Use of design patterns for mobile game development

Astahovs Ilja
Abstract

Design patterns have been a valuable asset to software developers for a long time. All kind of software, including games, benefit from re-using the well known solutions to the common problems. Building a game from scratch requires a carefully made design and this is where design patterns come handy. However, the approach to game development has changed lately. The importance to learn the design patterns has been de-emphasized as frameworks and complete game engines emerge. Many design approaches provided by them are used out the box, often taken as some kind of standard.

The purpose of this paper is to show how some of the classical design patterns can be used in game development and how some of the modern technologies adopt them. To research the importance of design patterns in game development, a small mobile game project has been done. As problems arose, a number of potentially useful design patterns were identified and analyzed. Those design patterns were practically applied to the project and their potential use with other related technologies was discussed. The paper emphasizes the important role of design patterns in game development and gives some insights into how design patterns are implemented in some of the modern middleware.
# Contents

1 Introduction..............................................................................................1
    1.1 Design patterns: historical review..................................................1
    1.2 Design patterns nowadays: do we need them?...............................1
    1.3 Design patterns and game development........................................1
    1.4 A game implementation without using any of design patterns........2
    1.5 Project’s goals and limitations......................................................4

2 Design Pattern: Model-View-Controller.............................................5
    2.1 Case Study: Building a game upon Cocos2d game engine..............6

3 Design Pattern: State (State Machine)................................................7
    3.1 Case study: Using State Machine pattern together with Cocos2d.....7
    3.2 In Unity3d.....................................................................................8

4 Design Pattern: Singleton.....................................................................9
    4.1 Singleton in the example project....................................................10
    4.2 Alternatives to Singleton pattern..................................................10
    4.3 In Objective C / Cocoa...................................................................12
    4.4 In Cocos2d.....................................................................................12
    4.5 In Unity3d.....................................................................................12

5 Design Pattern: Observer....................................................................13
    5.1 Notification and delegation in Cocoa..............................................13
    5.2 Event dispatching in Cocos2d..........................................................14
    5.3 Key-Value Observing in Cocoa.......................................................14
    5.4 Case Study: View/Model synchronization for Car selection...........14
    5.5 Case Study: View/Model synchronization for NPC status changes...17
    5.6 Case Study: Building an Event System..........................................18
    5.7 State change notification in Unity3d..............................................19

6 Conclusions..........................................................................................20

References..............................................................................................22

Appendix A: Valet Parking 1989 for the iPhone.....................................24
    A.1 Project description..........................................................................24
    A.2 The gameplay..................................................................................24

Appendix B..............................................................................................29
    B.1 Cocos2d's CCDirector.................................................................29
    B.2 Cocoa's NSNotificationCenter....................................................29
1 Introduction

1.1 Design patterns: historical review
The notion of a design pattern (DP) as a well documented approach to solving some common problem occurring in software development was first popularized by a book “Design Patterns: Elements of Reusable Object-Oriented Software” by a group of authors’ also known as “Gang of Four”. Originally, the patterns were grouped into three categories: structural, creational and behavioral [GoF]. Other classifications include architectural patterns and concurrency patterns [18].

Evolution of programming languages and introduction of different design patterns is all about controlling complexity of software systems. The emerging use of DPs was the next logical step after invention of object-oriented programming (OOP) [Flynt], which emphasized component-centered development inside a framework. Applying DPs for solving recurring problems improves the object-oriented approach and makes it possible to deal with ever-growing complexity of applications. Understanding the problems solved by certain DPs helps to describe the purpose of different software components [Flynt, Core]. Instruments such as software development kits (SDKs) and frameworks used in software development evolve, and it encourages the programmer to reuse not only some basic components (e.g., reuse of components belonging to STL, Standard Template Library, in C++) but also complete modules (e.g., parts of game engines: game state managers, event systems etc.).

1.2 Design patterns nowadays: do we need them?
Even though the design pattern themselves were a product of refactoring [Flynt], one can argue how well design pattern-driven development fits into iterative and incremental design, which is what modern game-development process tends to be nowadays. In the modern era of agile software development, initial ideas quickly turn into prototypes, which in turn can rapidly transform into actual code blocks. Growing competition inside the industry, tough budgets and deadlines simply does not leave any resources available for perfectly thought-out design. Moreover, even though the OOP strove for re-usability (especially on framework-level, where most of the code in the final products is the code which belongs to the framework components) [Flynt], the re-usability of the custom code is significantly lower. Writing a re-usable code can cost three times more than writing a single-use code [19, 20], and integrating design patterns into the development process requires a lot of effort [Schmidt], so why would one bother paying extra attention producing a “nicer” design?

DPs can also be grouped into universal patterns and situational patterns. The former, as the name suggests, does not depend on language or technology, and often describe project’s overall structure, meaning that one can apply it to any project, only with minor changes. Because the purpose of this thesis is to give a more general insight into what is the role played by the design patterns in the modern game development and the choice of the technology is a much minor issue, this thesis will focus on the universal patterns.

1.3 Design patterns and game development
Using the DPs has proved to be useful for the design of general application software, especially web applications where many of DPs fit naturally into client-server model. But what about game applications, is there something specific about them? The specifics of game applications (similarly to other simulation software) are as follows: even small games require dozens of well organized classes, thus code modularization is required. Compared to web and business applications, games are more dynamic. The gameworld is subject to constant changes, which requires fast communication between game objects, as well as keeping visual representation

---

1 Eric Gamma, Richard Helm, Ralph Johnson and John Vlissides
updated. Games allow intense interaction between player and the gameworld. Games are also more resource-consuming, so one needs to find a good compromise between code complexity (size, inheritance levels etc) and performance. In contrast to web applications, the gameworld simulation (data) and its representation can be perceived as being more coupled together, as compared to business applications [2]. Finally, game development involves constantly chasing ever-changing SDKs, but does it necessary mean using new techniques to solve the old problems? This thesis will try to show the role of DPs in game development in general and whether the DPs in question are outdated or not.

