1. User requirements

At this stage we want the application to have several types of instruments, namely Electric and String types. For now we want only have violin and guitar as a string instruments, sampling pad as an electric instrument, and electric guitar as both an electric and a string instrument. Every instrument has to have its name, price, description, optionally photo, quantity, and rating derived from reviews(about reviews later). In addition electric instruments have to have their voltage listed, string instruments have to have number of strings and material listed. Guitar has to have a type of strings ("Nylon", "Steel"). Violin has to have information about whether it has replacement strings or not and info if a shoulder rest is present. Sampling pad has to have info about a number of squares and about an input. Electric guitar has to have some info about an input, about a number of singles, and about a number of humbuckers.

For instrument we want to see a list of all instruments, get all instruments of a specific brand, and get description of a specific instrument with all its fields included.

Every instrument is associated with its brand. For every brand we work with we have a contract. In case we cut out the cooperation with a brand, all the contracts with the brand are removed. Every brand has to have its name, description, year of establishment, and optional logo. Every contract has a starting date, a end date, and a textual content.

There are two types of persons, customer and worker, a person can be simultaneously both a worker and a customer, types can change dynamically. Every person has its name, surname, unique email address, unique phone, address. Private data of a person can be changed.

A customer is able to order instrument(s) and pay for it. Also a customer is capable to leave a review on purchased instruments. Every customer has a balance field, a bonus points field.

Every order has an optional description, a delivery address, and info whether order was paid or not. Order can be added, edited, paid by a customer. We want to be able to get all the orders for a specific customer.

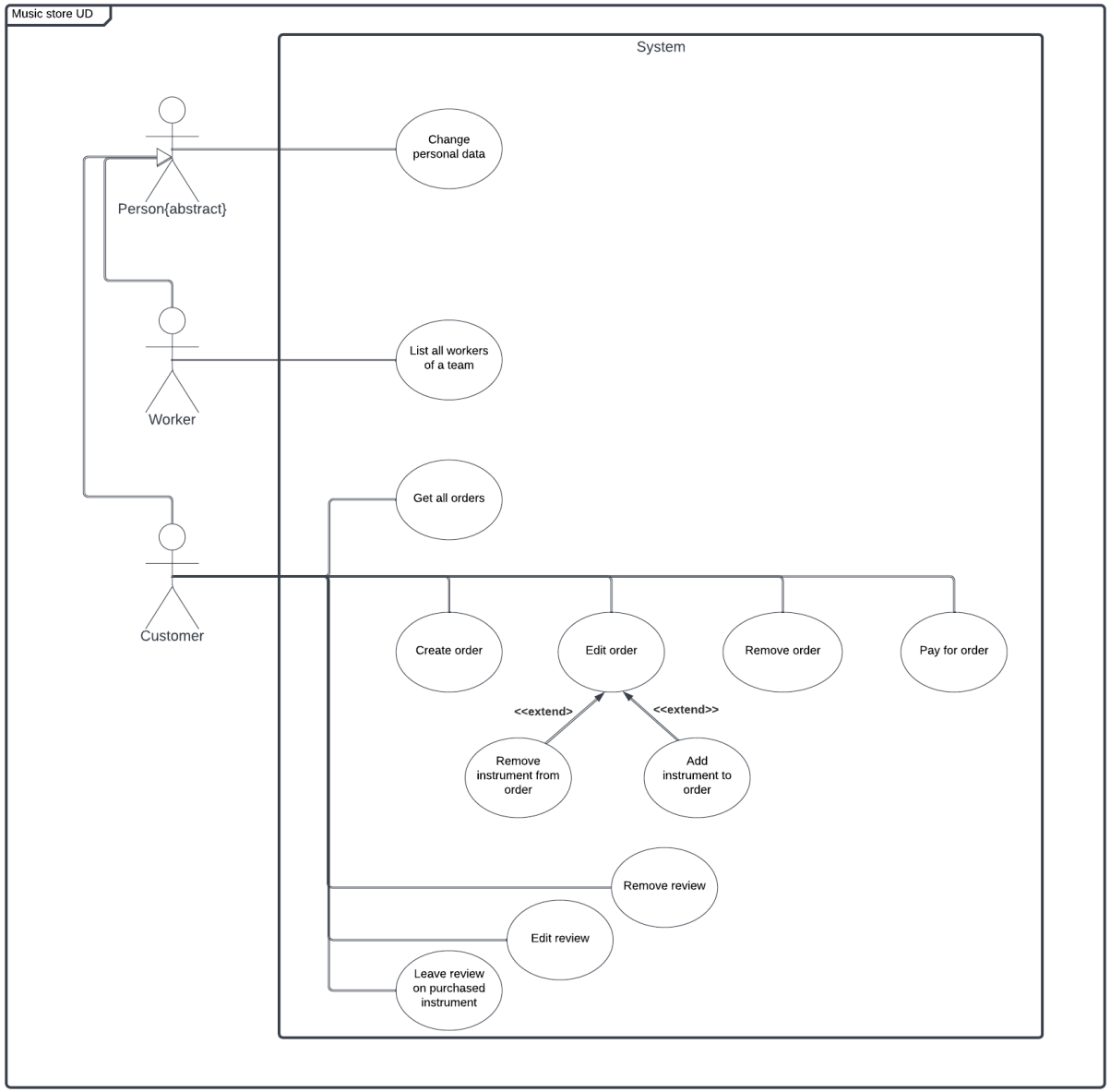
Every Review has its content, optional image, rating. Reviews can be added, removed, edited. We want to allow customer to edit reviews he/she left. We want to know once a review was edited.

