SOFTENG 306 Project 2 - Quality Report

Team 10
Team members: Simon Cheng, Josh Lim, JinKai Zhang

Inconsistencies

The new UML Diagram

There is an update in the UML Diagram, please refer to Appendix A for the comparison between the old one and the new one.

Explanation of inconsistencies

Activity Class

Initially, we thought we could use ListActivity as an abstract class to get rid of code repetition because ShoppingCartActivity, SearchActivity, CategoryActivity, and TopPicksActivity show a list of cards. However, we realized we could not as each activity requires a different onCreate method, and TopPicksActivity uses a different adapter for smaller cards. As a result, we changed ListActivity to a concrete class and replaced CategoryActivity with it.

Adapter Class

Due to the unfamiliarity of Android Studio, we have used ListView at first. However, we encountered a problem when we wanted to implement a top pick list that scrolls horizontally. This is not possible with ListView, hence we have decided to use RecyclerView instead. When we wanted to have a better layout for the UI, we also noticed that using ScrollView with ListView will end up with two scrolling components, so we chose to use RecyclerView for all adapters.

As a result, we changed ElectronicDeviceAdapter to ProductItemAdapter, added TopPickListAdapter, and ImageSliderPagerAdapter. ProductItemAdapter is used for displaying products in category list, TopPickListAdapter is used for top picks list, and ImageSliderPagerAdapter is used for image sliders. These are required because each adapter uses a different view component.

Model Class

At the beginning, we did not decide how to collect photos and choose top pick list products. The solution we found was by adding image fields in the ElectronicDevice class for retrieving the url link. For choosing top picks, a rating field is created and top-picked products are decided based on their rating.

In order to reflect the visual differences of each concrete product, we added a method to retrieve the background colour id of a category. This is used when we are constructing a card for a product. The background color id is retrieved and searched in the context. We also added the method populateSpecsTable() to generate a table for product specification in detail views.

Database

The images are uploaded to firebase cloud storage and store the access link in the image fields inside product documents.

The current design simplifies the shopping cart section as we only consider one user. We used one collection for the shopping cart to upload/remove the product when the user clicks the add_to_cart/remove_item button.

UI

Please see Appendix B to view the comparison between the prototype and the final product. We made the changes in UI to better follow the material design principles, ultimately enhancing user experience.

SOLID Principles

Single Responsibility Principle (SRP)

"A class should be responsible to one actor" and "a method should only perform one task". We have the following practices that follows SRP:

Activity and content view
We are limiting to one content view per activity to ensure that SRP is being followed. Each activity is responsible for initialising its own content view's components and their events.

Card view components
Each card component is separated into its own xml CardView component. This will not only allow us to reuse the card component, but also make sure that they only serve the purpose that they are created for. For example, the *product_list_view_item.xml*

NavigationListener
The NavigationListener class has the responsibility of handling all events that are happening in the bottom navigation bar.

RecyclerView adapter
We have one RecyclerView adapter per card view or pager, such as TopPickListAdapter for the top pick recycler view and ProductItemAdapter for the category activity view.

Open Closed Principle (OCP)

"software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification"

Use of IProduct Interface and Abstract class ElectronicDevice

The hierarchy and inheritance relationship of the Model Classes complies with OCP principles. The highest-level abstract class ElectronicDevice contains properties and methods common to all products. The middle layer abstract classes further refine the extra properties of products in different categories. Concrete products are implemented by inheriting the middle layer classes.

When a new product needs to be added, we can directly extend it from the middle layer class if the product's category belongs to any existing one. Otherwise, a new category abstract class can be extended from ElectronicDevice, and the product class can then be implemented by extending it. Therefore, for any new requirement, a change in a particular concrete class can be avoided.

Liskov Substitution Principle (LSP)

The principle suggests that objects of a superclass shall be replaceable with objects of its subclasses without breaking the application. In this project, we have the following practice that follows this principle.