Libraries and frameworks
Decisions on how to apply DPs when designing a game application depend on underlying technology. Few games are written from scratch today. Most games are developed by using existing 3rd-party libraries or frameworks², for example, a study from 2009 [DeLoura] shows that 55% of games which were in production at that time were utilizing a game engine. The study also shows the rising popularity of game engines and middleware. While adding libraries to the project usually does not affect the current design of the application, building a game upon an existing framework will directly affect the future design. Being partially-constructed applications, frameworks may adopt one or few DPs. Those design decisions propagate into user-written code. [Gregory]

1.4 A game implementation without using any of design patterns
What would an implementation of a typical game application look like if none of the DP were used? While sticking to object-oriented approach, let us forget about any further modularization (as in MVC which is one of the DPs discussed in this thesis) for the moment. Listing 1.1. shows a naïve way to put together basic game functionality: user interaction, update, simulation and rendering of the game world.

```cpp
GameObject* objects[]
GUIElement* elements[]
Player* player
Window* window

while (true) {
    // read player input
    if (key == DOWN) { update player's position }
    if (joystick is moved) { update player's position }
    // update each object belonging to the game objects
    for (each game object) {
        do physics simulation
        do AI
    }
    // render each object belonging to the game objects
    for (each game object in objects[]) {
        Window->render(object)
    }
    // render each GUI element
    for (each game element in elements[]) {
        if (element has been updated) {
            Window->render(element)
        }
    }
```

² here: game engines and game middleware which are structured as frameworks
Use of design patterns for mobile game development - Ilja Astahovs, 2012

What we got is a God object, a well known anti-pattern. As we add more features to our game, maintaining such “all-knowing” objects becomes a bottleneck of the project.

It is also worth noting that when doing rendering, the renderer needs to go through all the game objects to know which objects have been updated. The same applies to the GUI elements. We don’t use Observer pattern, so we can not notify the renderer about individual updates as soon as they happen.

Next, let us imagine handling mouse clicks during the game play without Delegation. If we don’t delegate it to Model (for moving the player inside the game world) or to GUI (for opening inventory), in order to figure out what to do we have to fetch all the required information from World or HUD4, respectively. Listing 1.2 illustrates the problem.

```java
if (mouse clicked) {
    // check each of the buttons belonging to GUI
    for (each button) {
        if (click is inside the button) {
            do smth
        }
    }
    // on the game world level, check if the user clicked on the empty space
    switch click
        case: click on the empty space
            player->move
        case: click on the other unit
            player->attack
}
```

Listing 1.2 Handling mouse clicks without Delegation

On the contrary, if we would delegate handling of the mouse events directly to GUI or game world, we could decouple "low level" player input handling from game logic details.

Let’s go further and add some “states” to our game (Intro, Main Menu, Game Play etc). To keep it simple, let’s use enum to map states to some integer numbers. If we don’t use State Machine pattern, our code would look like is shown on Listing 1.3.

```java
enum State {INTRO, MAIN_MENU, PLAY_GAME, …}
State state;
while (true) {
    switch (state)
        case State.INTRO :
            window->renderSomeMovie
            // read player input
            if (any key) {
                state = MAIN_MENU
            }
```

Listing 1.3 Minimal game implementation without using MVC pattern

---

4 heads-up display
case State.MAIN_MENU:
    // read player input
    if (new game clicked) {
        state = PLAY_GAME;
    }
    if (quit clicked) {
        break;
    }
}

Listing 1.3 States are managed in the main game loop

Without going further into details, it is obvious that such implementation results in several Monster-classes which are, in turn, closely coupled with other smaller classes. Such spaghetti code is hard to maintain. Some better structure is needed.

1.5 Project's goals and limitations
The goal of this project is to show the importance of DPs in the context of modern game development by presenting some practical examples of how different problems can be solved using today's instruments. Both software DPs (and they are many!) and the art of game development could be the subjects for dozens of different research project; this thesis does not pretend to do any deep research or come up with some brand new ideas. Instead, as the author of this paper is making his first steps into the world of game programming, the reader is presented with some basic problems which arise when building a game application, and the solutions involving the usage of certain DPs. Thus, the four DPs covered by this paper were chosen in a natural manner as the project evolved. The author doesn’t claim those DPs are the most important, but the paper certainly shows that they are sufficient to build a small mobile game.

Different types of games require different designs. For example, an approach used to implement a game of chess featuring a strictly linear turn-based user event driven gameplay will not scale up good if applied to a first-person shooter where a much more robust design is required. Moreover, just because of limited budget and production time, a simpler game does not need to re-use the same techniques suited for a big-budget title. The assumptions made by this thesis are based on a small mobile game (see Appendix A for the game description).

Other limitation is how well the choice of middleware to use for the case studies is justified. Cocos2d and Unity3D are two game engines referenced by this thesis, and they both are popular choices for making iPhone- and cross-platform mobile games, respectively. How well these engines represent the state-of-the-art tools for game development is up to the reader to find out.
2 Design Pattern: Model-View-Controller

Every game application in one way or another implements three basic functionalities:

- game world simulation (which defines rules of the game: both in the sense of explicitly defined rules the player consciously obeys, and implicit rules as physics);
- game world visualization (rendering);
- interaction with human player.

