Learn how to apply classic design patterns to iOS app development using Swift



Pro Design Patterns in Swift

Adam Freeman

Apress[®]

For your convenience Apress has placed some of the front matter material after the index. Please use the Bookmarks and Contents at a Glance links to access them.



Apress[®]

Contents at a Glance

About the Author	XXV
About the Technical Reviewer	xxvii
Part I: Getting Ready	1
Chapter 1: Understanding Design Patterns	3
Chapter 2: Getting Used to Xcode	9
Chapter 3: Creating the SportsStore App	25
Part II: The Creation Patterns	53
Chapter 4: The Object Template Pattern	55
Chapter 5: The Prototype Pattern	77
Chapter 6: The Singleton Pattern	
Chapter 7: The Object Pool Pattern	
Chapter 8: Object Pool Variations	
Chapter 9: The Factory Method Pattern	
Chapter 10: Abstract Factory Pattern	
Chapter 11: The Builder Pattern	233

Part III: The Structural Patterns
Chapter 12: The Adapter Pattern253
Chapter 13: The Bridge Pattern271
Chapter 14: The Decorator Pattern
Chapter 15: The Composite Pattern
Chapter 16: The Façade Pattern
Chapter 17: The Flyweight Pattern
Chapter 18: The Proxy Pattern357
Part IV: The Behavioral Patterns
Chapter 19: The Chain of Responsibility Pattern
Chapter 20: The Command Pattern401
Chapter 21: The Mediator Pattern423
Chapter 22: The Observer Pattern447
Chapter 23: The Memento Pattern473
Chapter 24: The Strategy Pattern491
Chapter 25: The Visitor Pattern
Chapter 26: The Template Method Pattern517
Part V: The MVC Pattern525
Chapter 27: The Model/View/Controller Pattern527
Index553

Part

Getting Ready

Chapter

Understanding Design Patterns

Design patterns are insurance policies for software development. Insurance policies work by trading a little cost now to avoid the possibility of a lot of cost later. The premium you pay to insure a car against theft, for example, costs a few percent of the value of the car, but when the car is stolen, your overall costs are minimized. You still have to go through the inconvenience of having your car stolen, but at least you don't have to bear the financial loss as well.

In software development, design patterns are insurance against the time taken to solve problems. The premium is the time it takes to add extra flexibility to your code now, and the payout is avoiding a painful and protracted rewrite to change the way the application works later. Like real insurance policies, you may not benefit from paying the premium because the problem you anticipate might never happen, but software development rarely goes smoothly and problems often arise, so that additional flexibility is usually a good investment.

This is a book for hands-on professional programmers. I focus on the practical applications of design patterns and focus on code examples instead of abstract descriptions. I describe the most important design patterns and demonstrate how they can be applied to iOS using Swift. Some of the patterns I describe are already implemented in the Cocoa framework classes, and I show you how use them to create more robust and adaptable applications.

By the time you have finished reading this book, you will understand the most important design patterns in contemporary software development, the problems they are intended to solve, and how to apply them to your Swift projects.

Putting Design Patterns into Context

Every experienced programmer has a set of informal strategies that shape their coding style. These strategies are insurance policies against the recurrence of problems from earlier projects. If you have spent a week dealing with a last-minute database schema change, for example, then you will take a little extra time on future projects making sure that dependencies on the schema are not hard-coded throughout the application, even though you don't know for certain that the schema will change this time around. You pay a little premium now to avoid the potential for a bigger cost in the future.

You may still have to make changes, but the process will be more pleasant, just like the process of shopping for a replacement car is made more pleasant when the insurance company pays for the stolen one.

There are two problems with informal strategies. The first problem is *inconsistency*. Programmers with similar experiences *may* have different views about what the nature of a problem is and *will* disagree about the best solution.

The second problem is that informal strategies are driven by personal experiences, which can be associated with strong emotions. Describing the difficulty of fixing a problem rarely conveys the pain and misery you endured, and that makes it hard to convince others of the need to invest in preventative measures. It also makes it difficult to be objective about the importance of the problem. Bad experiences linger, and you may find it hard to accept that there is little support for making the changes that would avoid problems you have encountered on previous projects.