A worker has minimal salary field(the same for all), salary field(cannot be less than minimal salary), worker type("delivery", "repairing service", "seller", "customer service", "manager"). Every worker belongs to one team, some of the team members may be managers of the team(only if their worker type is manager). For workers we want to be able to get a list of workers for a specific team.

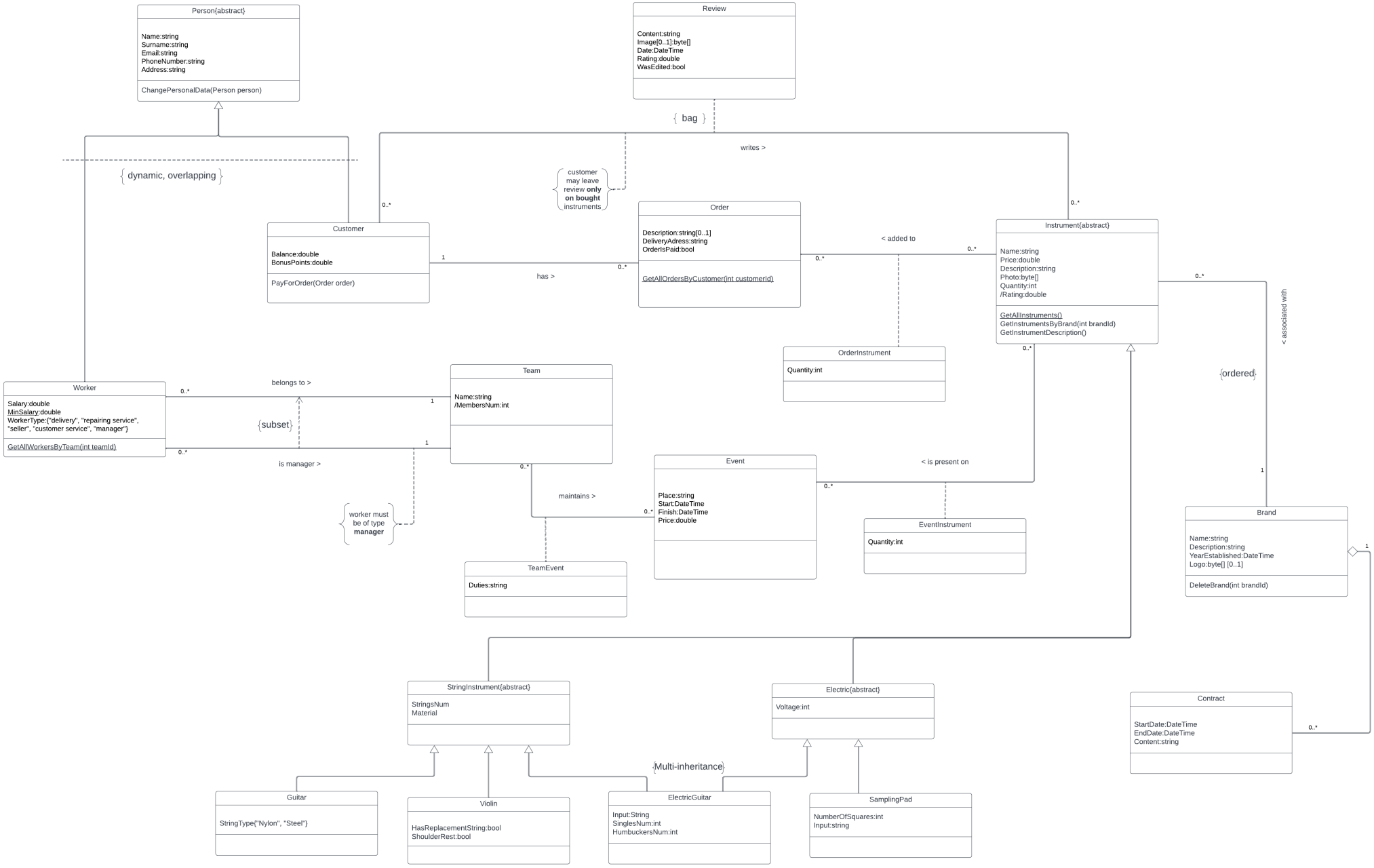
Every team has name, number of workers. Some teams are responsible for events.

Events may be maintained by several teams, also events may include some instruments from the store. Every event has place, start date, finish date, and price.

