PRO2/SEN1 Performance Assessment

Student identification

Key	Fill in at start of session.
STICK NR	
Name	
Signature	

Teachers:
Pieter van den Hombergh
Richard van den Ham
Thijs Dorssers

The assessment environment

Every candidate shall use a personal laptop. During the assessment that system will be running from the supplied **exam stick**, providing a working Linux-OS and containing everything you need, such as:

- a web browser to read any provided documentation
- an IDE

You will work alone.

Take care of where you type

The corrector will use the **editor-folds** to quickly find your solution. In fact anything outside those folds is invisible to the corrector or causes a lot of extra work. The correct and wrong way[a] to use the editor folds can be seen in the next examples.

Proper way of using the editor-folds. Your solution INSIDE.

```
//<editor-fold defaultstate="expanded" desc="PRO1_3; (1234567) Puk Piet ; WEIGHT 5">
//TODO complete a Hello world Method
System.out.println("Hello world"); // My solution
//</editor-fold>
```

WRONG, WRONG, oooohhh this is sooo WRONG

```
//<editor-fold defaultstate="expanded" desc="PRO1_3; (1234567) Puk Piet ; WEIGHT 5">
//TODO complete a Hello world Method

//</editor-fold>
System.out.println("Hello world"); // My solution is invisible to the corrector
```

[a] if you have to write a Hello World program

Hint: before you start coding, build the javadoc and study that first.
This will give you all the required documentation, without the distraction of too much detail.

//**TODO:** If working with NetBeans IDE, pressing CTRL + 6 will show the list of tasks. Every task is marked with //**TODO** and the task id. Typically you should work in id order.

Use the provided repository. For your convenience, the projects are committed to a subversion repository residing on the USB stick in your home directory. The projects on the desktop are the initial checkouts. You are advised to use the repository, by **commiting regularly**, so that you can save intermediate work and can roll back, in case of need. The use of this repository is not mandatory, but is the only way to show that you had had a working sub-solution at any time. In the post production process we will make a final commit into this repository and then use the HEAD as the correction baseline. Working with subversion on the command line is remarkably easy:

```
svn up
svn ci -m'completed TASK_1A'
```

