Mohammad Abdullah Atik

Applying Software Design Pattern on iOS Application

A case study Finnkino

Helsinki Metropolia University of Applied Sciences

Bachelor of Engineering

Information Technology

Bachelor's Thesis

3 November 2014

Helsinki
Metropolia
University of Applied Sciences

Author(s) Title Number of Pages Date	Mohammad Abdullah Atik Applying Software Design Pattern on iOS Application 31 pages 3 November 2014
Degree	Bachelor of Engineering
Degree Programme	Information Technology
Specialisation option	Software Engineering
Instructor(s)	Peter Hjort, Senior Lecturer

Software developers practice software design patterns and principles to solve commonly occurring problems while ensuring extensible robust and maintainable system. The thesis aimed to study a subset of software patterns and principles. The practical goal of the thesis was to develop an iOS application with proper patterns applied. The main focus was to recognize which pattern would suit for an application's various design challenges and what benefits would be harnessed by it.

The study aimed to answer two questions: how a pattern needed to solve the problem was recognized and what were the consequences of applying the pattern. An action research method was followed for this project. It involved design, analysis and implementation phase for developing the application.

Xcode IDE was used as the development environment to implement the practical task. Application usability testing and profiling were done with Instrument tools which are part of the IDE. Debugging and compiling were done with Apple's new tool LLVM (Low Level Virtual Machine).

The result of the practical task is an iOS application named Finnkino which allows searching and viewing movie information in nearby theatres within Finland. In addition it also shows current, upcoming and top movies around the globe. The application lets the user bookmark a movie for later view. A movie trailer editing and saving feature is also available.

Based on the final result, it can be concluded that software design patterns help to recognize the applicability of design principles to software development. Thereby applying design patterns produces robust, long lasting and maintainable software.

Keywords	SDK, MVC, OCP, PLK, LLVM

Helsinki Metropolia University of Applied Sciences

Contents

1 Introduction 1

2 Theoretical Background 2

2.1 Object-Oriented Design Pattern 2

2.1.1 Model-View-Controller (MVC) 2

2.1.2 Façade Pattern 3

2.1.3 Observer Pattern 3

2.1.4 Decorator Pattern 5

2.1.5 Command Pattern 6

2.2 Object-Oriented Design Principle 7

2.2.1 Open-Closed Principle (OCP) 7

2.2.2 Principle of Least Knowledge (PLK) 7

3 Method and Material 8

3.1 RESTful Web Services 8

3.2 iOS Software Development Kit 9

3.3 Development Tools 10

4 General Structure of the Application 11

4.1 Application Features 11

4.2 Application Implementation 12

5 Model-View-Controller 14

5.1 Model Object Tree 14

5.2 Tying up the Model, View and Controllers 14

6 Façade Design Pattern 16

6.1 Motivation 16

6.2 Applying Façade Pattern in Finnkino Application 16

7 Observer Design Pattern 18

7.1 Motivation 18

7.2 Observer Pattern (KVO) in Finnkino Application 19

8 Decorator Design Pattern 20

8.1 Motivation 20
8.2 Realization 21
8.3 Design Challenge in the Finnkino Application 21
8.4 Solution 22

9 Command Design Pattern 23

9.1 Design Challenge in the Finnkino Application 23
9.2 Solving Design Challenge 24

10 Multithreading and Responsiveness 25

10.1 Responsive UI in Downloading and Parsing Data 26
10.2 Choosing Callback Pattern 27
10.3 Multithreading in FKMovieViewController 28
10.4 Caching in FKMovieViewController 29
10.5 Responsiveness in RTMovieViewController 30

11 Conclusion 30

References 31

Helsinki Metropolia University of Applied Sciences

1 Introduction

To solve programming challenges, the object-oriented programming (OOP) concept has been introduced. OOP gives us a concept of abstraction, inheritance, encapsulation and polymorphism to build a system by using reusable objects and hence reducing system maintenance costs. There might be multiple approaches to solve a certain problem. It is important to identify the correct solution and model the design of the system before beginning actual programming. This will allow us to build a flexible design, and hence be able to adapt with future requirement changes. Reusable objects are not of much use if our solution does not allow flexibility.

Simply having the OOP tools at our disposal is not enough to build a flexible solution. Design principles give us guidance on how to use OOP concepts in a proper manner to build flexible and maintainable systems. By following the design principles many design patterns were developed to provide reusable solutions to common problems.

My motivation to study design pattern is: knowing the patterns opens our eyes to model and apply a design to solve a certain programming challenge in a way which produces more reusable and flexible solution along with other advantages. It educates us to formulate and solve programming challenges in a more generic way, such as without knowing the details about specific platform and programming language. Knowing design principles and patterns allow us to apply programming experiences achieved in one platform to another. Programming languages are merely tools to solve programming challenges. If design principles and patterns are practiced, we will not need to throw away our previous experiences achieved during many years in a specific platform when shifting to a new platform. Instead we can use the old experience in a new platform to solve a problem even in a more sophisticated way and thereby enhance our programming career.

In the first section the principles and patterns are studied and in the following section they are applied to solve practical problems in a real-life iOS application. Design patterns are a broad topic. Each design pattern has diverse applications. In this thesis the scope of studying the patterns is only confined to the aspect of solving different challenges of the Finnkino application. The parts or discussion that are not concerned with solving the design challenges has been left out. The goal of this project is two fold. The

main goal is to analyse and study the software-related design patterns, loosely coupled software. It will be helpful for any developer working on any mobile platform to apply the techniques studied in the thesis. The secondary goal is to build a mobile application for users/customers who would use the application to check information about current, upcoming and top movies in a theatre in Finland.

2 Theoretical Background

2.1 Object-Oriented Design Pattern

Design patterns are reusable solutions to common problems in software design. They are templates designed to help writing code that is easy to understand and reuse. They also help one to create loosely coupled code so that it can be changed or replaced by alternative components in the code easily.

2.1.1 Model-View-Controller (MVC)

The Model-View-Controller (MVC) consists of three types of object as shown in figure 1. This pattern assigns objects in an application to one of these three roles: model, view, and controller. [1]

a) **Model:** Model objects maintain an application's data by encapsulating it. They are reusable in a similar problem domain because they represent knowledge that is applicable to a specific problem domain. Model objects do not interact with view objects directly. They provide logic to manipulate the date they encapsulate. [1]

b) **View:** The view displays information contained in the model. In other words, the view is the presentation of model that user can interact with. The view object has knowledge about how to draw itself and respond to user actions. Since view objects can work with many different model objects, they tend to be reusable across different applications. [1]

c) **Controller:** A controller class acts as a bridge a between the model and view class. The model and view class interaction is done through the controller and hence reusability of the view and model is possible. Besides it also perform set-up and coordinate tasks for an application. [1]

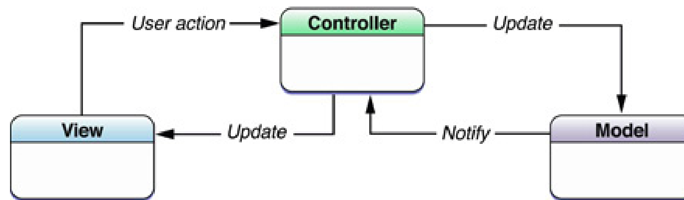

Figure 1. MVC Design Pattern. Reprinted from Apple Inc. [1]

Figure 1 shows user interaction with the view monitored by the controller and communicating any changes to the model.

