CSCI 5448 Project Final Report

Name

Omar Hammad

GitHub link

https://github.com/hammadojh/Bekam_OODA

Title

Online Used Market (Bekam)

Description

Bekam is an ios app that allows people to sell and buy used stuff. Bekam means “How much” in Arabic. Users can simply take a picture of something and post it to a shared page. If a user wants to buy a product he can communicate with the seller through the app provided chat. The app doesn’t include any online payment. It only connects buyers and sellers together.

Functionality

Implemented

ID	Title
F1	User can login
F2	User can logout
F3	User can create a new account
F4	User can view all products
F5	User can post new product
F6	User can modify a product

F7	User can send a message to another user
F8	User can view a message sent to another user

Not Implemented

ID	**Title**
N1	User can view account information
N2	User can modify account information
N3	User can delete account
N4	User can sort products
N5	User can filter products
N6	User can search for products
N7	User can view products categories
N8	User can view notifications
N9	User can delete a product
N10	User can set the product status as sold
N11	User can set the product status as not sold
N12	User can delete a message sent by him
N13	User can report another user
N14	User can block another user

System Architecture

The following figure (Figure 1) shows the high level architecture of the system. It is an MVC and Client Server architecture. In brief, We have a client side code which has three layers: Model, View and Controller which connects the two together. The controller also communicate with another layer which is the API services layer. It is basically a Facade to a bunch of API services that communicate with the cloud database. I am using Firebase as a cloud service.

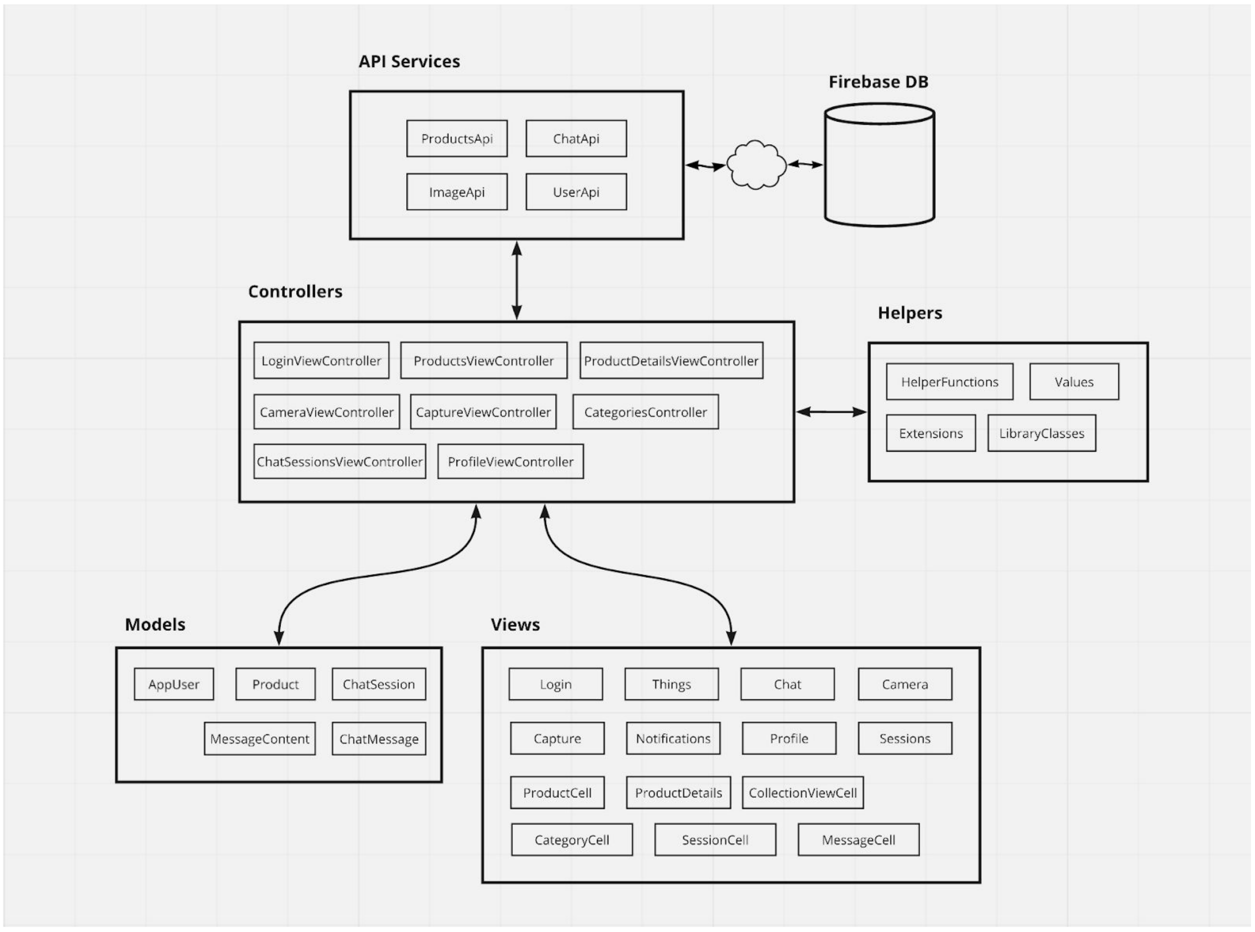


Figure 1 : System Architecture

Design Patterns Implemented

In total, I have implemented six design patterns other than Singleton and MVC. They are :

- Adapter
- Observer
- Facade
- Factory
- State
- and Flyweight.

Adapter

In my app, I have implemented a chat functionality. The building block of a chat is the messages. Messages can have different type of content. The simplest type of content is text. However, there are complex content types like image, url, location, etc. So I have used the Adapter design pattern to view these advanced content types.

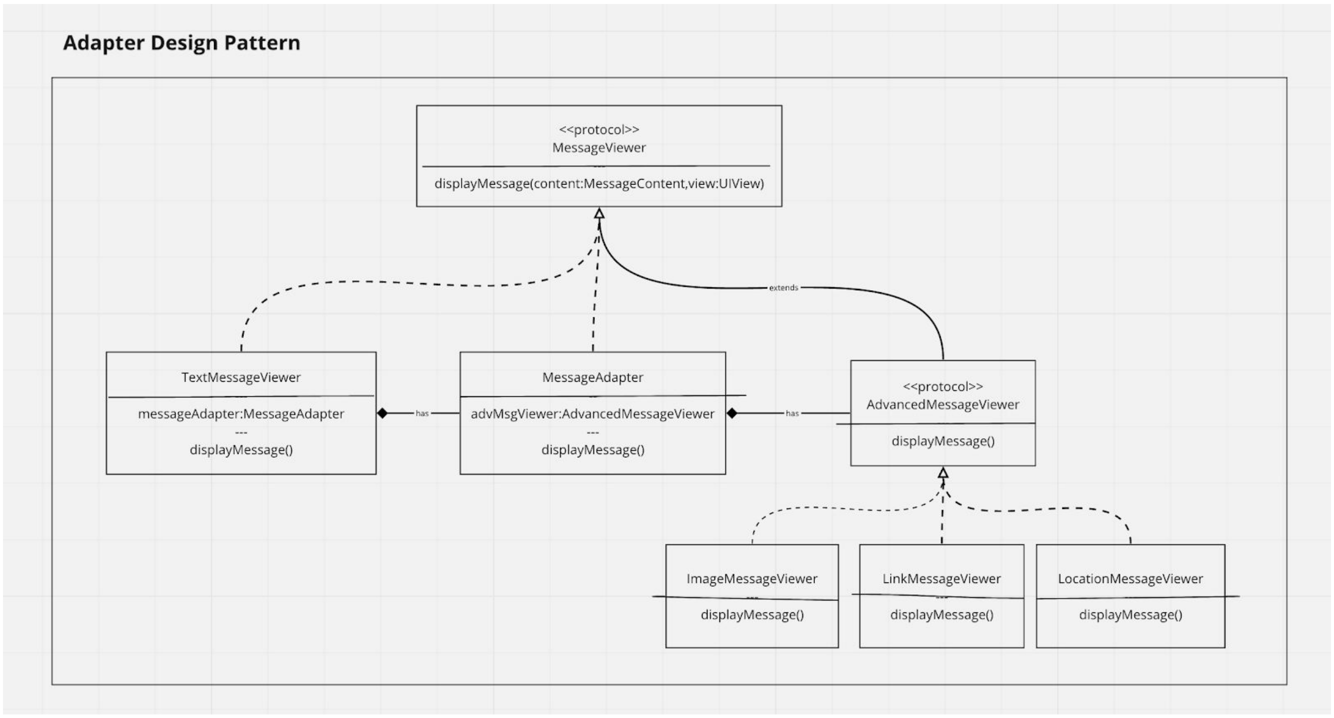
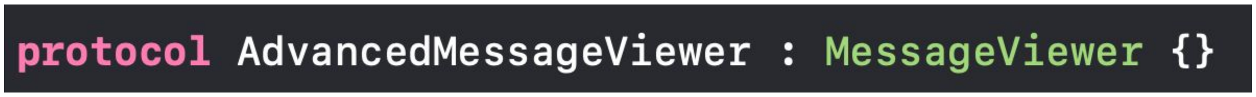
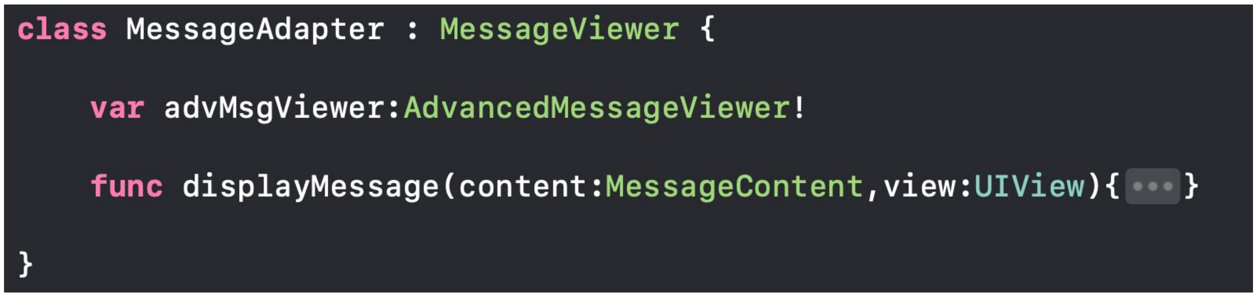