Each of them matches the components of MVC: *Model*, *View* and *Controller*, respectively. For an illustration of general usage of MVC in game design, see Fig. 2.1.[3, 4]

![MVC design pattern showing communication between modules and examples of components belonging to each module.](image)

**Model** is game logic, **View** is rendering, **Controller** is player's input and application flow not related to the gameplay. The Model does not depend on the View (which means the game simulation can run completely on its own without any further need of visualization). On the contrary, the View needs to somehow reference the Model in order to render the game objects (at the right position, with the updated set of properties). The Controller connects View to the Model to provide response to user input. On the iPhone device View usually provides visual elements representing the input controls, but it's recommended [6, 4] to use the Controller to translate the input into events which can be understood by the Model (more on that later, see section 5.6).

**Advantages:** to keep Model and View separated is essential for implementation which is built upon existing game- and/or graphic engines. View-Controller functionality of the application can to some extend be implemented with help of third party software. OpenGL, for instance, does View functionality, while DirectX library contain both View and Controller.

**Drawbacks:** in practice, when MVC is applied to small projects (including the example game), it brings a lot of additional code and complexity, which may slow down the production. Following the MVC pattern down to the smallest detail may require a lot of effort resulting in unnecessary levels of abstraction, unnecessary chain calls; sometimes a quick fix (e.g. mixing View- and Controller functionality together, or hardwire certain dependencies between modules instead of striving for pure separation etc.)
2.1 Case Study: Building a game upon Cocos2d game engine

Cocos2d is a game engine with support for 2D sprite graphics (using OpenGL ES API) and an integrated 2D rigid body physic engine, box2d. It is available on various platforms, and for the example project mentioned earlier Cocos2d for iPhone is used. It is build upon the Cocoa framework and adopts many of the techniques used by Cocoa, for example delegation. Its design resembles the MVC pattern. The core of Cocos2d architecture is presented by the following classes which fall into two categories: View: CCScene, CCLayer, CCSprite and Controller: CCDirector, CCEventDispatcher, etc.

Handling the player input

In the example application, the player interacts with the game environment by tapping the device’s screen. Delegation can be used by CCLayer (or CCSprite) to forward user input (player’s touches) further to some user-defined controller-type class [1]. TouchDelegate, as shown by Fig. 2.2. CCLayer receives the touch events from iOS and forwards them to TouchDelegate, which in turn identifies them and sends their coordinates to the current state. The current game state handles the touches accordingly to the current context: it could request the Model (Map) to determine whether the touch position co-insides with some car thus this car should be selected. Alternatively, if the game is in car driving mode the touch could result in the current car moving in requested direction. Finally, CCLayer "observes" the changes happened to the Model and updates the View accordingly (see Chapter 5 for detail).

![MVC pattern for overall design of the Valet Parking 1989 for iPhone](image-url)

Figure 2.2 MVC pattern is used for overall design of the Valet Parking 1989 for iPhone.
3 Design Pattern: State (State Machine)

The **State** pattern describes how an object can alter its behavior depending on its current state [GoF]. The solution usually involves a set of State classes each implementing a common interface. Each state describes the behavior associated with this state. Another class, a context class, stores a pointer to its state. Who is responsible for switching the states is up to the implementation. It may be carried by the states themselves, or by another object (a State Machine, which can be arguably implemented as a Singleton) [7].

In game development, the **State** pattern can be applied at least for two purposes:

- on macro level, to manage the overall application flow: different “screens” seen by user (the introduction screen playing some video clip, the main menu, the gameplay view, the credits, etc) can be implemented as separate states.

- to define behavior for game entities (artificial intelligence) in a structured way.

**Advantages:**

- gives better control over the application’s flow.
- divides the code into smaller self contained units.
- different resources can be dynamically allocated/deallocated between states allowing efficient memory usage: unload the artwork which corresponds to the previous level before loading the next one.
- this is a natural way for each state to have its own scope to define variables, that cannot be accessed from other state [8]. Can be considered a drawback, when used together with some game engines, e.g., *Unity3d* [9].
- efficient memory usage, because the assets required for the game can be loaded and unloaded between state transitions. It lets the game load its initial state (e.g., game intro) faster because it postpones loading of gameplay-related objects (levels, characters, etc.) until the corresponding states will be loaded [9].

**Drawbacks:**

- can be difficult to integrate into the existing game engines, and when done might introduce unnecessary complexity.
- the necessity to share data between multiple states. Though in Objective C / C++ project one can easily program the state methods to take parameters, in *Unity3d* the states are not allowed to send data to each other. Fortunately, one can still use *globals* (eventually, singletons) to store game information. *Unity3d* has also another built-in solution for objects to be persistent between scene transitions: `Object::DontDestroyOnLoad` function.
- different states sometimes may need to be combined. For example, one can introduce a separate state to display a menu, but what if the player wants to go to the menu while playing the game and be able return to the game immediately. This creates a problem: different parts of the application need to be run simultaneously [9].
- adds a lot of additional code, thus may be inefficient for smaller games.

### 3.1 Case study: Using State Machine pattern together with Cocos2d

Parts of Cocos2d game engine resemble a State Machine. *CCDirector* has functionality of a state manager, while objects of *CCScene* can be used as states. If combined with MVC, *CCScene*
becomes more View-oriented, and one may consider building a parallel state system for the Model, as done in the example project (Fig. 2.2).