2.1.2 Façade Pattern

The Façade design pattern is used to create a layer to abstract and unify a set of different interfaces in a subsystem. The higher level interfaces exposed to the user make it easier to use the subsystem by hiding lower level complexity. [2, 137]

2.1.3 Observer Pattern

The participants in the observer pattern are explained below:

a) Subject**:** Provides a gateway for observers to tie and untie with it. When the client class creates observers, it registers with the subject as shown in figure 2. Thus the subject knows who its observers are and it can have many observers. When the subject's property state is changed, it will call the "notify" method, which in turn calls the update method on all registered observers stored in an internal list. [3]
b) ConcreteSubject: Only notifying the observer is not enough. The Observer needs to know the current state of the subject. The task of the ConcreteSubject is to manage the internal state. [3]
c) Observer: Provides an update interface to receive a signal from the subject. [3]
d) ConcreteObserver: The subject notifies about its changed state but does not relay the information that was changed. The job of ConcreteSubject is to maintain a reference to ConcreteSubject. When the subject dispatches the notification, observers will make a call to the subject to obtain new data. [3]

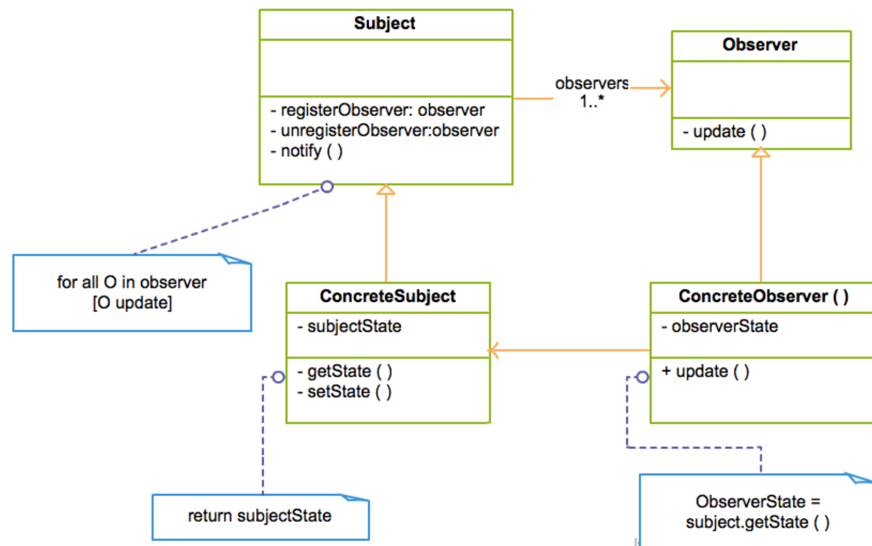

Figure 2. Observer design pattern class diagram [2, 166]

Three important aspects are shown in listing 1. First, observers are registered with the subject, which indicates the subject tracks observers. Second, observers keep a reference to the Subject, so that they can later retrieve the status after notified by Subject. Third, the client triggers the state changes by calling the setState method, which causes the notify and update method to be called in turn.

```
clientClass{
      void main() {
      Subject subject = new Subject();
      Observer observer1 = new Observer();
      Observer observer2 = new Observer ();
      subject.registerObserver(observer1 );
      subject.registerObserver(observer2);
     // assign subject to observer
      observer1.setSubject(subject);
      observer2.setSubject(subject);
     // set the state
      subject.setState(_state)
      }
}
```

Listing 1. Pseudo-code

2.1.4 Decorator Pattern

The Decorator Pattern attaches additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality. [4, 199] Participant classes in the decorator pattern are:

a) Concrete component: Is the object where new behaviours will be added dynamically. [4, 199]
b) Component: It is an interface for objects that can have responsibilities added to them dynamically. [4, 199]
c) Concrete Decorators: Extend the functionality of the component by adding state or adding behaviour. [4, 199]
d) Abstract decorator: Maintains a reference to the component object and maintains an interface conforming to the component's interface. [4, 199]

The participant classes relationship is shown graphically in figure 3 and figure 4.

Figure 3 shows that the abstract decorator class is unnecessary here since there is only one concrete decorator.

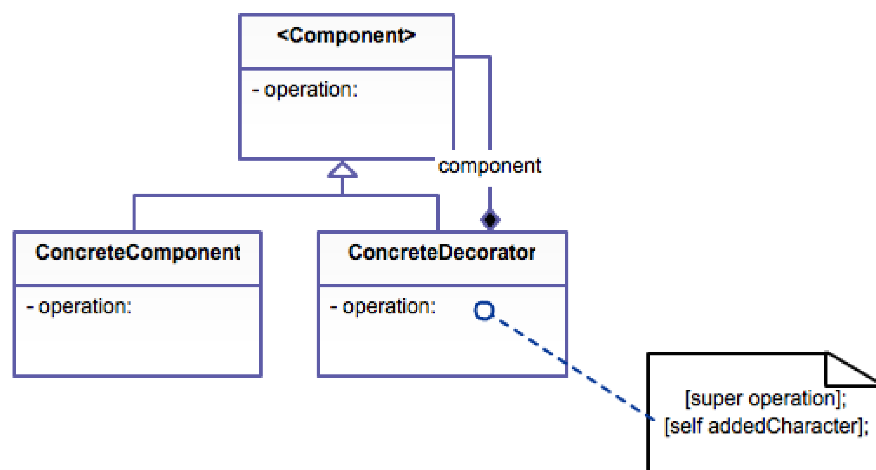

Figure 3. Class diagram without abstract decorator [2, 234]

It is shown in figure 4 that the abstract decorator class is forwarding a request to the concrete decorator class. It is only needed when one wants to add more than one responsibility. Its purpose is to forward the message to the concrete decorator.

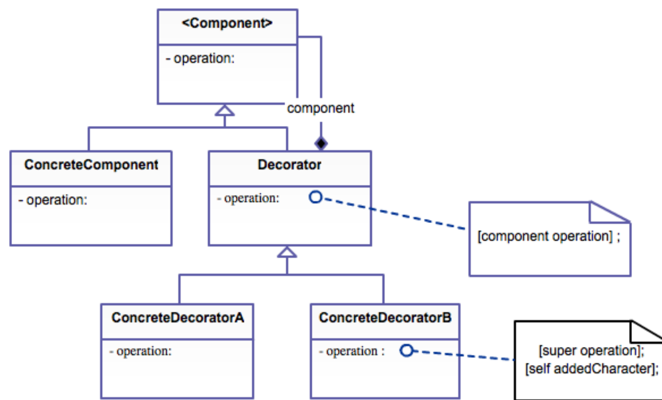

Figure 4. Class diagram with abstract decorator [2, 234]

Since there are multiple ConcreteDecorator class, the decorator class is necessary to forward the request to the appropriate ConcreteDecorator class.

2.1.5 Command Pattern

The participating classes in this pattern are the following:

a) Command**:** Is a generic interface for an operation to be executed. The invoker class only knows it. [4, 266-267]
b) ConcreteCommand: Extends the command class and implements the generic interface. It acts as an intermediary class between the Receiver and its action. [4, 266-267]
c) Receiver: Knows the details of how an action is to be performed. [4, 266-267]
d) Client: Creates different ConcreteCommand classes and sets the corresponding receiver. [4, 266-267]
e) Invoker: Asks the generic interface of the command class to perform the action. [4, 266-267]

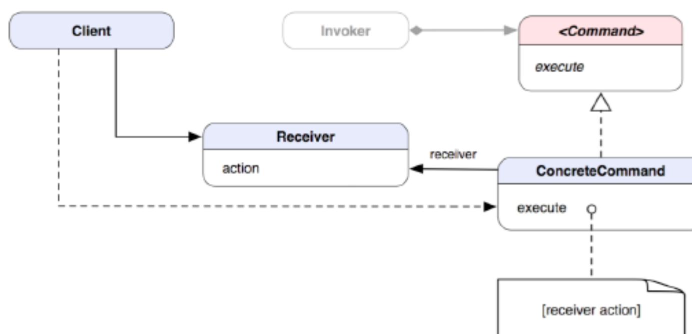

Figure 5. Restated from Pro Objective-C Design Pattern for iOS [2, 294]

As shown in figure 5, the client class creates every ConcreteCommand and hooks them with an invoker. Thereby each Command is encapsulated and saved in a queue for a later execution by the invoker class. This saving mechanism gives a chance to perform some pre action if needed, before actually carrying out the request. The ConcreteCommand responsible for the requested command performs an action on the receiver when invoker invokes the encapsulated commands that were saved in a queue.

2.2 Object-Oriented Design Principle

2.2.1 Open-Closed Principle (OCP)

Open-Closed principle states: "Classes should be open for extension but closed for modification" [5, 18].
Developers spent a long time and careful investigation implementing code. This particular piece of code is well tested and known to be working harmoniously with other pieces of code in the application. Allowing it to be changed will make it fragile by affecting existing functionality. Thus sometimes it is crucial to make sure that a class is not altered. This is ensured by the first part of the OCP. The second part of the principle states that classes should be open for modification. As we know sooner or later behaviour needs to be changed to support extension and/or modification as part of the ever-changing software requirements.

One of the many advantages of object-oriented design is the ability to extend functionality without requiring us to modify existing code. There are several ways to achieve this. While inheritance and composition are simple instances of OCP, we may need an extra layer of abstraction to make it possible to adhere to this principle. The decorator design pattern is an example that achieves this principle.

2.2.2 Principle of Least Knowledge (PLK)

The principle of the least knowledge known as the law of Demeter, states: "Talk only to your immediate friends". Concept of PLK is achieved in code if the following guidelines are followed. From Method (M) of any taken object (O) should invoke other methods that has following scope. [6]

- Object itself
- Object received as a parameter to methods but not objects returned from calling other methods.
- Any objects created within the method
- Direct component object.