Figure 3 : Adapter Design Pattern

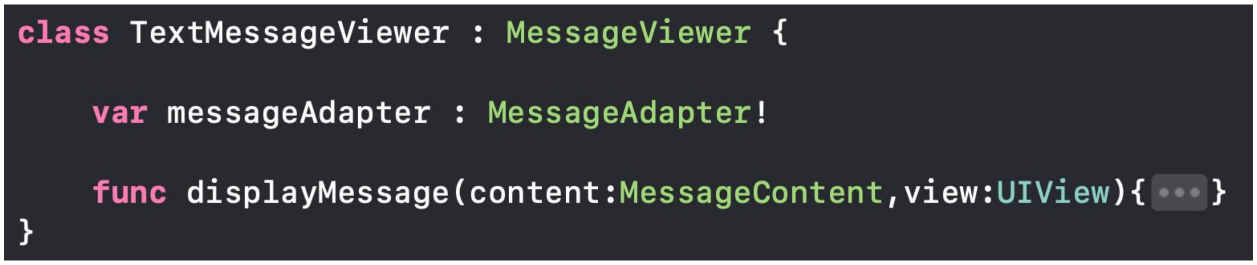
Implementation



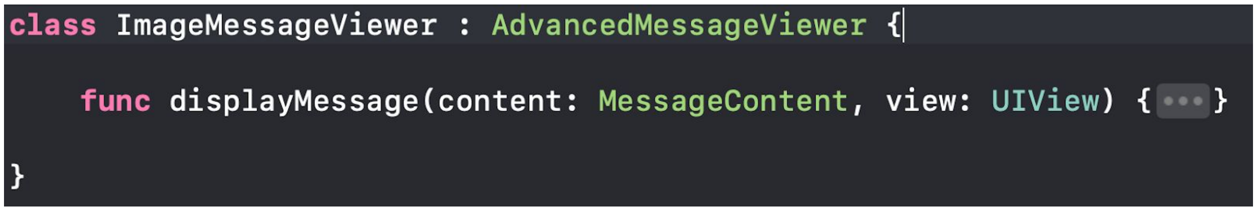
```swift
protocol AdvancedMessageViewer : MessageViewer {}
```



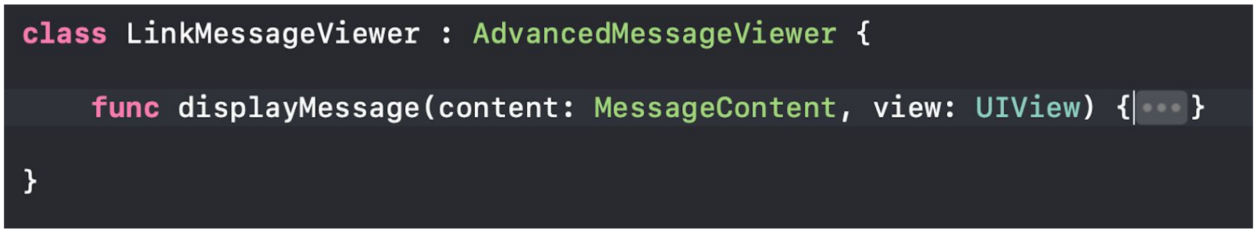
```swift
class MessageAdapter : MessageViewer {

    var advMsgViewer:AdvancedMessageViewer!

    func displayMessage(content:MessageContent,view:UIView){•••}

}
```



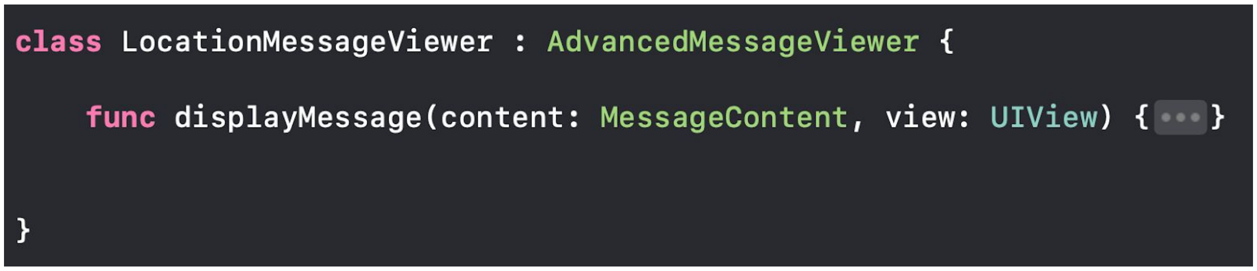
```swift
class TextMessageViewer : MessageViewer {

    var messageAdapter : MessageAdapter!

    func displayMessage(content:MessageContent,view:UIView){•••}
}
```



```swift
class ImageMessageViewer : AdvancedMessageViewer {

    func displayMessage(content: MessageContent, view: UIView) {•••}

}
```



```swift
class LinkMessageViewer : AdvancedMessageViewer {

    func displayMessage(content: MessageContent, view: UIView) {•••}

}
```



```swift
class LocationMessageViewer : AdvancedMessageViewer {

    func displayMessage(content: MessageContent, view: UIView) {•••}


}
```

Observer

In the view that have all the products, I have a collection of products. In each product there is a chat button. If the user clicks on the chat button he should be directed to chat with the owner of the product associated with this item. So, I let the view controllers observe the cell and once it is clicked, it notify the view controller and passes the needed values to start the chat.

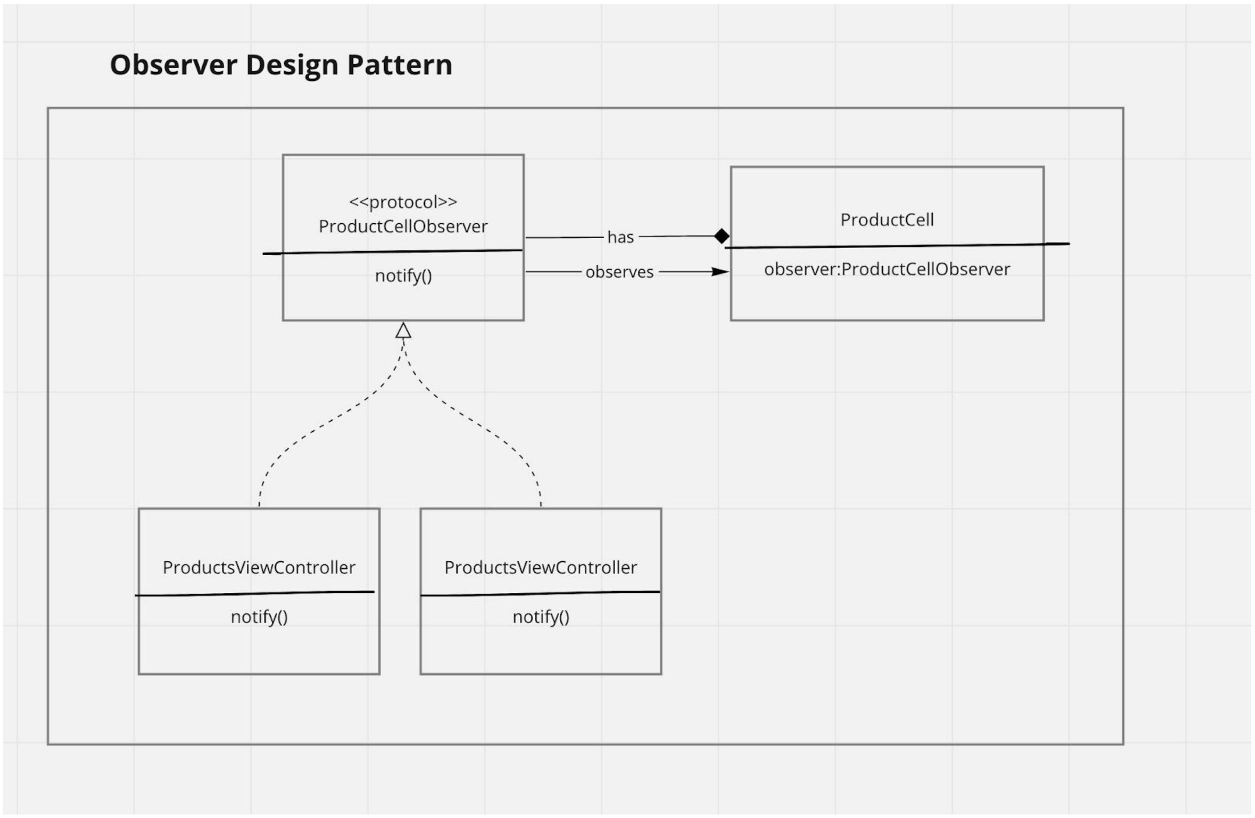
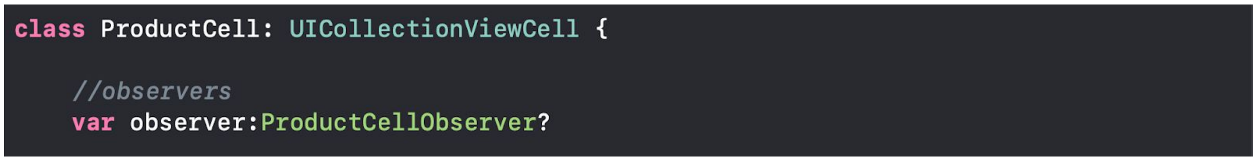
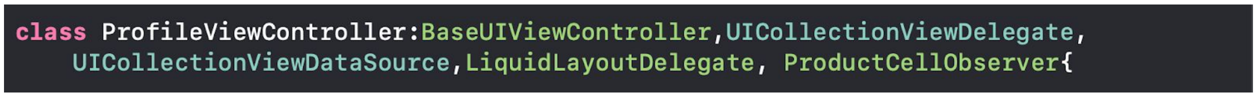