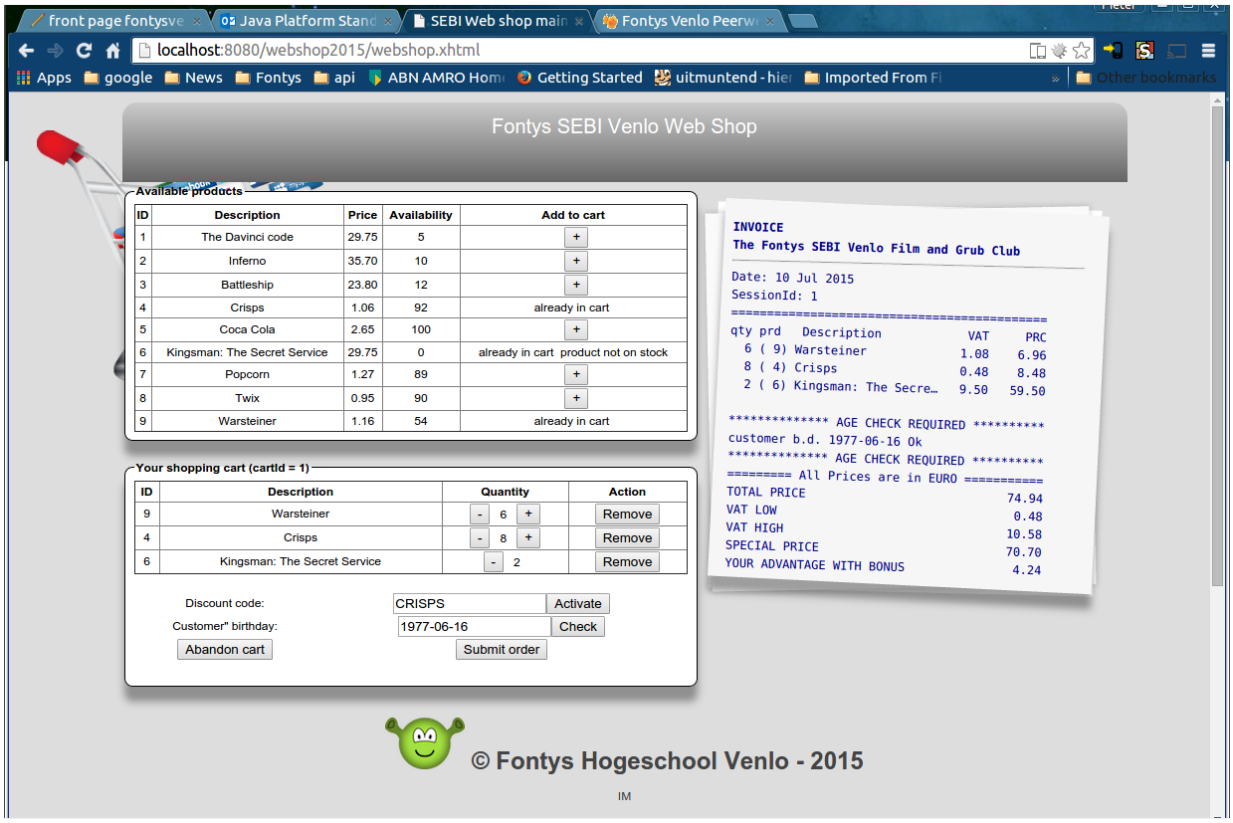

Figure 1: Screenshot of the working application

Contents

1 Webshop

In this assignment you will work on a webshop application.

The web frontend is implemented using JSF, the business logic part with handling the content of the shopping cart and the inventory is implemented in simple java objects (Pojos).

The following learning goals are addresses in this *exam*:

- Working in an existing application and understand the context.
- Understanding an API.
- Test Driven Development with unit testing.
- Working with polymorphic types.
- Working with exceptions.
- Writing and reading object stream files.
- Using Java 8 λ-expressions and function references.
- Using for loops and for each loops.
- Use and Java 8 λ-expressions and the Java 8 streams API.
- Use of JDBC with a data source.
- Use of JSF with a controller;
- Use of the mock framework Mockito.

This document provides an overall view of the application, design and realisation. Details to the specific test and programming tasks can be found in the source code. In those places where a class implements or overwrites a method, you can find the documentation of the method by clicking on the overwrite icon in the left gutter, between the source code line numbers.

2 Analysis

This simplified application models a webshop, in which customers can purchase goods, but can also abandon a cart, filled with products. You can find our initial ideas on the "napkin" in figure 2 on page 2.

Products in the cart are potential sales, and only if the customer selects an order, the sale will be completed.

To support both the success scenario (a sale), and the abandon scenario (no sale), the design is such, that a product can be put in a cart, making it into a potential sale. To let other customers know that potentially a good will be bought by the first customer, the potential sale is reflected in the inventory by transferring the good(s) from the inventory to the cart. The changed quantity is immediately reflected on the inventory part of the web site. Marketing says this will stimulate sales.

However, a customer may leave the shop and abandon a non empty cart. Here the design is such that a cart is created on first product selection by a customer. The cart has both a creation time stamp and an update time stamp. Every time the contents of the cart is changed, the update time stamp is set to that time (now() in the database). A background process (or thread) can then find all carts that have not been created or updated for some time, called **cart lifetime**. Such a cart is to be emptied, by transferring all goods back to the inventory and then deleting the cart. The cart's life time can be configured in the application at deployment time.