3 Method and Material

The Finnkino application used contemporary tools and frameworks at the time of development. The focus of development was on the client side. For the backend I used the popular RESTful web services available online. This section overviews the tools and resources used to develop the application.

3.1 RESTful Web Services

Any kind of bulk information that has business values are stored and maintained by backend servers. They have uniform business interfaces to communicate with. Such interfaces offer the consumers CRUD services through which they are capable of interacting with the information. [7]

Considering the present application's scope it was not possible to implement backend services to provide data for users. Rather I utilized the RESTful services offered by a popular website Rotten Tomatoes and Finland's largest film distributor Finnkino Oy.

Table 1. RESTful resources used in Finnkino Application

Resource	URI Path	HTTP Methods	Distributor
Gets a list of currently showing events (e.g. Movies) in Finnkino theater	/Events	GET	Finnkino
Retrieves list of news	/News	GET	
Retrieves list of news category	/NewsCategories	GET	
Retrieves schedule for a date	/Schedule	GET	
Movies listed in box office	/box_office.json	GET	Rotten Tomatoes
Gets a list of upcoming movies	/upcoming.json	GET	

Table 1 lists all the RESTful resources used by Finnkino Application. The first four resources are from Finnkino web service and the last two are from Rotten Tomatoes website. Since they were third-party web services, the choice was only limited to what they offered. As seen in table 1, they only offered GET interfaces to obtain information. Other web services such as POST, PUT or DELETE were not allowed.

3.2 iOS Software Development Kit

iOS Software Development Kit provides tools and resources to develop native iOS applications. iOS architecture is multi-layered, as shown in figure 6. It acts as an intermediary between hardware and the applications that run on top of it. Although it is possible to use interfaces from any layer of the architecture to write the code, the recommended way is to use a higher-level framework whenever possible. [8]

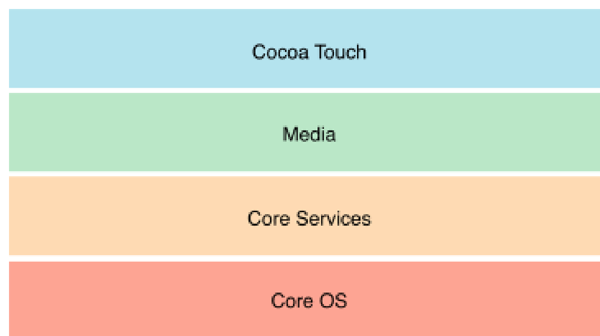

Figure 6. Multilayered iOS architecture

The system interfaces used to write iOS apps are called frameworks by Apple. The system libraries and functions used by Finnkino are listed in table 2. The following list overviews only the latest addition of the frameworks to iOS SDK that were utilized.

a. Storyboard: First introduced in iOS 5 SDK, storyboard holds the entire UI in a single file. It allows designing each screen and mapping transitions between the screens for user interactions. [9, 208]
b. Multi-threading: iOS SDK offers a multithreading programming environment. Multithreading capabilities are exposed via GCD (Grand Central Dispath) interfaces and NSOperationQueue. The latter is implemented on top of GCD. [10, 143]
c. Auto layout: First primered in iOS 6. It brings the capability to flexibly add and define view layouts. [15]

Table 2. List of all iOS frameworks utilized in Finnkino Application

Name	Description
AVFoundation.framework	Contains Objective-C interfaces for playing and recording audio and video.
AssetsLibrary.framework	Contains classes for accessing the user's photos and videos.
CoreData.framework	Contains interfaces for managing application's data model.
CoreGraphics.framework	Contains interfaces for Quartz 2D
CoreMedia.framework	Contains low-level routines for manipulating audio and video.
CFNetwork.framework	Contains interfaces for accessing the network via Wi-Fi and cellular radios.
Foundation.framework	Contains interfaces for managing strings, collections, and other low-level data types.
MobileCoreServices.framework	Defines the uniform type identifiers (UTIs) supported by the system.
QuartzCore.framework	Contains the Core Animation interfaces.
SystemConfiguration.framework	Contains interfaces for determining the network configuration of a device.
UIKit.framework	Contains classes and methods for the iOS application user interface layer.

Table 2 holds synopsis about the functionalities of the Frameworks. The detail information can be found in the iOS SDK.

3.3 Development Tools

Xcode is an Integrated Development Environment provided by Apple. Like any other IDE it contains an array of tools to facilitate application development [11]. The following section overviews a subset of Xcode tools that were used to develop Finnkino Application.

a) Compiler: Xcode 5 uses LLVM (Low Level Virtual Machine) as a default compiler and debugger. LLVM compiler is a major improvement over the old generation GCC (GNU C Compiler).
b) Interface Builder (IB): Previously Interface Builder used to be shipped as a separate application [13, 21]. The latest version of Xcode IDE has IB integrated. IB helps a developer to design the user interface by simply dragging and dropping the widgets on a design canvas. [13] It also allows designing the navigation and interaction of the UI.
c) Simulator: iOS simulators are used to run an iOS application and simulate the behaviour on a real device. In the Initial stage of development, developers do not need to run the application on a real device. Instead the simulator gives a

quick way to run and simulate the application, which makes the development easier.

d) Instrument: Instrument has a wide variety of performance analysis tools. For profileing Finnkino Application performance a subset of those tools were used. The Time Profiler tool was used to check time consumption of each method calls. It helped finding out the slowing bottleneck and thus indicating where to rewrite code to have faster excution. The memory allocation tool was used to analyze the application heap memory allocated by different objects. It was most effective to find out which part of the application consumes more memory and hence finding out a risky spot. The Leaks tool was particularly useful when finding out the object graph relationship and thereby preventing retain cycles.

4 General Structure of the Application

4.1 Application Features

Finnkino Application offers users various services to get information regarding films and events. It also allows the user to edit and save relevant information on the phone itself. This section will describe and document all the features available to the user. The following use case diagram defines interaction with the user. It represents a higher-level abstraction of available features.

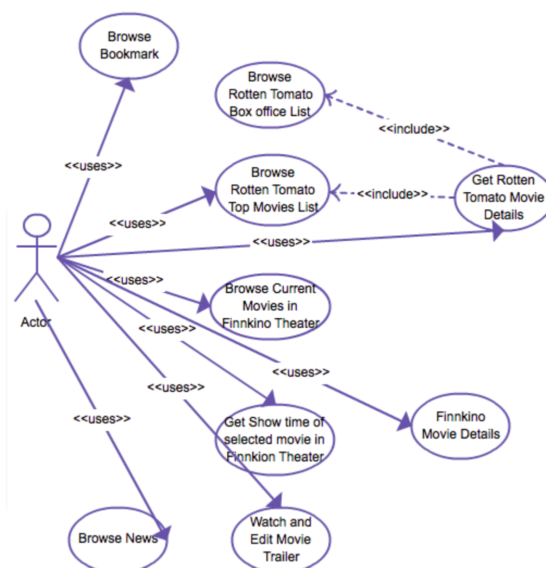

Figure 7. Use Case Diagram for Finnkino Application

The services shown in the use case diagram of figure 7 are described below:

a) Browse current movies in a Finnkino theater: This is the initial view a user sees on the application. It lists all the current movies and events in a Finnkino theater. Each block/entity in the view list represents single movie information, which holds: movie thumbnail image and title of the movie. This view also has a search field, which allows the user search for movie names with two different criteria: “Begin With” and “Contains”.
b) Get Finnkino movie details: From the list view mode when a user selects one movie it enters the detail view mode, where user can find the link to other views: watch trailer, read synopsis, add to favorites and get show time.
c) Get show time of a selected movie in a Finnkino theater: This view shows details about the movie theater name, location and time.
d) Watch and edit movie trailer: This view allows the user to watch the trailer of the user-selected movie and allows editing. The edit functions are trim, rotate, crop, add text and add song. After the editing is completed, the file will be saved in the phone video library.
e) Browse Rotten Tomato box office and top movies list shows box office movies and top movies list respectively from Rotten Tomatoes website. This information is international.
f) Get Rotten Tomato movie detail: This view shows critic information, audience score, genre, length, release date, title, starring characters and poster image.
g) Browse bookmark: This view allows the user to watch movie information that was previously bookmarked from other views. It also lets the bookmark list to be edited such as remove and undo remove.
h) Browse news: This view allows the user to browse different kind of news from the Finnkino website. The news are sorted and presented categorically.

4.2 Application Implementation

Figure 8 shows the high level abstraction of view controllers in Finnkino application. The main coordinating view controller is a UITabBarController, which holds four Navigation Controller. Four UIBarButtonItem on the UITabBarController allows navigation to each Navigation Controller. This Navigation Controller in turn holds the corresponding view controller of the views that the application user directly interacts with.