**Overall game flow (transition between scenes)**
Because the SM pattern is such a powerful concept, one might want to integrate it within Cocos2d’s architecture. Ideally, each state should create its own scene upon initialization and store a reference to it. To keep states and scenes decoupled, one can use void pointers and virtual methods as shown by Fig. 3.1, thus every time some changes happen to the state it can ask the corresponding scene to update itself.

![Diagram](image)

*Figure 3.1 Each state has to implement some minimal state functionality*

Unfortunately, in cases where the implementation is split between Obj C and C++ code, the Scene class (Obj C) cannot implement a C++ interface. Therefore, Scene::update() cannot be declared as virtual, and State (C++) cannot call it. On the contrary, scenes are allowed to refer to states. A problem with this implementation is to always keep the scenes and the corresponding states synchronized. One possible approach is to use the Observer pattern (see Chapter 5).

After all, we do not have to build a state machine upon Cocos2d’s classes. For iPhone developers, [Zirkle] provides a pure C++ implementation of a state machine which works with Cocoa.

**Behaviour of the game characters**
The characters’ behavior (see Appendix for details) can be formalized by a state machine, which uses the states shown by Listing 3.2.

```cpp
enum Behavior {
    StayingInQueue,
    Away,
    WantsToLeave,
    Leaving
};
```

```cpp
Character::update(float dt) {
    switch (behavior)
    case behavior.StayingInQueue: ...
    case behavior.HasGone: ...
    case behavior.WantsToLeave: ...
}
```

Listing 3.2 State pattern applied to define character’s behaviour.

### 3.2 In Unity3d

Generally, Unity3d uses scenes to split the game into several independent modules. A global object Application is used to switch between scenes. This mechanism resembles a state machine.
4 Design Pattern: Singleton

A singleton is a class which is restricted to having only one instance [GoF]. It can be achieved by hiding the class constructor and introducing a method which calls the constructor only when there is no instance of this class available. Good candidates for Singleton are different managers used to load resources (e.g. texture managers, sound managers), user input managers, and also objects which need to keep track of global game information (e.g. game score). Because of its purpose, singletons are often given a global scope, and this is why the Singleton can be perceived both as pattern and anti-pattern.

As the definition suggests, the main advantage of Singleton is that only one instance of the class can exist at any given time.

- Object becomes global and can be accessed from any class, the only thing one has to do is to add an `#include`. One do not need to pass references to the object of this class to other classes. Arguably, this makes the code easier to read, on the other hand, this makes the classes coupled together (more on this later).

As for the drawbacks, they are many:

- One have to make sure the user can not instantiate the singleton other than by using `getInstance` method, otherwise, we get bugs which are very hard to find (class properties get suddenly modified from "nowhere"). A commonly used way around this problem is to make default constructors private [Rucker]. However, it can’t be directly applied to Objective C class because in Objective C there is no notion of private methods.

- Abusing of the pattern can introduce too many global variables. If overused, one suddenly ends up with a bunch of “managers” visible from any part of the code.

- Using Singletons hides the dependencies (cannot be seen by looking at interface). Dependencies should be explicitly passed between classes as function (method) arguments [10]. One can argue, whether the dependencies get hidden completely because one can still figure out which classes depend on the singleton by expecting their interfaces trying to find `#include` statements. On the other hand, the `#include` statement may be included into the implementation file, or even worse, some `#includes` may be presented in the code by mistake (if someone forgot to remove them).

- Writing unit tests is difficult (because the subject to the test may depend on some globally available singleton).

- Problems with memory management: a singleton object itself does not know whether some other class keeps a pointer to it or not (being a global object it allows any other object to refer to it), so it does not know when it can be deallocated. Thus every singleton is potentially a memory leak [Si2, Rudy]. One can argue and say that the Singleton is usually applied to the classes which are instantiated when the application starts and remain in use all the way down until the application quits, thus deallocating the object when the application finishes can solve the problem.

- The pattern can easily lead to MVC violation: a new programmer unaware of the present design can easily access Model directly from the View, by simply including (`#include`) the corresponding Model class, thus mixing Model and View functionality [10]. Generally, it also violates a good object-oriented design: a class does not need to know about another class, unless it the task requires it.
4.1 Singleton in the example project

Some of the classes belonging to the game’s core functionality are designed as singletons. They include GameLogic, CarManager, CharacterManager, StateManager and MapManager. Strictly speaking, some of these classes (e.g. MapManager) could be equally well implemented by means of static methods, because they don’t have any instance variables, and all its methods can be made static. The main reason for using the Singleton DP was to make the “managers” and the “game rules” available across the whole Model. The first approach was cascade passing of arguments through the object hierarchy and using chains of getters, but after a while it became a problem: there were too many such method calls and wrapping the head around the whole hierarchy every time another call was needed became a tedious task. It became clear that the design had to be reworked. On the other hand, one could agree with [10] that one possible solution might be to optimize the hierarchy, rather than going with Singleton.

4.2 Alternatives to Singleton pattern

Passing the required object as a method argument

If some class needs additional functionality, for example a game state (PlayGameState) might need to read new maps from .xml files, it might get another class, MapManager, sent to it as an argument as shown by Listing 4.1.

```cpp
class PlayGameState {
public:
    void loadMap(MapManager* mapManager);
}
```

Listing 4.1 Passing the required object as a method argument

This approach certainly works for the given example, but sometimes these additional arguments don’t suit the method’s signature. Consider we want PlayGameState to log something into a file when loading a map. This would require us to extend the signature of loadMap as follows:

```cpp
void loadMap(MapManager* mapManager, Logger* logger);
```

which makes code harder to read and follow.