A List of IProduct

In the implementation of obtaining data from the firestore database, we define a list of types of **IProduct**, and any subclass of IProduct will be able to be added in this list.

```
List<IProduct> productList = new LinkedList<IProduct>();
// Code omitted
productList.add(product.toObject(Tablet.class));
```

Code segment in CategoryActivity

Minimise the occurrence of overriding superclass's method in subclass
The `getBackgroundColorId()` is the only method that was overridden in the Laptop, Phone and Tablet class. It was defined as an abstract method in ElectronicDevice and handed over its implementation to its subclasses. This method is made to get different background colours corresponding to each different category, such as yellow for Tablet, orange for Laptop etc. In this specific case of overriding this superclass's method would allow us to benefit from using the polymorphism feature in java to reduce code duplication in the implementation of setting different category's product a different background colour.

```
Product product = products.get(holder.getAdapterPosition());
// Code omitted
holder.productCard.setCardBackgroundColor(context.getColor(product.getBackg
roundColorId()));
```

Code segment in ProductItemAdapter

Interface Segregation Principle (ISP)

ISP states that no client should be forced to depend on methods it does not use. In our design, we managed to apply it with proper design of the IProduct Interface. To avoid the problem stated in ISP, we try to use multiple dedicated interfaces/subclasses instead of using a single total interface.

For instance, IProduct only contains methods that are universal to products. In our domain model, Phone and Tablet have a front camera however laptop does not. In this case, we exclude the `getFrontCamera()` method from the IProduct interface and put it into the MobileDevice class. Therefore, Laptop - a direct subclass of ElectronicDevice is not forced to implement the `getFrontCamera()` method.

By excluding unneeded methods from the IProduct interface, at the same time allowing to achieve ISP, it also enables better encapsulation among classes.

Dependency Inversion Principle (DIP)

By definition, DIP suggests that

1. High-level modules should not import anything from low-level modules. Both should depend on abstractions (e.g., interfaces).
2. Abstractions should not depend on details. Details (concrete implementations) should depend on abstractions.

In short, it can be summarised as - "The code of the high-level logic layer should not be changed due to changes in the code of the underlying module."

IProduct Interface

In practice, we encapsulate all Model classes dependencies (methods) in the IProduct interface and let ElectronicDevice class implement this interface. The methods will be implemented by the subclasses of ElectronicDevice class. Standardize the methods in underlying classes by defining new methods in the IProduct interface. In turn, all high-level modules thus present details independent of the implementation of low-level modules.

The populateSpecsTableFromParent() Method
For example, the `populateSpecsTableFromParent()` method is used to adjust the displayed format in the detail view by using the java reflection mechanism. The method is defined in the IProduct Interface and implemented in the ElectronicDevice Class. The subclasses of the ElectronicDevice can call this method directly instead of defining the method in their own classes. In this way, all low-level classes are dependent on their higher level class to implement the method.

Good Practices

We also maintain the following good practices in our project.

Readability

The whitespace between lines will allow the reader to view the code in smaller chunks, providing a better flow to the code (Appendix C). Additionally, we have descriptive package names, class names, method names, variable names for better understanding of modules (Appendix C & D)

Avoid design smell

To avoid design smell, we have used a utility class, DataUtil as a data provider to prevent code repetition. As a result, it allows us to have better modifiability as we only need to make changes to one class if required. We can also ensure that our data provider works by having a single class providing the functionality. The use of interface IProduct also allows us to avoid design smell as we are following the SOLID principles as mentioned above.

Comments and documentation

There are comments on top of a java class that describes a java class's purpose, comments on top of a method that describes its functionality, and inside the methods to explain what each chuck of code means. The good practice of commenting and documentation show our professionalism as software engineers. (Appendix C)

Consistency

Referring to the Java class naming and the Android resource naming convention, we use camel case for Java package and class; and underscore for xml classes.

The consistency of file naming enables better modifiability, since the name conventions are the same. If we have the convention of both underscore and camel case in a single class, it will be hard to add new methods or even to use it. This is because there might be existing methods that you did not notice due to different naming conventions. Therefore, keeping a consistent naming convention will prevent any confusion when needing to maintain and modify the source code in the future. (Appendix D)

UI design

Our final UI design follows the material design principles. For our colour scheme, we use a primary colour scheme of blue with three different layers of textures to achieve the goal of *"Color themes are designed to be harmonious, ensure accessible text, and distinguish UI elements and surfaces from one another."* that was suggested in the principle. (Appendix B)

