Shopping List Android

Albena Strupchanska	9332456
Alejandra Sánchez Bolaños	6699308
Luheng Wen	6812910
Klajdi Karanxha	6173780
Meng Jia	6421555

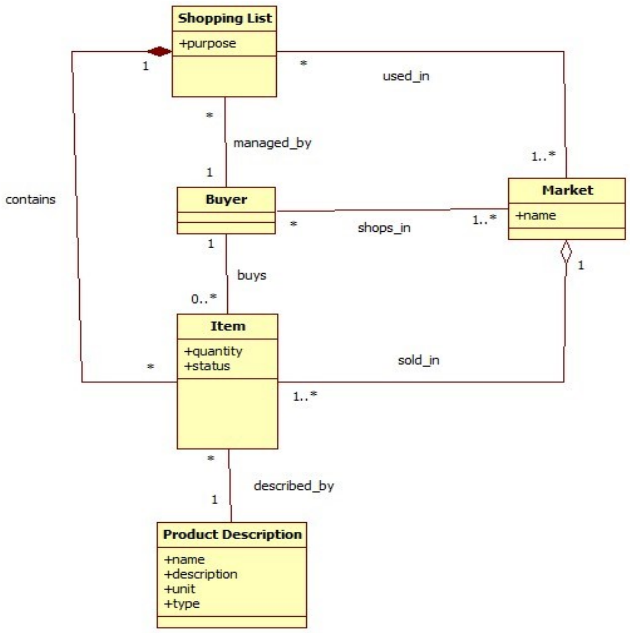
Due: March 3^{th}

Summary of Project

Shopping List is an open source mobile application that helps the user store, export, and exchange, manage and categorize their shopping items in order to simplify their shopping experience. The project is open source and targets mobile devices running the Android mobile operating system. The project was started by Mr. L. Prelle around one year ago and the repository is maintained in Github (https://github.com/prellele/shoppinglist-android). So far the application has only one active developer and our team is planning to contribute to it within the frame and aligned to the goals of this academic course. The project is currently in alpha version, available for distribution at the Google Play store and has room for a lot of flexibility and changes, it also has room for the application of a more regulated architecture and could be targeted in order to improve the maintainability by implementing many design patterns.