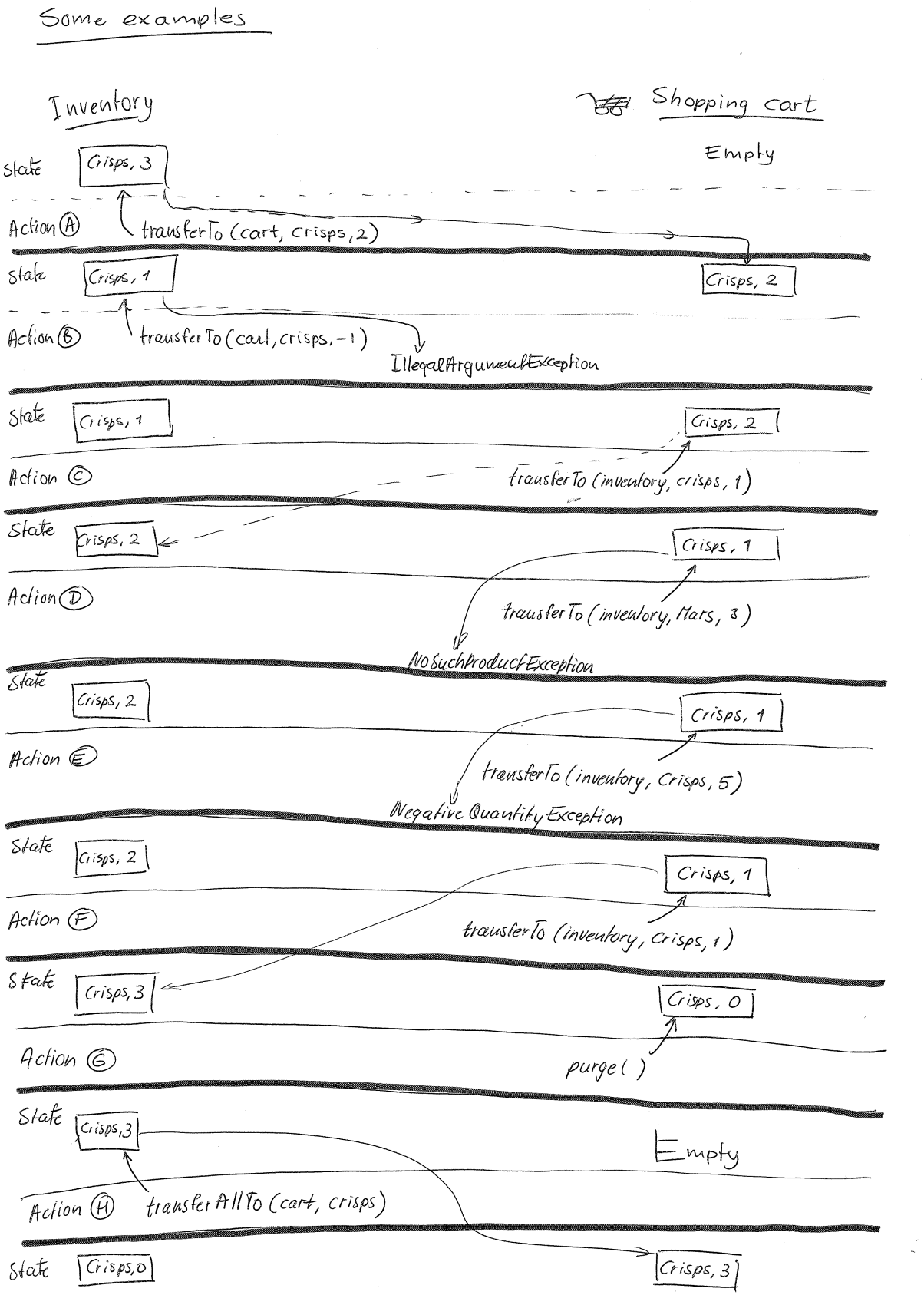

Figure 2: "Napkin" design

2.1 Basic but concrete scenarios

The two basic scenarios are given below.

Use Case	Customer makes a purchase
Code	UC-1.1
Package	webshop
File	**uc-purchase**.tex
Actors	Customer of webshop
Description	Successful purchase of a product in webshop
Precondition(s)	Customer is logged in.
Scenario	1. Customer visits sales page. 2. System creates an invoice. 3. Customer selects a product for purchase. 4. System takes product, quantity 1 from inventory and puts it (merges it) into cart. 5. System adds or updates invoice line in invoice. 6. System updates cart update time stamp. 7. System sets/updates the quantity of product in inventory and cart. 8. Customer can repeat steps 3-7 for other products. 9. Customer selects the submit option. 10. System persists the invoice. 11. System empties the cart (modelling that the cart stays at the shop, but the Customer receives the goods.).
Extensions	**3.1** This is the first product in the cart. 1. System creates cart record in database and sets create time stamp and update time stamp to now().
Exceptions	-
Result	Products are successfully purchased and booked as sales.
Version	1.0 | Author | HOM

Use Case	Customer abandoned cart
Code	UC-1.2
Package	webshop
File	**uc-abandon**.tex
Actors	- (Actions to be taken by the system autonomously)
Description	System detects that a cart exceeds the configured lifetime and transfers goods back to inventory
Precondition(s)	Cart has exceeded lifetime
Scenario	1. The goods that are in the cart are transferred back to the inventory. 2. The product is deleted from the cart table.
Extensions	-
Exceptions	-
Result	The cart is emptied and removed, all goods reclaimed for other (tentative) sales.
Version	1.0 | Author | HOM

3 Design

3.1 Simplifications

For this assignment, some simplifications are applied, business wise.

