What is a Design Pattern?

- A standard solution to a common programming problem

Example 1: Encapsulation

- **Problem:** Exposed properties can be directly manipulated
  - Violations of the representation invariant
  - Dependencies prevent changing the implementation
- **Solution:** Hide some components
  - Constrain ways to access the object
- **Disadvantages:**
  - Interface may not (efficiently) provide all desired operations to all clients
  - Indirection may reduce performance

Example 2: Inheritance

- **Problem:** Repetition in implementations
  - Similar abstractions have similar components
- **Solution:** Inherit default members from a superclass
  - Select an implementation via run-time dispatching
- **Disadvantages:**
  - Code for a class is spread out, and thus less understandable
  - Hard to design and specify a superclass ahead of time
  - Run-time dispatching introduces overhead
Example 3: Iteration

- **Problem:** To access all members of a collection, need a specialized traversal for each data structure
  - Introduces undesirable dependences
  - Does not generalize to other collections
- **Solution:**
  - The implementation provides traversal abstraction, does bookkeeping
  - Results are communicated to clients via a standard interface (eg: Sequence methods)
- **Disadvantages:**
  - Iteration order fixed by the implementation and not under the control of the client

Example 4: Generics

- **Problem:**
  - Well-designed data structures only hold one type of object
- **Solution:**
  - Programming language checks for errors in contents
  - Set<Int> instead of just Set
- **Disadvantages:**
  - More verbose types
  - Sometimes less understandable error messages

Other Examples

- Reuse implementation without subtyping
- Reuse implementation, but change interface
- Permit a class to be instantiated only once
- Constructor that might return an existing object
- Constructor that might return a subclass object
- Combine behaviors without compile-time extends clauses
- You could come up with a solution to all of these on your own, but why reinvent the wheel???

Design Pattern in More Detail

- A standard solution to a common programming problem
  - A design or implementation structure that achieves a particular purpose
  - A high-level programming idiom
- A technique for making code more flexible
  - Reduce coupling among program components
- Shorthand for describing software design
  - connections among components, heap structure, ...
- Vocabulary for communication and documentation
When To Use A Design Pattern

• Rule #1: Delay to avoid over-thinking
  – Get something basic and concrete working first
  – Improve or generalize it once you understand it
• Design patterns can increase / decrease understandability
  – Improves modularity and flexibility, separates concerns, eases description
  – But usually adds indirection, increases code size
• If you encounter a design problem, consider design patterns that address that problem

Canonical Reference

• aka: "Gang Of Four" Book

Three Kinds Of Patterns

• Creational patterns
  – constructing objects
• Structural patterns
  – combining objects, controlling heap layout
• Behavioral patterns
  – communicating among objects, affecting object semantics

Creational Patterns

• Initializers are inflexible
  – Can’t return a subtype of class they belong to
  – Create new object, and never re-use existing one
• Factory Patterns
  – ADT creators that are not Swift init()s
• Sharing Patterns:
  – Reuse objects to save space or share common state
Factories: Changing Implementations

- Supertypes support multiple implementations
  - protocol Matrix { ... }
  - class SparseMatrix : Matrix { ... }
  - class DenseMatrix : Matrix { ... }

- Clients use the supertype (Matrix) but still create objects:
  - let m : Matrix = SparseMatrix()
  - let m : Matrix = DenseMatrix()

- Switching implementations requires changing code

A Factory

class MatrixFactory {
    public static func createMatrix() -> Matrix {
        ...
        return SparseMatrix()
    }
}

- Clients call `MatrixFactory.createMatrix()` instead of a particular constructor
- To switch implementation, change only one place
- `createMatrix()` can do arbitrary computations to decide what kind of matrix to make

Example: Bicycle race

class Race {
    public init() {
        let bike1 = Bicycle()
        let bike2 = Bicycle()
        ...
    }
}

class TourDeFrance : Race {
    public init() {
        let bike1 = RoadBicycle()
        let bike2 = RoadBicycle()
        ...
    }
}

class Cyclocross : Race {
    public init() {
        let bike1 = MountainBicycle()
        let bike2 = MountainBicycle()
        ...
    }
}