Class Diagram of Actual System

Conceptual Diagram

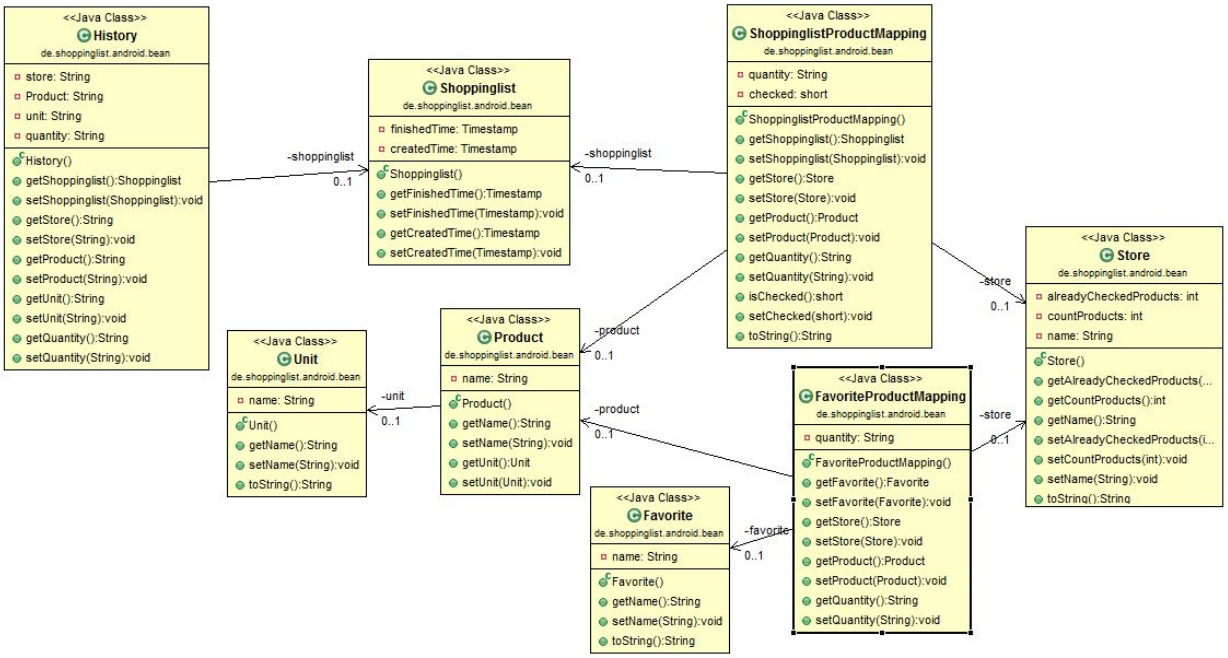


Application Overview and Class Diagram

The bean package has a description of important domain classes, but it doesn't show their relationships in terms of functionalities of the system. The main application functionalities are implemented in the activity package. We noticed that in this package, we have references to GUI classes (mainly in the adapter package), classes defined in the bean package and classes performing an access to the data storage (datasource package). We realized that we need not only the bean package but also the activity one in order to understand the application logic.

We have used ObjectAid reverse engineering tool to explore the application and generate the (partial) application class diagram. We noticed one disadvantage of the tool, mainly it doesn't show the associations between the inline classes and the actual class.

We have done some simplifications to the application complete class diagram, obtained by ObjectAid, in order to show only classes related to the application logic. We notice that the BusinessBean class is an abstract class, which is implemented by all other classes in the package. Since this is the case, we ignore (delete) the BusinessBean from the diagram. We have done the same to all GUI classes.



There are 8 domain classes in the actual architecture: Shoppinglist, ShoppinglistProductMapping, FavoriteProductMapping, Store, Product, Unit, Favorite and History. In general, associations between classes represent instance variables that hold references to other classes. We noticed that in our case, all associations hold references to either 0 or 1 class. Shoppinglist has been used by History and ShoppinglistProductMapping, thus a change in it would probably lead to a change in both classes. ShoppinglistProductMapping also

depends on Store and Product. This class has fields quantity and checked. Product holds a reference to Unit. FavoriteProductMapping depends on Product, Store and Favorite. These “mapping” classes contain similar information and have been used only to make indirect distinction between Favorite and Shoppinglist, which conceptually are both shopping list but one is static and the other dynamic (exist “checked” attribute).

There are no hierarchical connections between classes, instead all classes inherits directly from the abstract class (BusinessBean in our case, which was removed from diagram to improve readability).

In general, the relationships between classes are not clear from the architecture itself. Moreover, they seem in the opposite direction comparing to our domain model. We have reviewed the activity classes in order to derive real relationships. It turned out that there is no relationships in between classes (terms of OO) but rather pulling of information from the database in order to create the relationships.

Comparison of Diagrams

In our conceptual diagram, there are 4 conceptual classes, while in the real architecture there are 8 classes. So we will compare them at several levels.

1. Conceptual comparison
    - Shoppinglist class has the same meaning as our concept ShoppingList in terms that it keeps record of products and stores, but not in terms of relationship to other classes and attributes. Our conceptual class has an attribute “purpose”, whereas the real class has “createdTime” and “finishedTime”.
    - Favorite class has no correspondence to a conceptual class in our domain model. It is basically an static shopping list, so a kind of description class for a shopping list. This made us notice that we have missed to add a description class for the shopping list in the domain model for the sake of avoiding any information loss.
    - Product class corresponds to our ProductDescription concept. In terms of attributes they have one similar attribute “name”. Unit is an attribute of our conceptual class Product Description, whereas in the system, it is an actual class. This has some advantages as Unit is independent and easy for change and extension. Our concept has two more attributes that may lead to more functionalities, but they are missing in the real one.
    - Store class has the same meaning as our concept Market. It has more attributes concerning the status of products assigned to it .
    - ShoppinglistProductMapping class is conceptually similar to our Item concept. They have same attributes: quantity and checked (status). However in terms of relationships they are different. ShoppinglistProductMapping holds a reference to 0 or 1 Store, Product and Shoppinglist. In our domain model we assumed that an Item is sold in one Market, it belongs to one ShoppingList and is described by one Product Description. So we assumed that Item doesn’t exist without Market and Product Description. In real