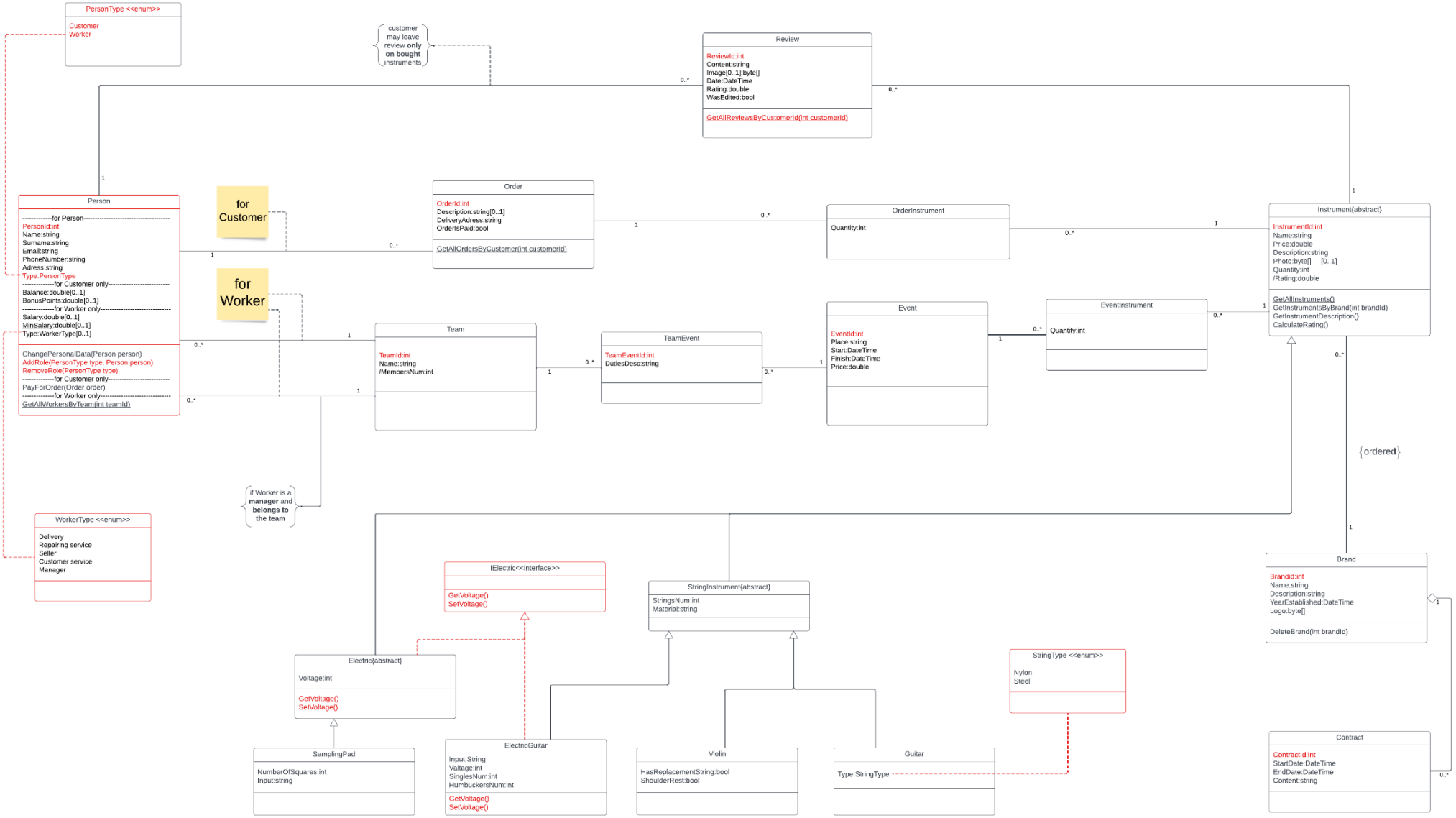
2. The use case diagram



3. The class diagram – analytical



4. The class diagram – design



5. The scenario of selected use case

1. Use-case: Edit review

1.1. Actors

Customer

1.2. Purpose and context

A customer bought an instrument and left a review, then customer wants to change some content in the review.

1.3. Dependencies

1.3.1. Included use-cases

None

1.3.2. Extended use-cases

None

1.4. Assumptions and pre-conditions

1. Customer has instruments bought and has review(s)

1.5. Initiating business event

Authorized customer wants to change his/her review

1.6. Basic flow of event

1. User clicks on “my instruments” button on the “Main” screen.
2. System shows “My Instruments” screen to the user.
3. User clicks on a selected instrument.
4. System shows to the user all the reviews he/she posted for the selected instrument (“My Reviews of Instrument” screen).
5. User looks for a review to modify and presses “modify” button on it.
6. System shows to user a window with the fields to modify(“Edit Review” screen).

7. User introduces modifications.
8. User presses “save” button.
9. Review is modified and saved to the database.
10. User is redirected back to the list of all the reviews for the selected instrument(“My Reviews of Instrument” screen).

1.7. Alternative flows of event

1.7.1. **User presses “back” button**

3a1. User presses back button.
3a2. User is redirected to the “Main” screen.
3a3. End of the use-case.

1.7.2. **User presses “back” button**

5a1. User presses back button.
5a3. Flow goes to point number 2.

1.7.3. **User decides not to modify the review**

8a1. User clicks “cancel” button.
8a2. Changes are discarded.
8a3. Flow goes to point 4.

1.7.4. **User provided either 0 or to many characters(more than 500).**

9a1. System shows the user a notification about an inappropriate number of characters.
9a2. Flow goes to point 7.

