Corso di Programmazione ad Oggetti – a. a. 2013/14
Corso di laurea in Ingegneria e Scienze Informatiche – Università di Bologna

Project relation: Profumeria Barbara Customers Manager

Entirely developed by Andrea de Rose.

# 1. Analysis

The owners of the local perfume store Profumeria Barbara commissioned AndreWare to develop an application which simplifies the customers database management.
Formerly, the database is an ODB file containing personal data about each customer and a “stamp” counter so that they can achieve promotions and discounts by purchasing products.
The application's main features are:

• Automatically load customers list from database on program start up;

• Add/edit/remove customers and their relative information;

• Fast increment or decrement customer's stamps count by button pressure;

• Store changes on database on demand or on program termination;

• Set the database directory (database files are stored on a network folder);

• Search customers by name and surname;

• Undo or redo one or more operations;

• Multiple language support.

# 2. Design

Although aims and requirements of the application seem pretty clear, there are many aspects to treat separately using different object-oriented design patterns.
The main and most suitable pattern for this application's aims is Model-View-Controller. This pattern allows the developer to deal with program data, user interface and their interaction in different contexts, making each part reusable or editable without interfering with each other.

- The Model contains a list of customers and a database pointer through which data is loaded and stored. Note that Model's customer list is separated and changes are written on database only after a proper save request. This helps preventing accidental entry errors and data corruption.

- Views use the Composite pattern in order to build themselves and add as many components as required under hierarchies. Their aim is to interact with users and communicate their choices to Controller.

- The Controller works as a bridge between views and Model. It handles all events caused by user interaction and performs consequent actions.

Other relevant patterns useful for application are:

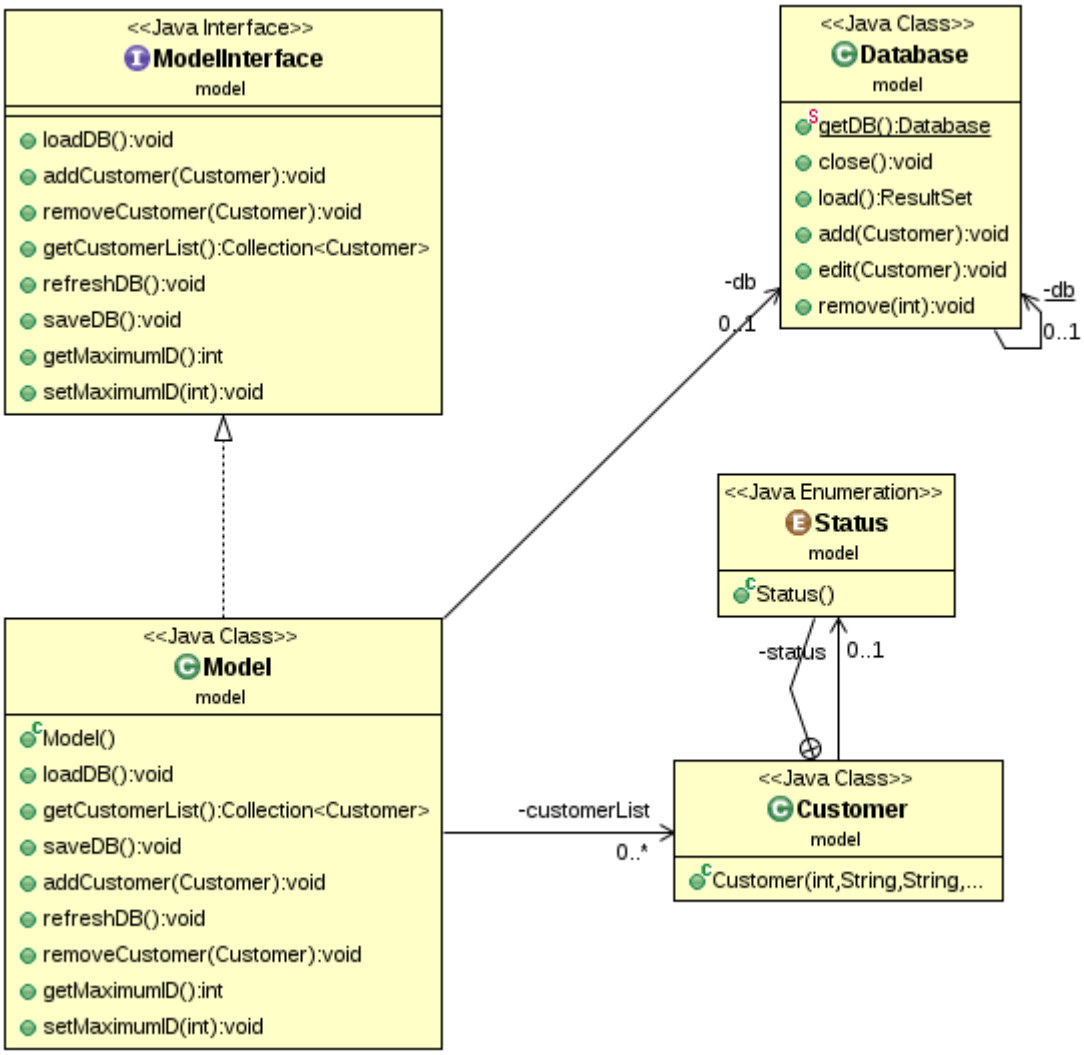
- Singleton: Language and Database objects have a single instance in the whole process. There is no need to have multiple instances of Language, because its only aim is to return strings by reading a file content. The Database must only have one instance; otherwise, a SQLException would be thrown since it would try to create more connections to a single structure.

