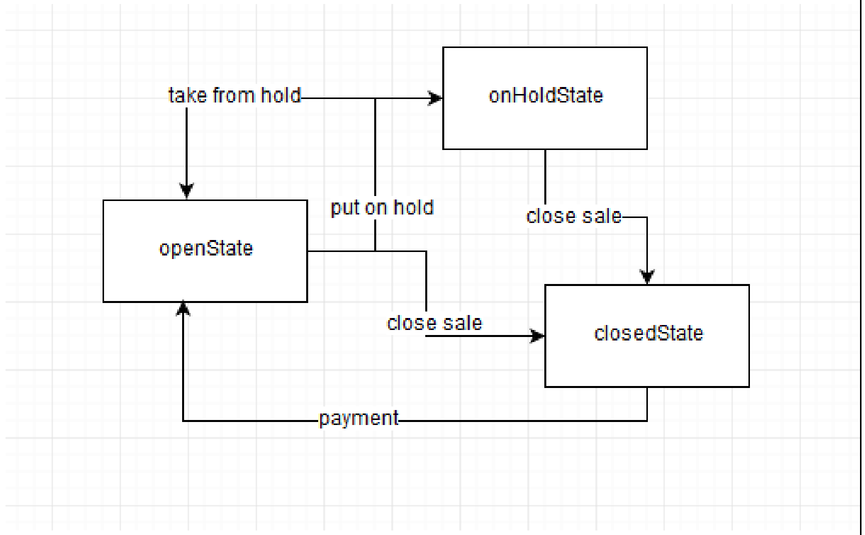
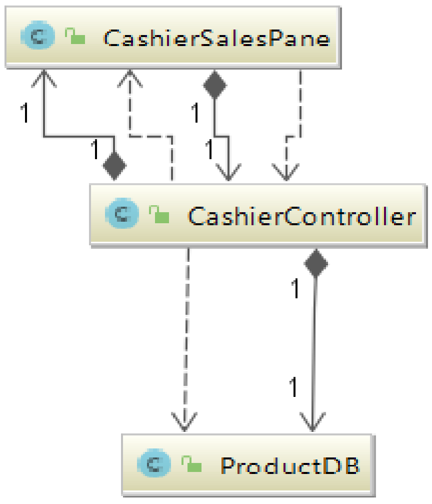
	Applied (yes/no) In which stories(no.) Why applied (advantage)	Associated class diagram (generated from your java code)
Observer	Yes, story 4 and next. The observer made sure that there is a simple connection between the cashier screen and the customer screen.	 Observer Observable ProductDB CashierController ClientViewController ProductOverviewController LogController 1 *
Strategy	Yes, story 7 for the different discount types. The context class in this example is CashierController where we use a DiscountStrategy to calculate the discount on a certain shopping cart.	 DiscountStrategy DiscountThreshold DiscountExpensive DiscountGroup
Simple Factory	For story 1 and 2 we used factory for the read and write function of an excel or txt file.	Load Save Factory  LoadSaveStrategy LoadSaveType ArticleLoadSaveExcelFile LoadSaveTextfileTemplate LoadSaveStrategyFactory ArticleLoadSaveTextfile