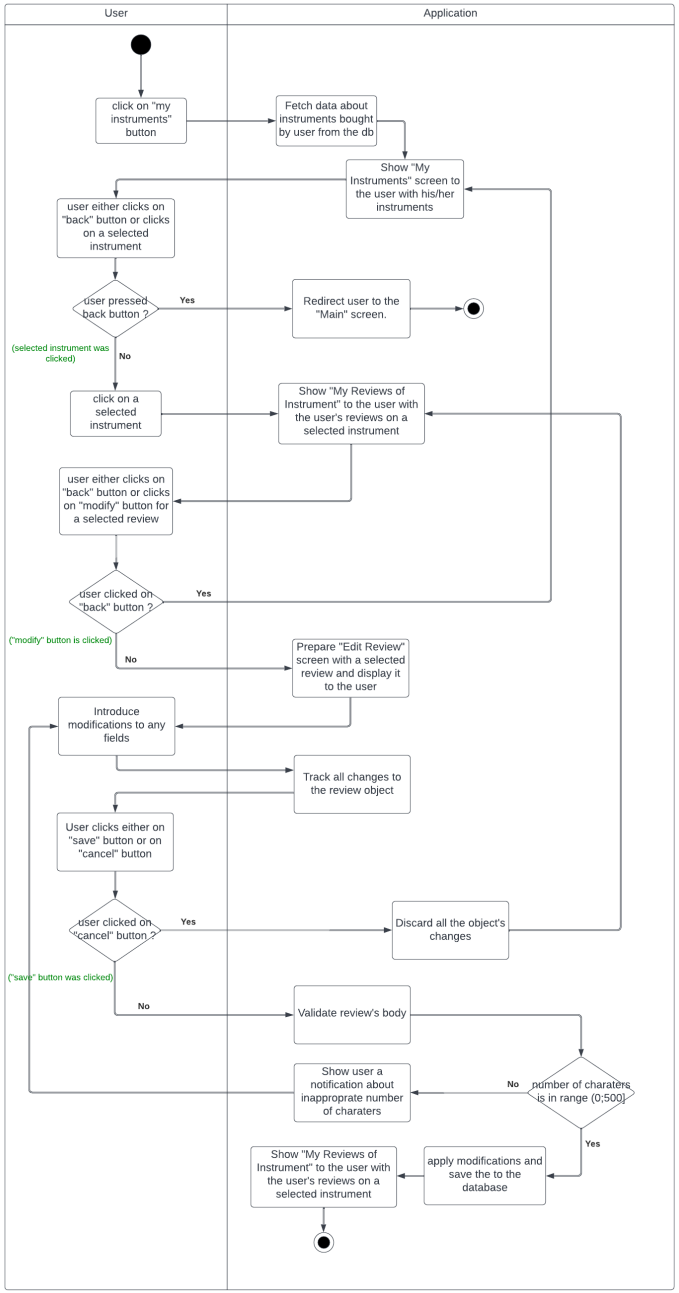
1.8. Extension points

None

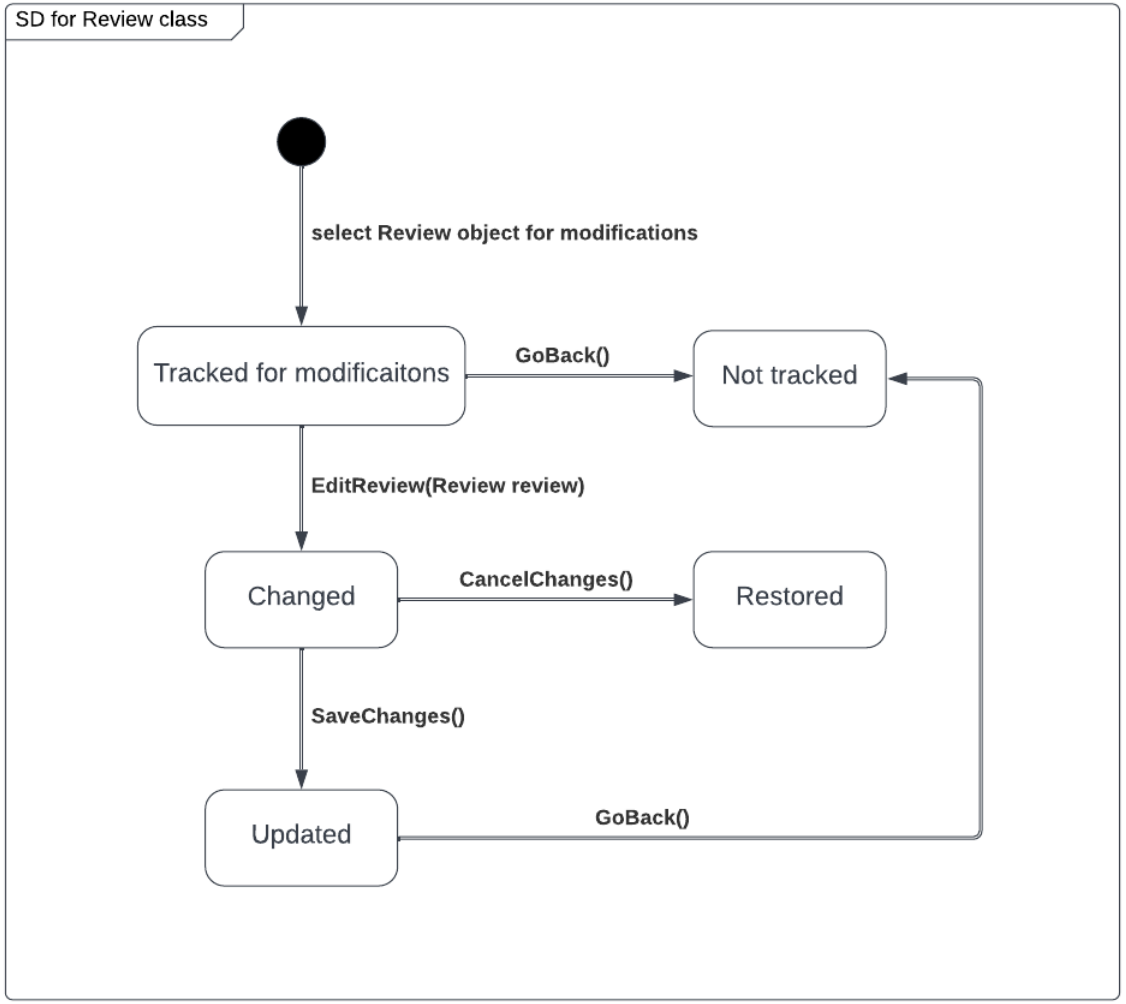
1.9. Post-conditions