application, ShoppinglistProductMapping could exist without being related to Store and Product.

- FavoriteProductMapping class is conceptually similar to our Item concept as it is related to Store (our Market) and Product (our Product Description). Moreover it is similar to ShoppinglistProductMapping. It only missed the checked (status) attribute and has get/set methods for Favorite, instead of for Shoppinglist. So it looks like a static version of the ShopppinglistProductMapping class.
- History has no correspondence to concepts in our domain model.

2. Functional comparison

Since there are 3 extra classes (Favorite, History, Unit) in actual diagram, which have no direct correspondence to our conceptual diagram, we can assume that each of those classes provides extra functionalities. The actual diagram has benefit in terms of presenting Unit as a class instead of a variable/attribute as it allows the user to create customized unit instance by manipulating methods within the Unit class. The advantage of Favorite class has already been mentioned above. One advantage of a History class could be reusing of a old shopping list to create a new one, but this was not addressed in the application. On the other hand, the existence of the purpose attribute in our concept ShoppingList presumes other functionalities, which are not addressed by the application.

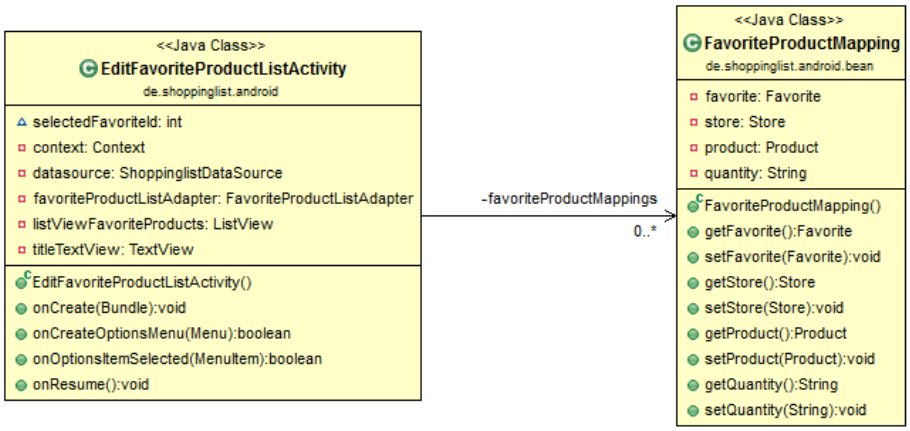
3. Structural comparison

It seems that the architecture of the model does not follow Object Oriented paradigm, rather it looks like a translation of database relations into classes. In the picture of the completel actual architecture, we noticed repetitions of classes with similar responsibilities, which we exclude from the picture to make it more readable. In general, the naming of the classes is a bit confusing e.g. HistoryOverviewAdapter is the linked to Shoppinglist instead of any History related class.

Analysis of the actual architecture

Usually, the name of a class should be descriptive enough to show its purpose. In our case this is partially true. So from here we directly can conclude that this will lead to a code smell “inappropriate naming”.
We have noticed that the relationships between our conceptual classes are mostly modeled by the addition of real “dummy” classes that have reference to the classes they should have a relationship with.
In general, the classes in the real architecture are highly coupled. There are couplings between different conceptual layers, for instance domain model and representation model etc.