Introducing Design Patterns

A *design pattern* identifies a common software development problem and provides a strategy for dealing with it, rather like the informal approach that I described earlier but that is expressed objectively, consistently, and free from emotional baggage.

The strategies that design patterns describe are proven to work, which means you can compare your own approach to them. And, since they cover the most common problems, you will find that there are design patterns for problems that you have not had to personally endure.

Most of the other design patterns in this book originate from a classic book called *Design Patterns: Elements of Reusable Object-Oriented Software* by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides (Addison-Wesley, 1995). The authors of this book are referred to as the *Gang of Four* (*GoF*), and the patterns they describe are some of the most important and fundamental in modern software development.

The GoF book is worth reading but is somewhat academic in nature. Design patterns are expressed abstractly without reference to a particular programming language or platform. This abstraction makes them hard to use; it can be difficult to figure out whether a pattern describes a problem that you are concerned about and difficult to be sure that you have correctly implemented the solution.

My goal in this book is to put design patterns in context and give you all the information you need to easily identify and apply the patterns that you need, along with a Swift implementation that you can apply directly to your project.

Understanding the Structure of a Design Pattern

Most design patterns apply to small groups of objects in an application and solve a problem that arises when one object—known as the *calling component*—needs to perform an operation on one or more other objects in the application.

For each of the design patterns in this book, I describe the problem the pattern solves, explain how the pattern works, and show you how to implement the pattern using Swift. I also describe common variations on the pattern and describe the pitfalls most closely associated with the pattern.

WHERE IS THE UML?

The Unified Modeling Language (UML) is often used to describe patterns, but I don't use it in this book. I am not a fan of UML for several reasons. First, most developers don't completely understand UML and take in little information from a UML diagram. There are exceptions, of course, and these tend to be people who work in large corporations where there is a detailed analysis and design phase before development commences. For the rest of the world, UML is a poorly defined and misinterpreted mess of boxes and lines.

I find that UML is good for expressing some kinds of relationship but fails dismally at representing others. To a great extent, understanding patterns means understanding where logic that represents knowledge of other components exists, which is hard to convey using UML.

Finally, and rather less objectively, UML is symptomatic of many aspects of software development that I don't like. All too often, UML is used as a weapon to enforce static and inflexible designs and inhibits adapting a development process to meet evolving customer needs because the UML becomes an unchanging reference point.

For these reasons, as subjective as they are, I don't use UML in this book. Instead, I'll use free-form diagrams to illustrate the points that I want to emphasize.

Quantifying the Value of Design Patterns

It is easy to accept that design patterns are a good thing. Everyone understands the appeal of proven solutions used on countless projects to solve difficult problems. It is much harder to convince other programmers on the team that a specific pattern should be adopted in a project.

You can assess whether an insurance policy represents value for money by asking yourself some questions:

- Does the policy address something bad that is likely to happen to me?
- How often does the bad thing occur?
- Is the cost of the policy a small fraction of the cost of dealing with the bad thing?

These simple questions make it easy to understand that there is no point in buying car insurance if you don't have a car or if there are no car thieves in your town. They also highlight the poor value in paying \$10,000 per year to insure an \$11,000 car unless you anticipate multiple thefts (in which case, you might also consider moving to a different area).

The point is clear even though this is a simplistic view of insurance: don't buy a policy unless it offers some benefit. The same is true for design patterns: don't adopt a pattern unless it offers value that you can quantify and articulate to others. The questions needed to assess the value for design patterns are similar:

- Does the pattern identify a problem that I am likely to encounter?
- How often does this problem occur?
- Do I care enough about avoiding the risk of having to fix the problem in the future to undertake the work of implementing the design pattern today?

It can be hard to answer these questions. There are no actuarial tables for software development, and it can be hard to estimate the amount of future effort that will be required to fix a problem (especially one that may not arise).

Instead, it can be easier to focus on the immediate benefits that a design pattern can offer. For example, those patterns that increase the modularity of an application generally do so to minimize the effect of a future change, but a modular application has fewer tightly coupled components, which means that it is easy to isolate units of code. Being able to isolate units of code is essential for effective unit testing, and so adopting a change-insurance pattern has an immediate benefit of improving the testability of code.