- Command: all actions performed on customers become command objects. This helps the implementation of a undo/redo mechanism, allowing users to revert or repeat actions before or after saving.

- Memento: when performing RemoveCommand or EditCommand and its derivate, the previous and next (not on RemoveCommand) customer statuses are stored in the command object so that they can be easily restored upon undo and redo performing.

- Observer: observer are interfaces mostly implemented by Controller and commands. Observers are used by views to notify changes and events.

Model structure UML diagram:



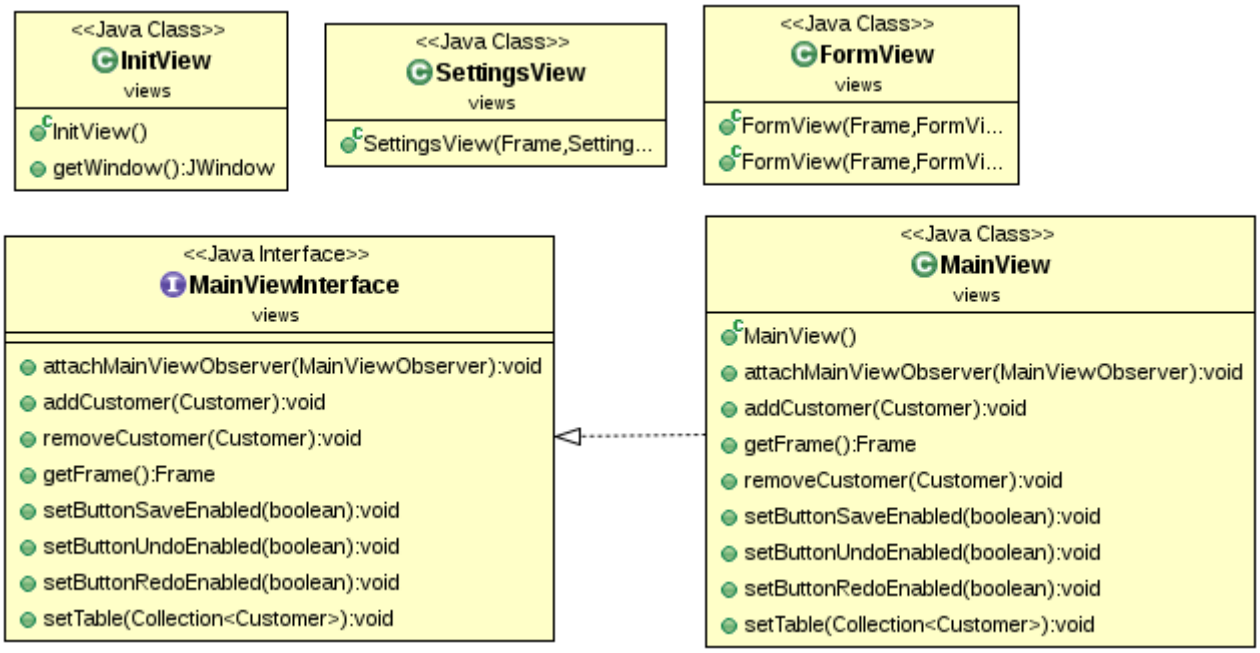