Example of a relationship between two classes




```
public class FavoriteProductMapping extends BusinessBean {
        //Attribute Declaration
        private Favorite favorite;
        private Store store;
        private Product product;
        private String quantity;which are used to manipulate
        public Favorite getFavorite() {return favorite;}
        // Getter, setter methods  the attributes
        public void setFavorite(Favorite favorite) {this.favorite = favorite;}
        public Store getStore() {return store;}
       public void setStore(Store store) {this.store = store;}
        …..
        }

public class EditFavoriteProductListActivity extends AbstractShoppinglistActivity {
        // Attribute declaration
        int selectedFavoriteId;
        private Context context;
        private ShoppinglistDataSource datasource;
        private FavoriteProductListAdapter favoriteProductListAdapter;
        private List<FavoriteProductMapping> favoriteProductMappings;
        private ListView listViewFavoriteProducts;
        private TextView titleTextView;
        // OnCreate method invoked on first created activity, which has instance of
favoriteProductMappings made up of a set of FavoriteProductMapping objects => relationship//
        public void onCreate(final Bundle savedInstanceState) {
                this.favoriteProductListAdapter=newFavoriteProductListAdapter(this.context,
        this.favoriteProductMappings);
        this.listViewFavoriteProducts = (ListView) this.findViewById(R.id.listProductsInFavorite);
        this.listViewFavoriteProducts.setAdapter(this.favoriteProductListAdapter);
        ...
```

// Specific operations about editing favorite product list through database by implementing the system defined methods, such as OnItemLongClickListener(),OnMenuItemClickListener(),etc.

….