Equally, design patterns that increase the amount of abstraction in an application allow new features to be added with less effort and less code duplication. Almost everyone can agree that quicker and easier development is a good thing, even if they don't agree with the need to avoid the problem that a design pattern is intended to guard against.

There are no easy answers, however, and the final decision to adopt a design pattern will be driven by the combined experience of the development team, the confidence in the completeness of the specification, and the competence of individual developers.

Using a Design Pattern After the Problem Occurred

You will find it hard to drive the adoption of patterns if you are the sole voice promoting them in a team that has no experience of them and little time to consider them. The chances are that you will fail to convince others. Don't be frustrated.

My advice is not to push too hard. If you force the team into following new practices, you will be held accountable for every problem and delay they cause, which will be especially difficult if the problem you are trying to guard against never happens. Advocates for design patterns are, sadly, often seen predictors of unlikely doom.

Don't lose hope, but put this book away and wait. If the problem you are concerned about doesn't occur—if, for example, the database schema doesn't change—then take pleasure in that the project dodged a bullet and move on to the next assignment.

Don't worry if the problem does occur; you can still benefit from design patterns. Your project is now in a situation that you had hoped to avoid, but you can use the patterns as a framework for digging yourself out of the hole. Select the most appropriate design pattern and use it as a framework to structure the clean code around which you drive the resolution of the problem. In this way, you can leverage a bad situation to introduce the team to a proven solution to the problem. This isn't as good as avoiding the problem in first place, but at least you will be able to create a long-term solution and lend some credibility to your enthusiasm for design patterns.

Understanding the Limitations of Design Patterns

There is a lot to like about design patterns, but they have their limitations. By their nature, design patterns are solutions to problems that other people have encountered on other projects. Design patterns are a starting point for avoiding or solving a problem and not a precisely tailored solution. That doesn't mean they are not useful, but they do require some work in order to adapt them to fit into your project.

Treat design patterns as recipes. Tinker, adapt, and adjust a pattern, and you will end up with something that works for you. You might need to refine your implementation a few times, and your use of a pattern is likely to get better with insight gained from several projects, but you'll end up with something that improves on your starting position and that helps minimize the impact of a common problem.

Some programmers treat design patterns as immutable laws. They are *pattern zealots*, someone who promotes the use of patterns as an inflexible "best practice" that should always be followed and cannot be adapted. That's rubbish. Applying patterns that you don't need or resisting adapting a pattern to fit into a project misses the point entirely.

There is no point trying to argue with a pattern zealot. They get their pleasure from being able to reference ever more obscure sources, and there is no effective way to ground their views in reality. My advice is to ignore them and focus on building good software by making it robust, scalable, and flexible enough to cope with changes, all of which can be aided by the design patterns described in this book.

About This Book

In this book, I describe how to use the most important design patterns using Swift and the Cocoa frameworks. Swift has attracted many new developers to the Apple platform, so I have written this book to provide all of the information you will need about Swift and the Cocoa classes I use. I also show you how to create the example projects using Xcode, which can be a confusing tool if you are new to the world of Swift development. You can download all of the examples from Apress.com if you don't want to type in the code yourself.

What Do You Need to Know?

You need to be an experienced developer to follow the concepts in book. No prior experience of Swift is required, but you should understand the basic concepts of object-oriented programming and have used a modern language such as Objective-C, C#, Java, or JavaScript.

What Software Do You Need?

You need to have a Mac running OS X 10.10 (Yosemite) and have downloaded and installed Xcode 6.1. To get Xcode, you will need to sign up as an Apple Developer, which you can do at https://developer.apple.com. Don't worry if you are new to Xcode; I'll show you everything you need to know to follow the examples in Chapter 2.

What Is the Structure of This Book?

This book is split into five parts, each of which covers a set of related topics. Part 1 contains this chapter and an Xcode primer for the techniques you will need to follow for the example in this book. I also build an example application called SportsStore that I return to throughout this book to demonstrate how to apply design pattern in context.

