Design Patterns

- A standard solutions to common programming problems
- Creational patterns
  - constructing objects
- Structural patterns
  - combining objects, controlling heap layout
- Behavioral patterns
  - communicating among objects, affecting object semantics

Structural Patterns: Wrappers

- Wrappers are a thin veneer over an encapsulated class
  - Modify the interface
  - Extend behavior
  - Restrict access
- The encapsulated class does most of the work

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Functionality</th>
<th>Interface</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adapter</td>
<td>same</td>
<td>different</td>
</tr>
<tr>
<td>Decorator</td>
<td>different</td>
<td>same</td>
</tr>
<tr>
<td>Proxy</td>
<td>same</td>
<td>same</td>
</tr>
</tbody>
</table>

Adapter

- **Problem:** interface to class doesn’t match what we want to use.
- Examples:
  - angles passed in radians vs. degrees
  - use “old” method names for legacy code
- **Solution:** Alter the interface without changing functionality
  - Rename a method
  - Convert units
  - Implement a method in terms of another
Adapter: Scaling Rectangles

- We have this Rectangle protocol
  ```swift
  protocol Rectangle {
    func scale(by: Double)
    ...
    var width : Double { get }
    var area : Double { get }
  }
  ```

- We have this class, but want one that conforms to Rectangle protocol:
  ```swift
  class NonScalableRectangle { // not a Rectangle
    var width : Double
    var area : Double
  }
  ```

  public mutable vars? Ugh, but trying to keep it simple...

Adapter with Subclassing

```swift
class ScalableRectangle : NonScalableRectangle, Rectangle {
  func scale(by amount : Double) {
    width *= amount
    height *= amount
  }
}
```

Adapter with Protocol Extension

```swift
extension NonScalableRectangle : Rectangle {
  func scale(by amount : Double) {
    width *= amount
    height *= amount
  }
}
```

Adapter with Delegation

```swift
class ScalableRectangle : Rectangle {
  let delegate : NonScalableRectangle
  init() { delegate = NonScalableRectangle() }
  var width : Double {
    get { return delegate.width }
    set { delegate.width = newValue }
  }
  func scale(by amount : Double) {
    width *= amount
    height *= amount
  }
  func circumference() -> Double {
    return delegate.circumference()
  }
}
```
**Subclass**

class ScalableRectable:
    NonScalableRectangle, Rectangle {
    func scale(by amount: Double) {
        width *= amount
        height *= amount
    }
}

**Extension**

extension NonScalableRectangle:
    Rectangle {
    func scale(by amount: Double) {
        width *= amount
        height *= amount
    }
    }

**Delegation**

class ScalableRectable: Rectangle {
    let delegate: NonScalableRectangle
    init() {
        delegate = NonScalableRectangle()
    }
    var width: Double {
        get { return delegate.width }
        set { delegate.width = newValue }
    }
    func scale(by amount: Double) {
        width *= amount
        height *= amount
    }
    func circumference() -> Double {
        return delegate.circumference()
    }
}

---

**Subclass vs Delegation vs Extension**

- **Subclassing**
  - automatically gives access to all methods of superclass
  - built in to the language (syntax, efficiency)
- **Delegation**
  - permits removal of methods (compile-time checking)
- **Extension**
  - lightweight, but limited
  - leads to poor code organization
  - no new properties
  - messy if other classes conform to Rectangle protocol

---

**Decorator**

- **Problem**: Want to add functionality to a class without changing the interface
- **Solution**: Extend existing methods to do something more than they currently do
  - (while still preserving the previous specification)
- **Not all subclassing is decoration**
  - can add new methods too!

---

**Example: Bordered Windows**

```swift
protocol Window {
    // rectangle bounding the window
    var bounds: CGRect { get }
    // draw this on the specified screen
    func draw() ...
}

class WindowImpl: Window {
    // ...
    }

class WindowImpl: Window {
    // ...
}
```
Bordered Window via Subclass

class BorderedWindow: WindowImpl {
    func draw() {
        super.draw()
        bounds.draw()
    }
}

Bordered Window via Delegation

class BorderedWindow: Window {
    let innerWindow: Window
    init(innerWindow: Window) {
        self.innerWindow = innerWindow
    }
    func draw() {
        innerWindow.draw()
        innerWindow.bounds.draw()
    }
}

Proxy

- Same interface and functionality as the wrapped class. So, uh, why wrap it?...
- Control access to wrapped object
  - Communication: manage network details when using a remote object
  - Locking: serialize access by multiple clients
  - Security: permit access only if proper credentials

Composite Entities: Bicycle

- Bicycle
  - Wheel
    - Hub
    - Spokes
    - Rim
    - Tire
  - Frame
  - Drivetrain
    - gears
    - chain
  - ...
Composite Pattern

• **Problem:** Want to manipulate a single unit and a collection of units in the same way.