Figure 4 : Observer

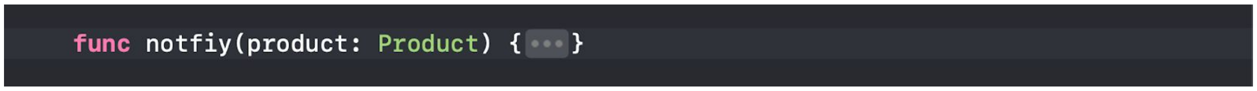
Implementation



```
class ProductCell: UICollectionViewCell {

    //observers
    var observer:ProductCellObserver?
```



```
class ProfileViewController:BaseUIViewController,UICollectionViewDelegate,
    UICollectionViewDataSource,LiquidLayoutDelegate, ProductCellObserver{
```



```
    func notfiy(product: Product) {•••}
```

```
class ProductsViewController: BaseUIViewController, UISearchBarDelegate,
    UICollectionViewDelegate, UICollectionViewDataSource,LiquidLayoutDelegate,
    ProductCellObserver {
```

```
    func notfiy(product: Product) {•••}
```

Facade

In the app there are a lot of API calls to the server. So instead of letting all the classes get coupled with each API class, I have created a facade class called API Services that link between the controllers and the API classes.

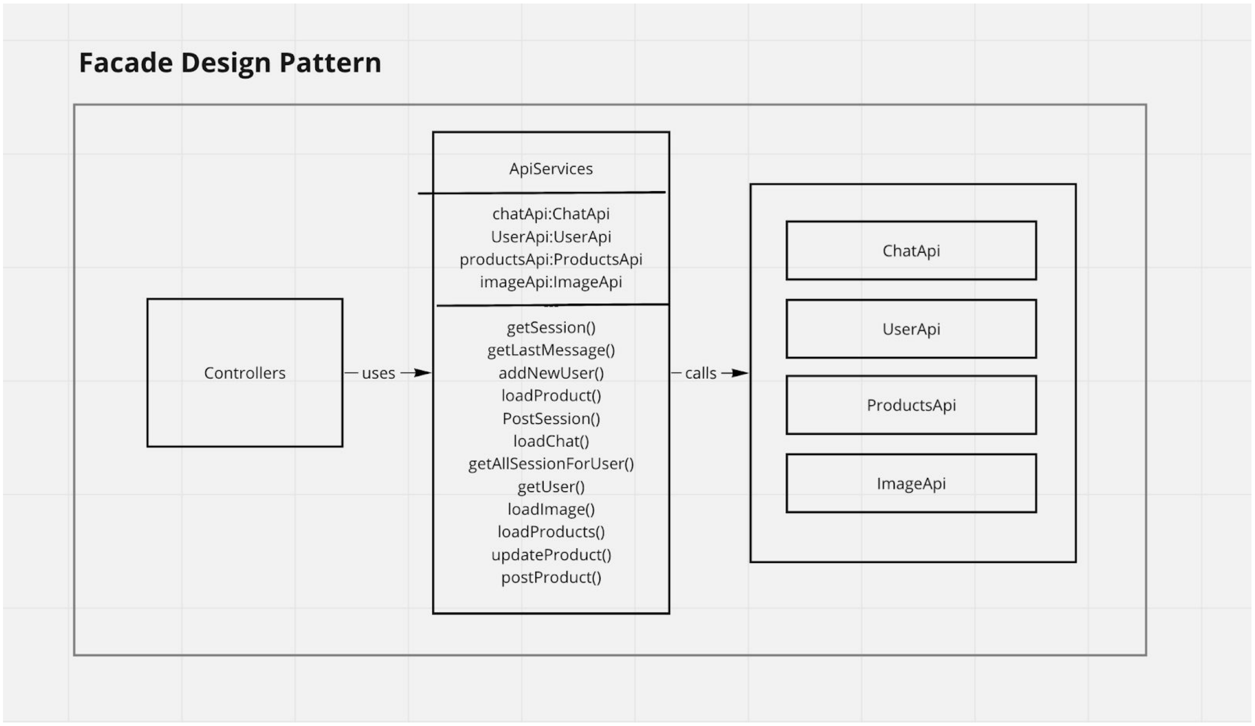
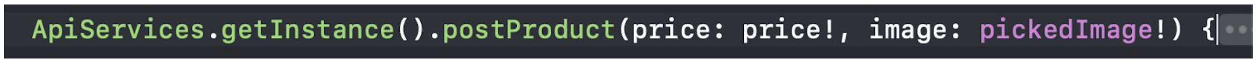


Figure 5 : Facade

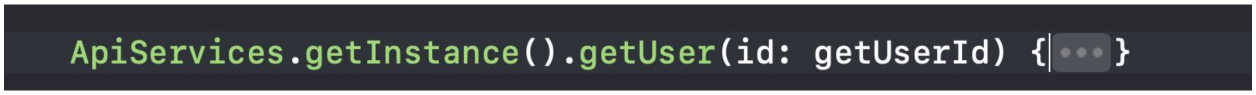
Implementation

```
class ApiServices {

    fileprivate let chatApi = ChatApi()
    fileprivate let productsApi = ProductsApi()
    fileprivate let imageApi = ImageApi()
    fileprivate let userApi = UserApi()
```

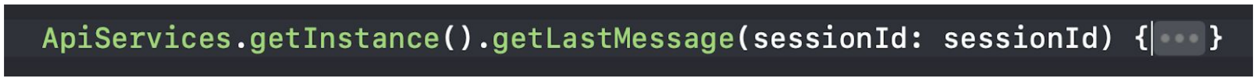
```
    ApiServices.getInstance().loadProducts{•••}
```



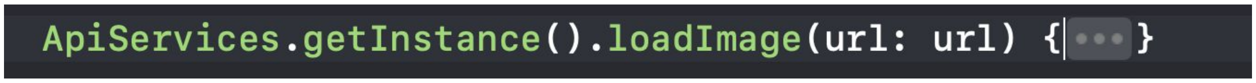
```
ApiServices.getInstance().postProduct(price: price!, image: pickedImage!) {•••
```



```
ApiServices.getInstance().getUser(id: getUserId) {•••}
```



```
ApiServices.getInstance().getLastMessage(sessionId: sessionId) {•••}
```



```
ApiServices.getInstance().loadImage(url: url) {•••}
```

Factory

There are three types of products prices in the app. Free products, Normal price products and rent price products. For each one of these the look of the price label is different. So I thought that a factory of the labels will handle the creation of these labels.