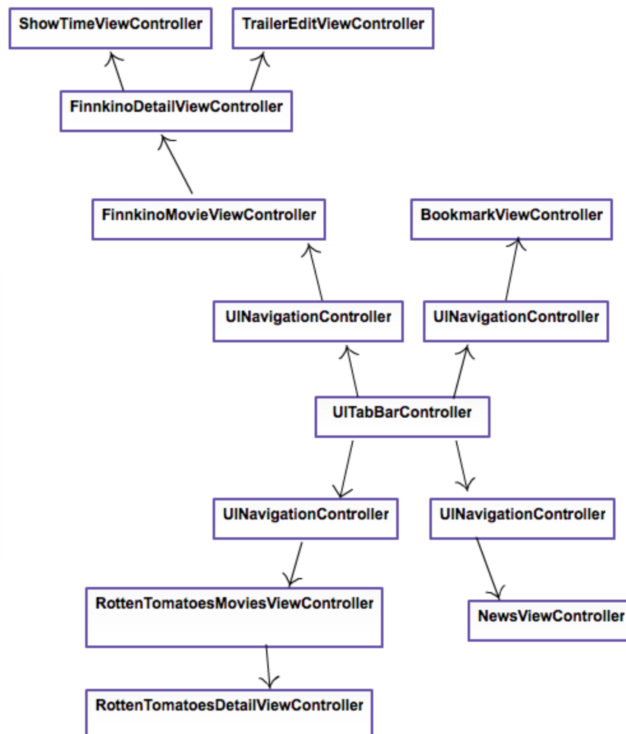

Figure 8. Coordinating view controllers in Finnkino Application

The following section shortly describes the view controller of the views that the user directly interacts with. The description includes the function and responsibilities of these controllers.

FinnkinoMovieViewController is a subclass of UITableViewController. It has a UISearchBar with index and UITableView to present data to the user. UIRottenTomatoViewController is also a subclass of UITableViewController. TrailerEditViewController has a player view to show the video trailer. It has play, pause button to control the playback and a menu, which holds export, trim, rotation and crop button. FinnkinoNewsViewController is a customized UITableViewController. Each of its cell holds another horizontally scorllable UITableView. The enclosing table views cell contains thumbnail images of the news.

Helsinki Metropolia University of Applied Sciences

5 Model-View-Controller

5.1 Model Object Tree

To make it easier to track the state of the XML tree I have separated the parsing logic in three different classes: FinnkinoEvent, FKOneMovieEvent and FKMovieContentDescriptor. Each of them implements the NSXMLParserDelegate protocol. Only one class can be a delegate of the parser at a time. So the delegate of NSXMLParser class needs to be changed when an appropriate starting tag is found and set back to the parent parser delegate class when a closing tag is found. Figure 9 shows the model object tree.

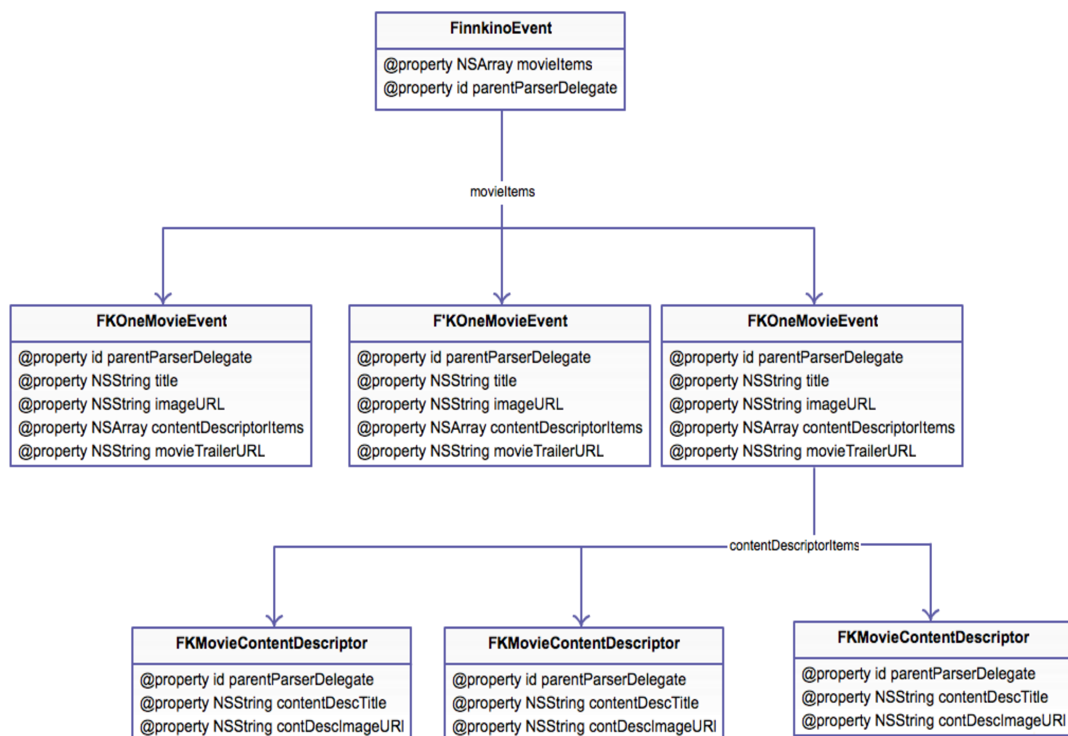

Figure 9. Model object tree for Finnkino Application

The model object tree encapsulates the parsed XML documents child elements information hierachially in object form.

5.2 Tying up the Model, View and Controllers

Figure 10 shows the MVC architecture followed in the Finnkino Application. It shows the boundary between each participant. By following MVC maximum the reusability has

been ensured. For example, view controllers can reuse the model object tree shown in figure 9 without rewriting it.

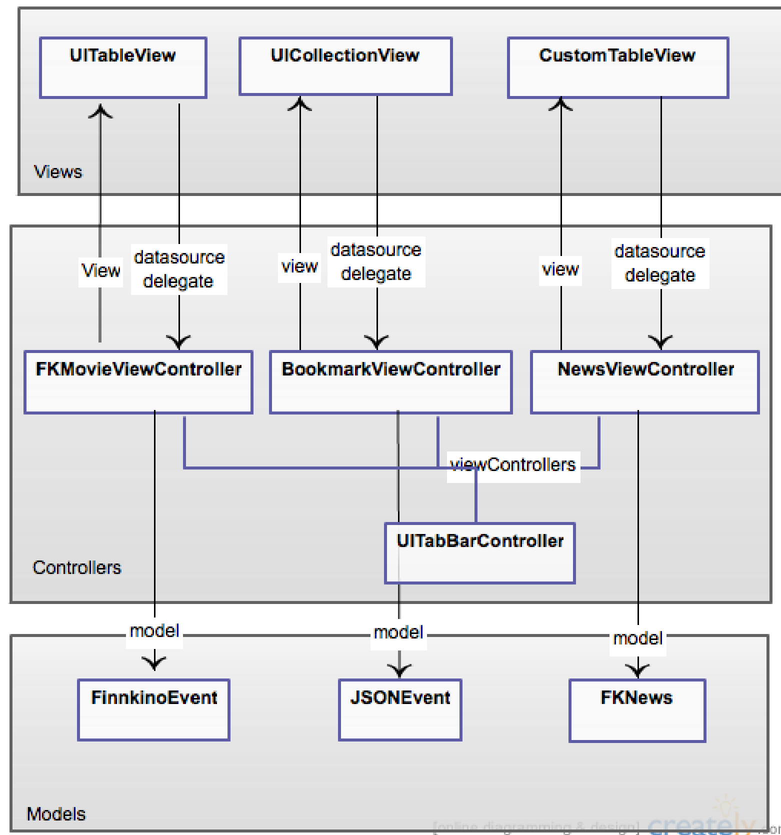

Figure 10. Finnkino object diagram

The role of the model, view and controller classes in the Finnkino object diagram shown in figure 10 are described below:

a) Model: FinnkinoEvent and FKNews map XML data payload downloaded from Finnkino web services as a model object tree. JSONEvent maps and holds JSON data payload from Rotten Tomato web service. They do not have any direct association with the user interface for presenting or editing it.
b) View: Generic views offered by framework such as UITableview and UICollectionView are reused by any application. This also holds true for custom views created in this application.
c) Controller: Acts as a middleman between view and model objects.

By following the MVC pattern, a clear separation between model, view and controllers has been achieved. The whole structure of the application is more reusable and easily extensible.

