Lecture 17: Design Patterns (part 2)

Kenneth M. Anderson
Object-Oriented Analysis and Design
CSCI 4448/6448 - Spring Semester, 2005

Last Lecture
Design Patterns
- Background and Core Concepts
- Examples
  - Singleton, Factory Method, and Adapter
Goals of Lecture

- Cover Additional Design Patterns
  - State
  - Iterator
  - Flyweight
  - Decorator
  - Observer
  - Composite

State

- Intent
  - Allow an object to alter its behavior when its internal state changes

- Motivation
  - TCPConnection example
  - A TCPConnection class must respond to an open operation differently based on its current state: established, closed, listening, etc.

- Previous Example
  - In lecture 12, we saw the State pattern being used in our MessageBuilder example
    - the MessageBuilder would respond differently to timeout events and selection events based on its current state
State, continued

- **Applicability**
  - Use State when
    - an object’s behavior depends on its state
    - operations have large, multipart conditional statements that depend on
      the object’s state

- **Participants**
  - **Context**
    - defines the interface of interest to clients
    - maintains an instance of a `ConcreteState` subclass
  - **State**
    - defines an interface for encapsulating the behavior associated with a
      particular state of the Context
  - **ConcreteState**
    - each subclass of State implements a different behavior that implements
      the correct behavior for a particular state

---

State’s Structure and Roles

```
Context
  Request()

State
  operation()

ConcreteStateA
  operation()

ConcreteStateB
  operation()
```

- `state->operation()`
State, continued

Collaborations

- Context delegates state-specific requests to the current ConcreteState object
- A context may pass itself as an argument to the State object handling the request
- Context is the primary interface for clients
- Either Context or ConcreteState subclasses can decide which state succeeds another and under what circumstances
- In the MessageBuilder example of Lecture 12, each ConcreteState object returned a pointer to the next state

Consequences

- State localizes state-specific behavior and partitions behavior for different states
- State makes state transitions explicit
- State objects can be shared

Example

- See the code distributed with Lecture 12 for an example of the State Pattern
Iterator

Intent

Provide a way to access the elements of an aggregate object (e.g. a collection class) sequentially without exposing its underlying representation

Also Known As

Cursor

Motivation

A collection may have multiple ways of being “traversed”; Iterator lets you keep traversal operations out of the core collection interface

Applicability

Use the Iterator pattern
- to access the contents of a collection without exposing its internals
- to support multiple traversals of collections
- to provide a uniform interface for traversing different collections

Participants

Iterator
- defines an interface for accessing and traversing elements

ConcreteIterator
- implements Iterator interface and keeps track of current position within collection

Aggregate (Collection Class Interface)
- defines an interface for creating an Iterator (factory method)

ConcreteAggregate (Collection Class)
- implements the factory method
Iterator’s Structure and Roles

Collaborations
- A ConcreteIterator keeps track of the current object in the aggregate and can compute the next object in the traversal

Consequences
- The Iterator pattern supports multiple traversals for each collection (e.g. inorder, preorder, postorder for trees)
- Iterators simplify Aggregate interface
- More than one traversal can occur on a single collection at once; as long as the traversal is read-only
Iterator, continued

- Implementation
  - The Iterator interface in the Java Collection classes
    - java.util.Iterator (interface)
    - java.util.List (interface)
    - java.util.LinkedList (class)
    - java.util.ListIterator (interface)
      - implementing subclass is private within List class

Flyweight

- Intent
  - Use sharing to support large numbers of fine-grained objects efficiently

- Motivation
  - Imagine a text editor that creates one object per character in a document
  - For large documents, that is a lot of objects!
    - but for simple text documents, there are only 26 letters, 10 digits, and a handful of punctuation marks being referenced by all of the individual character objects
Flyweight, continued

Applicability

- Use flyweight when all of the following are true
  - An application uses a large number of objects
  - Storage costs are high because of the sheer quantity of objects
  - Most object state can be made extrinsic
  - Many groups of objects may be replaced by relatively few shared objects once extrinsic state is removed
  - The application does not depend on object identity. Since flyweight objects may be shared, identity tests will return true for conceptually distinct objects

Participants

- Flyweight
  - declares an interface through which flyweights can receive and act on extrinsic state
- ConcreteFlyweight
  - implements Flyweight interface and adds storage for intrinsic state
- UnsharedConcreteFlyweight
  - not all flyweights need to be shared; unshared flyweights typically have children which are flyweights
- FlyweightFactory
  - creates and manages flyweight objects
- Client
  - maintains extrinsic state and stores references to flyweights
Flyweight’s Structure and Roles

if (flyweights[key] exists) {
    return existing flyweight
} else {
    create new flyweight
    add to pool of flyweights
    return the new flyweight
}

Collaborations
- Data that a flyweight needs to process must be classified as intrinsic or extrinsic
  - Intrinsic is stored with client; Extrinsic is stored with client
  - Clients should not instantiate ConcreteFlyweights directly

Consequences
- Storage savings is a tradeoff between total reduction in number of objects versus the amount of intrinsic state per flyweight and whether or not extrinsic state is computed or stored
  - Greatest savings occur when extrinsic state is computed
**Flyweight, continued**

- See code example (released with lecture 13)
- Simple implementation of flyweight pattern
  - Focus is on factory and flyweight rather than on client
  - Demonstrates how to do simple sharing of characters