Each of the other four parts of the book focuses on a specific type of pattern. Part 2 covers the creational design patterns, which are concerned with how objects are created in an application. Part 3 describes the structural design patterns, which define and manage the relationship between objects in an application. Part 4 of this book focuses on the patterns that describe how objects communicate with one another. In the final part of this book, I describe the Model/View/Controller (MVC) pattern, which applies to the structure of the entire application and is commonly used for Mac OS and iOS UI applications.

Where Can You Get the Example Code?

You can download all the examples for all the chapters in this book from www.apress.com. The download is available without charge and includes everything you need to re-create the examples without having to type them in. You don't have to download the code, but it is the easiest way of experimenting with the examples and cutting and pasting into your own projects.

If you do want to re-create the examples from scratch, then you will find that every chapter contains detailed listings of all the files that I create and modify. I never refer you to an external file or hand-wave about leaving the rest of the example as an exercise; every detail you need to re-create every example is contained within this book.

Summary

In this chapter, I outlined the content and structure of this book and set out the experience and software required. In the next chapter, I provide a brief primer to Xcode and describe the features I rely on in this book.

Getting Used to Xcode

In this chapter, I use Xcode to demonstrate the ways in which I present examples throughout this book. *Playgrounds*, which are one of the most useful features of Xcode 6, allow for code experiments to be created and evaluated without needing to create an application project. I use playgrounds to help describe the problems that patterns are intended to solve and use them to demonstrate simple implementations.

Many of the design patterns I describe in this book rely on restricting access to classes, methods, and properties. Swift supports access protection keywords, but they operate on a per-file basis. This is a problem for playgrounds because all the code is in a single file, and so access protections are not enforced. Many of the design patterns I describe require support for concurrent access, which is something that playgrounds do not handle at all well.

For these reasons, I also use OS X Command Line Tool projects, which are the simplest Xcode projects that support multiple files. Command Line Tool projects do not present a windowed UI to the user and are limited to reading from and writing to the console. The benefit of using such a simple project type is that it allows me to focus on just the design pattern I am describing and the code required to implement it, without the complexity of dealing with a user interface.

Few real-world projects are so simple, of course, and in Chapter 3, I create an iOS app called SportsStore, which does present a graphical user interface and which I use in every chapter to provide additional context for applying the patterns I describe.

Working with Xcode Playgrounds

Xcode playgrounds are a good way to prototype code and test ideas without needing to create an iOS application project. Playgrounds are a new feature in Xcode 6, and many developers are unfamiliar with them, especially those who have been attracted to iOS and Mac OS development by Swift and have not used earlier Xcode releases to develop in Objective-C.

Creating a Playground

When you start Xcode, you will be presented with a splash screen that lets you select between creating a new playground, creating a new project, or checking out an existing project from a source code repository, as shown in Figure 2-1.



Figure 2-1. The Xcode 6 splash screen

Tip Select New ➤ Playground from the File menu if the splash window is not shown.

Click "Get started with a playground." Xcode will prompt you for a location to name and save the playground file. Change the name to MyFirstPlayground and ensure that iOS is selected for Platform, as shown in Figure 2-2.

Name	MyFirstPlayground	
Platform:	iOS	٥

Figure 2-2. Naming the playground and selecting the platform

Click the Next button and select a location where you will be easily able to find the playground in the future, as shown in Figure 2-3.

Sav	e As: MyFirstPlaygroun	nd	•
	Playgrounds		Q Search
Favorites iCloud Drive Applications Desktop Documents Documents Downloads Devices Permete Disc	Books Misc Playgrounds	4 4	П
New Folder			Cancel Create

Figure 2-3. Changing the name of the playground file and selecting a save location

Xcode will create a file called MyFirstPlayground.playground in the location you selected, the contents of which are shown in Listing 2-1. If this is the first time you have used Xcode, you will be asked to enable developer mode.

Listing 2-1. The Contents of the MyFirstPlayground.playground File

import UIKit

var str = "Hello, playground"

Tip I don't show the comments added by Xcode in the listings in this book or add any of my own comments. In real projects I incessantly comment code, but in this book I'll be explaining the effect of the statements I write in the accompanying text.