• **Solution:** Make all units in a composite structure support the same interface.
  – So no need to “always know” if an object is a collection of smaller objects or not
  – Good for dealing with “part-whole” relationships

• An extended example...

Three Kinds Of Patterns

• Creational patterns
  – constructing objects

• Structural patterns
  – combining objects, controlling heap layout

• Behavioral patterns
  – communicating among objects, affecting object semantics
  – Observer Pattern

Methods on Components

```swift
protocol Component {
    func weight() -> Double
    func cost() -> Double
}

class Tire: Component {
    let price: Double
    func cost() -> Double {
        return price
    }
}
class Wheel: Component {
    let assemblyCost: Double
    let hub: Hub
    let tire: Tire
    func cost() -> Double {
        return assemblyCost + hub.cost() + tire.cost() + ...
    }
}
class Bicycle: Component {
    let assemblyCost: Double
    let frontWheel: Wheel
    let frame: Frame
    func cost() -> Double {
        return assemblyCost + frontWheel.cost() + frame.cost() + ...
    }
}
```

Traversing Composites

• **Goal:** perform operations on all parts of a composite

• **Idea:** generalize the notion of an iterator
  – process the components of a composite in an order appropriate for the application

• Example: arithmetic expressions
  – How do we represent: 2*5 + 8/4
  – How do we traverse/process these expressions?
Representing Expressions

2*5 + 8/4

Operations
- evaluate: 12
- description: "((2*5)+(8/4))"

Abstract Syntax Tree (AST)

protocol Expression {
    ...
}

class Num extends Expression {
    let val : Int
}

class Plus extends Expression {
    let lhs : Expression
    let rhs : Expression
}

class Mult extends Expression {
    let lhs : Expression
    let rhs : Expression
}

class Var extends Expression {
    let name : String
}

Object Model vs. Type Hierarchy

- AST for a + b:

  Plus
  / 
  /   
  a   b

  Var

- Class hierarchy for Expression:

  Expr
  /  
  /   
  Plus  Mult
  /  
  Num  Var

Operations on ASTs

- Need to write code for each entry in this table

<table>
<thead>
<tr>
<th>Type of Object</th>
<th>Operation</th>
<th>Eval</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Plus</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mult</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Questions:
- Should we group together the code for a particular operation or the code for a particular expression?
- Given an operation and an expression, how do we “find” the proper piece of code?
**Operations on ASTs**

- Need to write code for each entry in this table

<table>
<thead>
<tr>
<th>Type of Object</th>
<th>Number</th>
<th>Plus</th>
<th>Mult</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operation</td>
<td>eval</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>description</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Questions:**
- Should we group together the code for a particular operation or the code for a particular expression?
- Given an operation and an expression, how do we "find" the proper piece of code?

```swift
protocol Expression : CustomStringConvertible {
    var description : String { get }
    func eval() -> Int
}

class Num : Expression {
    let val : Int
    var description : String { return "\(val)" }
    func eval() -> Int { return val }
}

class Plus : Expression {
    let lhs : Expression
    let rhs : Expression
    var description : String { return "(\(lhs) + \(rhs))" }
    func eval() -> Int { return lhs.eval() + rhs.eval() }
}

class Mult : Expression {
    let lhs : Expression
    let rhs : Expression
    var description : String { return "(\(lhs) * \(rhs))" }
    func eval() -> Int { return lhs.eval() * rhs.eval() }
}
```

### Interpreter vs Procedural Pattern

- **Interpreter**: Collects code for similar objects, spreads apart code for similar operations
  - Easy to add types of objects
  - Hard to add operations
- **Procedural**: Collects code for similar operations, spreads apart code for similar objects
  - Easy to add operations
  - Hard to add types of objects

(Visitor Pattern: form of procedural... we won't cover...)
Alternative Representation

- Represent AST types as cases in an enum.

```swift
indirect enum Expression {
    case num(val: Int)
    case plus(lhs: Expression, rhs: Expression)
    case mult(lhs: Expression, rhs: Expression)
}

- **indirect** necessary when enum is *recursive*

- Not an OO design, but facilitates procedural pattern.
  - Easy to add new operations
  - Harder to add new cases

- Similar to datatypes in many functional languages.

```swift
func eval(_ expr: Expression) -> Int {
    switch expr {
    case .num(let val): return val
    case .plus(let lhs, let rhs): return eval(lhs) + eval(rhs)
    case .mult(let lhs, let rhs): return eval(lhs) * eval(rhs)
    }
}

func description(_ expr: Expression) -> String {
    switch expr {
    case .num(let val): return "\(val)"
    case .plus(let lhs, let rhs): return "\((description(lhs)) + \(description(rhs))\)"
    case .mult(let lhs, let rhs): return "\((description(lhs)) * \(description(rhs))\)"
    }
}