1. The review is successfully changed and saved to the database.

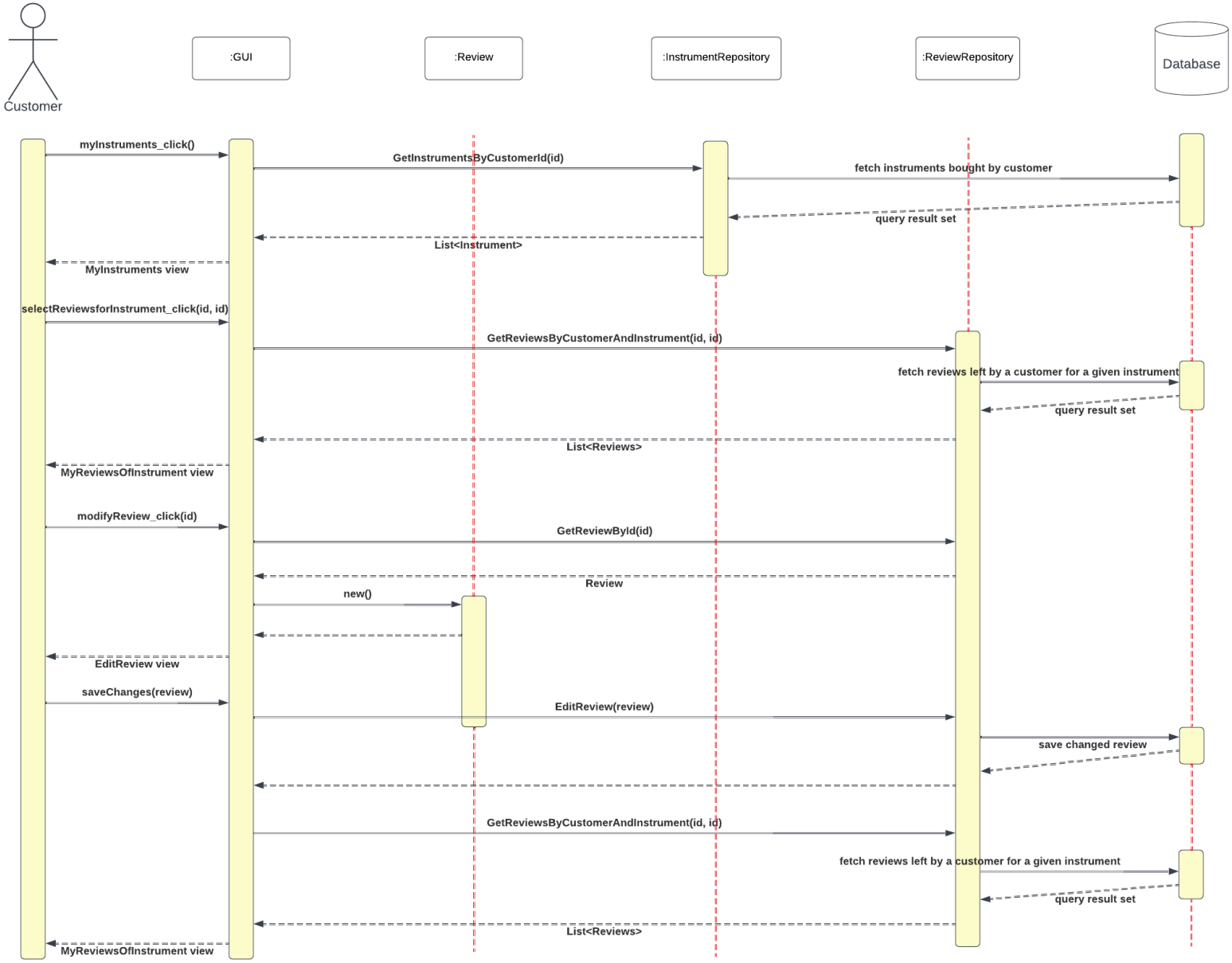
6. The activity diagram for picked use case



