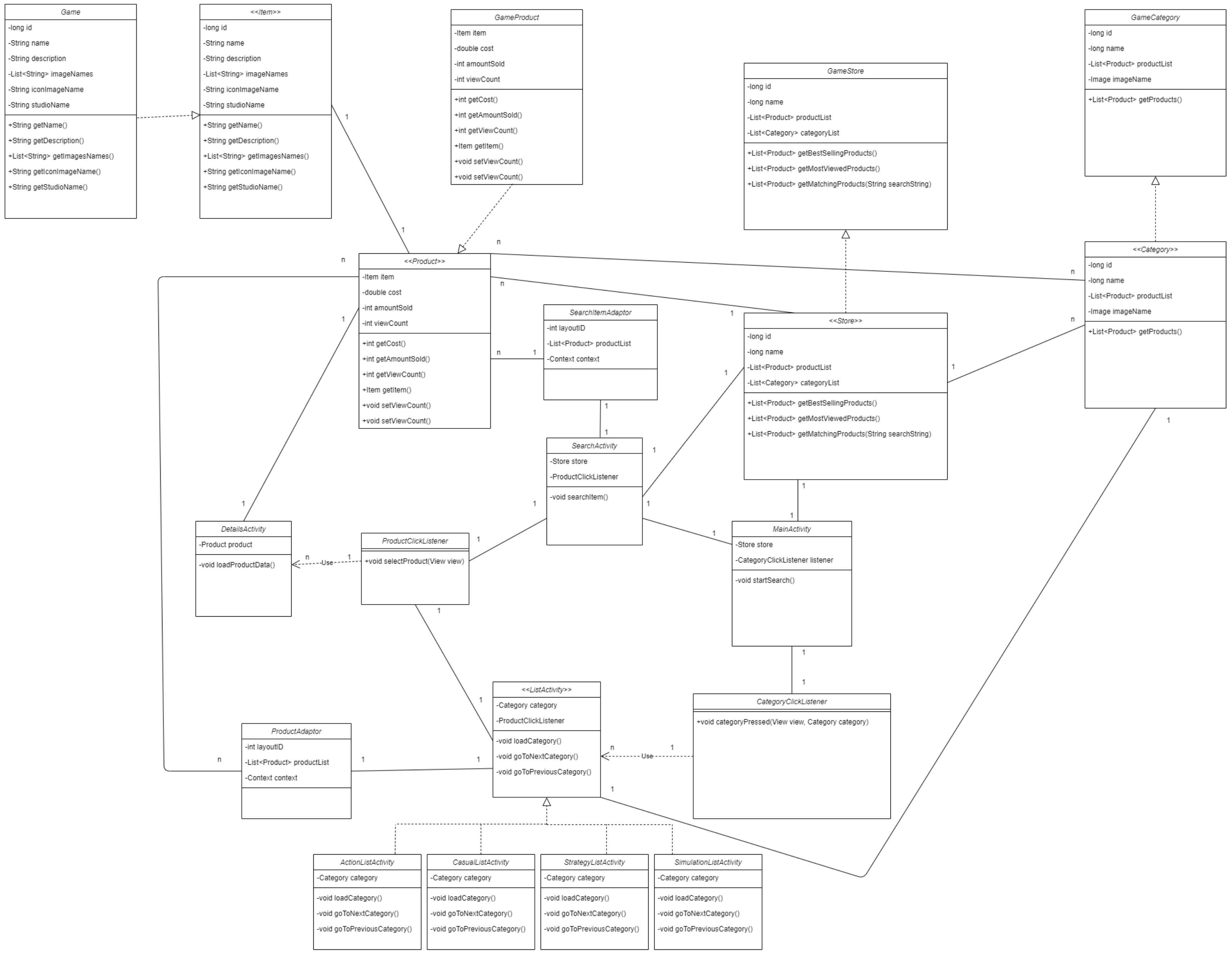
SOFTENG 306 Final Report

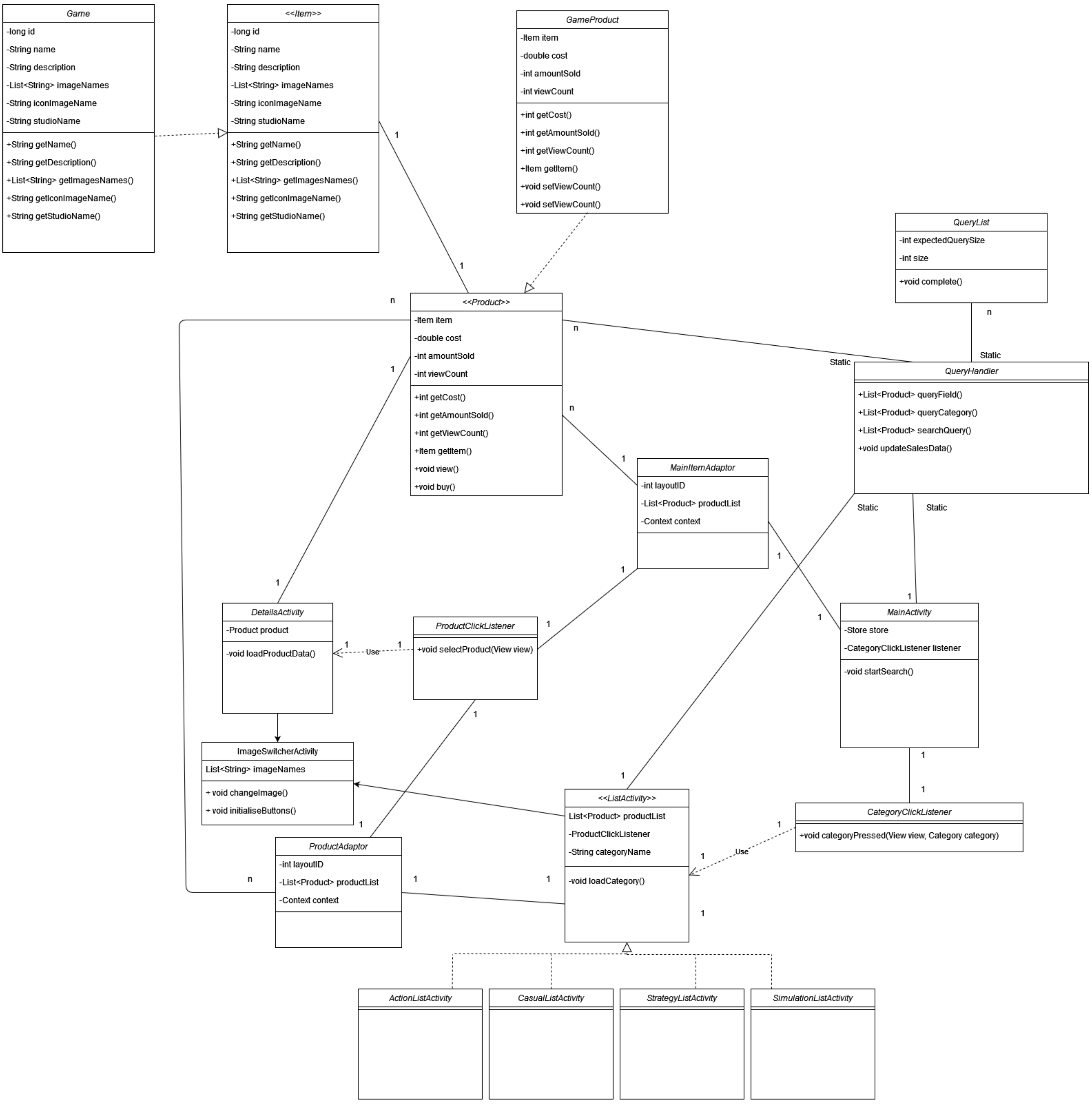
Group 24

Dylan, Braden, Jack

Initial Class diagram



Final Class diagram



Solid principles

Single Responsibility Principle

The application is roughly divided into the model, view, and controller layer.

In the model layer, the Game class is separated from the GameProduct class. The Game class contains information about the game (for example, the name of the game, and the category of the game) and the product class contains information about the sales (for example, price, and view count). Unlike in the initial design doc, the model classes have some extra methods such as writeToParcel() and createFromParcel(), moreover the interfaces are now Parcelable. This is because we realised we needed to pass the objects into intents. Furthermore, we removed the category class because we felt that it was redundant.