// code performing the manipulation to the favorite list through database //

…….}

… // GUI methods//

}

System Level Refactoring

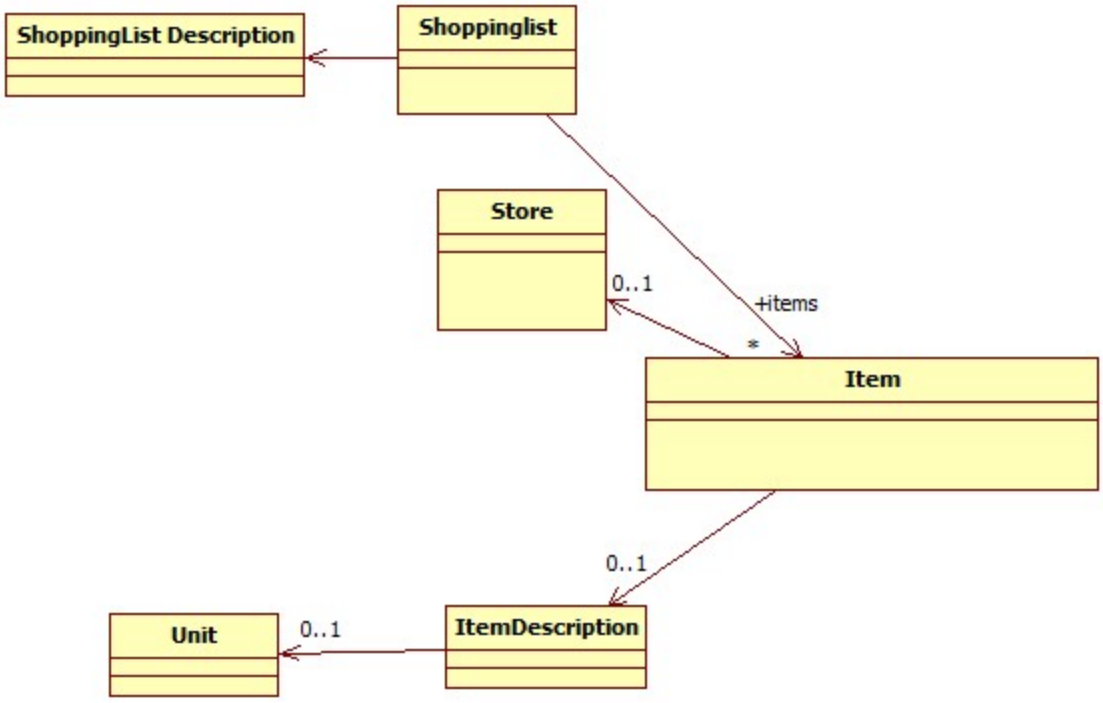
1. Functional Decomposition Pattern

We identified that having a data relational model expressed as OO, the application has many OO principles broken. It is not encapsulating the data and the behaviour correctly along with missing cohesion. The next steps describe how to gradually modify the domain in order to be used by the application when the datasource refactoring is completed.

1. We rename the domain classes with meaningful names to deal with the unappropriate naming smell. Example:

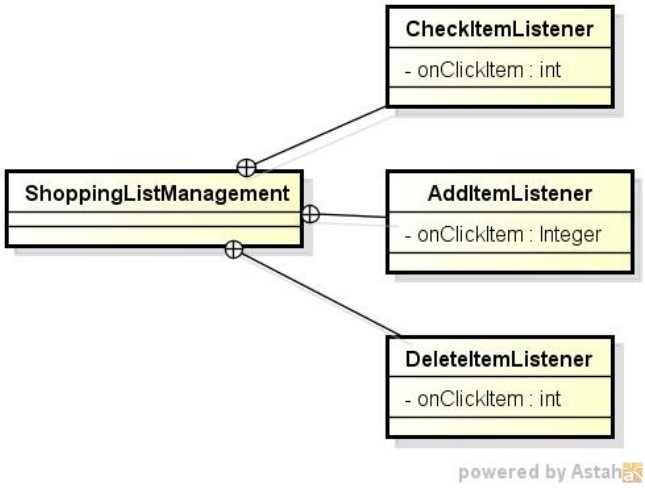
   - Favorite -> ShoppinListDescription - FavoriteProductMapping -> ItemDescription
   - ShoppinglistProductMapping -> Item -

2. We rename the methods and local variables that did reference to the names changed.
3. The concept History collapsed different concepts in attributes instead of using association of objects. So we replace the attributes product, unit and store with the objects Product and Store using delegation to conserve familiarity.
4. The concepts Item and ItemDescription share almost the same behaviour an attributes
5. For Item and I, having almost equal behaviour, we replace the attribute quantity from ShoppinglistProductMapping for ShoppingListProductMapping as itemDescription, and use delegation in the getters and setters to conserve familiarity.
6. The activity classes create routines to get the list of objects ShoppingListProductMapping and FavoriteProductMapping because the Shoppinlist and Favorite don't have access to them. This is signed of primitive obsession, temporary field and duplicated code. To remove the partially obsession smell, we create the bidirectional relationship between these classes with an ArrayList type.
7. To follow the principle Open/Close, the access to the lists is used with the refactoring encapsulate collection. i.e. addItem, removeItem for ShoppingList.