A playground provides insights into the code that is entered into the editor. There is a comment and two statements in the playground. Comments are ignored, and the import statement makes classes in the Cocoa UIKit framework available. It is the second statement that gives a hint of what playgrounds can do.

```
...
var str = "Hello, playground"
...
```

This statement creates a variable called str and uses a literal string value to set its value. If you look to the right of this statement in the playground, you will see that the value of the variable is displayed, as illustrated by Figure 2-4.



Figure 2-4. A simple playground

This is an interesting start but not especially useful. In the sections that follow, I'll show you the different playground features that I rely on in this book.

Displaying the Value History of a Variable

Modify the code in the playground to match Listing 2-2, and you will get a sense of the power of a playground.

```
Listing 2-2. Defining a Loop in the MyFirstPlayground.playground File
```

```
import UIKit
var str = "Hello, playground"
var counter = 0;
for (var i = 0; i < 10; i++) {
    counter += i;
    println("Counter: \(counter)");
}</pre>
```

Tip You don't have to compile—or even save—the playground file to see the effect of changes. The code statements are automatically evaluated after every edit.

I have defined a counter variable with an initial value of 0 and a for loop that increases the counter value with each iteration and writes the current value to the console using the println function. The right-hand panel updates, displaying (10 times) next to the statement that changes the counter value. To the right of the (10 times) message is a small circular plus button, as shown in Figure 2-5.

und.playground		
nd.playground		+
rlection		
can play	"Hello, playground" 0 (10 times) (10 times)	© 0

Figure 2-5. The button that shows value history

Tip Be sure to click the button next to the statement that changes the value of the counter variable and not the one that calls the println function.

This symbol is labeled *Value History*, and clicking it opens a panel that shows how the value of the counter variable changed as the code was executed and shows the console output generated through the println function. Figure 2-6 illustrates the view.



Figure 2-6. Displaying the value history in a playground

The chart shows how the value of the counter variable changes for each iteration of the for loop. You can display the value history for any variable defined in the playground, but numeric values are the most usefully presented.

A NOTE ABOUT CODING STYLE

You will notice that I use semicolons to terminate statements throughout this book, even though Swift doesn't require the use of semicolons after statements unless you need to separate multiple statements on the same line.

Although it's not a requirement of Swift, I have been writing code for decades with languages that do require semicolons, and—try as I might—I can't break the habit. There is something about an unterminated statement that just looks wrong to me, and I hit the semicolon key automatically. I considered going through each chapter and removing the semicolons, but that is a path that leads to broken examples, which I work hard to avoid in my books. And so, with a note of apology, I decided to let my preferences manifest themselves and use semicolons in the listings. You don't have to follow my style, however: one of the nice features of Swift is its relaxed approach to code style, and you are entirely free to express your own preferences and habits (including, if you want, the addition of unneeded semicolons).

Using the Value Timeline

At the bottom of the Value History panel is a slider that you can use to see how the value of variables changed during the execution of the code. The effect of this slider is easier to see when there are multiple variables to look at, and in Listing 2-3 I have updated the playground with some additional statements.

Listing 2-3. Adding Additional Statements to the MyFirstPlayground.playground File

```
import UIKit
var str = "Hello, playground"
var counter = 0;
var secondCounter = 0;
for (var i = 0; i < 10; i++) {
    counter += i;
    println("Counter: \(counter)");
    for j in 1...10 {
        secondCounter += j;
    }
}</pre>
```

Display the value histories for the counter and secondCounter variables by clicking the circular button to the right of the statements in the for loops. You will see two separate charts, and you can see the relationship between the values of the variables by dragging the playback head (the red vertical bar at the bottom of the panel) left and right to move to different points in the execution of the code, as shown in Figure 2-7.



Figure 2-7. Reviewing the variable value timelines in a playground

Displaying UI Components in a Playground

Playgrounds can also be used to display UI components, which I rely on to demonstrate how Cocoa implements some patterns. Listing 2-4 shows how I modified the playground to show a text field.