6 Façade Design Pattern

6.1 Motivation

There are three common situations when this pattern is useful.

a) Decoupling subsystem from clients: When one needs to modularize a system to subsystem components to reduce complexity. For better maintenance of those subsystems we need to minimize communication and interdependencies amongst them. The Façade design pattern helps to decouple the subsystem.

b) Simple interface for subsystem: Subsystems often gets more complex when different design patterns are applied to them. As the subsystems are evolved more subclasses are introduced to the system to handle underlying complexities. The façade design pattern provides a simpler interface to the subsystem classes.

c) Divide the subsystem into layers to reduce coplexities: A system may devide a subsystem into multiple layers. A façade class should define an entry point to each subsystem level. If the subsystems are dependent on each other, the façade simplifies the dependencies by communicating only through the façade entry points.

6.2 Applying Façade Pattern in Finnkino Application

The Finnkino application with and without façade looks like figure 11 and figure 12 respectively. In the previous MVC section, web service call and parsing logic was placed in view controller classes. View controllers are responsible for managing and updating their view. They should not be bothered with the details of dealing with external sources such as database, web service and file system as shown in figure 12.

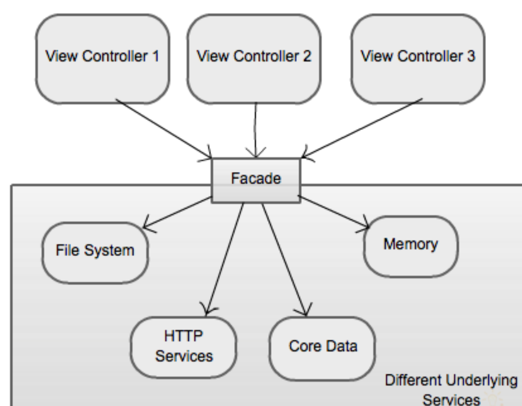

Figure 11. Implementation with Façade pattern

The client class (view controllers) should make a request to Façade class (FinnkinoFeedStore). Once the call has been made Façade class will call methods on the subsystem that handle communication with external sources. After processing it returns the control to the client class with the object it was interested in. The number of view controllers increased in the future would have caused havoc without Façade class since there would be redundant code, complexity of understanding the logic external source handling, strong coupling of client and subcomponents and hence less reusability.

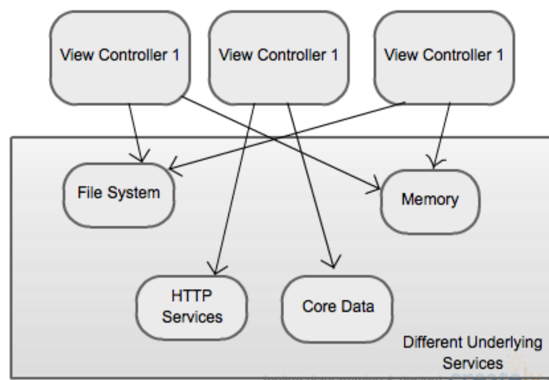

Figure 12. Implementation without Façade pattern

The sequence diagram shown in figure 13 shows the flow interaction from FinnkinoMovieViewController to the underlying classes via façade class FinnkinoFeedStore. The façade class hides the complex process by exposing a simple interface fetchRSSFeedWithCompletion: to the client classes.

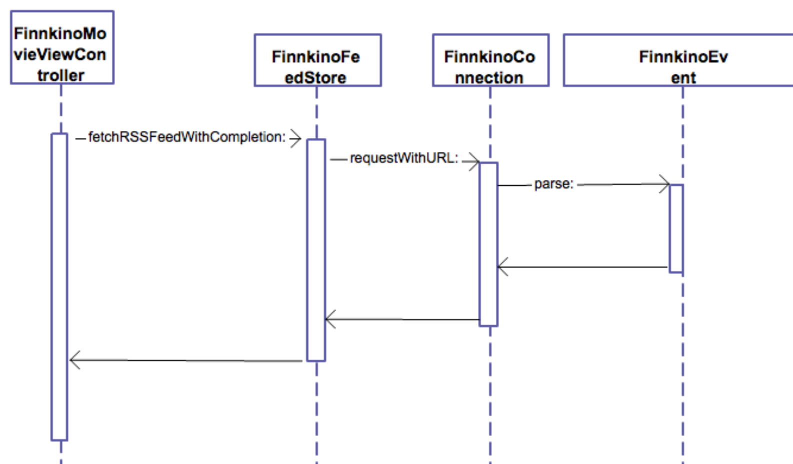

Figure 13: Flow of Finnkino Application via FinnkinoFeedStore from FinnkinoMovieViewController.

The consequences of applying façade patterns are the following. First, weak coupling is achieved by applying the façade pattern: underlying subcomponents can be replaced without affecting the client. For example, one will not need to change client code if the backend service is changed. Second, application programmers need to deal with fewer classes: reduces the number of objects the client deals with. Third, façade pattern also reduces dependencies of the external code on the inner working of the libraries. Fourth, façade pattern helps to reduce code redundancy.

7 Observer Design Pattern

7.1 Motivation

Object-Oriented programming is about objects and their interaction. This inherently requires one object to be informed about the changed status of another object. Interactions between objects are done via callback patterns such as delegate, blocks and target-action. In these callback patterns interacting objects need to have information about each other. The following describes the level of interdependencies between objects for different callback patterns.

a) Target-action pattern: Has exactly one action to be performed (this limitation is solved by using delegate pattern). One-to-one object relationship. Target object on which the action has to be performed needs to be known beforehand.
b) Delegate pattern: The concept of delegate pattern
    - A is delegate of delegating object B
    - B will have a reference of A
    - A will implement the delegate methods of B
    - B will notify A through delegate methods. Calling method selectors, which are declared in protocol, does this.

    The limitation of this pattern is that thedelegating object keeps a reference to other object. Communication is limited within the methods declared within the protocol, not just any method selector. Only one object can be a delegate at a time. As a result it is only good for one-to-one object relationship.[16]
c) Blocks: A method is passed around as a parameter to objects. It also lacks the one-to-many object relationship.

The discussion shows every interaction method has its own advantages. However if the design target is to design a one-to-many object relationship then observer pattern will be the right choice. The original observer pattern described in section 2.1.3 is not loose coupled. This issue is solved by NSInvocation and KVO mechanisms, which are the Cocoa adaptions of observer pattern. The Cocoa adaption (KVO) of the original observer pattern achieves the targeted goal.

7.2 Observer Pattern (KVO) in Finnkino Application

The goal to achieve in the FKTrailerEditViewController class is to be notified from different notifying classes: AVPlayer and AVPlayerLayerStatus, which requires a one-to-many relationship. Figure 14 shows how the observer pattern is applied to achieve the goal.

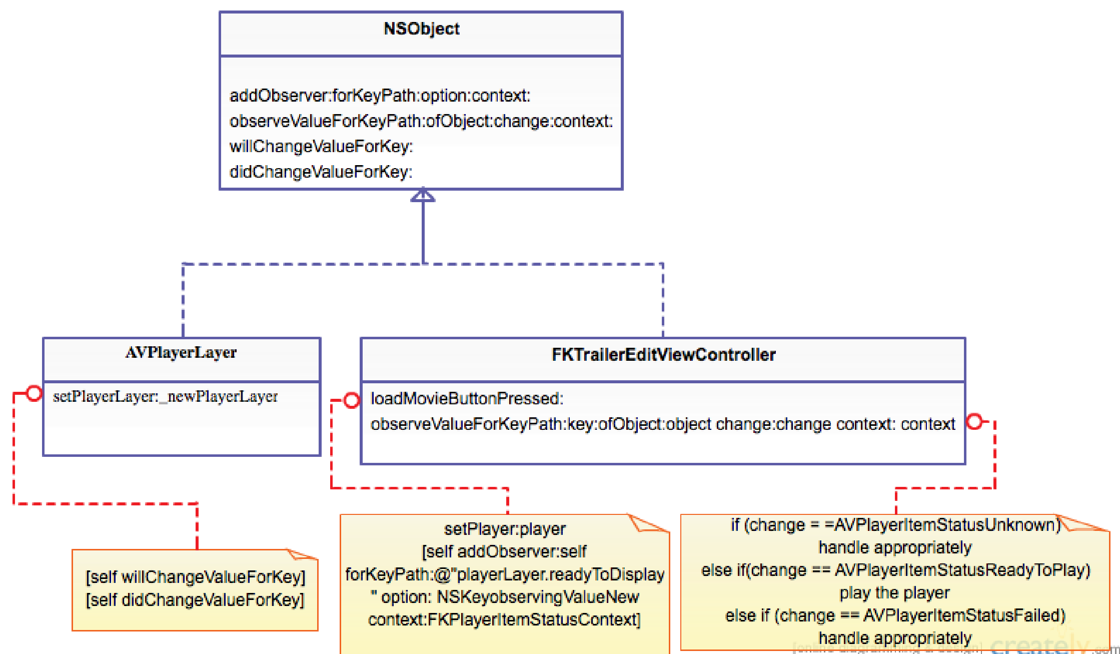

Figure 14. Observer Pattern applied in FKTrailerEditViewController

The following describes the mechanism of the class diagram in figure 14. The observer pattern is a publish-subscribe model. The observer subscribes to notifications from the subject. Once the subject needs to notify observers for any changes, it will start broadcasting the predefined notifications. KVO provides the similar publish and subscribe types of service offered by the original observer pattern and it offers additional benefits such as loose coupling amongst objects and flexibility of observing properties. There is

no extra overhead work in order to observe a property. We just have to conform to the NSKeyValueObserving informal protocol, which requires us to update the key matching properties through its accessor method. The framework will do rest automatically. With the help of auto synthesize we will not even need to write the accessor by hand.