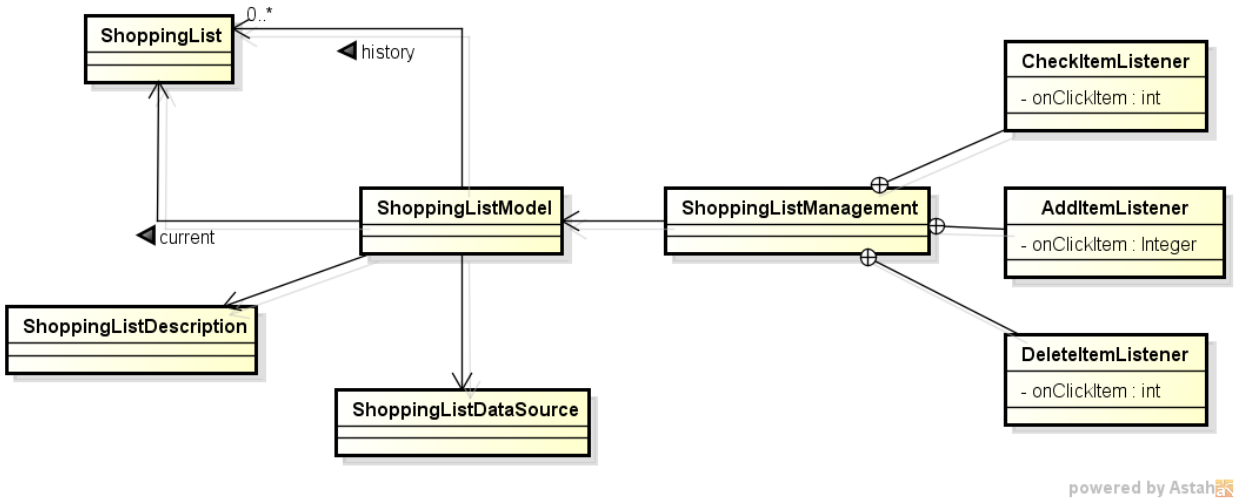
2. Data class and Duplicated code smells in the Domain model with setters: Explained in specific refactorings

3. Improve readability of Activity classes using Specific Requirement 1.

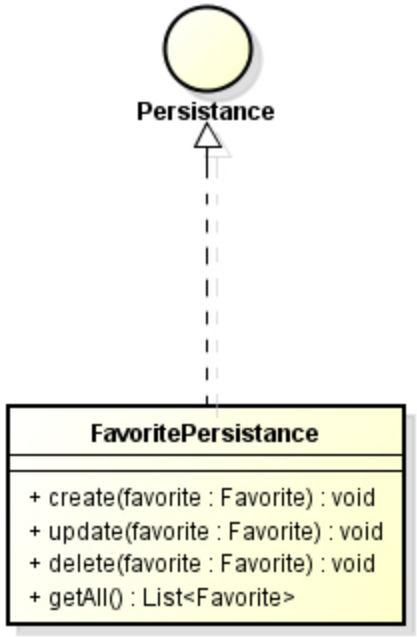


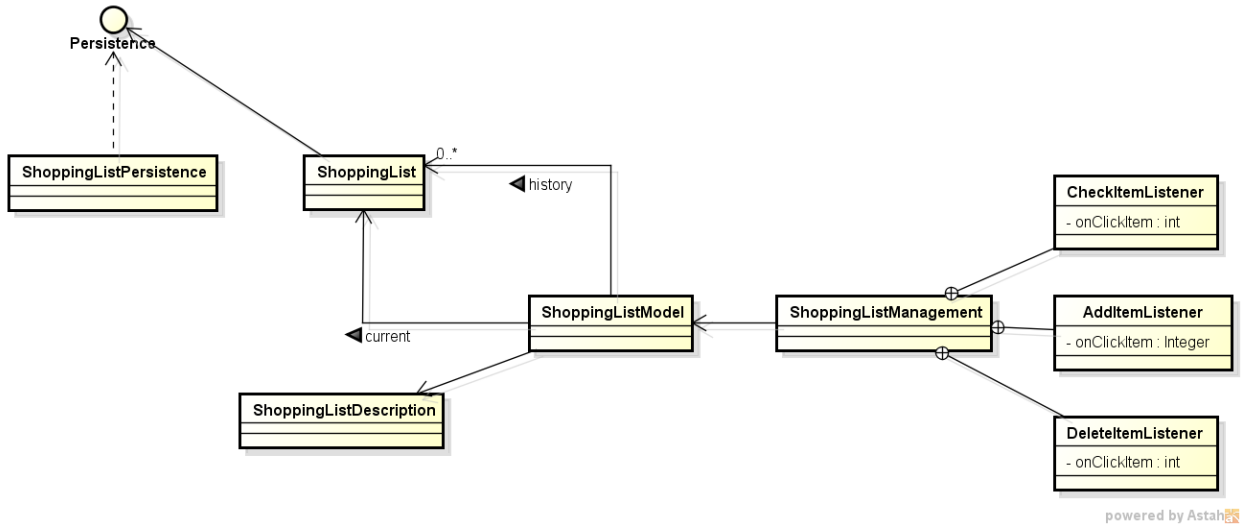
4. Code Smells: Docoupling smell with GUI and Model responsibilities
After encapsulating the methods depending if they affect GUI or the model, we identified that last group of methods contains the logic of our application along with the behavior of our domain. But at the same time this encapsulation created new smells that we have to fix in order to clean the application and improve its architecture. The steps are described next.