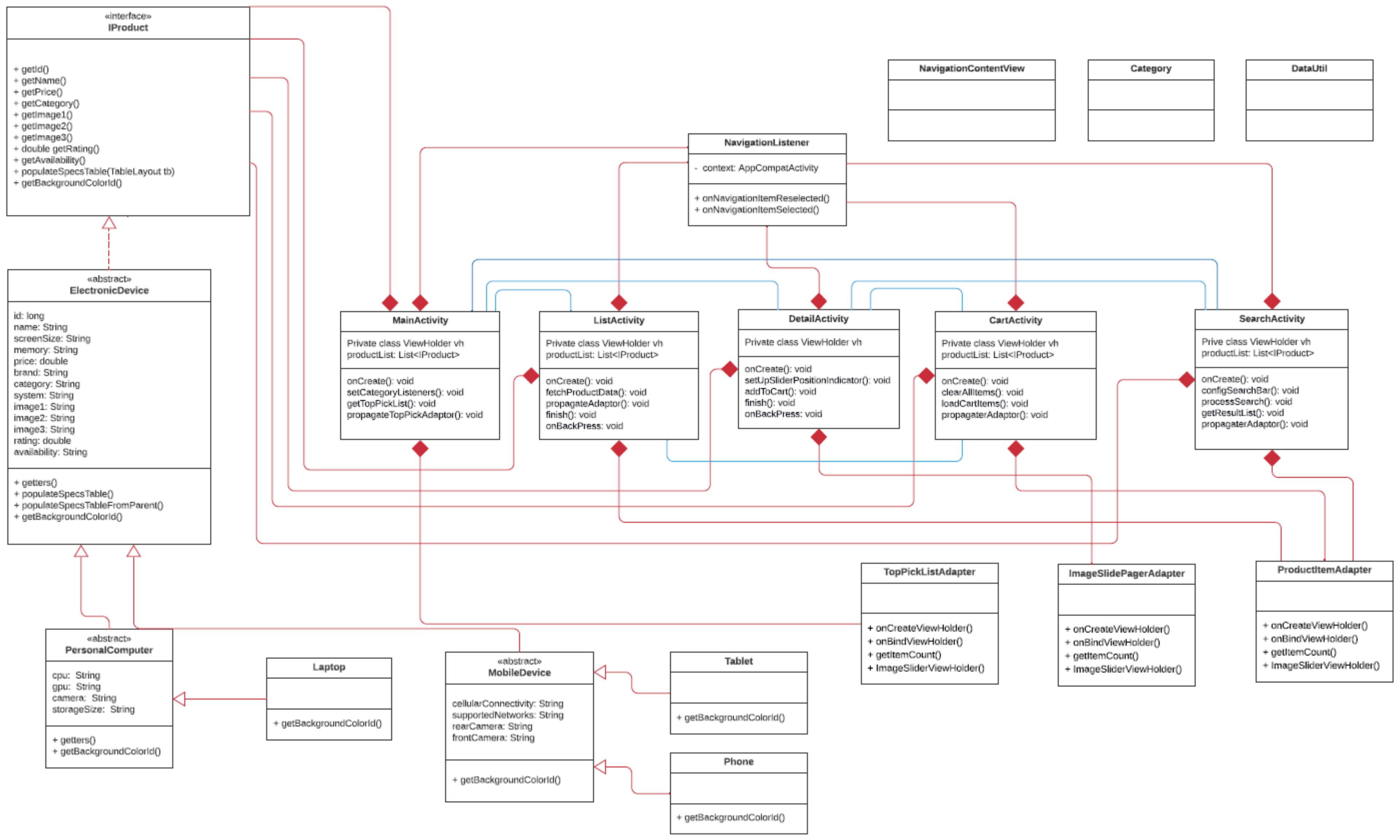
In addition, the font type scale in our application is also referring to the material design principle of designing fonts that are optimized based on Material typography guidance for readability. For instance the “Top Picks” in the Appendix B.2 are has a font size of 36 dp, which it is size for headline 6 suggested by the type scale for the font family Rubik

Ultimately, this will allow our final product to provide a better user experience.