Referencing global objects using the extern mechanism (C++)

In C++ an object can be defined in one place and be referenced by extern mechanism. The binding happens at linking phase, after the code has been compiled. A problem with this approach is that it creates a lot of dependencies [Dalmau].

Inheriting the required functionality

Another option (proposed by [10]) is to strive for a shallow but wide inheritance hierarchy between classes. Back to the example game, any class who wants to be able to read from files has to inherit this functionality from the base class, ResourceLocator, as shown by Fig. 4.2. The advantage of this approach is that only those classes inheriting from ResourceLocator have access to the required functionality, thus we made ResourceLocator available to those classes without making it visible to the rest of the application. Fig. 4.3 illustrates the same approach but the hierarchy is a bit deeper.

One drawback of this approach is eventual risk of running into diamond problem\(^5\), as illustrated by Fig. 4.4. Different programming languages take different ways to approach this problem, but

\(^5\)“Diamond problem” - an ambiguity that arises when two classes B and C inherit from A, and class D inherits from both B and C” [Wikipedia]
given that some languages do not allow multiple inheritance (e.g. Objective C), the safest way would be to avoid such structures.

![Diagram of inheritance hierarchy]

*Figure 4.2 Inheriting the required functionality: shallow but wide inheritance hierarchy*

![Diagram of inheritance hierarchy with static variables and methods]

*Figure 4.3 Inheriting the required functionality as an alternative to Singleton*

**Using static variables and methods**
The Singleton and the concept of static variables and methods have a lot in common. One major difference though is that a class which declares static members is not restricted to one instance.
An advantage of Singleton compared to the static variables is *lazy initialization*, which allows to save the memory.

**4.3 In Objective C / Cocoa**
The Singleton is a common DP for the components of Cocoa framework. Within the framework, such components are called “shared”. Usually they have the role of “managers”: UIApplication, NotificationCenter etc. Listing B.2 shows some code from NotificationCenter.m, which is an example of classical Singleton. Generally, Apple does not discourage the programmers from using singletons, even more, it provides the guidelines [11].

**4.4 In Cocos2d**
Cocos2d adopts Singleton pattern, using the classic implementation: see Listing B.1. CCDirector corresponds to what generally is known as game manager, i.e. the top-level class of the game framework hierarchy which creates the main window and does the job of switching game states (scenes). It can be made available to any class simply by importing (#import) CCDirector.h. Another application for the Singleton is CCTouchDispatcher, a Controller-type component which serves a event system for handling of touch events.

**4.5 In Unity3d**
Unity3d allows access to “global” run-time data via the Application class. This class is visible from any script by default, no #include is required. Application is a collection of static variables and methods. Worth noting, there is another interesting mechanism, which allows “global” user-defined objects: the method Object.DontDestroyOnLoad, which allows the objects to exist between the scenes. [12]

---

*[lazy initialization* - the tactic of delaying the creation of an object <...> until the first time it is needed” (Wikipedia)
5 Design Pattern: Observer

The Observer DP lets an object be notified when other object’s state changes. There are many cases when Observer can come handy, but let us focus on how it solves a problem of communication between the logical and visual representations of the game. This problem arises naturally when a modular approach, such as MVC, is used. To keep the View and the Model synchronized, one needs to provide a simple yet resource-effective way for View to be notified of changes which happen to Model's state. Such change notifications is a part of game’s event system. The implementation of the Observer is as follows: for each object whose state may change, create a list of dependent objects (listeners). Each of the listeners shares a common interface, Observer, and each has its own implementation of handleEvent(event) function. This function is called every time the object’s state is changed. The event which is sent as an argument to this function may include all the necessary information which View needs to update itself, or it may contain a reference to the object which sent the event.

Advantages of using the Observer are as follows:

- The object which is subject to state-changes can broadcast notifications about its status changes to all of its observers.
- The object does not need to know the exact type of its observers, which means it does not have to include any additional headers. All its observers are just of type Observer.
- If an Event System is used, the object does not even need to manage the list of its observers, because it is done by Event System.
- If as Event System is used, is is easy to introduce new types of events.

A typical game engine usually has some mechanism for handling various application-wide events, so called event system. The way such event system is implemented usually depends on what is used by the underlying architecture (framework, etc.). For instance, Cocos2d has borrowed idea of delegation from its underlying framework, Cocoa. Other event system to mention are Microsoft Windows’ window messages, Java's AWT, delegate and event in C# [Gregory].

There are two types of changes that may happen to the Model: changes which occur as an immediate response to user input (e.g. tapping the screen may select a car), and changes which are triggered by internal game events, timers, etc. (e.g. character’s patience meter rises as time is passing). How to handle both of these events will be discussed shortly.

5.1 Notification and delegation in Cocoa

Cocoa’s framework has its own mechanisms for creating and managing connections between objects. It introduces two similar concepts: Delegation and Notification. Delegation occurs when one object handles a task over to another object. For example, every visual component in Cocoa has a delegate which handles how the component responds to user input (e.g., touches). A delegate is specified dynamically, e.g. via dependency injection. For the component to function properly there has to be a delegate linked to the component. When delegate’s method is called, a value is returned, thus Object-to-Delegate relationship is 1-to-1.