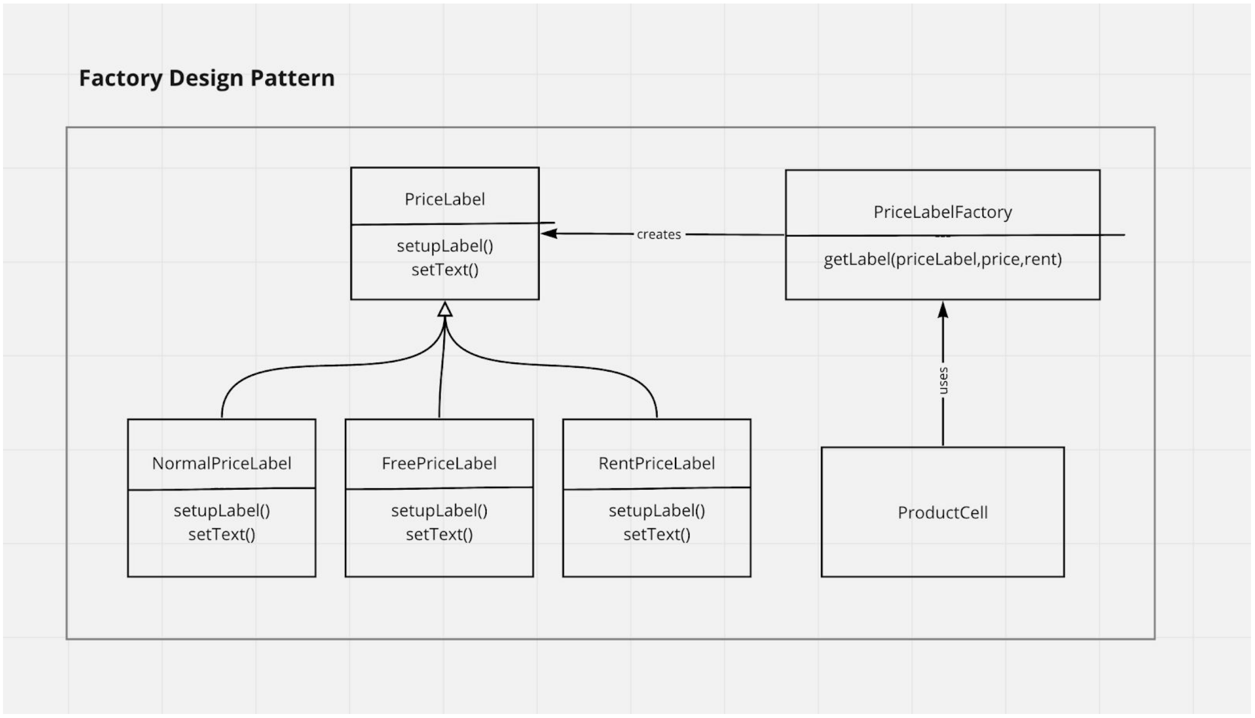
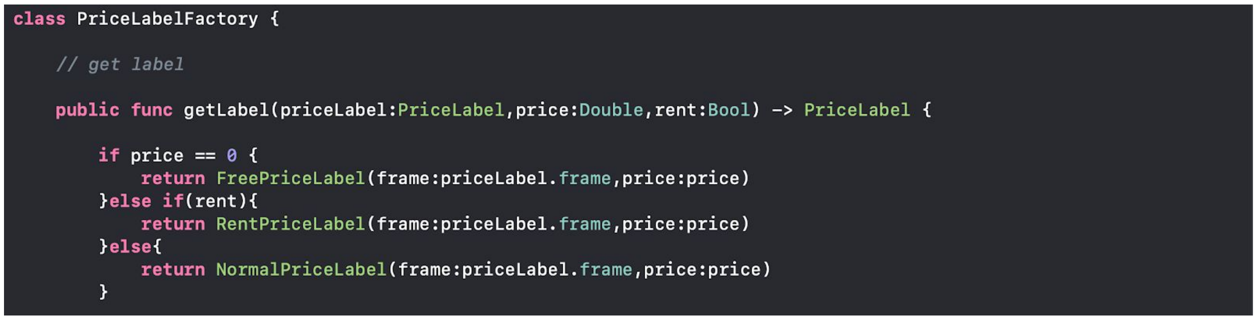
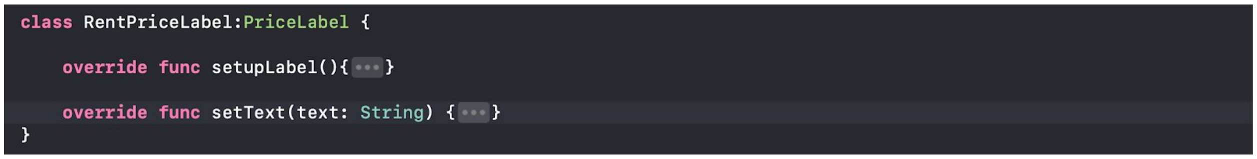


Figure 6 : Factory

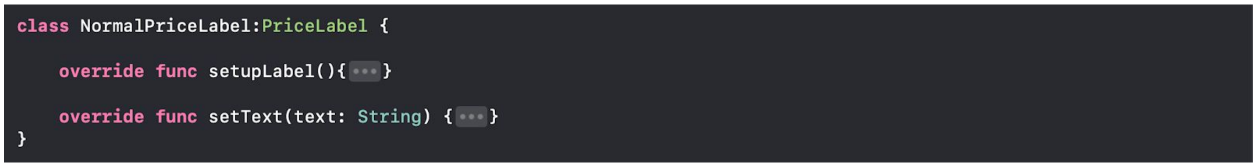
Implementation



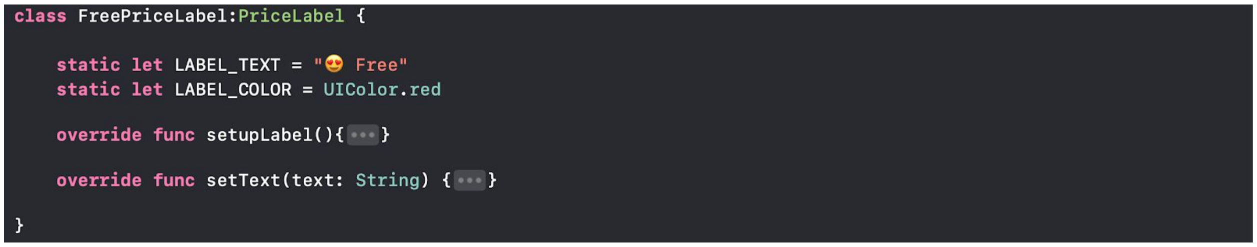
```swift
class PriceLabelFactory {

    // get label

    public func getLabel(priceLabel:PriceLabel,price:Double,rent:Bool) -> PriceLabel {

        if price == 0 {
            return FreePriceLabel(frame:priceLabel.frame,price:price)
        }else if(rent){
            return RentPriceLabel(frame:priceLabel.frame,price:price)
        }else{
            return NormalPriceLabel(frame:priceLabel.frame,price:price)
        }
```



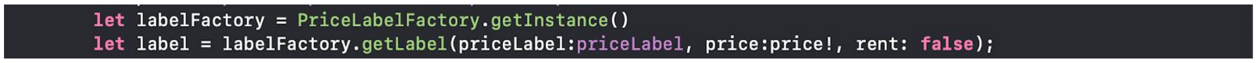
```swift
class RentPriceLabel:PriceLabel {

    override func setupLabel(){•••}

    override func setText(text: String) {•••}
}
```



```swift
class NormalPriceLabel:PriceLabel {

    override func setupLabel(){•••}

    override func setText(text: String) {•••}
}
```



```swift
class FreePriceLabel:PriceLabel {

    static let LABEL_TEXT = "😍 Free"
    static let LABEL_COLOR = UIColor.red

    override func setupLabel(){•••}

    override func setText(text: String) {•••}

}
```



```swift
        let labelFactory = PriceLabelFactory.getInstance()
        let label = labelFactory.getLabel(priceLabel:priceLabel, price:price!, rent: false);
```

State (a.k.a Delegate in Swift)

When the user opens the product details screen. This screen has two states : either the product is owned by the current user, or it is owned by another user. In these two cases the screen should act differently. So I thought that the controller should delegate the behaviour to the state object or as called in swift the delegate.