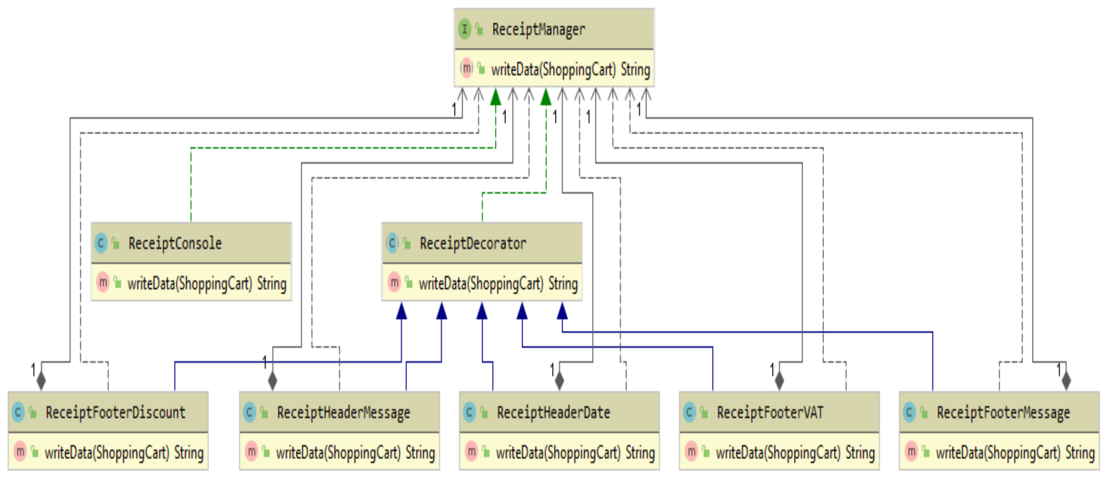
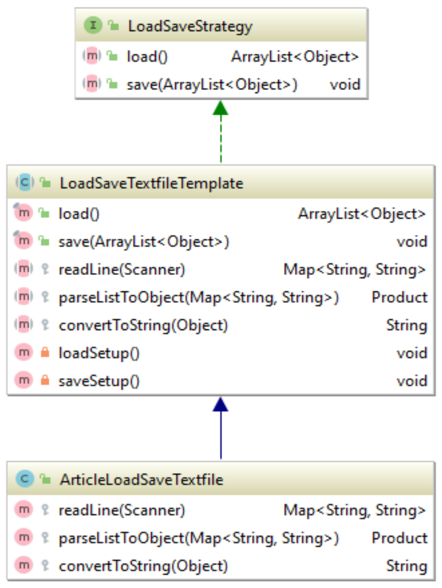
	For story 7 a factory is made for the different kinds of discounts	Discount Factory  DiscountStrategy calculateDiscount(ShoppingCart) double DiscountType getName() String getClassname() String DiscountThreshold calculateDiscount(ShoppingCart) double DiscountFactory create(String) DiscountStrategy DiscountGroup calculateDiscount(ShoppingCart) double DiscountExpensive calculateDiscount(ShoppingCart) double
	For story 10 for the different types of discount	Receipt Factory  ReceiptType getClassName() String getName() String ReceiptDecorator writeData(ShoppingCart) String ReceiptFactoryDecorator createObject(String, ReceiptManager) ReceiptDecorator ReceiptHeaderMessage writeData(ShoppingCart) String ReceiptFooterVAT writeData(ShoppingCart) String ReceiptFooterDiscount writeData(ShoppingCart) String ReceiptHeaderDate writeData(ShoppingCart) String ReceiptFooterMessage writeData(ShoppingCart) String
Facade	Yes, in our project we create ProductDB class which provides a simple interface to complex subsystems.	 ProductDB save() void getProduct(int) Product load() ArrayList<Product> add(Product) void remove(Product) void isProductExist(int) boolean updateStocks(List<Product>) void decreaseStock(Product) void payment(ShoppingCart) void closeSale(ShoppingCart) void addOnHold(ShoppingCart) void takeFromHold() ShoppingCart addObserver(EventType, Observer) void updateObservers(EventType, Object) void removeObserver(EventType, Observer) void

Singleton	Yes, for story 2 we implement singleton pattern for LoadSaveStrategyFactory in that case we make sure that there will be only one object instance.	
	Yes, for story 7 we implement singleton pattern for DiscountFactory in that case we make sure that there will be only one object instance.	

```
public class LoadSaveStrategyFactory {
    private static LoadSaveStrategyFactory instance = null;
    private LoadSaveStrategy loadSave = null;

    private LoadSaveStrategyFactory()
    {

    }

    public static LoadSaveStrategyFactory getInstance()
    {
        if (instance == null)
            instance = new LoadSaveStrategyFactory();
        return instance;
    }

    public LoadSaveStrategy create(String type){
        LoadSaveType lsType = LoadSaveType.valueOf(type);
        String className = lsType.getClassName();
        try{
            Class dbClassName = Class.forName(className);
            Object object = dbClassName.newInstance();
            loadSave = (LoadSaveStrategy)object;
        } catch (Exception e){
            e.printStackTrace();
        }
        return loadSave;
    }
}
```