On the other hand, Notification pattern is a variation of Observer DP. The object (notification sender) is allowed to have multiple observers. The object does not send notifications directly to its observers, instead, all notifications are posted to Notification Center. The observers need to register with the Notification Center to receive notifications. It is possible for observers to receive different types of notifications from multiple senders, thus a many-to-many relationship.
Apple recommends using Delegation over Notification when there's only one observer which responds to object's changes. [5]

5.2 Event dispatching in Cocos2d
Cocos2d relies heavily on Observer DP to handle touch events. It

`CCTouchDispatcher` has a list of observers (called *delegates*). Each object can implement `CCStandardTouchDelegate` protocol to handle touches: it needs to register itself with `CCTouchDispatcher` by calling `CCTouchDispatcher::addStandardDelegate`. Then, every time `CCTouchDispatcher` receives a touch it broadcast it to all of its delegates.

5.3 Key-Value Observing in Cocoa
Another Cocoa's technique, *Key-Value Observing (KVO)* [14], allows observers to be notified when one of the object's properties has been changed. Cocoa defines a protocol, `NSKeyValueObserving`, and each class implementing this protocol can register other classes as its observers. For instance, by inheriting from `NSObject`, any class gets a ready-to-use method `addObserver forKeyPath:<...>`. Thus, from the programmer's perspective, only minimal code is required to set up the Observer/Subject relation. How the corresponding references are stored and how the notifications are dispatched is hidden from the programmer.

What happens under the hood is the following [13]: when an object registers an observer for a certain property (`key`), the object is subclassed and the setter for that `key` is overridden to include code to notify the corresponding observers. This requires the setter to be used each time the key needs to be changed, therefore to be observable each key has to be *KVO-compliant*.

5.4 Case Study: View/Model synchronization for Car selection
Depending on where on the screen the touch occurs, there are different scenarios possible. One possible scenario: when the player taps over a car, it results in player taking over control of this car (the car gets "selected"). This Model event should trigger View update, which in turn should, among other things, center the view on the currently selected car and show its owner's portrait (i.e. update the HUD).

A reasonable question to ask is why not update the HUD every frame as we do for the game world? We can send a query to Model each frame to get an owner of the currently selected car, and then we can fetch the corresponding sprite. We can erase and re-create the contents of the HUD between each frame. Indeed, rebuilding the HUD every frame will keep it updated. What is different though, compared to redrawing the game world every frame, is that most of the HUD elements are unlikely to change every frame. The game score, the portrait of the character whose car the player is driving at the moment, the queue of customers – all these elements may remain unchanged for longer periods.

**Solution alternative #1: using method's return value**
A trivial solution might be to create a method `<Model>::tryToSelectACarAtPos(pos)` which, upon success, would return an instance of `Car`. When the user taps the screen, this method is called. Then, depending on the return value, the `View` can immediately update itself, as shown by Fig. 5.1.
Drawbacks

What if tapping the screen can have different results depending on the state of the Model? In our case, the user taps the screen when he/she wants to select a car. He/she also taps to steer the car that already have been selected. In case of design where Controller is totally unaware of any game-related details, only Model knows how the game world will react on taps at the very current moment. That’s why the only thing Controller can do is to call \texttt{<Model>::playerTouchesOnce()} method.

Solution alternative #2: polling the Model for changes

As the main game loop runs, View checks for Model updates every frame. Model maintains a number of flags (booleans), one flag for each event the View could be interested in. Thus, Model could use \texttt{carWasSelected} and \texttt{carWasDeSelected} flags to be used when the player has selected or has deselected a car, respectively. As soon as the View notices that \texttt{carWasSelected} became true, it immediately updates itself and sets \texttt{carWasSelected} to false.

Drawbacks

This approach uses more CPU resources. While not having any noticeable impact on game’s speed when the project is small (i.e. number of objects the player can interact with is small), doing more stuff each frame will inevitably reduce frame rate as number of objects increases. One possible way to save CPU time would be doing update check once per n frames (as opposed to doing it every frame). In the example game, a new guest arrives once per few seconds, thus there is no need for HUD to check for guests queue updates every frame.

As functionality grows, introducing and maintaining new flags can become a tedious task. Extra caution is needed to keep the flags updated. Code which does polling can become a huge switch statement.
**Solution alternative #3.a: a list of observers**

Each object of Car can have a list of observers interested in receiving a notification when the corresponding car gets selected. One can use dependency-injection to add new observers during run-time.

**Limitations and restrictions of the example application**

The example application is implemented partially in C++ (Model) and Objective C (View and Controller), which imposes certain restrictions on interaction between Model and View. According to Observer pattern, View has to inherit the Observer interface and define its own implementation of notify() method. For Model (which is purely in C++) to be able to call myObserver.notify(), Observer has to be implemented in C++. Because View is implemented in Objective C it cannot subclass a C++ class. Thus, View cannot subclass Observer and there is no way for Model to keep a reference to View.

The same problem when GameLogic needs to notify PlayGameLayer; GameLogic cannot register PlayGameLayer as its observer, because PlayGameLayer cannot implement Observer interface.

**Solution alternative #3.b: setters do notification dispatch**

Extend the setters’ functionality: when called, setters post notifications to all registered observers.

**Drawbacks:** requires a lot of additional code, adding new properties would mean writing a lot of bookkeeping-code.

**Solution alternative #4: posting notifications to Notification Center, Obj C**

See Fig. 5.2 for details.
5.5 Case Study: View/Model synchronization for NPC status changes
In the game, each non-player character (NPC) has an attribute, patience, which changes over

time. The game's HUD shows a portrait and patience meter for some NPCs. Each time the patience
level changes the corresponding picture of patience meter has to be updated. The problem is how to notify the HUD of such changes.