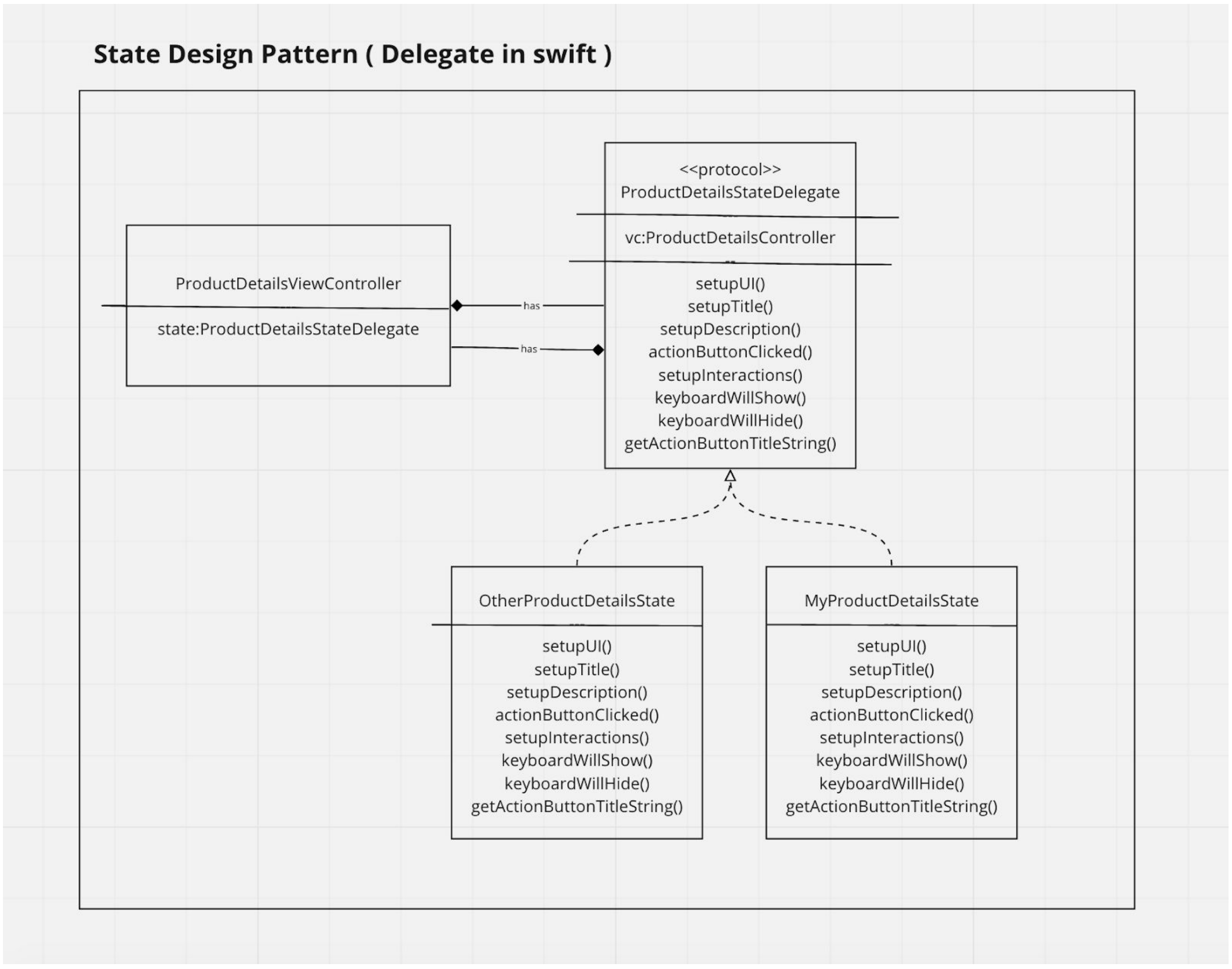
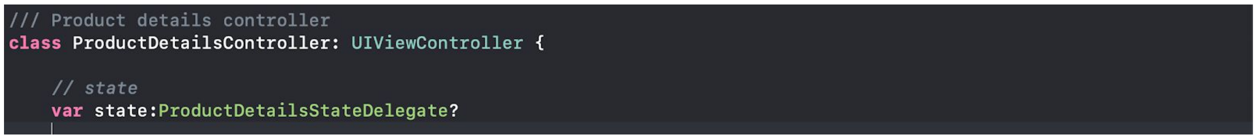
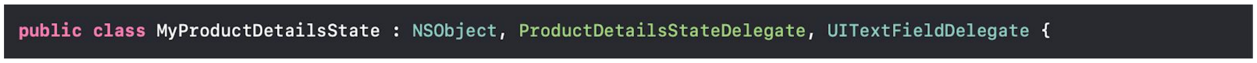


Figure 7 : State

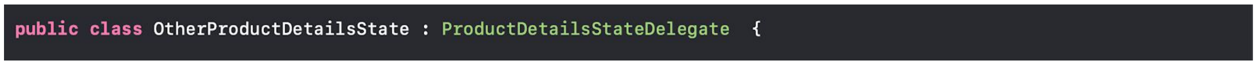
Implementation



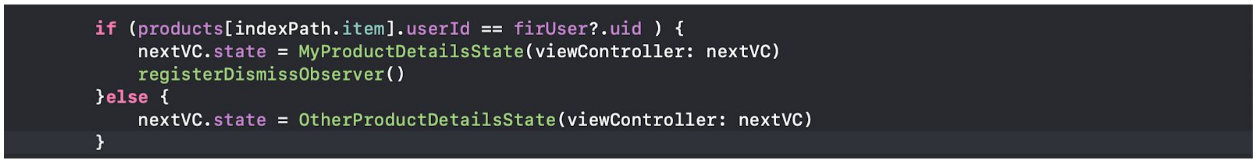
```
/// Product details controller
class ProductDetailsController: UIViewController {

    // state
    var state:ProductDetailsStateDelegate?
```



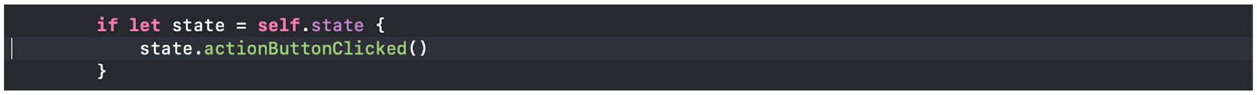
```
public class MyProductDetailsState : NSObject, ProductDetailsStateDelegate, UITextFieldDelegate {
```



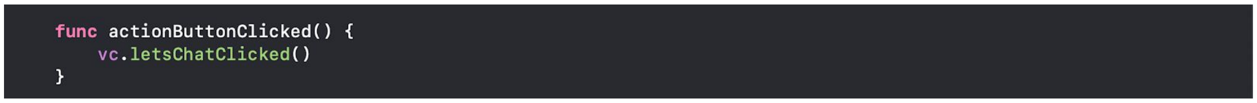
```
public class OtherProductDetailsState : ProductDetailsStateDelegate  {
```



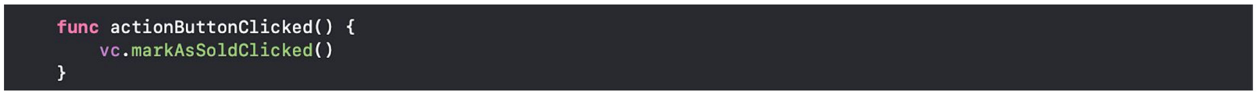
```
        if (products[indexPath.item].userId == firUser?.uid ) {
            nextVC.state = MyProductDetailsState(viewController: nextVC)
            registerDismissObserver()
        }else {
            nextVC.state = OtherProductDetailsState(viewController: nextVC)
        }
```



```
        if let state = self.state {
            state.actionButtonClicked()
        }
```



```
    func actionButtonClicked() {
        vc.letsChatClicked()
    }
```



```
    func actionButtonClicked() {
        vc.markAsSoldClicked()
    }
```

Flyweight

The app depends on images a lot. Each product must have an image. These images are used more than one time in the app. Before, every time the user scrolls to a new item, it loads the new image from the server. I thought why don't I implement the flyweight design pattern and save the image to a cache and only load the image if it is not already there.

