Functional Design Patterns

Chris Woodard
Tampa Bay Cocoaheads

Design Patterns in General

- Methods and conventions for writing code
- Became a “thing” with the publication of the “Gang of Four” book
- Initially just for OOP to decouple classes
- Represent solved metaproblems
- Tons of examples on the Internet

Problems Design Patterns Try To Solve

- ***Brittle*** programs caused by tightly coupled components
- ***Fragile*** programs caused by dependence on complex state
- ***Untestable*** programs caused by tight coupling and dependence on complex state
- ***Unreadable*** programs written by developers in the process of injecting the first two problems.

Familiar Patterns

- ***Template*** - create identical instances of a general type of object (alloc/init instances of *class*).
- ***Delegation*** - objects that agree to implement methods listed in a *protocol* can act as *delegates* of another object (e.g. UITableViewDataSource, UITableViewDelegate).
- ***Observer*** - instance objects can designate methods to be called when some condition or change occurs (e.g. NSNotification handlers).

Familiar Patterns (continued)

- ***Singleton*** - A class specially built so that there is only ever one instance per app.
    - Uses a class method for allocating this instance the first time it’s called and then returning the instance thereafter. This method is often named “sharedXXX” or “defaultXXX” (in Objective C; in Swift it doesn’t have to be).
    - Meant to control access to shared or fragile system resources or to represent external resources that there is only one of.

Familiar Patterns (continued)

- ***Factory*** - Class or method designed to create instances of different classes based on different inputs. (UITableView's dequeueReusableCellWithIdentifier method)
  - Factory-created classes should share a common base class.
  - Factory method must know the mapping from the input to the class it is supposed to create as a result.

Familiar Patterns (continued)

- ***Builder*** - A class *pair* intended to solve the "telescoping initializer" problem.

    - For a class named "MyXYZ" there is a related class called "MyXYZBuilder". MyXYZBuilder contains all of the properties needed to build MyXYZ with suitable default values.

    - The constructor for MyXYZ includes a block called a "builder block" that is called with a MyXYZBuilder **builder* parameter. The caller then sets the properties of *builder*

Functional Design Patterns - Common Design Patterns

Thoughts…

- **OO patterns** are meant to help decouple objects in an app by:
  - Controlling the access that one object gives to other objects
  - Providing a finite set of *externally invokable* methods to other objects
  - Encapsulating knowledge
- **There** are a lot of patterns, and a given object can be built from more than one of them. They are useful and can help an OO application if they are used.

However…

- We're still faced with problems, like:
    - One object needing to give another object *write access* to its properties, requiring extra checks and validation to avoid silent and abrupt failure.
    - Complexity and "spaghetti-codedness" of overusing the delegation pattern leading to difficult-to-reason-about code.
    - OO Design Patterns don't inherently address the problem of *shared mutable state*.

Functional Programming

- As much as possible, program execution is built from *functions* rather than *objects*.

- State isn't shared among functions; a function only knows the values of its parameters and how to compute its return value.

- Functions are *first class values* that can be created at runtime and supplied as arguments to other functions.

Map/Filter/Reduce

- Processing a list with one or more of these operations:
  - *map* - iterate over a list and compute a new value for each element in the list. Result is a list of these new values
  - *filter* - iterate over a list and compute a subset of the elements in the list. Result is this subset.
  - *reduce* - iterate over a list and accumulate the list elements into a single value.

Map/Filter/Reduce

- Performed by calling functions (or methods) or each operation and supplying them with functions or closures:
    - *map* - mapping function that creates a (maybe) modified copy of each element and returns it to be included in the result list.
    - *filter* - predicate function that decides whether an element should be included in the result list.
    - *reduce* - accumulator function that accumulates the value of each element into the result value.

Chain of Execution

- Constructed by applying functions to the output of other functions:
- *filter(map( ))*
- *map(filter( ))*
- *reduce(filter(map()))*

Functional Design Patterns - Common Design Patterns

Chain of Execution

- Constructed by applying functions to the output of other methods *if that output is conformant with those methods*:
- *map().filter()*
- *filter().map()*
- *map().filter().reduce()*

Functional Design Patterns - Common Design Patterns

Chain of Execution - Continued

- Models the original Unix method of constructing applications by piping commands together so that the output of one command becomes the input of another.
- Makes the order and nature of operations explicit

Functional Design Patterns - Common Design Patterns

Recursion

- **A recursive function:**
  - Calls itself inside of its own body
  - Is most appropriate to a hierarchical problem like binary search, tree operations, or modeling a hierarchy such as an HTML or SVG document object or a tree-structured set of related items such as blog post comments.

State/Event

- “State” is some variable whose value describes a property of a system, such as velocity or direction.

- “Event” is some value that, when applied to the state in some way, causes that state to change to a new value.

- “State/Event” is a pattern that involves a set of known states, a set of events, and a *state change function* that produces a new state from the current stat and a given event.

Functional Design Patterns - Common Design Patterns

Resources

- http://natashatherobot.com
- “Functional Programming in Swift” - http://www.objc.io/books/
- http://www.raywenderlich.com/82599/swift-functional-programming-tutorial
- “Functional Thinking” - http://shop.oreilly.com/product/0636920029687.do
- The original “Gang of Four” book - http://books.google.com/books/about/Design_Patterns.html?id=6oHuKQe3TjQC