In the Finnkino application the observer (FKTrailerEditViewController) class overrides observeValueForKeyPath:ofObject:change:context: to receive any callback changes in the subject (AVPlayerLayer). When AVPlayerLayer gets the property "readyToDisplay" changed, it will broadcast an update. There is no need to broadcast manually since Apple's default KVO mechanism handles it. WillChangeValueForKey: and didChange-ValueForKey: methods correspond to the "notify" method in the original observer pattern.

This loose-coupled pattern has some side effects. They are the following: First, it is not possible for the broadcaster to control who would be able to listen to the brodcast, in other words anyone can listen to the broadcast. Second, strict naming conventions must be followed by the methods for the automatic message broadcasting to work. Third, listeners need to be stopped observing manually before they are deleted. Otherwise the broadcasting message will cause the program to crash due to not being handled by the registered observers. [14]

8 Decorator Design Pattern

8.1 Motivation

To add responsibilities to an object we normally tend to think about inheritance or subclassing a base class. However there will be cases when subclassing is not an option. As an alternative Decorator pattern will help us out. Subclassing allows us two types of modification.

1. Method overriding: Allows child class to provide its own implementation of a method that is already provided in the superclass.
2. Adding new methods in subclasses.

When we use Decorator Pattern in Objective C as an alternative to subclassing there are two corresponding ways.

1. Subclassing an abstract interface, if the intended target is method overriding.

2. Category, if our intended target is to add new methods. However, the category approach is not suitable if our intended modification type is overriding since Apple documentation discourages it.

8.2 Realization

The following explains how to recognize when the command design pattern is the right choice to solve a design goal.

1. Class definition/implementation might not be available, for instance a framework class. Therefore, it simply may not be possible to make changes in the superclasses to support the subclass implementation.
2. Even if a class implementation/definition is available, a good design principle is that a class should be open for extension but closed for modification. [5, 18]
3. Superclasses are not available for subclassing. For instance in Java one can make a class Final to avoid extension. However, in objective C it is not a problem.
4. Too many subclass to support combination of extended features.
5. When features are optional or we want to add it dynamically at runtime.

These above mentioned reasons are justified in the next section by solving the Finnkino aplication specific design challenge.

8.3 Design Challenge in the Finnkino Application

The UIImage class has limited attributes such as size, scale and orientation. The goal is to extend its capabilities such as adding shadow at the edge and transformation to it. If one wants to support more features in the future, the number of subclasses will grow greatly to support all combinations. For instance 5 features would result in 15 subclasses. Moreover subclasses are instantiated statically, which means it will not be possible to choose the feature at runtime dynamically. So subclassing is not a flexible solution since client classes cannot control how and when to decorate the component. I wanted to add or change the properties dynamically at runtime rather than instantiating subclasses statically.

8.4 Solution

The target will be fulfilled if ImageComponent is able to handle all draw* methods. For that a common interface to share between the concrete component and decorator class is needed. We want to decorate the draw* methods since we are interested in the draw* methods of the UIImage class to decorate.

The concrete component UIImage and the concrete decorator classes such as ImageTransform filter and ImageShadow filter are subclasses of the abstract interface ImageComponent. So they have the same object type ImageComponent. Now the decorator class holds the reference to the component.

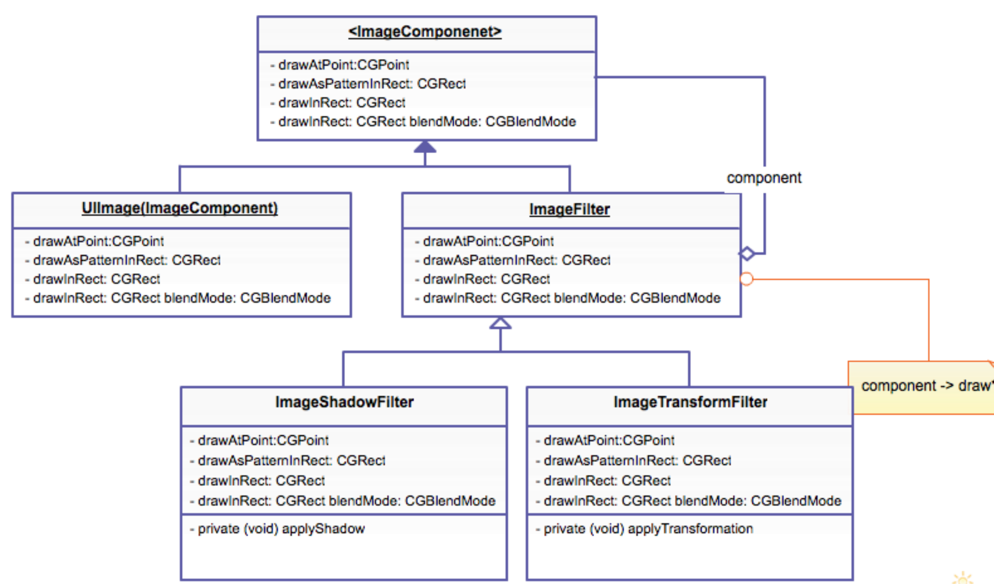

Figure 15. Decorator pattern applied to Finnkino Application

Descriptions of each participant class shown in figure 15 are given below:

a) UIImage (Concrete Component): Is an object to which additional responsibilities can be added. It returns the original image.
b) ImageComponent (Component): Abstract Interface for UIImage. It contains all draw* methods from UIImage, since these are the methods we want to decorate to.
c) ImageFilter (Decorator): Maintains a reference to an ImageComponent object. It simply forwards draw requests to its component, and decorator subclasses can extend this operation. The important characteristics of dynamically adding behaviour with the class is made possible by this composition and simple forwarding /delegating the object to the next desired decorator object.

d) ImageShadowFilter (Concrete Decorator): Handling drawInRect: message. Returns the original image with the shadow filter applied.
e) ImageTransformFilter (Concrete Decorator): Another concrete decorator handling drawInRect: message. Returns the orginal image rotated.

```
// Returns the original image
UIImage id <ImageComponent> *image = [UIImage imageNamed: @"Image.png"];
 [image drawInRect:rect];
// Returns the original image decorated with shadow
 id <ImageComponent> *decoratedImage2 = [[ImageShadow-Filter alloc] init];
[image drawInRect:rect];
// Returns the original image decorated with trnasfor-mation
id <ImageComponent> *decoratedImage1 = [[ImageTrans-formFilter alloc] init];
[image drawInRect:rect];
```

Listing 2. iOS code snippet showing usage of Concrete Component and decorator

As shown in listing 2, the same old draw* messages are called in order to instantiate both the original image and decorated images.

9 Command Design Pattern

9.1 Design Challenge in the Finnkino Application

Pseudo code shown in listing 3 is the logic of a menu action. Firstly it becomes difficult to read the code as the menu item increases. Secondly the logic to perform each menu action might be complex and the code might extend to a few hundred lines. Using the command pattern solves this problem.

```
- (IBAction) getAction:(id) sender
{
    int tag = [sender tag];
    switch (tag)
    {
```

```
            case kTrimIndex:
                 // Trim the video asset
                 break;
          case kRotateIndex:
               // Rotate the video asset
               break;
          case kCropIndex:
               // Crop the video asset
               break;
          case kAddMusicIndex:
               // Add Music to asset
               break;
          case kAddWatermarkIndex:
               // Add Text to asset
               break;
     }
}
```

Listing 3. iOS pseudo code

In listing 3, the 5 switch-case corresponds to the 5 menu actions in the TrailerEditViewController of the Finnkino application.

9.2 Solving Design Challenge

Instead of writing the complex logic in the client class we will move the required action to the object through encapsulation. Thus we will need to create object/class for each command. All these classes will inherit from the command interface to abstract the child action. After that an invoker class is needed to map the action to corresponding command objects.