1. To remove the smell of high coupling of the application, extract a class named ShoppingListModel.
2. For each activity class, move methods encapsulated as model methods to remove the long class smell. Also move the variables related with these methods.
3. We define the relationships of buyer concept of our domain model in this class.
4. We include in the methods the parallel behaviour done in the database with our domain classes.
5. We extract the persistence behaviour from the classes obtained and include them this behaviour into the domain classes.



6. We manage the relationship between the database and the domain classes with refactoring 2 of specific refactorings.





Specific Refactorings

1. Code Smells: Divergent change, Long method, Chain Message and Feature envy in Activity classes

We identified that all activity classes contain long methods that set the listeners for the events of the activity. Those listeners are anonymous classes, which increase the length of the methods and makes this class divergent to change because every time that we require to modify the logic of a listener, we will have to modify this class. Moreover, because of the use of anonymous classes, the access to objects of the outer class presents a message chain smell. To solve this smells we suggest the next steps for each activity class:

1. We rename the activity class to fit betters its responsibilities.
2. We rename the methods responsible for setting the action listeners to reflect its purpose.
3. To deal with long method and divergent change smells, we Extract the anonymous classes defined inside the setting listeners methods as inner classes.

Before	public class ShoppingListManagementActivity extends AbstractShoppinglistActivity { ….
	 public void actionsToPerformInStoreViewType() {.....}
	 public void actionsToPerformInAlphabeticallyViewType(){
	 this.listStore.setOnItemClickListener(new OnItemClickListener() {
	 public void onItemClick(...) { ….}); …..
	 }

After	public class ShoppingListManagementActivity extends AbstractShoppinglistActivity { …. public void setItemListViewActionListener() { this.listStore.setOnItemClickListener(new CheckItemActionListener()); …. } …

	public void setStoreListViewActionListener() {....} …. private class CheckItemActionListener implements onItemClickListener(){....} }

4. To deal with chain messages, we remove the access *ActivityClass.this* from the call of methods.
5. To reduce coupling between GUI and Model, for each listener we extract methods split-off by their use in model or in view.

Before	public void onItemClick(...){ …
	if (clickedMapping.isChecked() == GlobalValues.NO) {
	ShoppinglistActivity.this.shoppinglistProductMappingsToShow.get(..).setChecked(GlobalValues.YES);
	ShoppinglistActivity.this.datasource
	.markShoppinglistProductMappingAsChecked(clickedMapping.getId());
	} else if (clickedMapping.isChecked() == GlobalValues.YES) {
	ShoppinglistActivity.this.shoppinglistProductMappingsToShow.get(..).setChecked(GlobalValues.NO);
	ShoppinglistActivity.this.datasource
	.markShoppinglistProductMappingAsUnchecked(...);
	}....}

After	public void onItemClick(...){ …
	boolean isItemChecked clickedMapping.isChecked();
	updatetemStatusView(isItemChecked);
	updateItemStatusModel(isItemChecked);
	}....}

2. Code smells: Large class, long parameter list, divergent change in ShoppinglistDataSource

We identified that the class ShoppinglistDataSource manages the persistence behaviour of all domain objects. Deriving in a large class divergent to change. To solve these smells we suggest the next steps:

1. We rename the methods to fit better to its responsibilities and to increase readability.
2. For the long parameter list as arguments of a class, we can instead pass an object which contains all parameters the class needs, to make it much easier to understand and maintenance.