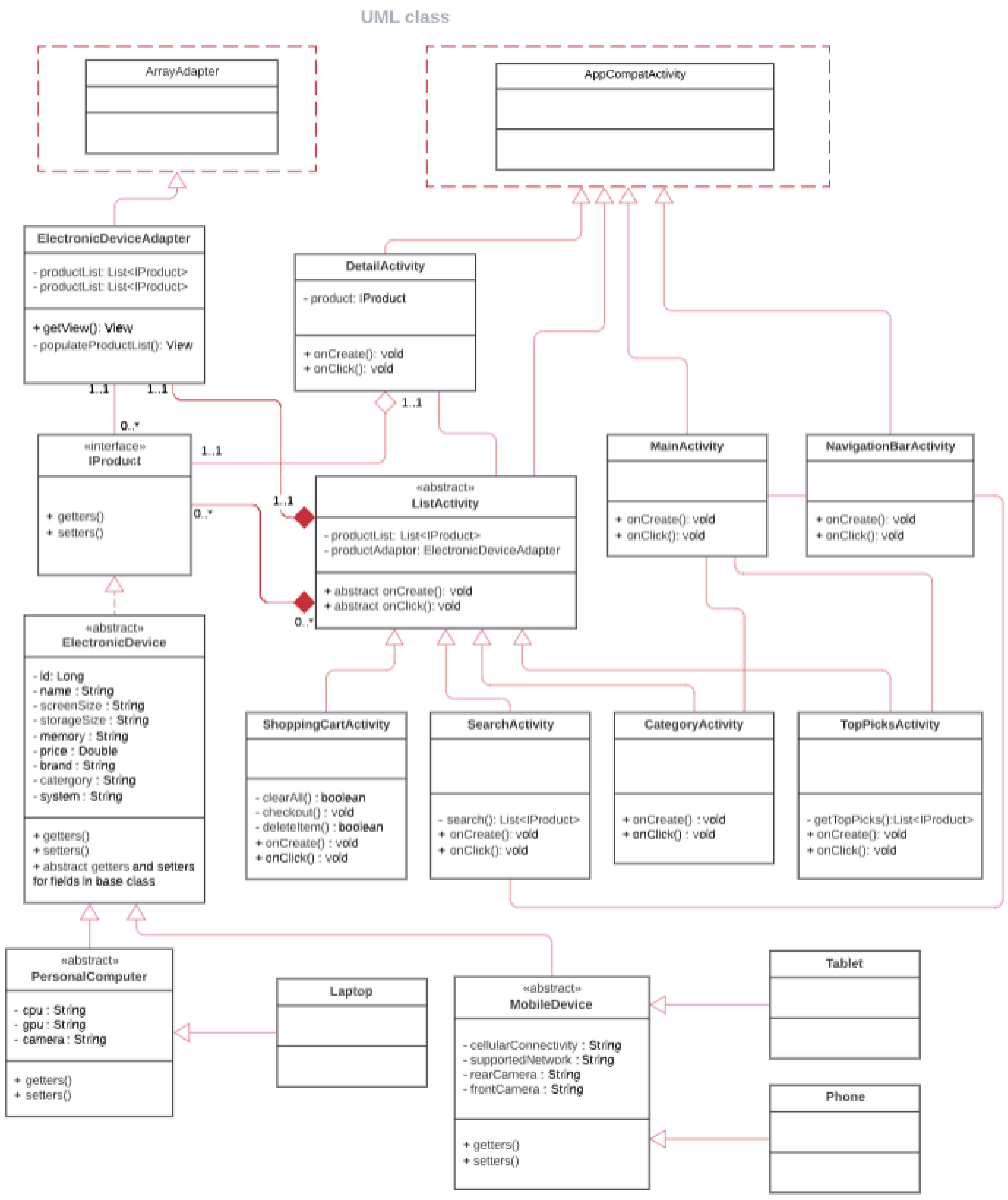
Appendix

Appendix A - Comparison of UML Diagrams

Appendix A.1 - The New UML Diagram

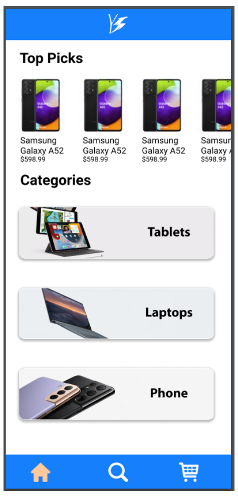


Appendix A.2 - The Old UML Diagram



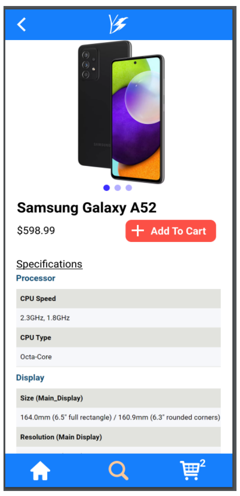
Appendix B - Comparison of prototype and final product