Listing 2-4. Adding a UI Component to the MyFirstPlayground.playground File

```
import UIKit
var str = "Hello, playground"
var counter = 0;
var secondCounter = 0;
for (var i = 0; i < 10; i++) {
    counter += i;
    println("Counter: \(counter)");
    for j in 1...10 {
        secondCounter += j;
    }
}
let textField = UITextField(frame: CGRectMake(0, 0, 200, 50));
textField.text = "Hello";
textField.borderStyle = UITextBorderStyle.Bezel;</pre>
```

textField;

There are two important differences when using UI components in a playground from a regular Xcode project. The first is that you must use the initializer with the frame argument and generate a frame to contain the component using the CGRectMake function, like this:

```
int textField = UITextField(frame: CGRectMake(0, 0, 200, 50));
...
```

The arguments to the GCRectMake functions are the bounds of the frame that will contain the component, where the third and fourth values define the width and height. I have specified a frame that is 200 pixels wide and 50 high, which is sufficient for a text field.

The second difference is the last statement in the playground, which simply contains the name of the variable to which I have assigned the UI component.

textField;

• • •

. . .

This is required so that Xcode will provide the plus icon to the right of the statement; clicking the icon displays the component in the assistant editor, as shown in Figure 2-8. The assistant editor panels display the result of statements, so a statement that returns the configured UI component object is required.



Figure 2-8. Displaying a UI component in a playground

Working with OS X Command Line Tool Projects

OS X Command Line Tool projects are ideally suited to demonstrating design patterns in Swift. This kind of project supports multiple files and concurrency, which makes it possible to demonstrate the effect of access protections and the effect of working with multiple threads.

Creating a Command-Line Project

Click "Create a new Xcode project" on the Xcode splash screen or select New \succ Project from the Xcode File menu if the splash screen isn't visible. Select the Command Line Tool template, which is found in the OS X \succ Application category, as shown in Figure 2-9.

iOS Application Framework & Library Other OS X	Cocoa Application	Game	Command Line Tool	
Application Framework & Library System Plug-in Other				
	Command Line T This template create	ool as a command-line t	001.	
Cancel			Previous	Next

Figure 2-9. Selecting the Command Line Tool template

Click the Next button, and Xcode will prompt you for details of the project you want to create. Set the name to MyCommandLine and ensure that the Language option is set to Swift, as shown in Figure 2-10. I have specified Apress as the organization for the project, but I don't rely on these values in this book, and you can set them to your own organization.

Product Name:	MyCommandLine	
Organization Name:	Apress	
Organization Identifier:	com.apress	
Bundle Identifier:	com.apress.MyCommandLine	
Language:	Swift	0

Figure 2-10. Specifying details for the project

Click the Next button, and Xcode will prompt you to specify a location for the project. Select a convenient location and click the Create button. Xcode will create the project files and open the main project window.

Understanding the Xcode Layout

When Xcode shows the project, you will see a layout that is similar to the one shown in Figure 2-11. You may see a slightly different layout, but I'll explain how to open each panel.



Figure 2-11. The Xcode layout for a Command Line Tool project

I have numbered the main panes that Xcode presents, and I describe them in Table 2-1 for readers who are new to Xcode. The content of some panes changes based on the task being performed by Xcode, so I have included details of how the content can be selected.

Number	Description
1	This is the navigator pane, which presents the contents of the project in different views that are selected using the row of buttons at the top edge. I use the Project Navigator view, which is displayed by clicking the first button in the row. The visibility of the navigator pane is controlled through the View ➤ Navigators menu.
2	This is the main editor window, which adapts to the file being edited. Different editors are available, includes one for project settings (which is what is shown when the project is first created), a code editor for .swift files, and a drag-and-drop editor called UI Builder that deals with .storyboard files. (I use .storyboard files in Chapter 3.) Select View > Standard Editor > Show Standard Editor to open the code editor.
3	This is the inspector pane, which reveals information about components in the application and is used when creating the application layout. I describe this pane further in Chapter 3.
4	This is the utility pane. The content of this pane is set using the four buttons at the top edge, and the view shown in the figure is the Object Library, which contains the UI controls used to create an application layout, which I use in Chapter 3. You can display the Object Library by selecting the View ➤ Utilities ➤ Show Object Library menu.
5	This is the Debug pane, which is used to interact with the debugger and to display console messages written using the Swift println function. This is where the output from Command Line Tool applications appears. This pane is controlled through the View > Debug Area menu.