Main aspects:

- **ModelInterface** and **Model** define and implement an interface to load customer data onto a collection, manipulate and save it when required.

- **Customer** and **Customer.Status** define data structure of single customers, Customer uses setters and getters to easily access fields.

- **Database** is a singleton object using HSQLDB library to connect to database and execute queries upon loading or saving data.

Views structure UML diagram:




Main aspects:

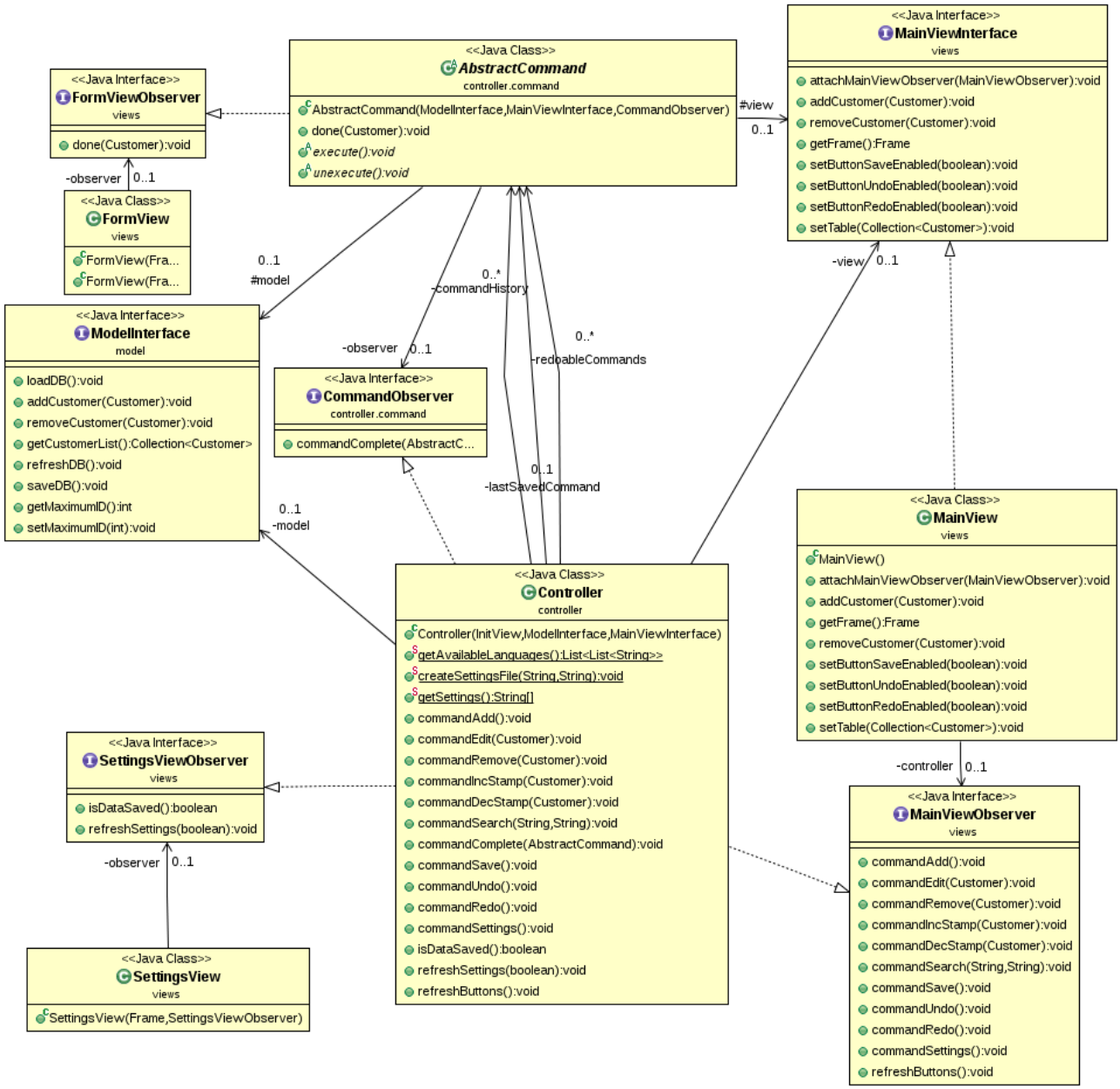
- **InitView** contains a simple JWindow displaying the application logo while it is loading.