- There is no separate sales record, although accounting rules require it. However, such a **Sales** record (product, qty, sales price and VATLevel) contains the same conceptual information as the **InvoiceLine**. This is sufficient for the exercise.
- Because of the above simplification, the model does not support returning goods. This is Okay, because we do not intend to sell this implementation anyway. We rather want to keep it stupid and secret at the same time.

3.2 Architecture

The *project architecture* actually consists of two separate Netbeans projects. The idea behind this division into components is testability of the business component called **webshopModel**. The **webshop** is the web application which depends on this webshopModel.

3.3 Design of the product container

Both cart and inventory can be considered as *product container*, meaning they can contain tuples of (product,quantity). This tuple is modelled in a separate class `ProductQuantity`.

The logical interaction between inventory and cart is a transfer of (product,quantity) information. Note however, that these two containers should never refer to the same (product,quantity) tuple.

The operation to *take* products from a container is `take(Product,Quantity)`. This operation takes the specified amount from the container and returns a `ProductQuantity` tuple if successful. To *put* products into the container we use the `ProductQuantity merge(ProductQuantity)` operation which *adds to* or *sets* the current quantity to/with the new quantity.

It is quite natural to use the result of the take operation as the parameter to the merge operation: merge with the content of the cart what you took from the inventory.

Cart and inventory should behave differently when the amount is set to zero by means of a `transferToXXX()` operation. For the inventory, records with zero quantity are allowed, whereas for the cart, records with zero quantity should be deleted from the cart.

This matches the natural business situation that a quantity of zero in an inventory leaves the shelve space intact, only empty.

In the cart however, the (product, quantity) tuple with quantity set to zero has no meaning.

To support this difference in behaviour, the ProductContainer has a method called `purge()`, which removes any tuple having quantity equals zero.

Therefor: In the business logic one should call `cart.purge()`, but never `inventory.purge()`; This difference between cart and inventory is expressed by the fact that the classes `Inventory` and `Cart` both inherit from `ProductContainer`. See the class diagram in figure 6.

In chapter 6 you see that the project provides two implementations for the persistence layer.

In the database persistence implementation , the inventory is backed by table `inventory`, the cart by table `carts`.

4 Additions to webshop for assessment retake

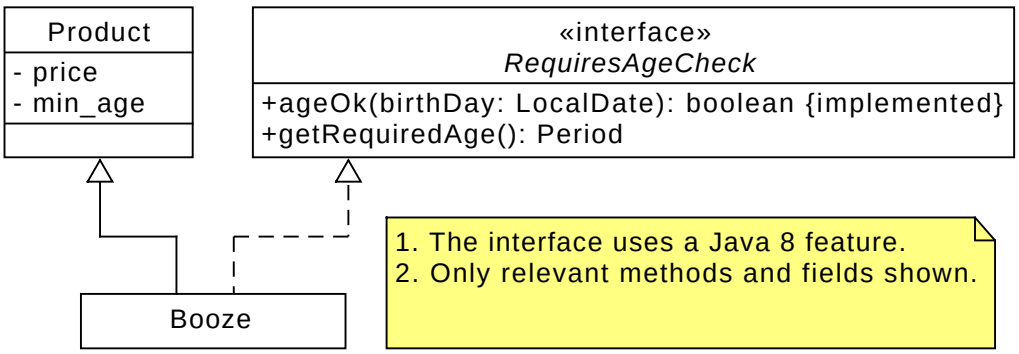

Figure 3: Product that requires age check.

The following additions have been made to the webshop in comparison to the lab exercise:

java.time API. In this new implementation the deprecated ~~`java.util.Time`~~ class and friends are avoided. Instead we use `java.time.LocalDate` to represent Date values and `java.time.Period` to represent a time span, such as age.