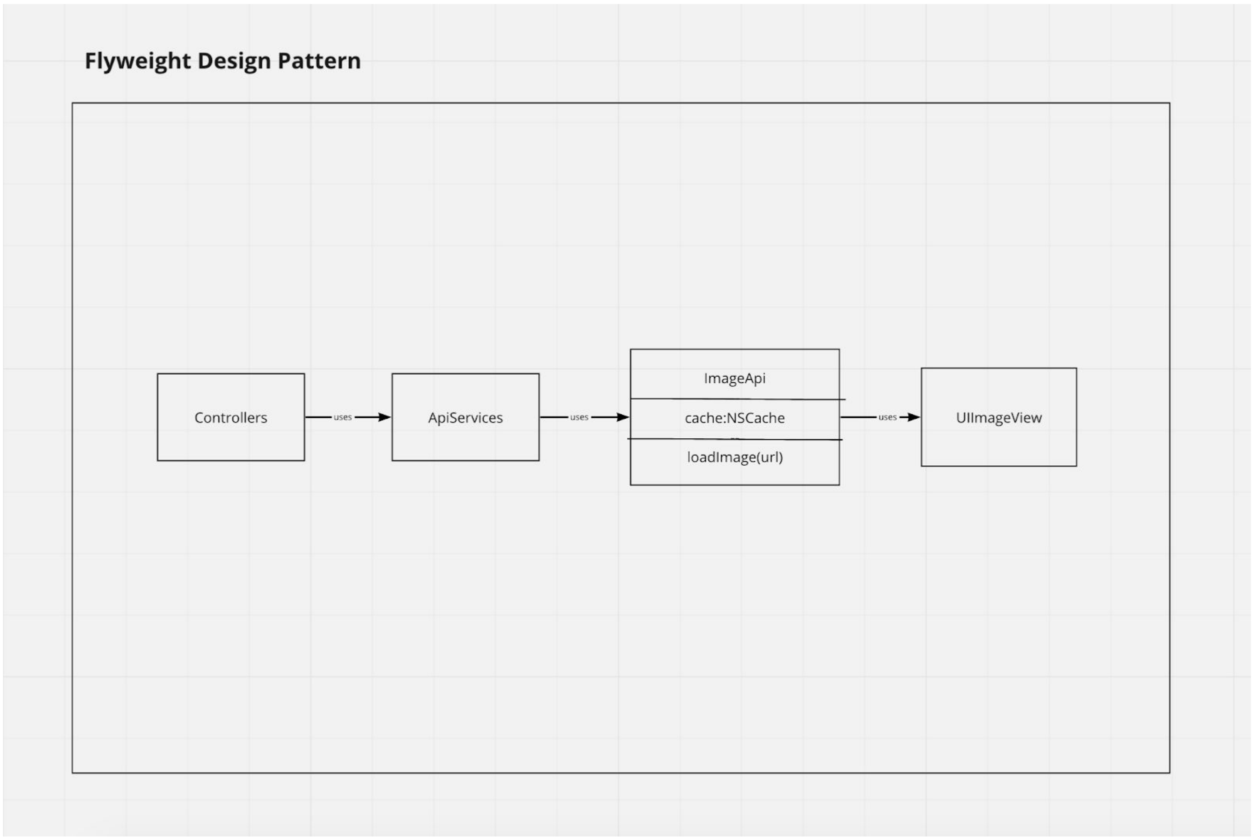
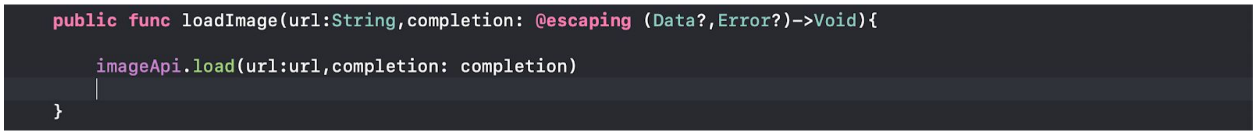
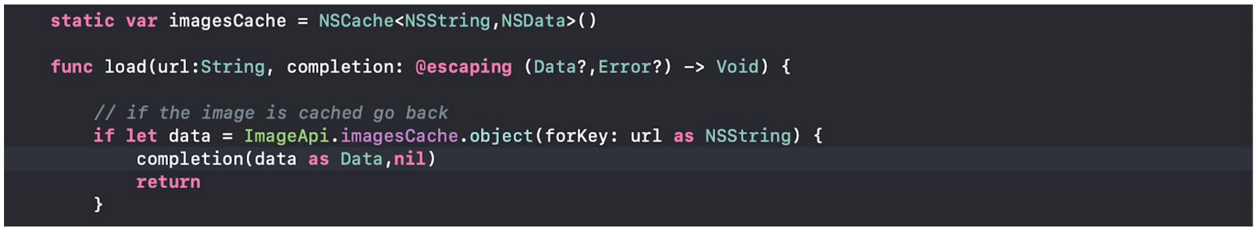


Figure 8 : Flyweight

Implementation



```
public func loadImage(url:String,completion: @escaping (Data?,Error?)->Void){

    imageApi.load(url:url,completion: completion)

}
```



```
static var imagesCache = NSCache<NSString,NSData>()

func load(url:String, completion: @escaping (Data?,Error?) -> Void) {

    // if the image is cached go back
    if let data = ImageApi.imagesCache.object(forKey: url as NSString) {
        completion(data as Data,nil)
        return
    }
```

New vs Old Class Diagrams

In my old class diagram, I did not include the controllers. So my current one is a bit different than the old one. The model classes are quite similar, except some differences in the names of the methods and properties. I have used a tool to generate all the classes in my system. If I did not mistake counting there are about 92 classes/interfaces in the system. This is the result.

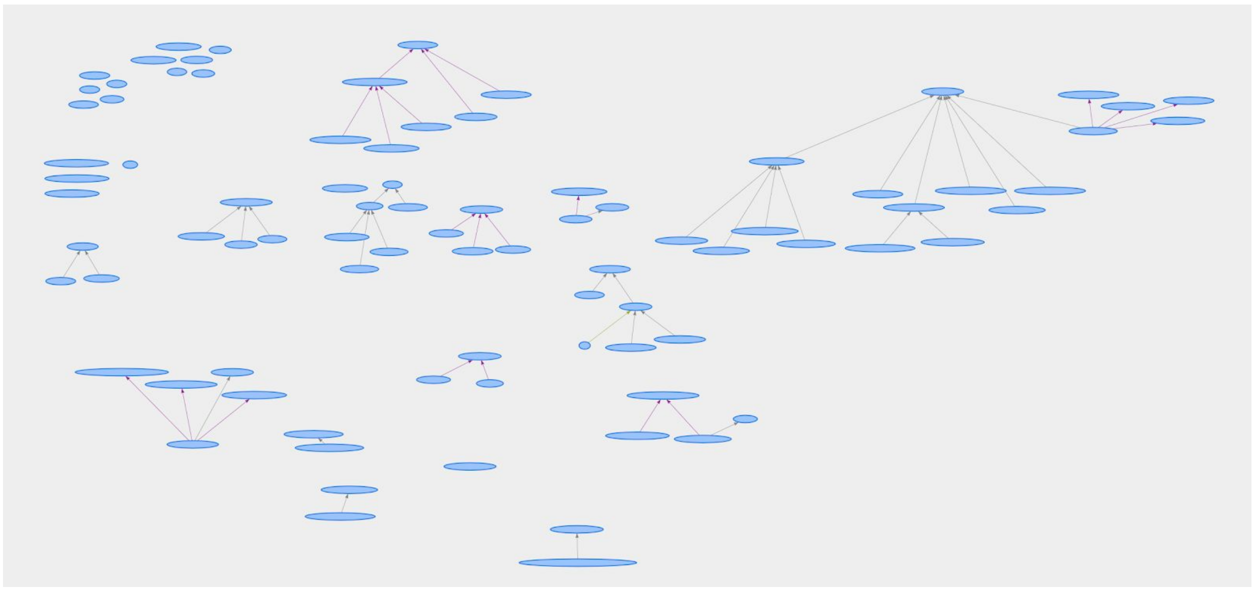


Figure 2a : All current classes

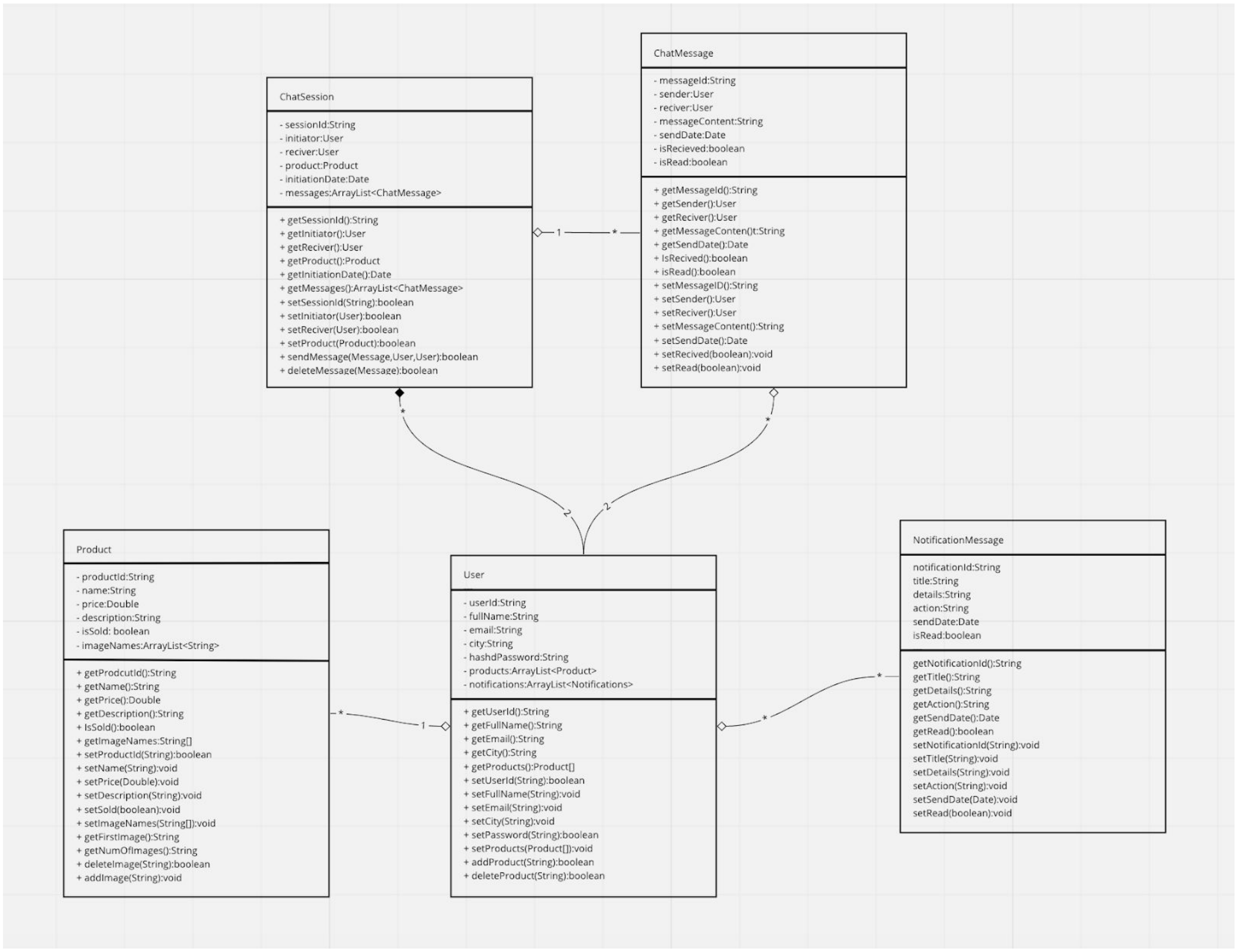
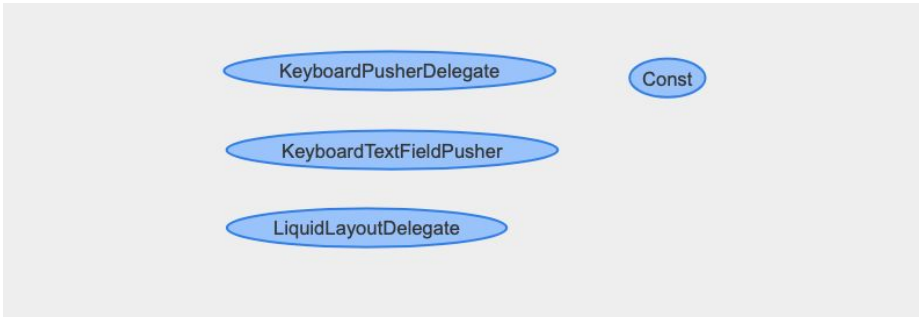
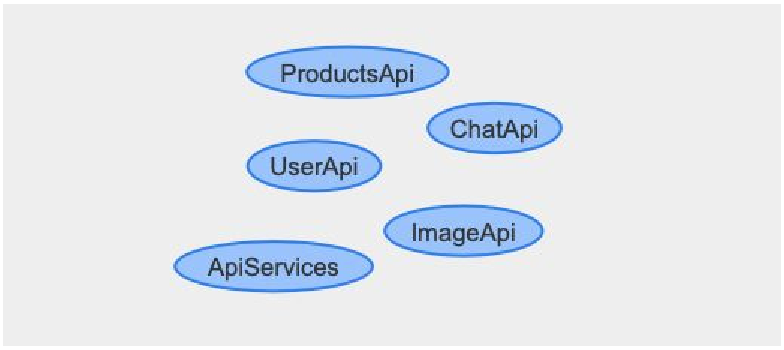
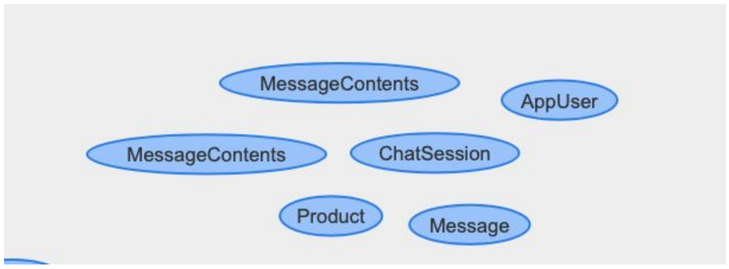