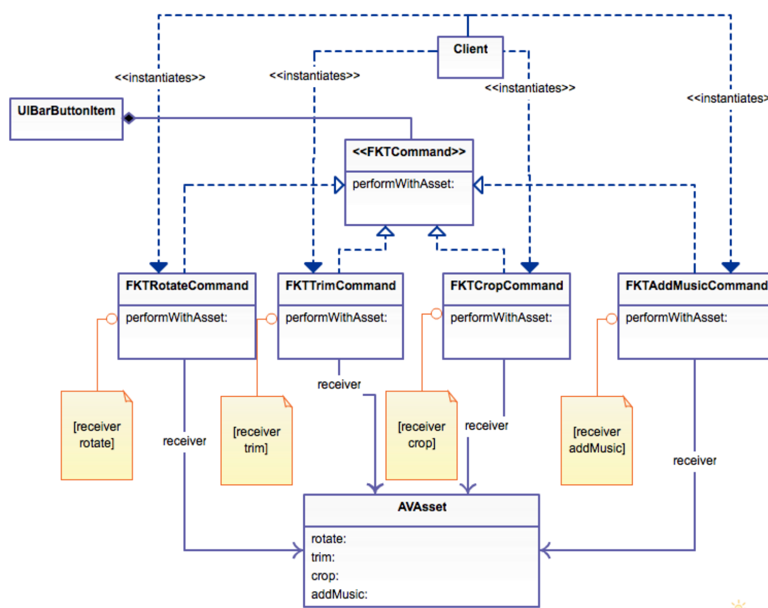

Figure 16. Command design pattern applied in Finnkino

The mechanism of communication among related classes in the class diagram shown in figure 16 are explained below:

a) Define command interface FKTCommand with a method signature -performWithAsset:.
b) From the command interface FKTCommand creates four derived classes, FKTTrimCommand, FKTRotateCommand, FKTCropCommand, FKTAdd-MusicCommand. These classes encapsulate the receiver class AVAsset. Each of them calls the receiver class's specific action method inside the -performWithAsset: method and knows what argument to pass to the receiver. By having that we have successfully encapsulated the receiver class, methods to invoke and arguments to pass in the concrete command objects.
c) Instantiate each concrete command object in the client class for deferred execution request.
d) The client instantiates the receiver object (AVAsset) and concrete command objects. After that it hooks up the invoker (in our case it is the UIBarButton items created in the storyboard) to call the command.
e) The invoker and in turn the user of the application decides when to call – performWithAsset: command.

The benefits achieved by applying the command pattern are the following. Firstly, it helped decoupling the client class and receiver class by means of encapsulation of commands into objects/classes and hence the client class does not have to worry about details of how to perform actions on the receiver. Secondly, this pattern enables extensibility since we can add new commands without altering the existing code. Finally, a more readable client class is achieved.

10 Multithreading and Responsiveness

The ultimate goal is to achieve the best possible and responsive user interface experience by utilizing the functions offered by the application development framework and hardware capabilities of the device. To reach this goal I had to make several choices. These are explained in the following sections.

10.1 Responsive UI in Downloading and Parsing Data

The task of the view controllers is to update the data to their corresponding views. It comprises two tasks: downloading the data payload (XML or JSON) and parsing them to hold in memory for later processing by the application. Downloading and parsing is a time-consuming task especially in a slow network and if the data payload is large. To keep the user interface responsive while data is being downloaded and parsed they need to be performed asynchronously. Implementing delegate protocols provided by the framework class NSURLConnection and NSXMLParser solves this.

As shown in figure 17, the FinnkinoConnection class is set as a delegate of NSURLConnection. When the connection initiates, the protocol methods implemented in the delegate class will be called to handle the downloaded data.

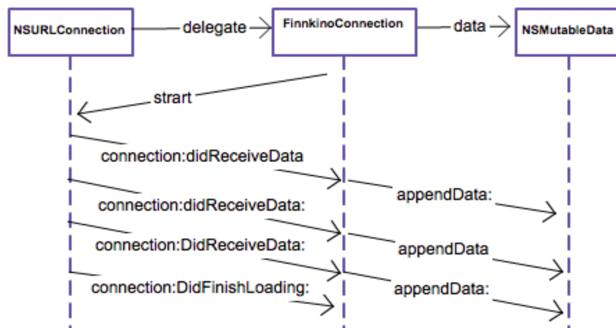

Figure 17. Data downloaded asyncronously

Once the data downloading is finished NSXMLParser is initiated and parsing is kicked off. As shown in figure 18, the parser class calls the protocol methods implemented on the class that are set as its delegate (FinnkinoEvent).

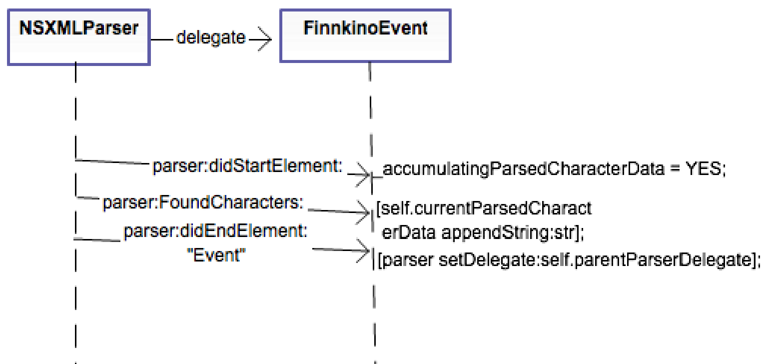

Figure 18. Data parsed asynchronously

The benefit achieved of this asynchronous nature shown in figure 17 and figure 18 is, the UI will not freeze while the data payload is being downloaded and parsed. The user can still navigate away to another tab. Other UI elements such as buttons and a search bar are responsive to user touch.

10.2 Choosing Callback Pattern

Since I had applied the façade design pattern in our application the downloading and parsing were handled in a separate class. The data just downloaded and parsed must be handed off to the consuming view controller. I had four choices for solving the callback problem. They are delegation, target-actions, notification and block.

a) Delegation**:** By analysing Delegation pattern as a solution of our callback problem I had found that it is more suitable when the interested class requires to be notified of multiple events and only one class can be a delegate at a time. Firstly, our view controllers are only interested in receiving already downloaded and parsed data. We do not need to be informed of multiple events. So it is an overkill to write protocols just for one event. Secondly, our façade class required to be created as a singleton. We know multiple classes might make a request to the singleton at the same time. So we cannot use the delegation pattern here as a callback.

b) NSNotification**:** Although NSNotification decouples the receiver and sender it is more suitable when multiple classes need to be informed of the same message. It is like broadcasting rather than straight communication between two objects. It is an overkill setting up and removing observers when our intention is to send the message to only one class.

c) Target-action: It is more suitable when we have a close relationship of two objects such as view controllers and one of their views, since we will have to know the method selector name to make a callback, which requires us to know the details of another class. Our callback is from a different class and so it would not be a good design practice to have such interdependencies.

d) Blocks**:** I chose block to solve my callback problem. The reasons are the following:

    - The callback is only going to be called once.
    - Handing off downloaded and parsed data to the client is a quick process.

10.3 Multithreading in FKMovieViewController

Now having the processed data handed off to the view controllers, they need to update their views with the data. The table view is updated with title, duration, genre and an Image. The title, duration and genre information were sent as string from the server. However the image was sent as an URL address instead of raw data, which meant we need to further download the image from the received URL address. If the image is downloaded in the main thread, the view will be stalling while the user scrolls of the table view. This is where multithreading offers a solution. Figure 19 shows how we make use of Multithreading to solve our problem. The figure only demonstrates the fundamental concept of threading in iOS application. For Finnkino Applications thread management I used NSOperation and NSOperationQueue. It is possible to use NSThread to implement multithreading, but it is harder to manage multiple threads. NSOperation is a higher-level class, which simplifies thread management.