Age check Product has a sub class called `Booze` which requires an age check on sale. From the class diagram in figure 3 you can see the relation of the product subcategory `Booze`, the `Product` class and the new interface `RequiresAgeCheck`.

Persist invoice On sale, the Invoice information is persisted in the database or in the in-memory persistence implementation.

File system persistence the local file system is used as a 'database', in which the entities, e.g. invoice, are persisted to one file each. In the invoice example, the whole invoice, including its lines, are persisted to said file. Each type has its own sub directory below the directory `./fsdb`.

The mapper for the entities should be able to save the entity and restore the entity.

To be able to implement this, you should use the **`java.io`** or **`java.nio`** API. HINT: look at **`ObjectOutputStream`** and **`ObjectInputStream`** classes Javadoc.

See the TODO list for the tasks these additions imply.

5 Classes and responsibilties

The business is about Cart, Inventory, Invoice, InvoiceLine, Product, ProductQuantity, PriceReductionCalculator and VATLevel. The class, responsibilities and collaborations (CRC) are given below:

Product

The good we sell	
Responsibility	**Collaborators**
Product describes good and specifies price (including VAT) and VATLevel.	InvoiceLine VATLevel Cart Inventory

Invoice

The payment request to the buyer	
Responsibility	**Collaborators**
Specifies the money amount we request for the price stated and the VATLevel determined by financial law. Optionally states a special price or price reduction.	InvoiceLine VATLevel Cart SpecialPriceCalculator

InvoiceLine

Details of product, qty and vat	
Responsibility	**Collaborators**
Specifies details of product, price and vat level	Product Invoice VATLevel

Cart

The virtual container the customer puts the (product,quantity) tuple in.	
Responsibility	**Collaborators**
Collects customer product selections and shows product and quantity. Cart **is a** ProductContainer[a].	Inventory Invoice ProductContainer (super)

[a] This relationship is a design aspect and should normally not go into a crc card, but is mentioned here because this is an exam.

Inventory	
The stock of products.	
Responsibility	**Collaborators**
Holds the products in quantities. Shows what is available at any moment. Inventory **is a** ProductContainer[a]. [a] This relationship is a design aspect and should normally not go into a crc card, but is mentioned here because this is an exam.	Cart ProductContainer (super)

PriceReductionCalculator	
Computes a price reduction when the customer provides a bonus code.	
Responsibility	**Collaborators**
When the customer enters a bonus code, a price reduction will be calculated. This price reduction depends on the implementation.	Cart Invoice InvoiceLine

RequiresAgeCheck	
To check if the customer is mature enough to purchase the products in the cart.	
Responsibility	**Collaborators**
A cart may contain products that may only be sold to people of a certain age. Products that require such checks implement this interface. The type is used by the invoice. The mapper decides how products are classified.	Mapper Invoice Cart

Booze	
Example type for age check	
Responsibility	**Collaborators**
Booze extends Product implements RequiresAgeCheck	Product RequiresAgeCheck

In this exam, `Cart` and `Inventory` share a common super type, `ProductContainer`. This is an attempt to keep design and implementation DRY[1].

3 Both Cart and Inventory understand an *ownership* concept. This makes it possible to connect a Cart and cart entry to each other. In a web implementation this can be used to either associate a web session or a revisiting customer to a cart. It also allows to have all cart-entries reside in one table.

[1] Don't Repeat Yourselves

ProductContainer	
Super type of `Cart` and `Inventory`.	
Responsibility	**Collaborators**
Both cart and inventory share the operations (take and merge), so the implementation of `Cart` and `Inventory` can mostly be given in this super[a] type.	ProductQuantity Cart (extends) Inventory (extends)

[a] This relationship is a design aspect and should normally not go into a crc card, but is mentioned here because this is an exam.

WebshopFacade	
Façade for use by the view implementation in MVC.	
Responsibility	**Collaborators**
Manipulates the product(s) by putting them from inventory to cart or the other way around and represents a visit of a customer to the shop. Provides easy access point to all the functionality provided in the business package and as such is the Model in MVC.	Cart Inventory Invoice PriceReductionCalculator

All classes above are in the webshopmodel project.