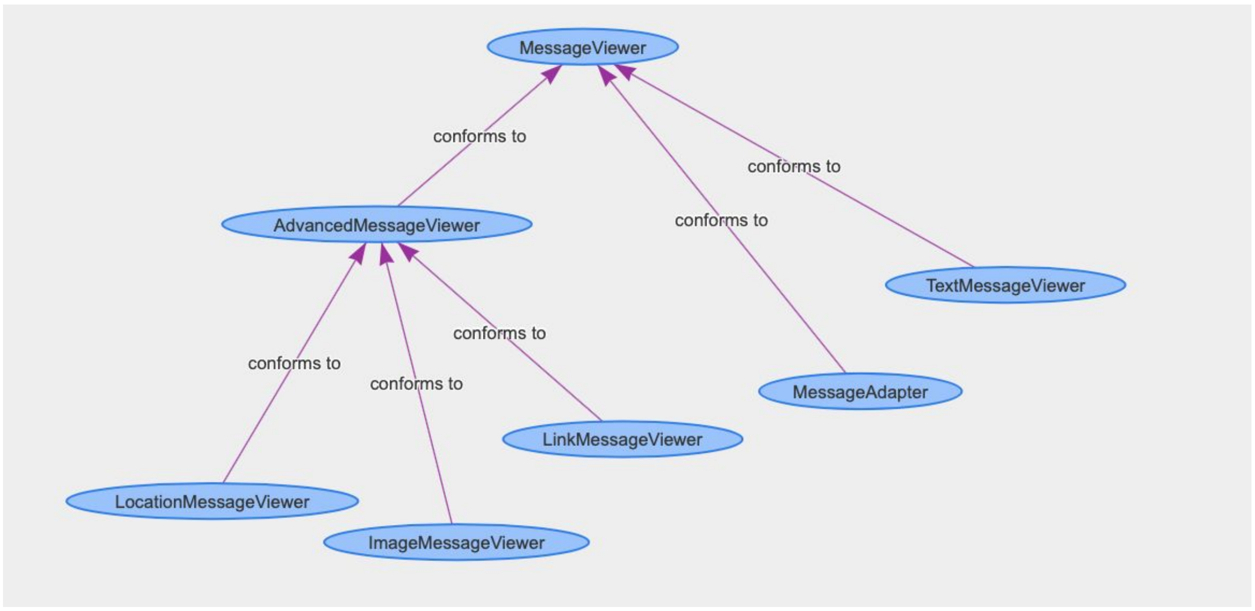
Figure 2b : Old Class Diagram

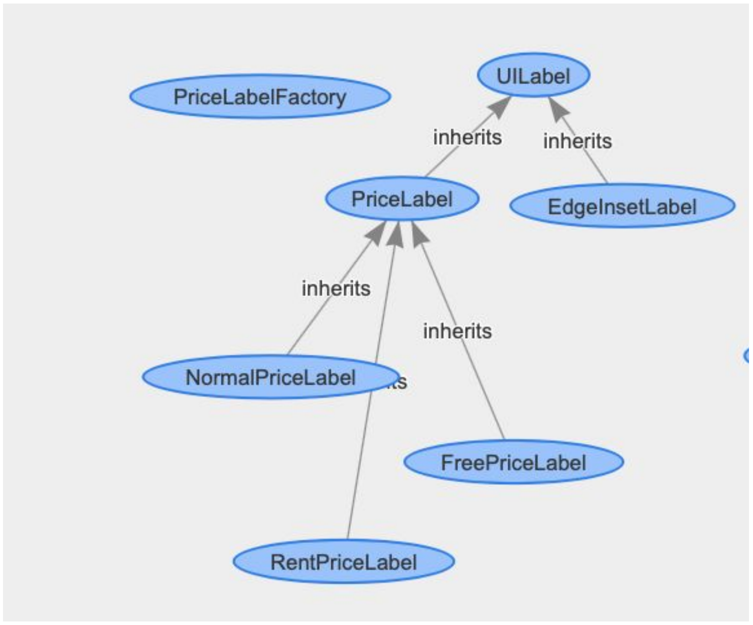
The following figures will be separate figures for each part of the class diagram. I will only design the ones that I have created. I will not be able to show all the links in the system.