- **MainViewInterface** defines the main application view. **MainView** impements the interface and contains a frame displaying the customers list on a table and a set of buttons to perform any implemented action.

- **SettingsView** contains a JDialog with components letting the user change database path and language settings.

- **FormView** contains a JDialog with components letting the user add or edit customers.

MVC structure and interactions UML diagram:




Main aspects:

- Both **Controller** and any **AbstractCommand** have access to **ModelInterface** and **MainViewInterface** in order to update relative customers list.

- **Controller** is both a **MainViewObserver**, **CommandObserver** and **SettingsViewObserver**, it handles all requests and works as a bridge between most of application parts.

- **AbstractCommand** is a **FormViewObserver**, it's an abstract class defining a generic command performed by user.

- **Controller** holds two **AbstractCommand** collections and a single pointer in order to manage undo/redo functions and saving status.

- All views notify their own observer when an event occurs.

# 3. Packages

- **controller**: contains the main application **Controller** class.
- **controller.command**: contains the **CommandObserver** interface, **AbstractCommand** class and all implemented commands.
- **core**: contains the **Language** class which manages text shown in views and the **Init** class containing the main method.
- **model**: contains interfaces and classes relative to main application **Model**.
- **test**: contains a JUnit test verifying all **Model** components work properly.
- **views**: contains interfaces and classes relative to all application views.
- **resources.zip**: this zip file contains logos, button images and language files. It is unpacked in application folder on first launch.

# 4. Testing

Class **test.ModelTest** was created to test application **Model** and it's interaction with **Database**.
The test consists in creating a dedicated database folder, adding a new **Customer** to **Model**, saving them on **Database**, reloading and verify that customer data was stored with no alteration or problems.
The same action sequence is repeated to edit customer and then to remove it.

Manual testing consisted in adding, changing and removing one or more customers, undoing-redoing operations. Testing also consisted in saving changes to database, restarting the application and verifying that data prior to restart was not changed.

# 5. Final Notes:

Development work has mostly been spiral.
Because there was not a precise plan, the project was nearly started over after one month: only the views and database pointer structure were kept. Most of the work was done after the start over.
The biggest issues were knowledge lacks about design patterns and, more generally, projecting and designing. This was summed to the fact of working alone and having no experience in software development.
When writing the code, the Database class took long to work properly, because of syntax mistakes writing queries and even the ODB database structure was not clear (many chars columns where int was expected and vice versa). Many other issues slowed down the work for days.
A final (but not simple) rework was made so that application can be ran by executing a jar file: program does not use ResourceBundle as previously scheduled, because language files are stored in the self-generating program folder and no longer in a source package.

The resulting product should is capable to self-generate a configuration file by asking the user which language to use and the database location. If no database is found or the required table is missing, the application will generate it automatically.
The user should not see any error occur (unless they alter configuration file or language files structure); sometimes “useless” actions may be performed (ex. querying the change of a customer data whthout effectively changing anything), but this would not be visible to user.

Of course the whole program can be refactorized to improve performances and work in more efficient ways. Perhaps a new branch of this project will be started in the future.