Solution alternative #1: polling the Model for changes
As in previous example, polling the Model for changes each frame would impact the performance. Keeping the HUD updated would require either redrawing the HUD each frame (which means manipulating the corresponding sprites each frame) or keeping track of all updates manually (which is, in this case, an unnecessary complication).

Solution alternative #2: adding a listener, pure C++ implementation
In pure C++ code, dependency injection can be used to establish a connection between an NPC and the HUD.

Solution alternative #3: Key-Value observing, pure Obj C
For this solution to work, Character class has to be implemented in pure Obj C. The implementation details are presented by Fig. 5.3.
The benefits of this approach are obvious:

- no additional bookkeeping-code is required (flags etc.). All required code is a pure functionality, the rest is taken care by Cocoa;
- one can easily add new properties to Character, (e.g., health) together with the corresponding HUD elements (health bar).

The initial ambition was to separate the application into Objective C/C++ and pure C++ code, which led to the problems when trying to adopt some of DPs, e.g., Observer. Different frameworks and programming languages often integrate some of DPs very tightly into its API which is good because it adds an abstraction layer and saves a lot of work, but may also cause problems in some specific situations.

5.6 Case Study: Building an Event System

The purpose of Event System in the context of a game application is to manage communication between different objects, including GUI-elements, game world’s objects, game flow elements (like, states), etc. One can use the approach taken by Cocoa to implement a system to identify and process game events. One can suggest to center the whole system around one global object, an event manager, which is in fact a candidate for being a singleton. For instance, in Cocoa, it is the `NSNotificationCenter`. What it does is creating a dictionary where for each event there is a list of observers [15]. By making the event manager a singleton we let it be visible from any class belonging both to Model and View. It plays the role of Controller, setting up and processing all the communication between Model and View. Thus, such Event System is an example of when MVC, Singleton and Observer are used together. Its simple implementation is shown on Fig. 5.4.

![Map: event-to-index](image)

<table>
<thead>
<tr>
<th>Key (name of event)</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>“carSelected”</td>
<td>0</td>
</tr>
<tr>
<td>“carDeselected”</td>
<td>1</td>
</tr>
<tr>
<td>“newGuestArrives”</td>
<td>2</td>
</tr>
</tbody>
</table>

![Game event system which uses Observer, MVC and Singleton](image)
5.7 State change notification in Unity3d

There is no built-in event system in Unity3d in terms of a separate object which would keep track of observers for each game object belonging to the scene, instead, each game object is updated every frame when its `update()` function is called. Since each Unity3d script object is a child to `MonoBehaviour`, every script object inherits some default functions (e.g. `update()`).

It is up to the programmer to implement a custom event system [17] which would broadcast state changes of one game object to other, using the techniques discussed earlier.
6 Conclusions

The design patterns mentioned in this thesis are certainly not outdated. They are still valid tools for solving common problems in game development, and they are used directly or indirectly via frameworks.

Whether a DP in question should be used or not depends on whether use of the pattern can make the source code easier to manage. The implementation of the example game project showed the practical advantage of some of the patterns, the patterns included in this thesis. At various points during the development process, the design (more precisely: the lack of good design) did threaten project's success. Some design patterns (e.g. MVC and Observer, State Machine and Singleton) are best suited when used together. Both MVC and State (State Machine) patterns have shown its great potential as a foundation for small game applications. The MVC is a decent choice for overall game structure and this pattern is adopted by other middleware. The State Machine can be used to split the Model further into smaller modules.

The frameworks which were reviewed in this thesis do implement the mentioned DPs under the hood, and it proves their usefulness. To exploit the technologies to its full potential one needs to be familiar with underlying mechanisms, which are variations of different DPs. By learning to recognize certain design patterns when looking at how different frameworks approach the same problems, it is much easier to familiarize oneself with new technologies. Even though the mentioned design patterns date back to 1994, they are still used by the popular frameworks.

Such DPs as MVC and State Machine are best suited for building a game engine or a small game from scratch because they define the whole structure of the application.

The usage of Singleton pattern is somewhat controversial. While the concept of globally accessed objects is widely perceived as an anti-pattern, and even the "creators" of DPs discourage the new generation of programmers from using it [16], there are many cases where Singleton comes in handy. In particular, Singleton is widely used by Apple when it comes to the services every application has access too, the same applies to Cocos2d. Unity3d also uses singletons for the global application data. Although the main purpose of the Singleton is to limit number of instances to one single instance, it is mainly used to allow a global access to the class variables. There are multiple alternatives to the Singleton, but there is always a price to pay. Finally, as the example project has shown, Singleton has its niche, at least if used in small projects.

Implementation-wise, the decision to implement game logic as pure C++ classes turned out to be controversial. It allowed versatile techniques to be applied to the same project, but on the other hand, it introduced some limitations for the design (limited usage of Observer and State Machine patterns, for instance).

The example game was developed in agile manner: new features were introduced continuously and the design has gone through repeated re-factoring. Using certain DPs came naturally and by the end of the project one can clearly see the positive effect of this process: the code became more modular and more flexible allowing to reuse it in the next projects.
Acknowledgments

I would like to thank Zordix AB for providing the work environment for me and giving me the opportunity to use Valet Parking 1989 as a starting point for my project. I am also grateful to my teachers Pedher Johansson and Marie Nordström from Umeå University for guiding me through the whole writing process and supporting me from the very first day. My supervisor Johan Eliasson and my opponents Kim Nilsson, Joakim Hagberg, Javid Jou, thank you for giving me the valuable feedback. Jakob Ronestjärna, thank you for saving me hours of work by pointing out my mistakes in code.
References


---

7 all the web resources mentioned here were retrieved during the period of April-June 2012


Appendix A: Valet Parking 1989 for the iPhone

A.1 Project description
To find out which design patterns are commonly used nowadays in game development and to get more insight into how they can be practically applied, a small game project has been undertaken. A small casual arcade game, Valet Parking 1989, originally developed by Zordix AB (www.zordix.se) for Nintendo DS console was a perfect candidate to play with either while porting it to some other device or while making a remake of it.