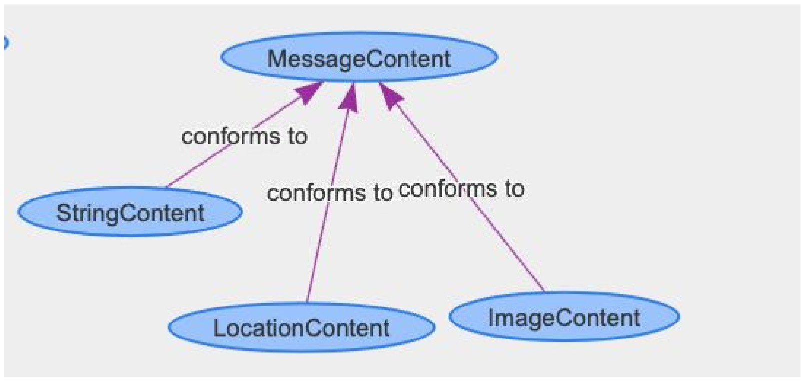

KeyboardPusherDelegate
Const
KeyboardTextFieldPusher
LiquidLayoutDelegate

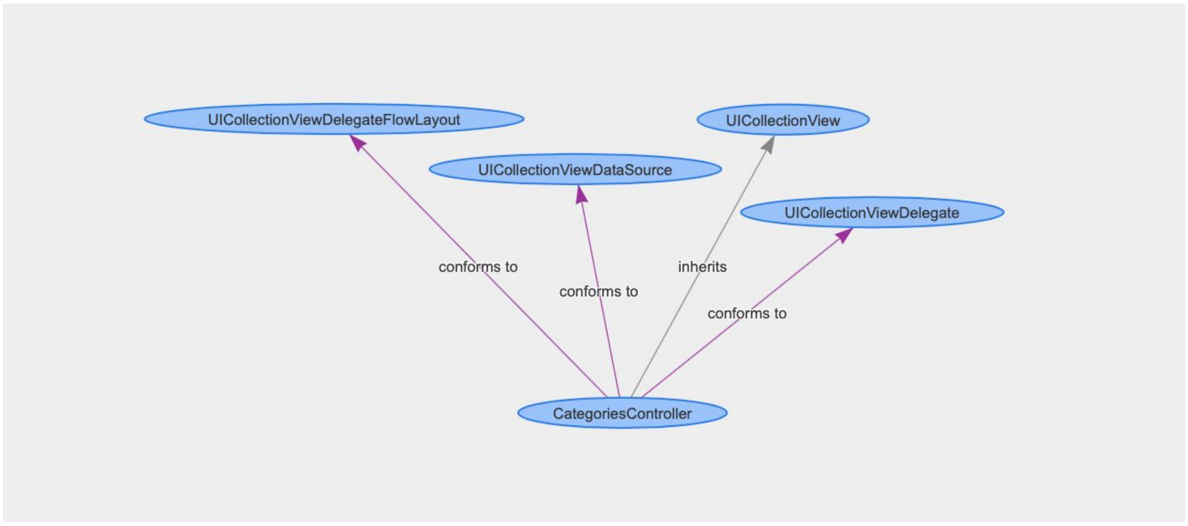

ProductsApi
ChatApi
UserApi
ImageApi
ApiServices

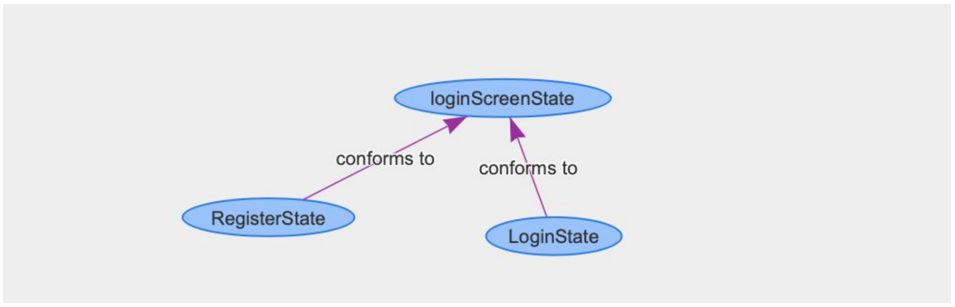

MessageContents
AppUser
MessageContents
ChatSession
Product
Message

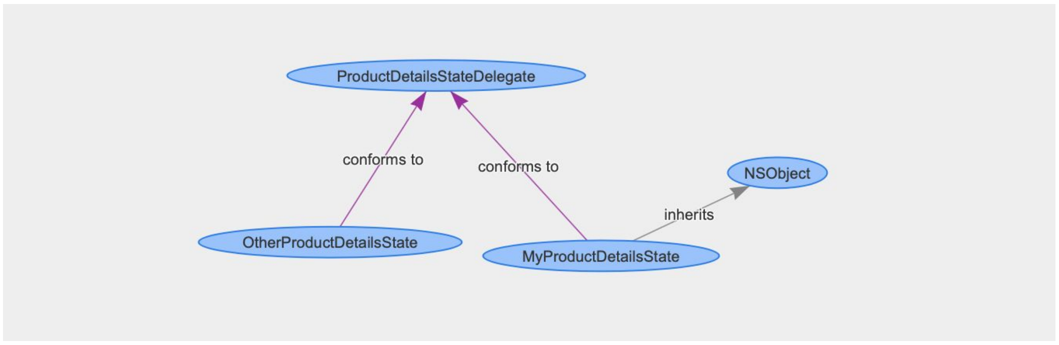

MessageViewer
conforms to
conforms to
conforms to
AdvancedMessageViewer
TextMessageViewer
conforms to
conforms to
conforms to
MessageAdapter
LinkMessageViewer
LocationMessageViewer
ImageMessageViewer

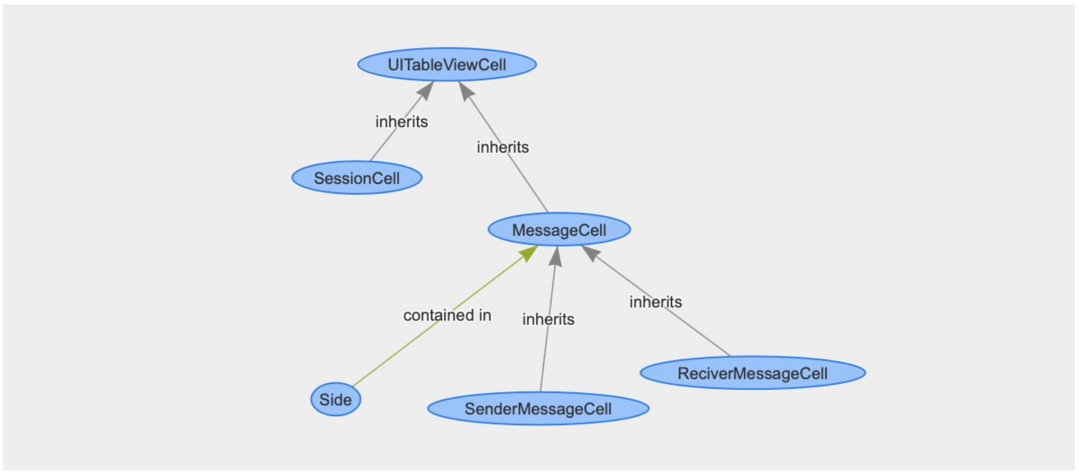

PriceLabelFactory
UILabel
inherits
inherits
PriceLabel
EdgeInsetLabel
inherits
inherits
NormalPriceLabel
FreePriceLabel
RentPriceLabel

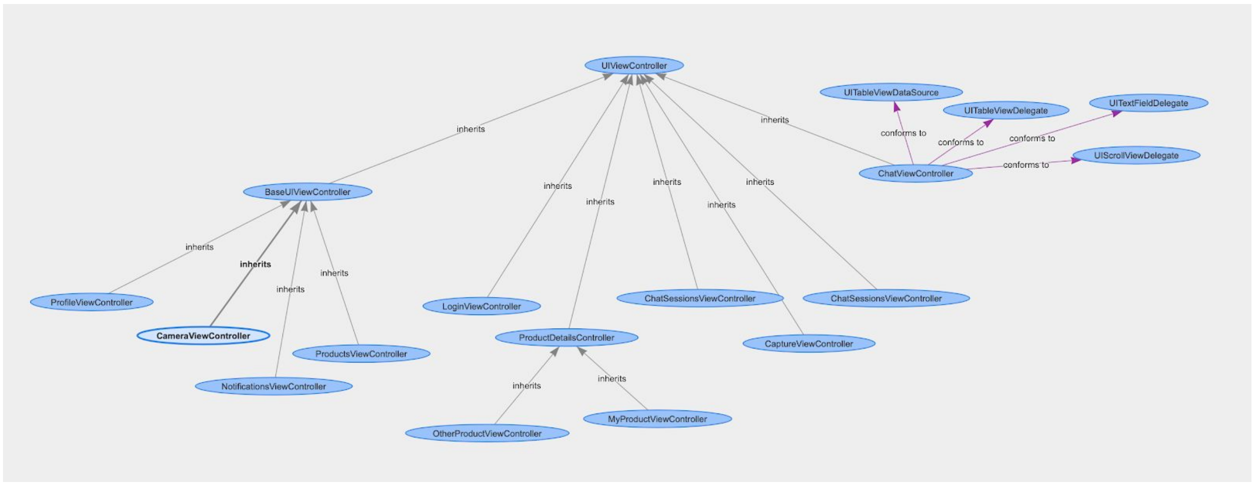

MessageContent
conforms to
StringContent
conforms to
conforms to
LocationContent
ImageContent


UICollectionViewDelegateFlowLayout
UICollectionView
UICollectionViewDataSource
UICollectionViewDelegate
conforms to
conforms to
inherits
conforms to
CategoriesController


loginScreenState
conforms to
conforms to
RegisterState
LoginState


ProductDetailsStateDelegate
conforms to
conforms to
NSObject
inherits
OtherProductDetailsState
MyProductDetailsState


UITableViewCell
inherits
inherits
SessionCell
MessageCell
contained in
inherits
inherits
Side
SenderMessageCell
ReciverMessageCell


UIViewController
UITableViewDataSource
UITableViewDelegate
UITextFieldDelegate
inherits
inherits
conforms to
conforms to
conforms to
conforms to
UIScrollViewDelegate
ChatViewController
BaseUIViewController
inherits
inherits
inherits
inherits
inherits
inherits
inherits
inherits
inherits
inherits
ProfileViewController
CameraViewController
ProductsViewController
NotificationsViewController
LoginViewController
ChatSessionsViewController
ChatSessionsViewController
ProductDetailsController
CaptureViewController
inherits
inherits
MyProductViewController
OtherProductViewController

What I learned

I really learned a lot from this class especially the design patterns part. The most interesting thing is how the code is much more maintainable when applying these good design standards. When I think of adding new features in the futures, which I aim to, I feel that it will be more approachable than before. I also learned that without trying yourself you will not learn. For example, the design patterns that I have applied gets to my mind now very fast when I face the same problem.