Why Design Patterns and Smalltalk?

- Design Patterns help explain Smalltalk’s extensive class libraries
  - Lots of examples in vendor-provided code
- Smalltalk raises programmers above the bits & bytes level
  - They can think about design rather than implementation

Designing for Extension

- Designing classes that are meant to be subclassed is HARD
  - Especially when they are concrete classes themselves
- Often you want to let a superclass control the way its subclasses will operate
  - The Hollywood principle

Template Method (325)

- Define the outline of an algorithm in an abstract superclass
- Defer some steps to subclasses
- Subclasses specialize the algorithm by overriding abstract methods
- Template Method uses three types of methods
  - Concrete operations
  - Abstract (primitive) operations
  - Hooks
Template Method Structure

```
AbstractClass
   \templateMethod
      \primitiveMethod1
      \primitiveMethod2

ConcreteClassA
   \primitiveMethod1
   \primitiveMethod2

ConcreteClassB
   \primitiveMethod1

ConcreteClassC
   \primitiveMethod1
```

Template Method Example

```
Object>>printString
   "Answer a String that is an ASCII representation of the receiver."
   | aStream aString |
   "aStream contents
   printString is a Template Method
   printOn: is a hook method
```

```
Template Method Example

Object>>printOn: aStream
   "Append the ASCII representation of the receiver to aStream. This is the default implementation which prints 'a' ('an') followed by the receiver class name."
   | aString |
   aString := self class name.
   (aString at: 1) isVowel
      ifTrue: [aStream nextPutAll: 'an '
      ifFalse: [aStream nextPutAll: 'a '
      aStream nextPutAll: aString
   This is the "default" printOn: implementation
```

```
Template Method Example

String>>printOn: aStream
   "Append the receiver as a quoted string to aStream doubling all internal single quote characters."
   aStream nextPut: $'.
   self do:
      [ :character |
      aStream nextPut: character.
      character = $'
         ifTrue: [aStream nextPut: character]].
   aStream nextPut: $'
   so 'abc' printString yields 'abc', not aString
```

Benefits of Template Method

- Makes classes easier to subclass and specialize
  - Simpler, smaller methods
- Often arises when refactoring common behavior from subclasses
  - "Refactoring to Generalize"

Object-Oriented Trees

- The next three patterns are commonly used together in OO programs in tree structures
- The structure of the algorithms involved is different from in procedural programming
OO Trees (Top-Down)

OO Trees (Bottom-up)

OO Trees (doubly-linked)

Whole-part structures

- Example: Financial portfolio composed of individual stocks and bonds
- You would like to have both the portfolio and the individual stocks and bonds act alike in some ways

Composite (163)

Composite Structure

- Compose objects into tree structures to represent part-whole hierarchies
- Lets clients treat individual objects and compositions of objects uniformly
- Abstract class represents both primitives and their containers
**Typical Composite**

```
+aComposite
  \______\   \_______\ 
 | aLeaf   |   | aLeaf   |
 \         |   \         |
 | aComposite |   | aLeaf |
 \______\   \_______\ 
 | aLeaf   |   | aLeaf |
```

**Composite in VisualWorks**

```
VisualComponent
    \_______\       \_______\ 
 | VisualPart |       | CompositePart |
 \          |       \          |
 | DependentPart |   | Wrapped |
 \          |       \          |
 | View |   | TranslatorWrapper |
 \      |       \      |
 | BoundedWrapper |   | BorderedWrapper |
 \    |       \    |
```

**Adding new behavior**

- Sometimes you need to customize the behavior of an object at runtime
  - Subclassing not an option
- Example: adding borders or scroll bars to a View

**Decorator (175)**

- Also known as: Wrapper
- Attach additional responsibilities to an object dynamically
- A flexible alternative to subclassing for extending functionality
- Forwards requests to the component, may perform additional actions
- May be nested recursively

**Decorator Structure**

```
AbstractComponent
    |   1
    |   |
        |   |
        |   |
    Decorator |   Leaf
```

**Decorator in VisualWorks**

```
VisualComponent
    \_______\       \_______\ 
 | VisualPart |       | CompositePart |
 \          |       \          |
 | DependentPart |   | Wrapped |
 \          |       \          |
 | View |   | TranslatorWrapper |
 \      |       \      |
 | BoundedWrapper |   | BorderedWrapper |
 \    |       \    |
```
Using Tree Structures

- How do you handle requests made to an OO tree?
  - Who is responsible for doing the work?
    - A tree is NOT an object -- it's a collection of objects
  - Example: Context sensitive help system

Chain of Responsibility (223)

- Avoid coupling the sender of a request to its receiver
- Request passes through multiple handlers until one handles it
- Handler should be ascertained automatically
  - Who handles request is not determined until request is processed
- Set of handlers is dynamic

Chain of Responsibility Structure

Chain of Responsibility (object)

Chain of Responsibility (message)

Chain of Responsibility (example)
Chain of Responsibility (example)

Incompatible Interfaces

- Often you have two objects with incompatible interfaces that need to work together
- Example: Electrical Voltage Adaptor

Adapter (139)

- Also known as: Wrapper
- Converts the interface of a class into another interface clients expect
- Lets classes with incompatible interfaces work together
- Adapter can add functionality the Adaptee doesn't provide
- Enables use of several existing subclasses

Adapter (Structure)

Maintaining Consistent State

Observer (293)

- Also known as: Dependents, Publish-Subscribe
- Define a one-to-many dependency
  » When one object changes state, all dependents are notified automatically
  » Dependents can update automatically
- Decouples Subject from Observer
  » Hides Observers from Subject, each other

- Often you have several objects that need to be kept consistent with the state of another object
- Example: multiple windows showing different views of the same object
Observer Structure

Observer (object)

Observer (interaction)

Summary

Summary (2)