Table 2-1. The Default Xcode Panes

Adding a New Swift File

The reason that I use Command Line Tool projects is so I can create multiple code files and enforce access protection with keywords such as private. To add a new file to the project, select the Project Navigator view in the navigator pane and right-click the MyCommandLine folder item, which currently contains a file called main.swift, as shown in Figure 2-12.

•••	MyCommandLine: R	eady Today at 12:44
		MyCommandLine.;
🗖 🖬 Q 🛆 🤆	ショロ 日 田 く ン MyCom	mandLine
MyCommandLi 1 target, OS X SD	ne MyCommandLine :	C Build Settings Buil
► MyCommapc	Show in Finder Open with External Editor Open As Show File Inspector	Levels +
	New File New Project Add Files to "MyCommandLine"	s chitecture Only
	Delete	
	New Group New Group from Selection	forms ures
	Sort by Name Sort by Type	-
	Find in Selected Groups	
	Source Control	
	Project Navigator Help	

Figure 2-12. Adding a new file to the example project

Xcode will display the set of file templates available for new files. Select the Swift File template, as shown in Figure 2-13.

OS Source User Interface Core Data	Cocoa Class	Test Case Class	Playground	Swift File
Resource Other DS X	m	h	c	C++
Source User Interface Core Data Resource Other	Objective-C File Swift File An empty Swift file.	Header File	C File	C++ File
Cancel			P	avious No

Figure 2-13. Selecting the Swift file template

Click the Next button and set the name of the new file to MyCode.swift, as shown in Figure 2-14.

Sa	ve As:	MyCode.swift		Ĺ
	Tags:			
		MvCommandLine	0	

Figure 2-14. Setting the name of the code file

Click the Create button, and Xcode will create the file and open it for editing. Replace the default content with the statements shown in Listing 2-5.

Listing 2-5. The Contents of the MyCode.swift File

```
class MyClass {
   func writeHello() {
      println("Hello!");
   }
   private func writePassword() {
      println("secret");
   }
}
```

I have defined a class called MyClass and added two methods. The writeHello method has no access control keyword and can be called from anywhere in the same module or project. The writePassword method is decorated with the private keyword, which means it can be accessed only from types that are defined in the MyCode.swift file.

The main.swift file contains the statements that will be executed when the application is executed. In most of the examples in this book, I will add statements to the main.swift file to represent the calling component in a design pattern. Listing 2-6 shows the statements I added to the file for this example.

Listing 2-6. The Contents of the main.swift File

```
let myObject = MyClass();
myObject.writeHello();
```

I create an object from the MyClass class and call the writeHello method. To compile and run the application, click the play icon at the top of the Xcode window, as shown in Figure 2-15.



Figure 2-15. The Xcode button that compiles and runs the application

Tip If you don't see the button shown in the figure, then select Show Toolbar from the Xcode View menu. You can also control the compilation and execution of the project using the Product menu.

Xcode will compile the code and run the application, and the following output will appear in the debug console window:

Hello! Program ended with exit code: 0

The first line of the output comes from the call to the println function. The second line indicates that the example program has terminated. I won't usually show the second line, and some of the example applications that I build won't terminate on their own.

Summary

In this chapter, I explained how Xcode can be used to create playgrounds and Command Line Tool projects, which are the main ways in which I introduce design patterns in this book. In the next chapter, I walk through the process of creating an iOS application called SportsStore, to which I apply every pattern in this book, in order to provide as many examples as possible and to put the patterns into a more realistic setting.

Creating the SportsStore App

In the previous chapter, I showed you how Xcode can be used to create playgrounds and command-line tools, which are how I introduce each of the design patterns in the chapters that follow.

I like to provide as many code examples as I can in my books, so in this chapter I create an iOS app called SportsStore. The app I create is entirely unstructured, which means I simply bolt the code and the UI together as directly as possible without any thought to the long-term consequences. This is, of course, the antithesis to design patterns, but it is a surprisingly common development style. Throughout this book, I apply the design patterns to the unstructured application to provide additional context for their use.