Appendix B.1 - Prototype in the first deliverable


Top Picks

Samsung Galaxy A52 $598.99 | Samsung Galaxy A52 $598.99 | Samsung Galaxy A52 $598.99 | Samsun Galaxy $598.99

Categories


Samsung Galaxy A52

$598.99

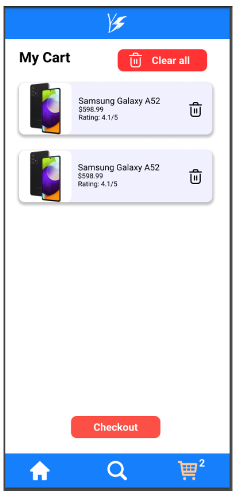
Specifications

Processor

CPU Speed
2.3GHz, 1.8GHz
CPU Type
Octa-Core

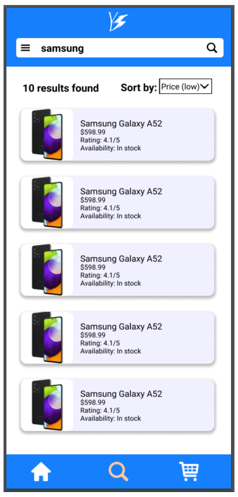
Display

Size (Main_Display)
164.0mm (6.5" full rectangle) / 160.9mm (6.3" rounded corners)
Resolution (Main Display)


My Cart

Samsung Galaxy A52
$598.99
Rating: 4.1/5

Samsung Galaxy A52
$598.99
Rating: 4.1/5


samsung

10 results found Sort by: Price (low)

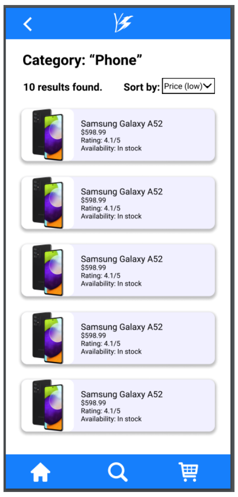
Samsung Galaxy A52
$598.99
Rating: 4.1/5
Availability: In stock

Samsung Galaxy A52
$598.99
Rating: 4.1/5
Availability: In stock

Samsung Galaxy A52
$598.99
Rating: 4.1/5
Availability: In stock

Samsung Galaxy A52
$598.99
Rating: 4.1/5
Availability: In stock

Samsung Galaxy A52
$598.99
Rating: 4.1/5
Availability: In stock


Category: "Phone"

10 results found. Sort by: Price (low)

Samsung Galaxy A52
$598.99
Rating: 4.1/5
Availability: In stock

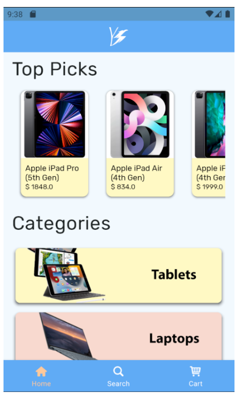
Samsung Galaxy A52
$598.99
Rating: 4.1/5
Availability: In stock

Samsung Galaxy A52
$598.99
Rating: 4.1/5
Availability: In stock

Samsung Galaxy A52
$598.99
Rating: 4.1/5
Availability: In stock

Samsung Galaxy A52
$598.99
Rating: 4.1/5
Availability: In stock

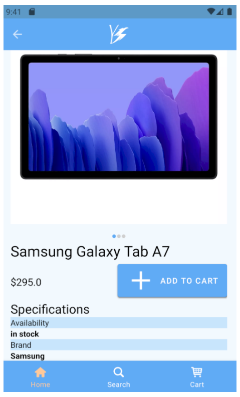
Appendix B.2 - Final Product



Top Picks

Apple iPad Pro (5th Gen) $ 1848.0 | Apple iPad Air (4th Gen) $ 834.0 | Apple iP (4th Ger $ 1999.0