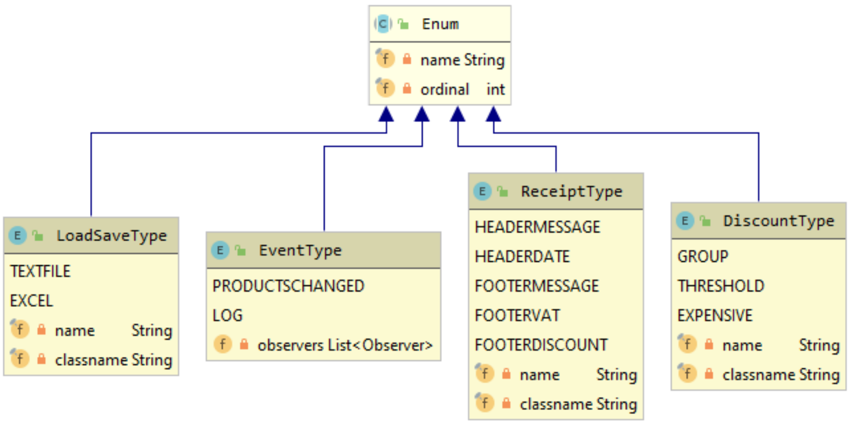
```
public class DiscountFactory {
    private static DiscountFactory instance = null;
    public DiscountStrategy discount;

    public DiscountStrategy create(String type){
        DiscountType dType = DiscountType.valueOf(type);
        String classname = dType.getClassname();
        DiscountStrategy discount = null;

        try{
            Class dbClassName = Class.forName(classname);
            Object object = dbClassName.newInstance();
            discount = (DiscountStrategy)object;
        } catch (Exception e){
            e.printStackTrace();
        }
        return discount;
    }

    private DiscountFactory()
    {

    }

    public static DiscountFactory getInstance()
    {
        if (instance == null)
            instance = new DiscountFactory();
        return instance;
    }
}
```

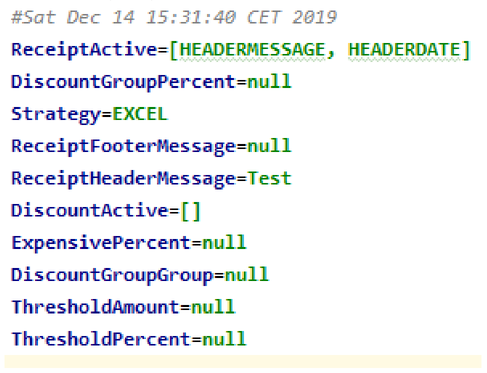
State	In story 9 we used a state design pattern to keep track of the different states of the shopping cart.	 State OpenState OnHoldState ClosedState ShoppingCart «create» 1  take from hold onHoldState openState put on hold close sale close sale closedState payment
MVC	Yes, throughout the whole project we have used MVC where the Controller class has a reference to model and view. this is just one of many examples where ProductDB is the model, CashierController the controller and CashierSalesPane the view.	

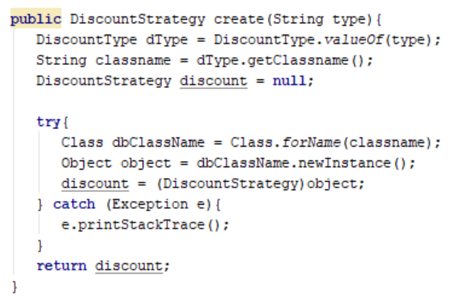
Decorator	In story 10 a decorator was used to print out the ticket.	 ReceiptManager writeData(ShoppingCart) String ReceiptConsole writeData(ShoppingCart) String ReceiptDecorator writeData(ShoppingCart) String ReceiptFooterDiscount writeData(ShoppingCart) String ReceiptHeaderMessage writeData(ShoppingCart) String ReceiptHeaderDate writeData(ShoppingCart) String ReceiptFooterVAT writeData(ShoppingCart) String ReceiptFooterMessage writeData(ShoppingCart) String 1
Template method	Yes, in story 1 and 2 where ArticleLoadSaveTextFile inherit from the template class LoadSaveTextfileTemplate. The subclass overwrites the methods readLine and parseListToObject.	 LoadSaveStrategy load() ArrayList<Object> save(ArrayList<Object>) void LoadSaveTextfileTemplate load() ArrayList<Object> save(ArrayList<Object>) void readLine(Scanner) Map<String, String> parseListToObject(Map<String, String>) Product convertToString(Object) String loadSetup() void saveSetup() void ArticleLoadSaveTextfile readLine(Scanner) Map<String, String> parseListToObject(Map<String, String>) Product convertToString(Object) String

Special Topics

For each "special topic", indicate whether you used it or not, and if so, where. Demonstrate with a class diagram if necessary.