In the view layer, the main activity, list activity, and details activity each have their own individual xml file, and each of these files only contain information pertaining to the view layer. Unlike our original design, we removed the search activity. We found it made for a smoother experience to search directly from the MainActivity, reducing the number of clicks required to reach the desired function for the user, and making their experience more intuitive. Our GUI is similarly faithful to the mockup. The only addition is colour variation between list items in different categories in order to meet project requirements.

In the control layer, we have a class from each activity. For example, we have MainActivity, Details Activity, and a class for each of the different category list activities. Generally, these classes are only responsible for generating the components and adding things like button functionality and transitions.

We also had to add new classes such as ImageSwitcherActivity in order to implement the image switching functionality. This was because the image switching functionality was used in both the DetailsActivity and CategoryListActivity. Thus, extracting this code and having these classes inherit from ImageSwitcherActivity allowed us to reuse the same code for both. This helped DetailsActivity and CategoryListActivity follow the single responsibility principle and it also helped reduce repetition.

Due to the removal of the search activity, we renamed the SearchItemAdaptor to MainItemAdaptor.

In the initial design doc, we had the GameStore app handle all the queries, however we decided to rename it to QueryHandler as we felt that that name was more intuitive.It is inefficient to query for all games when the app is launched, as users are unlikely to want to view every game in the store. For a very large store, it may not even be feasible. We realised it made more sense to make queries directly to the database, so that games are only in local memory when needed. To accomplish this, the QueryHandler class was created with static methods for more specific queries. All query functionality is in this class, rather than distributed between individual activity-specific classes, as we felt this reduced coupling between query handling and UI.

Open Closed Principle

As specified in the initial design doc, all activities have dependencies to interfaces instead of the concrete version of a model class. For example, activities will have dependencies to Product, Store, and Category instead of GameProduct, GameStore, and GameCategory. This makes it so that if there are any new types of concrete classes that are added we can easily add in those classes and have them work with the rest of the code without making any significant changes.

ListActivity is an interface, allowing future category implementations to have significant differences if needed. The game model is represented by an interface so that future game implementations should require no changes to the existing one. We didn’t add any new categories during development, however if we chose to do so, adding a new category would be relatively simple.

Liskov Substitution Principle

All implementations of each of our interfaces should behave the same way externally. For example, all our ListActivity implementations should, and will, use the same adapter.

Dependency injection is used in most of the classes. For example, the Category class, search ItemAdaptor, and ProductAdaptor all use a List to store the products. That way, many different types of lists can be used based on the requirements.

Furthermore, interfaces such as Product are dependent on other interfaces such as Item, that way if we ever decide to add new types of items and new types of products, the new code will easily work with the existing code. As shown in the class diagram, all subclasses implement all the methods in the super class.

Interface Segregation

Interfaces should only include the functionality required. Many of our classes implement interfaces, but it seems unnecessary to fully adhere to this principle when the scope of this project is so limited, and it is known that once completed, it will not be expanded. For this reason, we do not consider it so important. Nevertheless, every class only has dependencies that are necessary. For example, the DetailsActivity only has a dependency to the Product class so that it can access and present the product information. It uses and needs every method inside of Product, therefore it is following interface segregation principle.

Dependency Inversion

The classes in the model layer all have interfaces. For example, there is the Product interface, Store interface, and the Category interface. The activities in the application interact with the interfaces instead of the concrete class. ListActivity is also an interface so that we can implement different functionality for the different list activities if required, and also easily implement more list activities based on changing requirements. The GameProduct class in particular had a lot of dependencies, which was why it was important to create the Product interface.

The principle was not applied to most of the other activity classes and adapter classes because it would dramatically increase the complexity of the project due to the large number of classes. We believed that if necessary, we could implement the interfaces later on when required and it wouldn’t take too much effort to do so. During development, we felt this was the right choice because we didn’t have to waste time creating interfaces that we weren’t going to use.