Categories



Samsung Galaxy Tab A7

$295.0

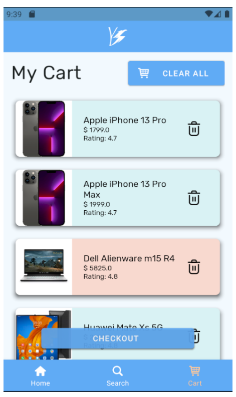
Specifications

Availability

in stock

Brand

Samsung



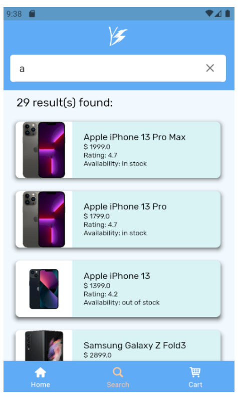
My Cart

Apple iPhone 13 Pro
$ 1799.0
Rating: 4.7

Apple iPhone 13 Pro Max
$ 1999.0
Rating: 4.7

Dell Alienware m15 R4
$ 5825.0
Rating: 4.8

Huawei Mate Xs 5G



a

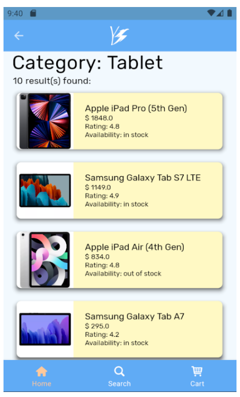
29 result(s) found:

Apple iPhone 13 Pro Max
$ 1999.0
Rating: 4.7
Availability: in stock

Apple iPhone 13 Pro
$ 1799.0
Rating: 4.7
Availability: in stock

Apple iPhone 13
$ 1399.0
Rating: 4.2
Availability: out of stock

Samsung Galaxy Z Fold3
$ 2899.0



Category: Tablet

10 result(s) found:

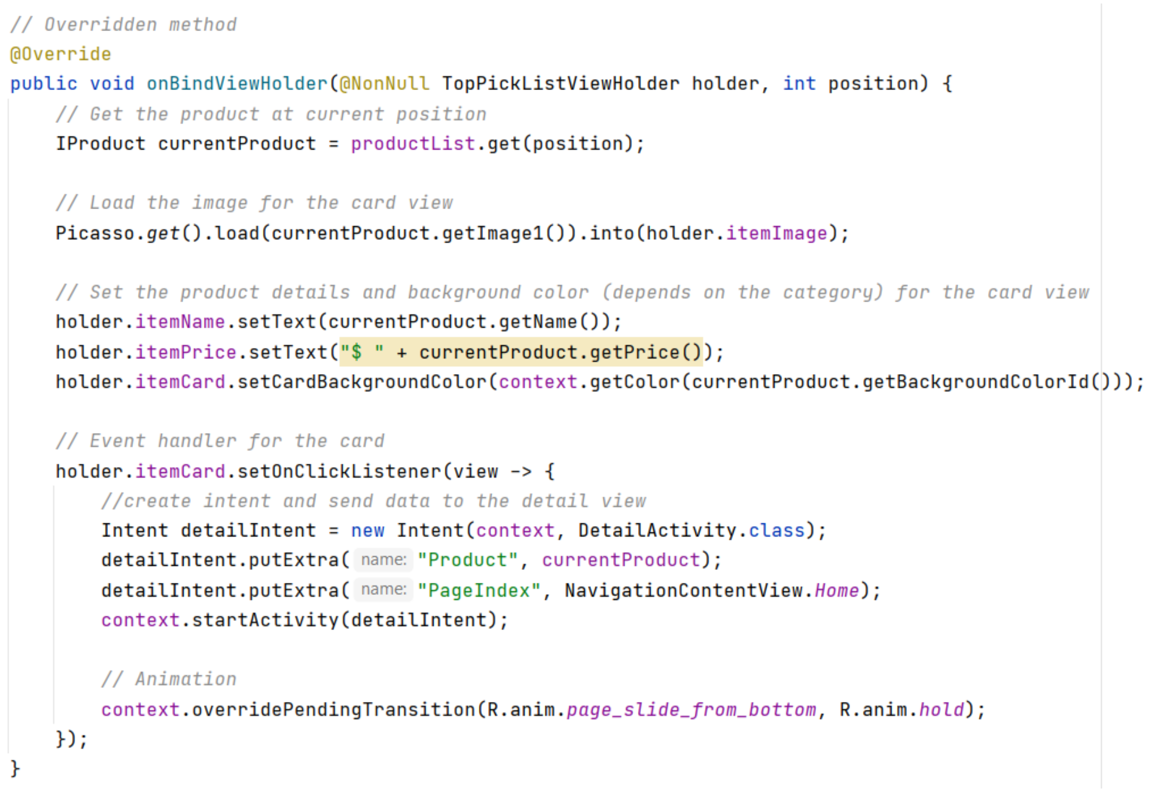
Apple iPad Pro (5th Gen)
$ 1848.0
Rating: 4.8
Availability: in stock