WebshopController	
The JSF managed bean	
Responsibility	**Collaborators**
Webshopcontroller provides access for the JSF pages to business logic. It is the controller in MVC. This is the only Java class that resides in the web shop project.	WebshopFacade

6 Realisation

In figure 4 you see a simplified model of the most important business use case. The application provides two implementations of the interface `ProductContainer`:

1. `IMProductContainer` which is a hybrid file system / in memory implementation.
2. `PGDBProductContainer` which uses a database (in particular postgresql) as the persistence service.

See the class diagram figure 6 on page 9 for details on class relations.

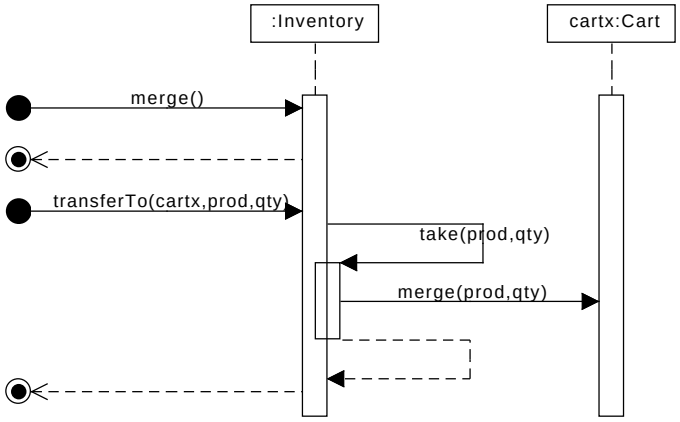

Figure 4: Fill the inventory (merge), then put a product from inventory into a cart.

In this exam you will have to test and implement some of the methods. The test names and documentation should give you hints on what to test. After you have written the tests, you can implement the methods.

6.1 The Façade between business and web

In the design you find a Façade class, name **`WebshopFacade`**, which provides one entry point to the whole business model. In the class diagram you can find it too to see what methods are provided by the **`webshopModel`** project to the **`webshop`** project.

6.2 File system persistence

The file system persistence implements a very simple approach, using one file per (composite) object to persist. The only type that is currently persisted in this way is the **`Invoice`**. The files a stored in a sub directory, simple named by type (Invoice in the example). The entity is stored in a file named after its *"primary key"*, with the file extension `.ser`. For instance, invoice with number 31 is saved in file `fsdb/Invoice/31.ser`

6.3 RDMBS Persistence implementation

In the realisation of the persistence version (**`PGDBProductContainer`**) of the product container, the responsibility of some of the constraints (e.g. quantity of a product must be non-negative) can be delegated to the database. When such a constraint is violated, the database layer will throw an exception, which is wrapped into a **`java.lang.RunException`** but should be caught, inspected (unwrapped with **`Exception.getCause()`** and dealt with in a business appropriate way.

The database schema can be found in figure 5.

Note that the **`owner_session`** column identifies the cart, in case you wonder why there is no `cart_id` column in the `carts` table.

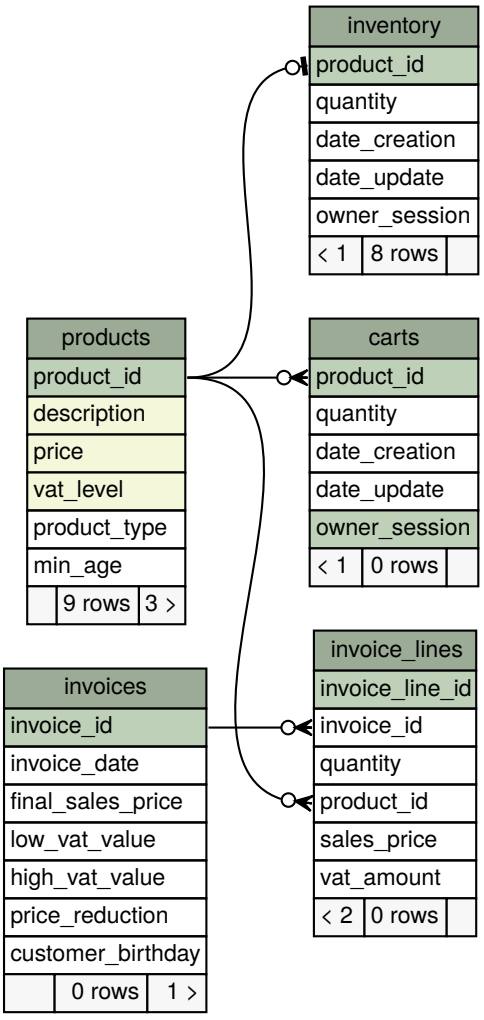

Figure 5: Simplified database schema

6.4 Create and load the postgresql database

You must create a postgresql database named **`webshop`** and load it with the initial data. To do that follow these steps:

1. Open the database GUI pgadmin III.
2. Connect to the exam database. The user/role name is **exam** with password **exam**.
3. Open the *servers* node, then open the server node *exam*.
4. In the object browser (left panel with tree like structure) right-click on databases node in the tree and select **New database**
5. In the *new Database* dialog enter **webshop** in the name field and leave the rest to their defaults. Click Ok.
   - Click on the webshop database to activate the sql buttons.
6. Select the execute sql button, the one with the magnifying glass.
7. Open the file browser (the folder icon).
8. In the folder **`.../dbscripts/`** you find the file schema.sql. Open, then execute it using the **ExecuteScript** button.
9. In the object browser, you can now verify that the script executed successfully, by checking that the webshop database exists and the tables, views and sequences as are defined in the script.

When you want to restore the database to it's original state with a filled inventory, you can re-execute the schema.sql script in **pgAdmin III**.

You can also use NetBeans to access and load the database. To make the jdbc driver available to the projects, you should first create a Netbeans Library, named as the one used in the project.

7 Your practical tasks

You can find the SEN1 tasks in the **`webshopModel`** project. There are no SEN1 tasks in the webshop web project.

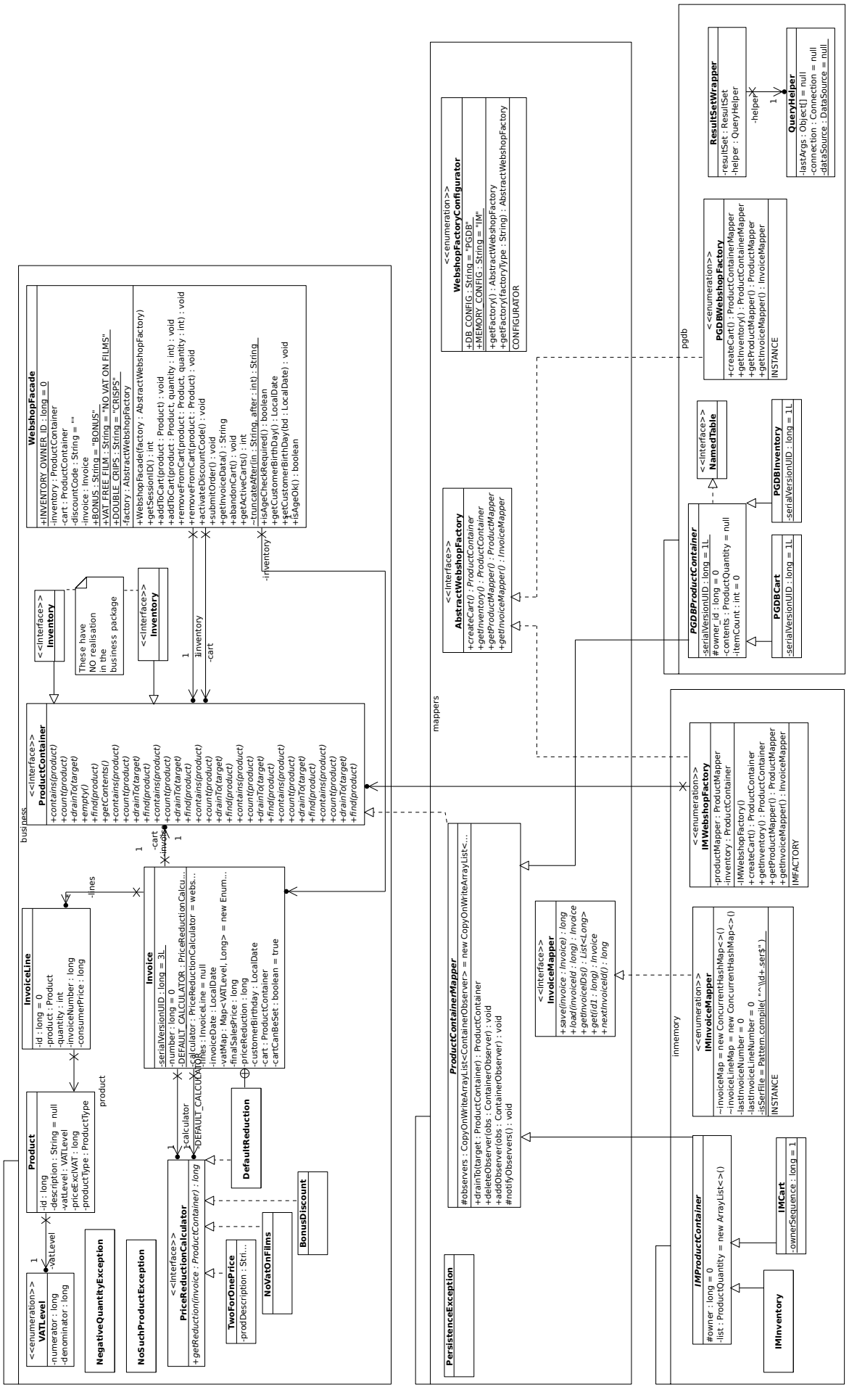

Figure 6: Class diagram of the important types in the business and persistence packages

In essence you should test and develop (yes in that order) methods in the ProductContainer class. The tests and methods you have to program are marked as such in the source code.

It is best to follow the given task numbering scheme, for instance task T01_A (the test) should be followed by T01_B (the implementation to that test).

The projects are available in a repository, which resides on the USB stick as well. The projects have been checked out into a sandbox called `examproject-EXAMxyz` on your desktop. NetBeans will do the right thing if you **`svn commit`** or **`svn checkout`**. In this way you can use subversion as a safety net and also keep the good way of working of TDD: RED, GREEN, REFACTOR.

Important hint: Work Test Driven

Develop a test: (Test one aspect of the method that must be implemented). When the test compiles but fails (red) you have a working test. Now commit with log-comment `test` **`red`**.