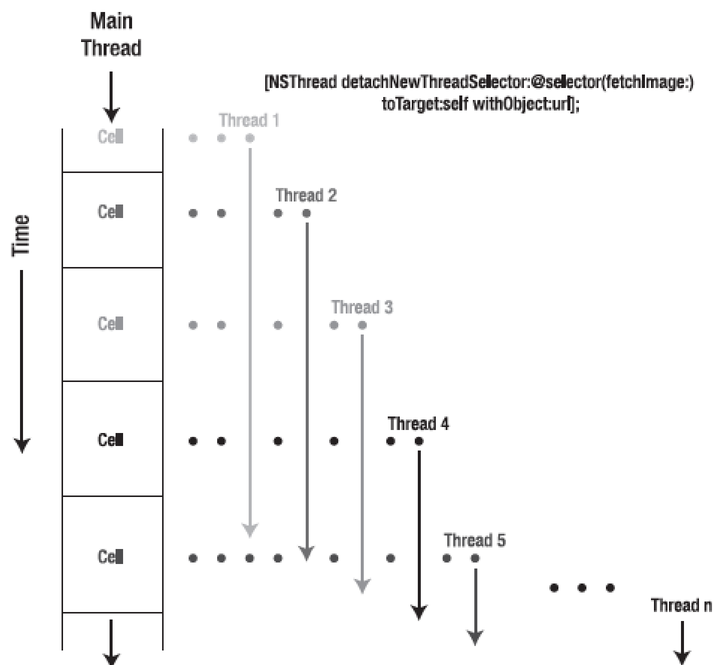

Figure 19. Multithreading concept, Copied from Danton Chin [15, 23]

How multithread functioning in the table view:

As the user scrolls the table view the image will be downloaded for the corresponding table view cell and the view will be updated with the image. If we download the image

on the main thread the UI will be blocking while image downloads. As a result the user will experience that table view is not scrolling smoothly.

10.4 Caching in FKMovieViewController

Data are cached in two steps. First the whole XML data payload is cached when FKMovieViewController's table view requests for data as shown in figure 20.

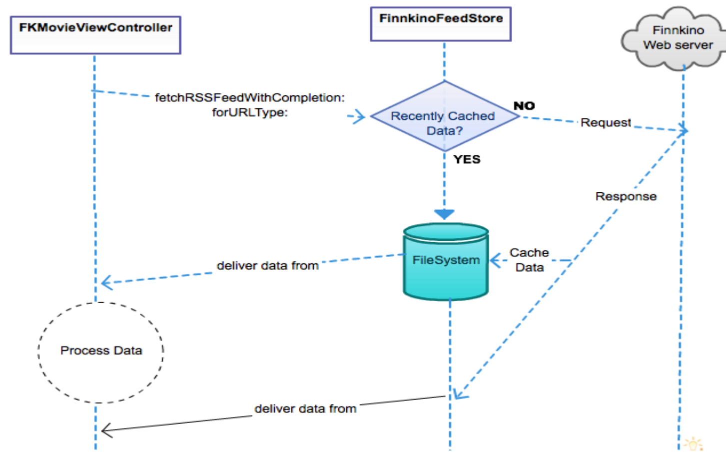

Figure 20. Caching mechanisms in Finnkino Application

The whole XML payload was cached, but the data did not include the image data. Instead it had the location URL of the image. The code is set up in a way that each table view cell makes a request to the image URL to load it inside of it. Now it is a performance penalty if each cell initiates the download request every time it is reloaded

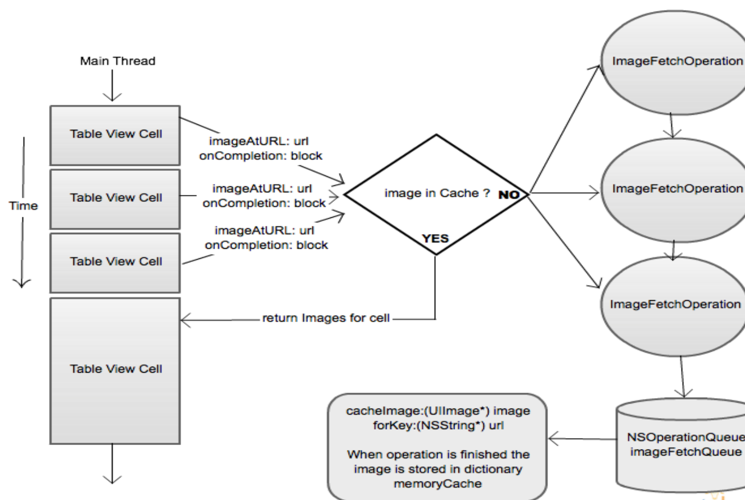

Figure 21. Image chaching in Finnkino Application

It is shown in figure 21 the solution to perfomance penalty is to cache the downloaded image in the file system and only initiate the download if it is not already in the cache.

10.5 Responsiveness in RTMovieViewController

Although RTMovieViewController does not request to cache the data, it has sophisticated functionalities comparing to FKViewController. The features are following:

a) The image is downloaded in the background thread.
b) The image will be filtered after downloading in the background thread.
c) The table view cell is only updated if the view has stopped decelerating and the user has taken the fingers off the screen. As the user scrolls away from the table view the off screen cells will not download or filter images for them. Cancelling the request to the download image of the already scrolled off-screen cells does this. In that way it stops making unnecessary network request to the server to download images and hence saves battery life.
d) Once the image has been downloaded the view is updated. It is updated again when the filtering has finished. In that way the user will have to wait less time to see the image.

11 Conclusion

The goal of the project was to develop an iOS application for users in Finland to browse different movie-related information, which was accomplished by successful completion of the practical part of the project. The second goal was to study the software design patterns on an iOS platform. The theoretical part of the project consisted of applying proven design patterns to solve commonly occurring design challenges. Simultaneously studying theory and applying it to the application helped gaining crucial understanding of software design patterns. The benefits, drawbacks and side effects of applying software design patterns are demonstrated in the thesis.

However, the application can be configured and functionalities can be extended for more features. The possiblity for improvement is open for extension and thereby study of various other software design patterns can be taken further. The study of these design patterns can be applied on many other development platforms.

References

1 Apple Inc. iOS Developer Library. Model-View-Controller [Online]. Septermber 2013.
URL: https://developer.apple.com/library/ios/documentation/general/Conceptual/DevPedia-CocoaCore/MVC.html. Accessed 20 August 2014

2 Chung C. Pro Objective-C Design Patterns for iOS. New York, NY: Appress Media LLC; 2011.

3 Kulandai J. Observer Design Pattern [Online]. March 2013.
URL: http://javapapers.com/design-patterns/observer-design-pattern/. Accessed 25 August 2014.

4 Gamma E, Helm R, Ohnson R, Vlissides J. Design Patterns: Elements of Reusable Object-Oriented Software. Indianapolis, IN: Pearson Education Corporatre Sales Division; 2009.

5 Knoernschild K. Java design objects uml and process. Bostom, MA: Pearson Education; 2002.

6 Bock D. The Paperboy, The Wallet, and the Law of Demeter [Online].
URL: http://www.ccs.neu.edu/research/demeter/demeter-method/LawOfDemeter/paper-boy/demeter.pdf. Accessed 25 August 2014.

7 Oracle Corporation. The JavaEE 6 Tutorial [Online]. 2013.
URL: http://docs.oracle.com/javaee/6/tutorial/doc/gijqy.html. Accessed 15 September 2014.

8 Apple Inc. iOS Developer Library. iOS Frameworks [Online]. Septermber 2014.
URL: https://developer.apple.com/library/ios/documentation/miscellaneous/conceptual/iphoneostechoverview/iPhoneOSFrameworks/iPhoneOSFrameworks.html. Accessed 12 September 2014.

9 Sadun E. The iOS 5 Developer's Cookbook: Core Concepts and Essential Recipes for iOS. Bostom, MA: Pearson Education; 2012.

10 Napier R, Kumar M. iOS 7 Programming Pushing the Limits. Chichester, West Sussex: John Wiley & Sons; 2014.

11 Knott M. Beginning Xcode. New York, NY: Appress Media LLC; 2014.

12 Napier R, Kumar M. iOS 6 Programming Pushing the Limits: Advanced Application Development. Chichester, West Sussex: Josh Wiley & Sons; 2012.

13 Apple Inc. Xcode The complete toolset for building great apps [Online].
URL: https://developer.apple.com/xcode/interface-builder/. Accessed 26 February 2014.

14 Gallagher M. Five approaches to listening, observing and notifying in Cocoa. [Online]. June 2008.
URL: http://www.cocoawithlove.com/2008/06/five-approaches-to-listening-observing.html. Accessed April 23 2014.

15 Danton C, Höfele C, Kazez B, Mora S, Palm L, Penberthy S. More iPhone Cool Projects. New York, NY: Appress Media LLC; 2010.

16 Galloway M. How best to use Delegates and Notifications in Objective-C. [Online]. June 2013.
URL: http://www.informit.com/articles/article.aspx?p=2091958. Accessed F23 ebruary 2014.

17 Echessa J. Getting Started with Auto Layout in Xcode 5. [Online]. June 2014. URL: http://code.tutsplus.com/tutorials/getting-started-with-auto-layout-in-xcode-5--cms-21016. Accessed 21 October 2014.