7. The state diagram for selected class



8. The interaction (sequence) diagram for selected use case



9. The GUI design

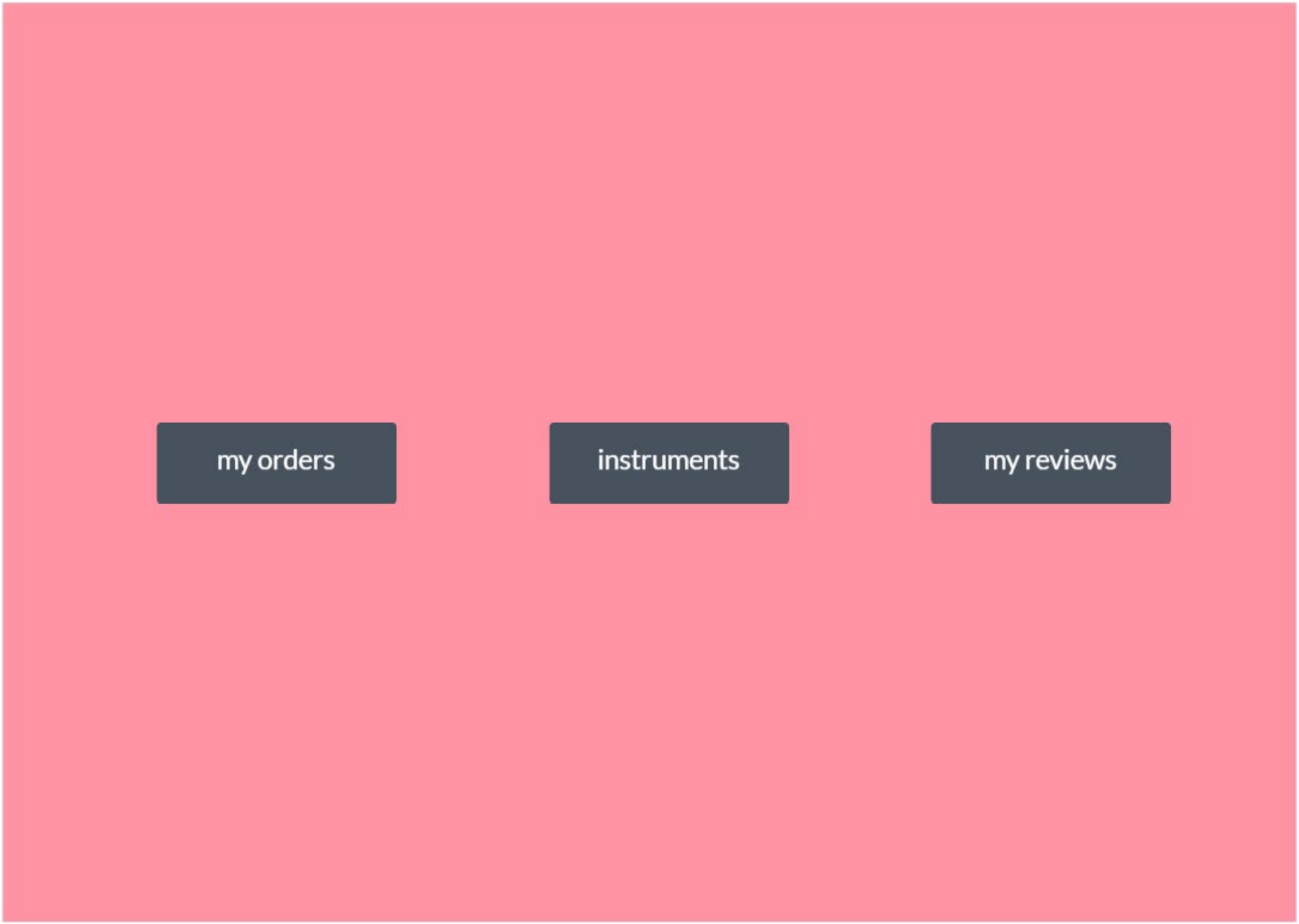


Figure 1. Main screen

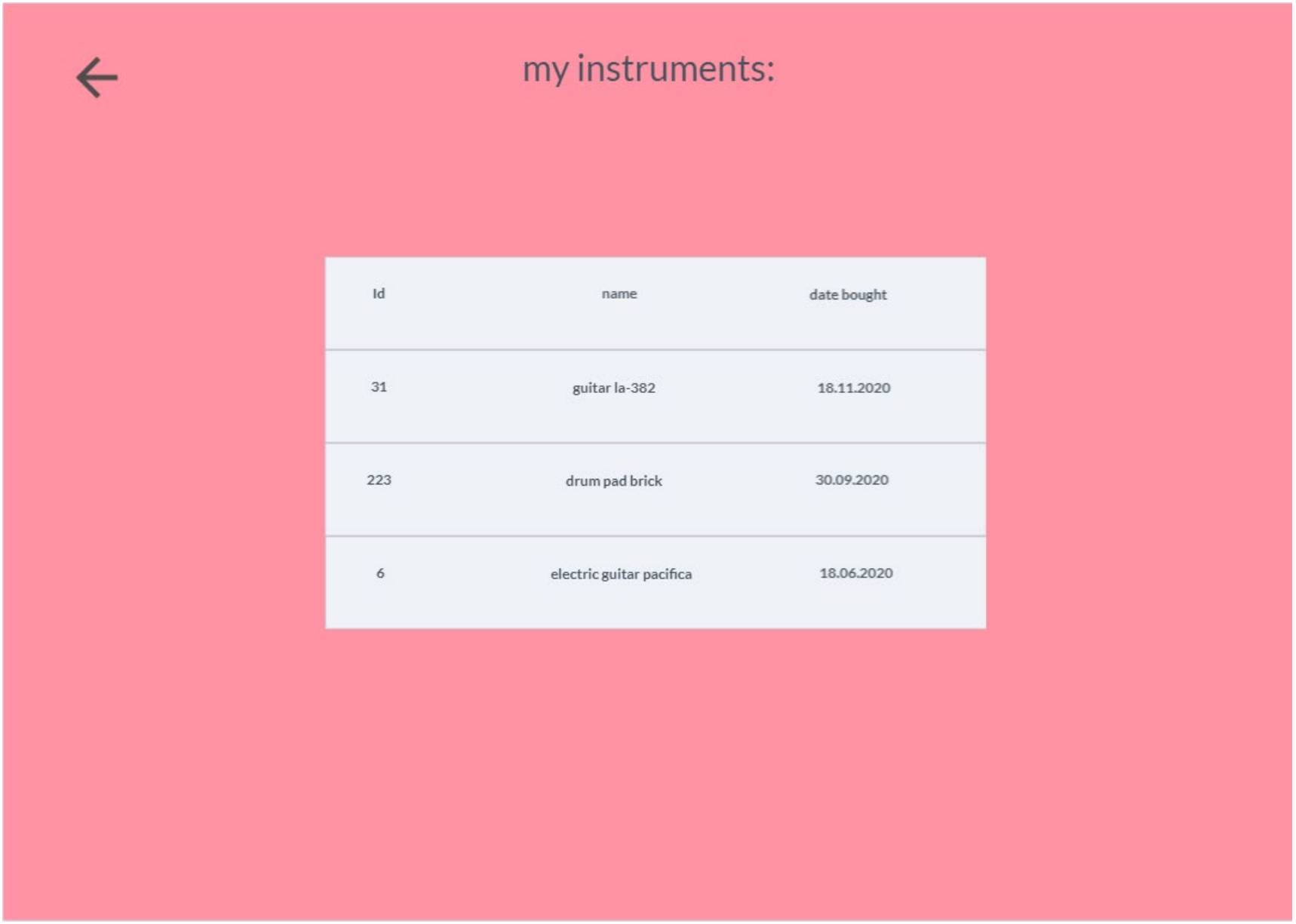


Figure 2. My Instruments screen

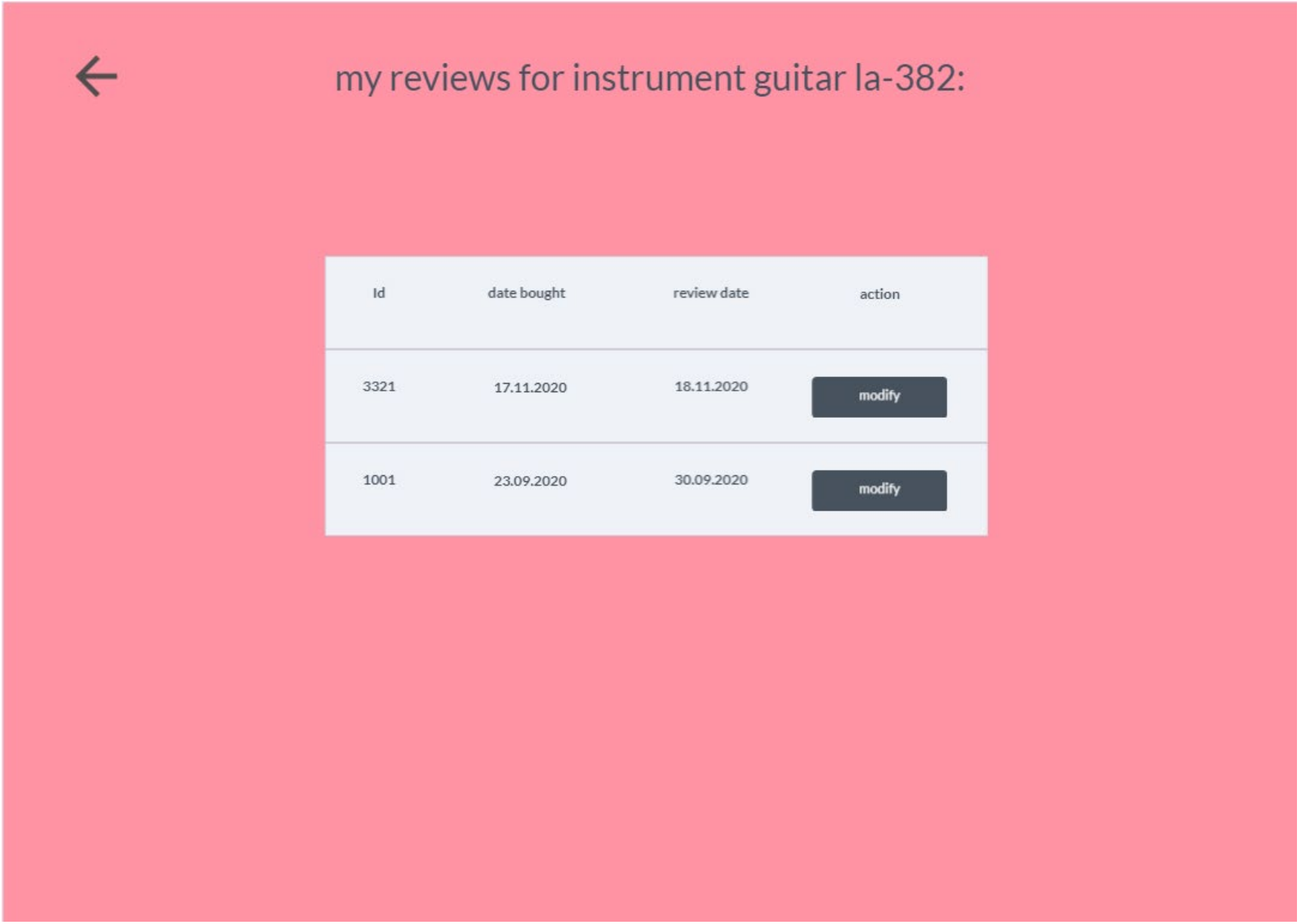


Figure 3. My Reviews screen

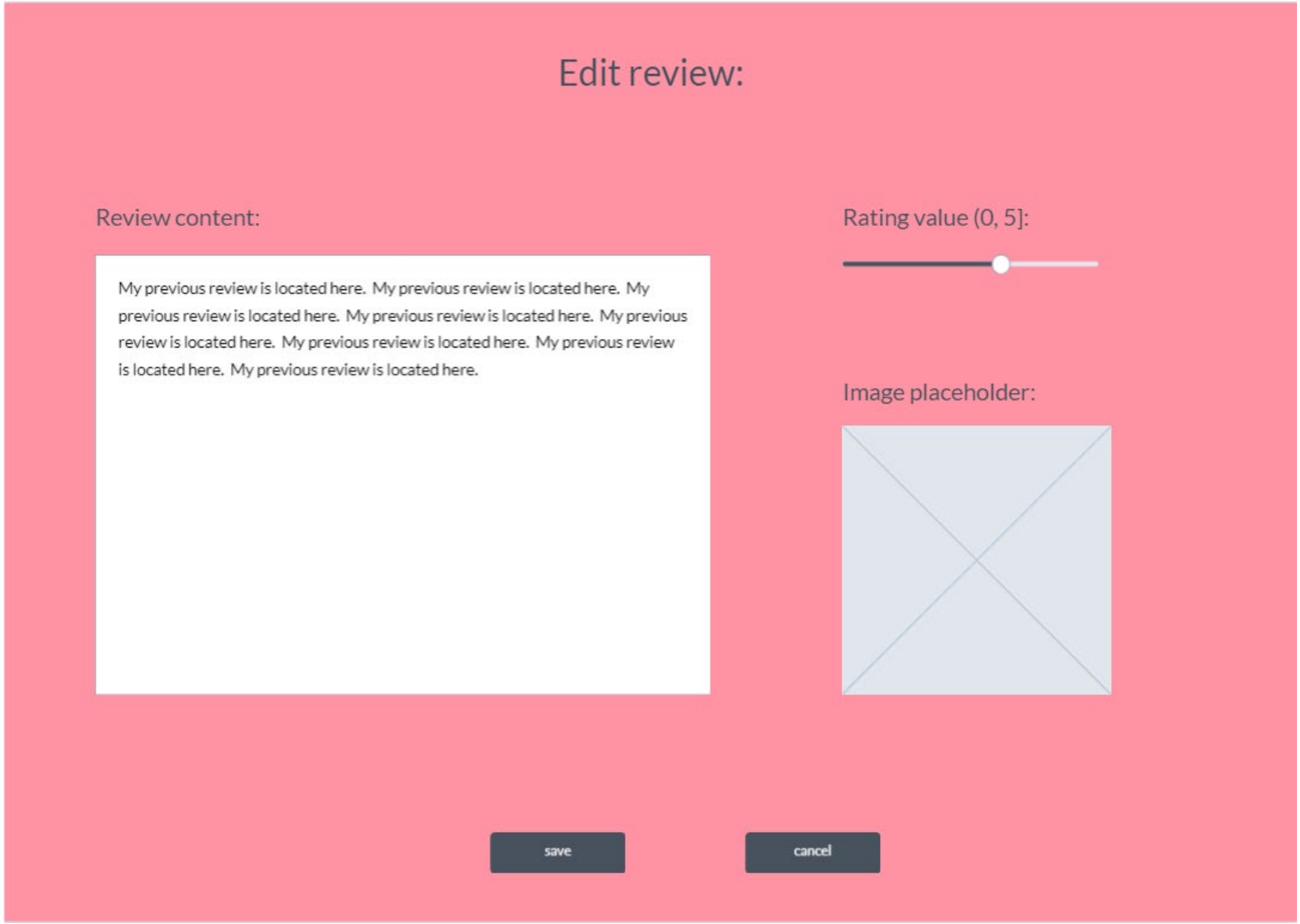


Figure 4. Edit Review screen

10. The discussion of design decisions and the effect of dynamic analysis

Dynamic analysis exposed a need to add a helper method related to a Review entity(means that these methods will be logically connected with the Review model, however the method will not necessarily be located exactly in the Review class). Introduced method is called: *GetAllReviewsByCustomerId(int customerId)*, most likely it will be located in a corresponding repository. **Other methods for CRUD operations are not listed on the diagrams because they are assumed to be trivial**.