**Decorator**

- **Intent**
  - Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality

- **Also Known As**
  - Wrapper

- **Motivation**
  - Sometimes we want to add responsibilities to individual objects, not to an entire class (like adding scrollbars to windows in GUI toolkits)
Decorator, continued

- **Applicability**
  - Use Decorator
    - to add responsibilities to individual objects dynamically
    - for responsibilities that can be withdrawn
    - when extension by subclassing is impractical

- **Participants**
  - **Component**
    - defines interface of objects to decorate
  - **ConcreteComponent**
    - defines an object to decorate
  - **Decorator and ConcreteDecorator**
    - Decorator maintains a reference to component and defines an interface that conforms to Component's interface; ConcreteDecorator adds responsibilities to the component

Decorator’s Structure and Roles
Decorator, continued

- **Collaborations**
  - Decorator forwards requests to its Component object. It may optionally perform additional operations before and after forwarding the request.

- **Consequences**
  - More flexibility than static inheritance
  - Avoids feature-laden classes high up in the hierarchy
  - A decorator and its component are not identical
  - Lots of little objects

Observer

- **Intent**
  - Define a one-to-many dependency between objects so that when one object changes states, all its dependents are notified and updated automatically

- **Also Known As**
  - Dependants, Publish-Subscribe

- **Motivation**
  - Need a way to update dependant objects while avoiding tight coupling
    - User Interface Example
Observer, continued

Applicability
- Use Observer
  - when an abstraction has two aspects, one dependent on the other
  - when a change to one object requires changing others
  - when an object should notify objects but should not make assumptions about which objects need to be notified

Participants
- Subject
  - provides interface to add and delete observers
- Observer
  - defines an updating interface for dependants
- ConcreteSubject
  - stores the state being observed
- ConcreteObserver
  - stores state that must be consistent with observed state

Observer’s Structure and Roles

Subject
- Attach(observer)
- Detach(observer)
- Notify()

for all o in observers
  o->Update()

ConcreteSubject
- GetState()
- SetState()

subjectState

Observer
- Update()

ConcreteObserver
- Update()

observerState

subject->GetState()...
  update observer state...
Observer, continued

Collaborations

- ConcreteSubject notifies observers whenever it changes its observed state
- After receiving a notification, ConcreteObserver gets state from ConcreteSubject
- See sequence diagram on page 295 of Design Patterns

Consequences

- Abstract coupling between Subject and Observer
  - Subjects do not know the concrete subclasses of their observers
- Support for broadcast communication
  - Subject does not know who is listening
- Unexpected updates
  - Change in state may update an unintended object, one we didn't suspect was an observer, or should only be observing at well-defined times
Composite

**Intent**
- Compose objects into tree structures to represent part-whole hierarchies
- Composite lets clients treat individual objects and compositions of objects uniformly

**Motivation**
- Image programs that allow graphic primitives to be grouped into collections of objects
  - Many operations are shared, such as move(), copy(), paste(), draw(), etc.

Composite, continued

**Applicability**
- Use Composite when
  - you want to represent part-whole hierarchies
  - you want clients to be able to ignore the difference between compositions of objects and individual objects
Composite, continued

- **Participants**
  - **Component**
    - declares the shared interface
    - declares child management operations
    - empty methods for leaves
    - defines an interface to retrieve parent
  - **Leaf**
    - implements shared interface
  - **Composite**
    - stores children
    - implements shared interface by delegating to children
    - implements child management operations
  - **Client**
    - Manipulates objects using the Component interface

Composite’s Structure and Roles

- **Client**
  - **Component**
    - Operation()
    - Add(Component)
    - Remove(Component)
    - GetChild(int)

- **Leaf**
  - Operation()

- **Composite**
  - Operation()
  - Add(Component)
  - Remove(Component)
  - GetChild(int)

for all g in children
  g->Operation()
Composite, continued

- Collaborations
  - Client uses the Component interface to interact with all objects
  - If the recipient is a leaf, then the request is handled directly
  - If the recipient is a composite, then the request is delegated to its children

- Consequences
  - Composite allows primitive objects and composite objects to be treated transparently
    - especially since the child management functions are defined in the Component interface
  - Composite simplifies code in the client
  - It makes it easy to add new types of “leaves”
    - nothing needs to change to add a new type of component (not even the client)
  - Disadvantage: Difficult to create composites that have only certain types of leaves; you need to subclass the Composite class and use run-time checks to make sure that only “legal” children are added to it
Summary

- Patterns are a design technique that can help you create more flexible software designs
- They describe generic solutions that can be applied to many different software systems
- We have now seen a number of patterns
  - Adapter, Blackboard, Composite, Decorator, Double Dispatch, Factory Method, Flyweight, Iterator, Observable, Singleton, State
- Note: some patterns built on previous ones
  - Factory Method appeared in Iterator
  - A variation of Singleton appeared in Flyweight

What’s Next?

- Refactoring
  - How to improve the structure of a software system without changing its functionality
- Test-Driven Design
  - How to evolve software systems by writing test cases FIRST!
- Spring Break!
- Design Patterns, part 3
  - Command, Facade, and more...
- Refactoring to Design Patterns
- Domain-Driven Design
  - Information from the second textbook...