Creating an Unstructured iOS App Project

In this section, I create a simple iOS app that allows a user to buy products from a retailer called SportsStore. I am going to implement only part of the shopping process in order to create an easy-to-understand example that doesn't require complex visual layouts and lets me focus on the structure of the code.

This is fortunate because I am one of the worst interface designers in the world. You will get a sense of my lack of aesthetic when you see the iOS layout I create—let's just call it minimalistic chic and move on. (In my own projects, I work with a professional designer, and I encourage you to do the same if you are similarly style-challenged).

The style of development I use in this section is known as a *single-class application*, which has no structure or design. This is a common development style, especially for programmers who have little experience of object-oriented languages.

This is the "before" in my proposition for the value of design patterns, and I introduce the "after" as I show you each pattern. I have, therefore, built this app so that I can emphasize the impact of the design patterns I describe, but this style of development is pretty common, and you can see code like this in just about any project, especially from developers who have recently made the transition to object-oriented languages. I am confident that if you think about the programmers you know, there will be at least person you can think of who writes similar code.

Tip I describe the process for creating the application step by step because many readers will be new to the world of Xcode and Swift. If you are an experienced Xcode user, then you can skip to Chapter 4 and just download the project from Apress.com.

The application that I am going to create is a simple stock management tool for an imaginary sports equipment retailer called SportsStore. I use SportsStore in one form or another in most of my books, and it lets me highlight the way that common problems can be addressed using different languages, platforms, and patterns. Figure 3-1 shows a mock-up of the initial interface that I will create for the SportsStore application.

240		-	
Product Product Description	No. 194 PPH	23	
Produc1 Product Description		- 23	
Product Product Description		- + 23	
Produc1 Product Description		- 23	•
Produc1 Product Description		- + 23	
Product Product Description		· 23	
Product		• 23	
Our door to Construction	34 Items in Stock		

Figure 3-1. The mock-up for the SportsStore example application

The user will be presented with a list of products displayed in a table. For each product, the name and a description will be displayed along with the current number of items in stock. The user will be able to edit the stock level directly using a text field or increment and decrement the value with a stepper.

THE VALUE OF EXAMPLE APPLICATIONS

I am not for a second going to pretend that the SportsStore application is useful in its own right—but that is not the point of an example. The goal is to give me a framework that I can use to demonstrate different patterns in a broader context than just a fragment of code in a playground or command-line tool.

The SportsStore example is just complex enough to me to demonstrate how to apply patterns without causing me to deal with difficult issues such as data persistence, security, data validation, and all of the other matters that have to be taken into account in a real project.

The problem with creating a more realistic application is that too much of the book is then given over to writing code that isn't directly related to the subject at hand. That is not the kind of book I want to write and, I hope, not the kind you want to read.

Creating the Project

To create a new project, select New \succ Project from the Xcode File menu. You will be presented with the range of project types that Xcode supports. Select Single View Application from the iOS \triangleright Application section, as shown in Figure 3-2.

105		\square		
Application	-		1	
Framework & Library Other OS X Application Framework & Library System Plug-in	Master-Detail Application Game	Page-Based Application	Single View Application	Tabbed Application
Other	Single View Application This template provides a starting point for an application that uses a single view. It provide a view controller to manage the view, and a storyboard or nib file that contains the view.			

Figure 3-2. Selecting the Xcode project type

Click the Next button, and you will be asked to choose options for the new project. Set Product Name to SportsStore and enter **Apress** and **com.apress** for Organization Name and Identifier, respectively. Ensure that Swift is selected for Language, that iPad is selected for the Devices option, and that the Use Core Data option unchecked, as shown in Figure 3-3.

Product Name:	SportsStore	
Organization Name:	Apress	
Organization Identifier:	com.apress	
Bundle Identifier:	com.apress.SportsStore	
Language:	Swift	0
Devices:	iPad	0
	Use Core Data	
Cancel		Previous

Figure 3-3. Choosing options for the new Xcode project