	Applied (yes/no) In which stories(no.) Why applied (advantage)	Accompanying class diagram or additional explanation
Enum	We have four different enum classes. One for the LoadSaveType in story 1 & 2. To make a difference between textfile and excel. One for the factory that was used in combination with the decorator in story 10. To make a difference between the different types of messages on the receipt. One for the different types of discount to keep in story 7. One for the different event types in story 9	
	 Enum name String ordinal int LoadSaveType TEXTFILE EXCEL name String classname String EventType PRODUCTSCHANGED LOG observers List<Observer> ReceiptType HEADERMESSAGE HEADERDATE FOOTERMESSAGE FOOTERVAT FOOTERDISCOUNT name String classname String DiscountType GROUP THRESHOLD EXPENSIVE name String classname String	

Properties	In story 2, 7 and 10 we used the properties file, config.properties. There is a class that writes and reads to it.	Content properties file
		`#Sat Dec 14 15:31:40 CET 2019`
		`ReceiptActive=[HEADERMESSAGE, HEADERDATE]`
		`DiscountGroupPercent=null`
		`Strategy=EXCEL`
		`ReceiptFooterMessage=null`
		`ReceiptHeaderMessage=Test`
		`DiscountActive=[]`
		`ExpensivePercent=null`
		`DiscountGroupGroup=null`
		`ThresholdAmount=null`
		`ThresholdPercent=null`

Reflection	Yes, in story 1,2,7,10. reflection is used in all our factory classes to make the right class and return it.	`public DiscountStrategy create(String type){`
		`    DiscountType dType = DiscountType.valueOf(type);`
		`    String classname = dType.getClassname();`
		`    DiscountStrategy discount = null;`
		
		`    try{`
		`        Class dbClassName = Class.forName(classname);`
		`        Object object = dbClassName.newInstance();`
		`        discount = (DiscountStrategy)object;`
		`    } catch (Exception e){`
		`        e.printStackTrace();`
		`    }`
		`    return discount;`
		`}`

Other...		

Distribution of work

Indicate in percentages how much you approximately spent on this task.

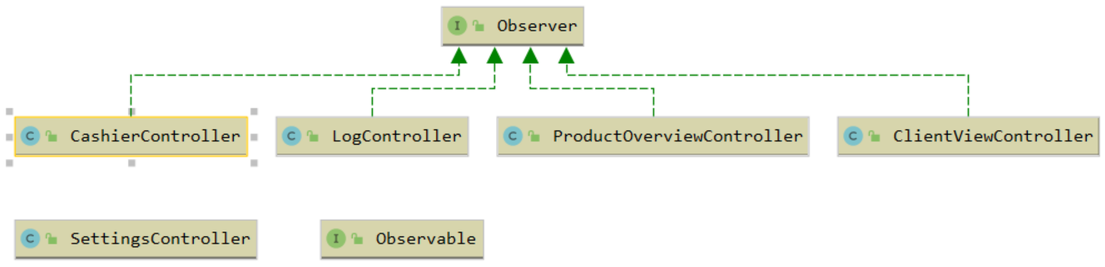
	Backx	Vlaeyen	Adomavicius	Total
Design	30%	40%	30%	100%
Class diagrams	40%	20%	40%	100%
Implementation	35%	35%	30%	100%
Report	20%	60%	20%	100%

Class Diagrams

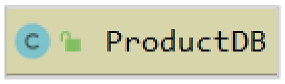
Add the final generated class diagram of your code), as a separate image file. This class diagram should be easy to read. Spread it out over several pages (e.g. 1 sheet per package (MVC) and 1 overview class diagram (without attributes and methods). DO NOT PRINT A CLASS DIAGRAM WITH A BLACK BACKGROUND COLOR!!!!!!!!!!!!!!!!!!!