Factories Method for Bicycles

class Race {
    func createBicycle() -> Bicycle {
        Bicycle()
    }
}

class TourDeFrance : Race {
    func createBicycle() -> Bicycle {
        RoadBicycle()
    }
}

class Cyclocross : Race {
    func createBicycle() -> Bicycle {
        MountainBicycle()
    }
}
Factory Object for Bicycles

class BicycleFactory {
    func createBicycle() -> Bicycle { ... }
    func createWheel() -> Wheel { ... }
    func createFrame() -> Frame { ... }
}

class RoadBicycleFactory: BicycleFactory {
    override func createBicycle() -> Bicycle {
        RoadBicycle()
    }
}

class MountainBicycleFactory: BicycleFactory {
    override func createBicycle() -> Bicycle {
        MountainBicycle()
    }
}

Separate Control of Bicycles / Races

class Race {
    init(factory: BicycleFactory) {
        let bike1 = factory.createBicycle()
        let bike2 = factory.createBicycle()
        ...
    }
}

class TourDeFrance: Race {
    init() {
        super.init(factory: RoadBicycleFactory())
    }
}

class Cyclocross: Race {
    init() {
        super.init(factory: MountainBicycleFactory())
    }
}

Passing Factory Objects Around

class Race {
    init(factory: BicycleFactory) {
        let bike1 = factory.createBicycle()
        let bike2 = factory.createBicycle()
        ...
    }
}

class TourDeFrance: Race {
    init() {
        super.init(factory: RoadBicycleFactory())
    }
}

class Cyclocross: Race {
    init() {
        super.init(factory: MountainBicycleFactory())
    }
}

External Dependency Injection

- **Java Example:**
  - BicycleFactory f = new UnicycleFactory();
  - Race r = new TourDeFrance(f);
- **With external dependency injection:**
  - BicycleFactory f = ((BicycleFactory) DependencyManager.get("BicycleFactory"));
  - Race r = new TourDeFrance(f);
- **Plus an external file:**
  <service-point id="BicycleFactory">
    <invoke-factory>
      <construct class="Bicycle">
        <service>Tricycle</service>
      </construct>
    </invoke-factory>
  </service-point>
  + Change the factory without recompiling
  - External file is essential part of program
External Dependency Injection

- Interface Builder and Storyboards...

Factories: Summary

- **Problem**: Want more flexible abstractions for what class to instantiate.
- **Factory method**
  - Call method that can do any computation and return any subtype
- **Factory object**
  - Bundles factory methods for a family of types
  - Can store factory object, pass to constructors, etc.
- **Dependency Injection**
  - Put choice of subclass in a file to avoid source-code changes or even recompiling when decision changes

Design Patterns for Sharing

- **Problem**: Swift initializers always return a new object, never a pre-existing object
- **Singleton**: only one object exists at runtime
  - Factory method returns the same object every time
- **Interning**: only one object with a particular (abstract) value exists at run time
  - Factory method returns an existing object, not a new one

Singleton

- Only one object of the given type exists
- Good for unique, shared resources
  - UserDefaults.standard
  - DispatchQueue.main
  - UIApplication.sharedApplication()
  - Logger for diagnostic messages
- Better than lots of global properties
  - Logically group related values
  - Can use initializer / factory to customize
  - Eg: Internationalization: messages in a particular language
Creating Singletons

• In Swift class:
  – public constant property to hold singleton object
  – private initializer

  class Logger {
    static public let instance = Logger()
    private init() { ... }
  }

• In client:
  – Refer to the single instance of the Singleton class
  – Logger.instance.print("button clicked")

Interning pattern

• Reuse existing objects instead of creating new ones

  class Address : Hashable {
    let street: String
    let town: String
    ...
  }

• Less space
• objects can be compared with === instead of ==
• Sensible only for immutable objects

Simple Interning Mechanism

• Maintain a collection of all objects
• If an object already appears, return that instead

  var interned = Set<Address>()

  func intern(_ n : Address) -> Address {
    // inserts if not present, returns elem == n in set.
    let (_, memberAfterInsert) = interned.insert(n)
    return memberAfterInsert
  }

• Create the object, but perhaps discard it and return another when interning.
java.lang.Boolean and Interning

public class Boolean {
    private final boolean value;

    // construct a new Boolean value
    public Boolean(boolean value) {
        this.value = value;
    }

    // Singletons for true and false
    public static Boolean FALSE = new Boolean(false);
    public static Boolean TRUE = new Boolean(true);

    // factory method that uses interning
    public static Boolean valueOf(boolean value) {
        return value ? TRUE : FALSE;
    }
}

Boolean b = Boolean.valueOf(true);
vs
Boolean b = new Boolean(true);

Should have never been made public