The initial goal was to port an existing Nintendo DS game to the iPhone platform. Zordix AB provided all the source code belonging to the original game, but the porting process turned out to be a challenging task because the code was lacking any documentation, even the comments were missing. Another problem was that the code relied on different libraries owned by Nintendo, for example a library containing mathematical functions. To access such libraries one had to acquire a license from Nintendo. One way out this situation was to implement the missing functionality otherwise carried by Nintendo’s code. These and some minor issues (e.g., different coordinate system used by Nintendo’s rendering routines) led to problems when debugging the existing code: every time an unexpected behavior happened it was hard to know whether this was the intended purpose of the code (no comments in code, as mentioned earlier) or if the eventual problems were caused by those newly implemented functions which replace Nintendo’s code, or was it due to some coordinate system transformation. Thus, because it was so difficult to work with the original code, and also because the project wasn’t meant to be commercial one decided to implement almost everything from scratch. Only some minor functionality, like XML file parsing, was re-used. However, some of the design decisions taken in the Nintendo DS version of the game were applied to the iPhone version, these for example includes using State Machine to manage transitions between game states and implementing different game managers as Singletons.

The original code was written completely in C++, and to ensure better future portability a decision to re-implement core game logic in pure C++ was made. The target platform was iPhone, thus Objective C/C++ became a primary language for programming the game application layer and game view layer.

A.2 The gameplay
Valet Parking 1989 is a casual 2D arcade game with time management elements. The player takes the role of a valet whose duty is to manage a parking lot by means of driving cars to- and from it. Similarly to other time management games, new guests arrive and leave with some frequency, and players task is to decide what is the most effective way to place their cars on the parking lot. Every time the player succeeds to serve a customer in time he or she gets some points (in-game cash) as a reward. On the opposite, when the player crashes the car into other vehicle or into some game-world obstacle some points are taken off. To win a game the player has to serve the predefined number of guests. The game ends when it has taken too much time to serve a guest. Fig. A.1 – A.8 show what the game looks like and describe its flow and elements.

---

8 Game application layer, game logic and game view are the categories for the game subsystems presented by [Game Coding Complete], something similar to MVC.
Figure A.1 A splash screen pops up when the game has started. Can be thought of as the first state of the game flow state machine.

Figure A.2 Game-play view, four guests have already arrived and are waiting in the waiting queue before they get served.

Figure A.3 The player has picked up a guest to take care of and now he or she is parking the car.
Figure A.4 The player's car hit another car, 2 points are taken off the player's score.

Figure A.5 The car has skidded and drifted before it has hit an obstacle. Even though the spectacular smoke and tire marks imply the player might deserve some points for being artistic, 2 points are taken off the player's score.
A guest wants his car to be taken to the EXIT area. There are 4 guests in queue waiting to be served. They have been waiting for quite a long time as their patience meters suggest, the player better start working faster! Also, here one can see what elements the graphical user interface (here: the HUD, heads-up display) contains: a score panel, a queue panel depicting cars, a queue panel depicting the guests, their patience meters and their the cars they want the player to fetch, as well as a portrait of an owner to the currently selected car with a speech balloon giving orders to the player.

A car has been taken to the EXIT area. The player earns 10 points for serving the guest well.
A.3 The game logic

A.3.1 Characters' behavior
Each character's behavior depends on its current state. There are 4 states a character can be at the moment:

- **StayingInQueue**: a character has just been spawned on the map. Its patience meter starts running. If the meter has reached the critical value (happens when the character gets tired of waiting), the player loses the game.

- **Away**: a character has left the car. Its patience meter gets stopped and reset to zero. The player can drive the car around the parking lot and park it. The player can also re-enter the car and re-park it.

- **WantsToLeave**: a character wants his car to be driving into the exit area. Its patience meter starts running. If the meter has reached the critical value (happens when the character gets tired of waiting), the player loses the game.

- **Leaving**: a character is driving his own car away. Its patience meter is not running.
Appendix B

B.1 Cocos2d's CCDirector

```c
static CCDirector *_sharedDirector = nil;
+(CCDirector *)sharedDirector
{
    if (!_sharedDirector) {
        if (![[CCDirector class] isEqual:[self class]])
            _sharedDirector = [[CC_DIRECTOR_DEFAULT alloc] init];
        else
            _sharedDirector = [[self alloc] init];
    }
    return _sharedDirector;
}
```

Listing B.1 A code snippet taken from CCDirector.h which is part of Cocos2d framework, illustrating Singleton pattern usage

B.2 Cocoa's NSNotificationCenter

```c
static NSNotificationCenter *defaultCenter = nil;
+(void)initialize
{
    static BOOL initialized = NO;
    if (!initialized) {
        initialized = YES;
        defaultCenter = [self alloc];
        [defaultCenter init];
    }
}
```

Listing B.2 A code snippet taken from NSNotificationCenter.m which is part of Cocoa framework, illustrating Singleton pattern usage