Implement an aspect: Now implement the aspect as in "make the test pass". When it is green, again commit with log-comment `test` **`green`**.

Repeat: The above until all aspects have tests and are implemented and your world is green.

Commit after *every successful run*, **Red** or **Green**. A failing (red) test is a good test!
Getting the test green afterwards is only better if it was red before.

Last remarks:

- Your development environment provides code coverage by means of the **tikione/jacoco** NetBeans plug-in. This can create html coverage reports which shows you what code you missed, test wise.
- If a test annoys you, you can temporarily switch it off by adding the **`@Ignore`** annotation to that test or even to the test class.
- There is NO **`JSF-xhtml`** work in this exam. You could of course use the application (start webshop project in glassfish).
- Bonus codes understood by the application are: "BONUS", "NO VAT ON FILMS" and "CRISPS".
- You can find the tasks and documentation in the source code of both projects. Because the documentation uses Javadoc features, so before you start coding, you should generate the java-doc from the **`webshopModel2015`** project.
- The tasks (**TODO**'s) can be found in NetBeans IDE by pressing CRTL + 6, or on a German keyboard STRG + 6
- The tasks are numbered. We tried to make these numbers guide you through the exam. Most tasks have and A and a B part. In such cases, the A part counts for testing (SEN1), the B part for programming (PRO2, practical part).
- The first task (create a "refreshment") should resolve the issues the compiler has with the initial state of the webshopModel project.

7.1 Exam tasks

The tasks list in this exam, collected from the source code is listed below.

T01_A1 Type of Warsteiner test, asserts
T01_A2 Create predicate to check if cart correctly checks for Booze
T01_B1 implement Booze class.
T01_B2 implement ageOk
T02_A set up mock cart with return values.
T02_B lambda for two for one price
T03_A testSaveAndLoad of im persistence
T03_B implement IMInvoiceMapper.save(Invoice). read javadoc first
T04_A testGetTotalPriceExcludingVAT
T04_B implement getTotalPriceExcludingVat using for-loop
T05_A reduction on Film
T05_B compute reduction.
T06_A test mapCartToLines
T06_B impl Invoice.mapToCartLines
T07_A test save and load
T07_B implement part of PGDBInvoiceMapper.load()

When you are done, close the IDE and shut down your computer.
Hand in the USB stick and all received papers.