Implementation of the **overlapping** and **dynamic** inheritances in case of classes *Worker* and *Customer* inheriting from *Person{abstract}* class was decided to be implemented in a flat structure, namely putting everything together in one class called *Person* and restricting usage of specific fields and methods by some metadata being present in every instance of a class as an enum type. For **overlapping** inheritance purposes there was added a discriminator field *PersonType*. For **dynamic** inheritance two methods were added *AddRole(PersonType type, Person person)* and *RemoveRole(PersonType type).*

In case of **multi-inheritance**, namely class *ElectricGuitar* inheriting from *Electric{abstract}* and *StringInstrument{abstract}*, the inheritance was decided to be implemented using normal inheritance from class *StringInstrument{abstract}* as well as adding and later inheriting from an interface *IElectric*.

As a result of dynamic analysis it is possible to state that chosen use-case (edition of a review left by a customer) has four alternative flows. One of them leads to termination of the use-case, and three bring flow several points back. One of the alternative flows is used for validation, to inform user in case of constraint violation and bring flow back to the moment of modification without saving the new state of the object. For the selected use-case there

were no new methods added to any of the business models explicitly, because of the fact that all the logics will happen in the corresponding repositories, that is why I do not consider showing these methods on the class diagrams as a necessity.

For the UI prototyping MarvelApp web platform was used. A minimalistic approach was chosen not to overkill with functionality and thus not to confuse the user. The interface is intended to be as intuitive as the functionality it is supposed to go along with. Several elements and colors are there in use, so they are reusable and sufficient for now. Soft pinkish color was used as a background to make the user feel the coziness and the easygoingness of the service the user intends to use.

Done by Adrian Chervinchuk s20357.