Before	public void checkWhetherProductIsNotInUse(...){...} public void markShoppinglistProductMappingAsChecked(...){...} public void updateFavoriteProductMapping(final int favoriteProductMappingId, final int storeId, final int productId, final String quantity) {...}

After	public void isProductIsNotInUse(...){...}
	public void checkShoppinglistProductMapping(...){...}
	public void updateFavoriteProductMapping(FavoriteProductMapping favoriteProductMapping) {
	final int favoriteProductMappingId = favoriteProductMapping.getId();
	final int storeId = favoriteProductMapping.getStore().getId();
	final int productId = favoriteProductMapping.getProduct().getId();
	final String quantity = favoriteProductMapping.getQuantity();
	...}

3. We extract classes by splitting-off responsibilities and move private variables used in methods from ShoppinglistDatsource to the classes created.
4. We rename the methods of each class eliminating the smell type embedded in name dependency from the method signature. i.e. updateShoppingList to update.
5. Then we extract interface based on methods with common names of the new classes.

Before

```
// Only one class to deal with database.
```

After

```
public interface Persistance{
        public void create(Object object);
        public void update(Object object);
        public void delete(Object object);
        public List<Object> getAll();
}
public class FavoritePersistance implements Persistance{
        public void create(Favorite favorite){}
        public void update(Favorite favorite){}
        public void delete(Favorite favorite){}
        public List<Favorite> getAll(){}
}
```

6. To deal with data clumps smell, having the domain object as a parameter in many method signatures, for each class we declare the domain object as a private variable and declare the class constructor of the class.

3. Data class and Duplicated code smells in the Domain model with setters

All domain classes are dumb data holders without constructor methods generating duplicated code every time that a new class is instantiated because the values of the class are assigned using setters. The steps described below solve these smells.

1. For each domain class, we identify all the classes that instantiate the domain model along with setters used for assigning value to the variables, to identify duplicated blocks of code.
2. We declare constructor methods with these variables as parameters to reduce the data class smell.

Before

```
public class Shoppinglist extends BusinessBean {
        private Timestamp finishedTime;
        private Timestamp createdTime;
        public Timestamp getFinishedTime() {return finishedTime; }
        public void setFinishedTime(Timestamp finishedTime) {
                this.finishedTime = finishedTime;
        }
        public Timestamp getCreatedTime() {return createdTime;}
        public void setCreatedTime(Timestamp createdTime) {
                this.createdTime = createdTime;
        }
}
```

After	public class Shoppinglist extends BusinessBean {
	final private Timestamp finishedTime;
	final private Timestamp createdTime;
	public Shoppinglist(){
	createdTime = new Timestamp(System.currentTimeMillis());
	}
	public Timestamp getFinishedTime() {return finishedTime; }
	public void setFinishedTime() {
	finishedTime = new Timestamp(System.currentTimeMillis());
	}
	public Timestamp getCreatedTime() {return createdTime;}
	}

3. For each class where the domain classes are identified, we replace the duplicated code with the call to the constructor of the class.
4. If we have a case where some variables are generated, we use polymorphism to create a different constructor, to reduce the duplicated code from the generation of the assigned value.
5. To remove unused setting methods from the domain classes, we identify which variables are defined only when instantiating the class and delete the setters and define these variables as final.

GitHub

We decided to create our own branch of source code in order not to interfere with the master branch. It will be up to the repository maintainer to merge our branch with his in the end.
Steps:

1. Checkout: *git clone https://github.com/**[username]**/shoppinglist-android.git*
2. Create new branch: *git checkout -b soen6471-w2014*
3. Change code and commit: *git commit [path_to_changed_directory] -m "Code changed, methods refactored."*
4. Push the changes to the online repository on our branch: *git push*

This were our idealistic steps. However, we could not commit to the branch that we created above and thus we committed to the main.