Samsung Galaxy Tab S7 LTE
$ 1149.0
Rating: 4.9
Availability: in stock

Apple iPad Air (4th Gen)
$ 834.0
Rating: 4.8
Availability: out of stock

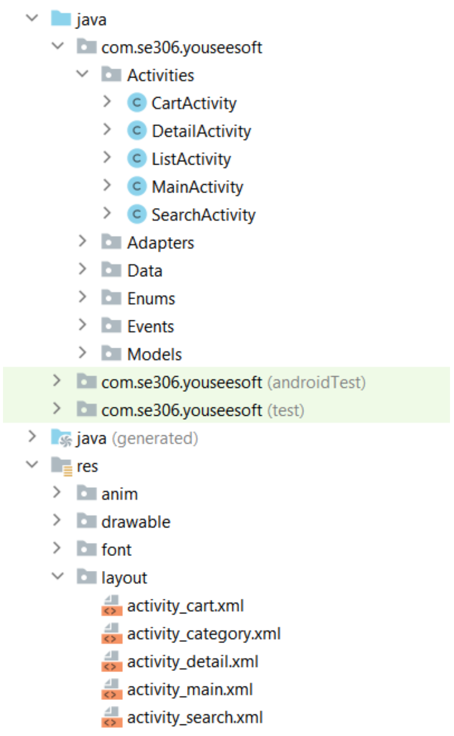
Samsung Galaxy Tab A7
$ 295.0
Rating: 4.2
Availability: in stock

Appendix C - Code segment in the TopPickListAdapter class



```java
// Overridden method
@Override
public void onBindViewHolder(@NonNull TopPickListViewHolder holder, int position) {
    // Get the product at current position
    IProduct currentProduct = productList.get(position);

    // Load the image for the card view
    Picasso.get().load(currentProduct.getImage1()).into(holder.itemImage);

    // Set the product details and background color (depends on the category) for the card view
    holder.itemName.setText(currentProduct.getName());
    holder.itemPrice.setText("$ " + currentProduct.getPrice());
    holder.itemCard.setCardBackgroundColor(context.getColor(currentProduct.getBackgroundColorId()));

    // Event handler for the card
    holder.itemCard.setOnClickListener(view -> {
        //create intent and send data to the detail view
        Intent detailIntent = new Intent(context, DetailActivity.class);
        detailIntent.putExtra( name: "Product", currentProduct);
        detailIntent.putExtra( name: "PageIndex", NavigationContentView.Home);
        context.startActivity(detailIntent);

        // Animation
        context.overridePendingTransition(R.anim.page_slide_from_bottom, R.anim.hold);
    });
}
```

Appendix D - Consistency of naming conventions



```
java
  com.se306.youseesoft
    Activities
      CartActivity
      DetailActivity
      ListActivity
      MainActivity
      SearchActivity
    Adapters
    Data
    Enums
    Events
    Models
  com.se306.youseesoft (androidTest)
  com.se306.youseesoft (test)
java (generated)
res
  anim
  drawable
  font
  layout
    activity_cart.xml
    activity_category.xml
    activity_detail.xml
    activity_main.xml
    activity_search.xml
```