Application Package:

Controller Package :



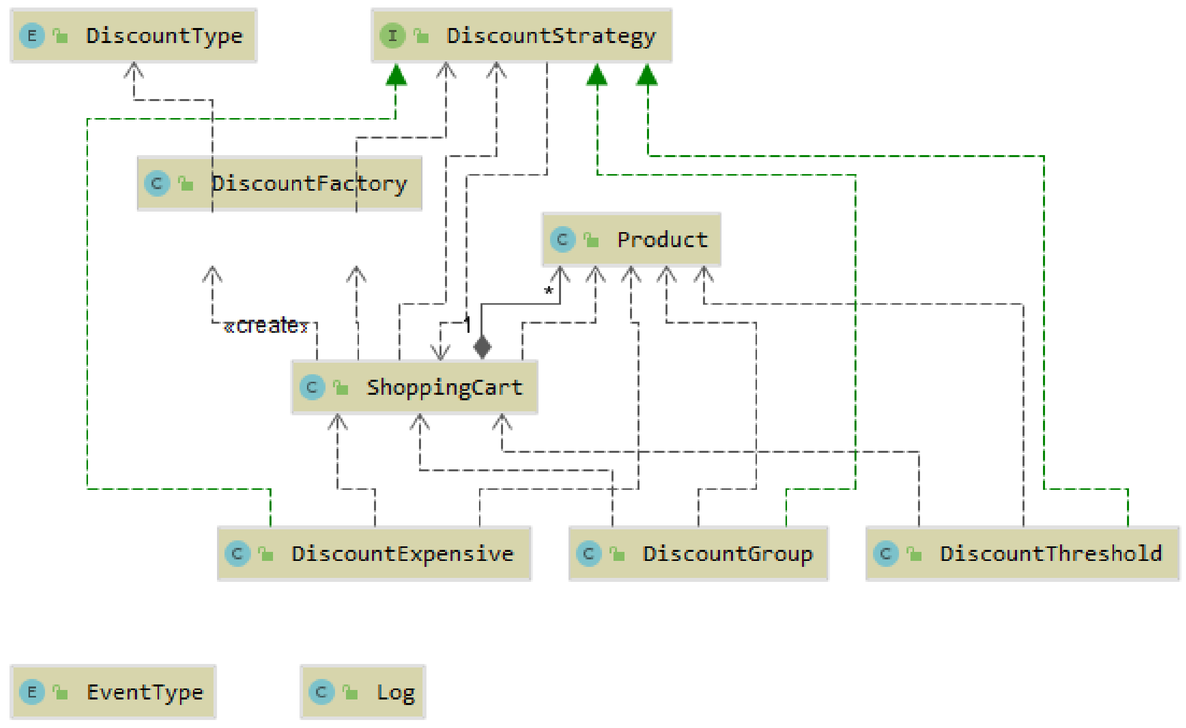
Database Package:



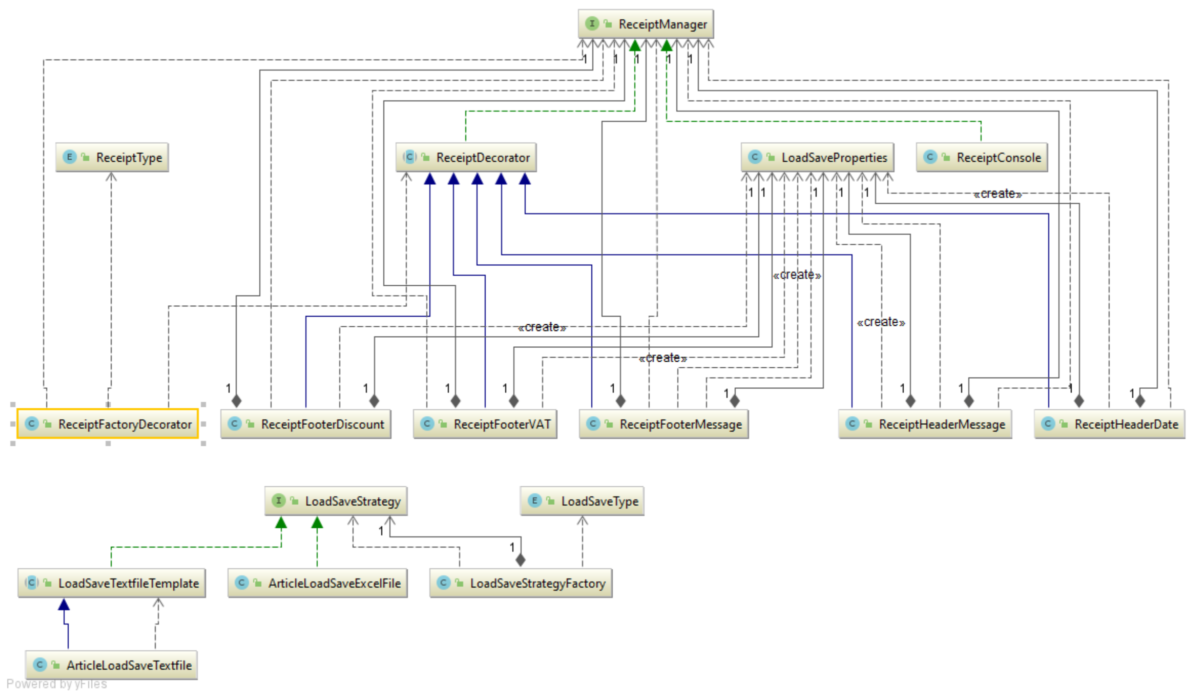
Files Package:

/ no java classes only files (article.xls, article.txt, config.properties).

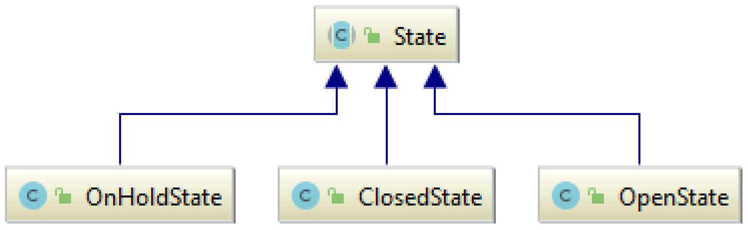
Model package:



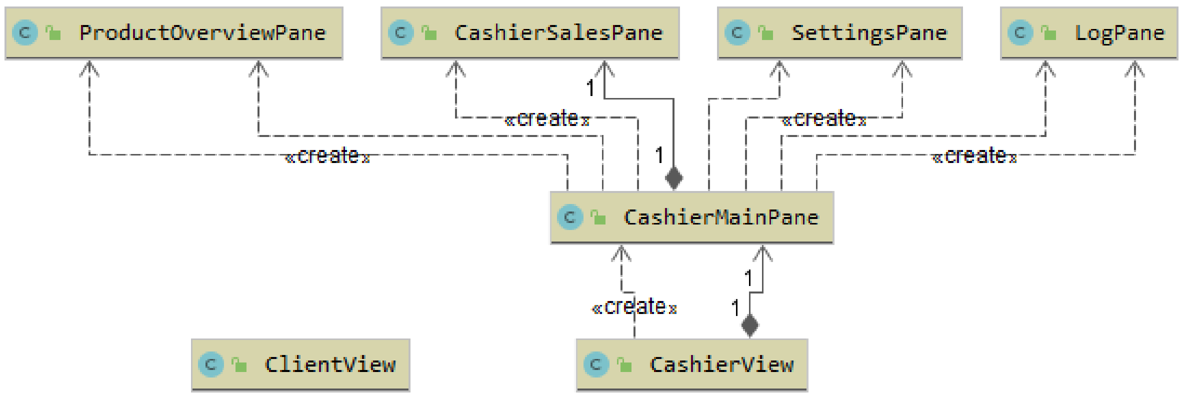
IO package (in model package):



States package (in model package):



View package:


ProductOverviewPane
CashierSalesPane
SettingsPane
LogPane
«create»
«create»
«create»
«create»
1
1
CashierMainPane
1
«create»
1
ClientView
CashierView