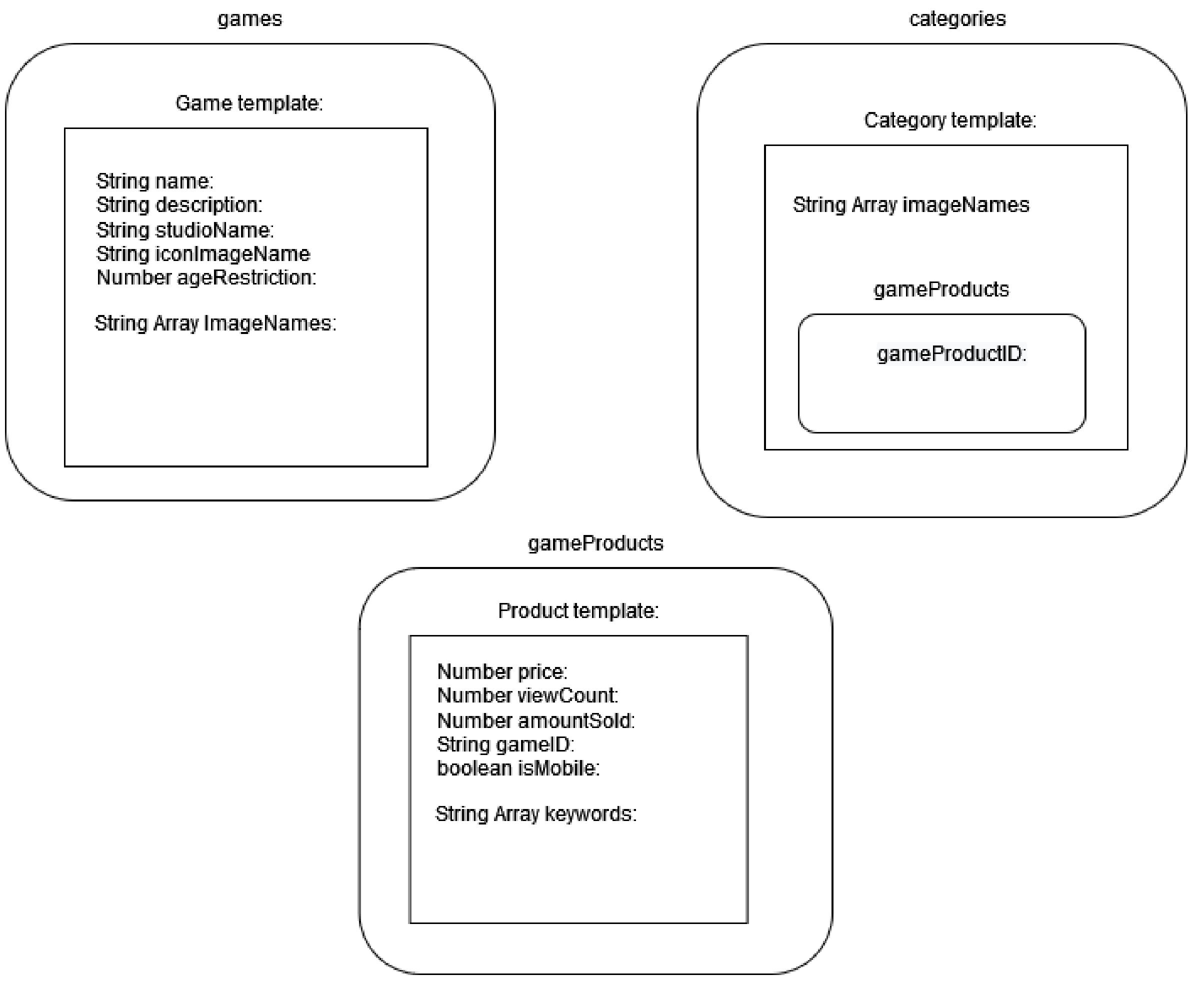
Naming Conventions

We used the camelcase convention for naming our variables. The names for all our variables were similar to that of the class diagram, which allowed our code to be consistent.

Package Use

Within our project, there were a series of packets that all of the code was put into in order to help keep everything organized and make it easy to find any one thing. These consisted of the activities package, containing all of the activities, the adaptor packages for adaptors, the data package for all database and query related classes, the listener package for the categoryClick and productClick listeners and model for the implementation of the game and game product classes. Whenever we had interfaces that were being implemented by a class, we put this interface into the same package that its implementation was in.

Database Schema



The document database schema was implemented as specified in the design document, with two minor additions - the ageRestriction and isMobile fields. These were added to introduce more variation between different categories, and are visible in the action and casual categories, respectively.