Lecture 27: OO Design Patterns

Kenneth M. Anderson
Object-Oriented Analysis and Design
CSCI 6448 - Spring Semester, 2001

Goals of Lecture

- Cover OO Design Patterns
  - Background
  - Examples

Pattern Resources

- Pattern Languages of Programming
  - Technical conference on Patterns
- The Portland Pattern Repository
  - http://c2.com/ppr/
- Patterns Homepage
  - http://hillside.net/patterns/patterns.html

Design Patterns

- Addison-Wesley book published in 1995
  - Erich Gamma
  - Richard Helm
  - Ralph Johnson
  - John Vlissides
- Known as “The Gang of Four”
- Presents 23 Design Patterns
- ISBN 0-201-63361-2

What are Patterns?

- Christopher Alexander talking about buildings and towns
  - "Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice"
  - Alexander, et al., A Pattern Language. Oxford University Press, 1977

Patterns, continued

- Patterns can have different levels of abstraction
- In Design Patterns (the book),
  - Patterns are not classes
  - Patterns are not frameworks
  - Instead, Patterns are descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context

Patterns, continued

- So, patterns are formalized solutions to design problems
  - They describe techniques for maximizing flexibility, extensibility, abstraction, etc.
- These solutions can typically be translated to code in a straightforward manner

Elements of a Pattern

- Pattern Name
  - More than just a handle for referring to the pattern
  - Each name adds to a designer's vocabulary
    - Enables the discussion of design at a higher abstraction
- The Problem
  - Gives a detailed description of the problem addressed by the pattern
  - Describes when to apply a pattern
    - Often with a list of preconditions

Elements of a Pattern, continued

- The Solution
  - Describes the elements that make up the design, their relationships, responsibilities, and collaborations
  - Does not describe a concrete solution
    - Instead a template to be applied in many situations

Elements of a Pattern, continued

- The consequences
  - Describes the results and tradeoffs of applying the pattern
    - Critical for evaluating design alternatives
  - Typically include
    - Impact on flexibility, extensibility, or portability
    - Space and Time tradeoffs
    - Language and Implementation issues

Design Pattern Template

- Pattern Name and Classification
  - Creational
  - Structural
  - Behavioral
- Intent
- Also Known As
- Motivation
- Applicability
- Structure
- Participants
- Collaborations
- Consequences
- Implementation
- Sample Code
- Known Uses
- Related Patterns

Examples

- Singleton
- Factory Method
- Adapter
- Decorator
- Command
- State

Singleton

- Intent
  - Ensure a class has only one instance, and provide a global point of access to it
- Motivation
  - Some classes represent objects where multiple instances do not make sense or can lead to a security risk (e.g. Java security managers)

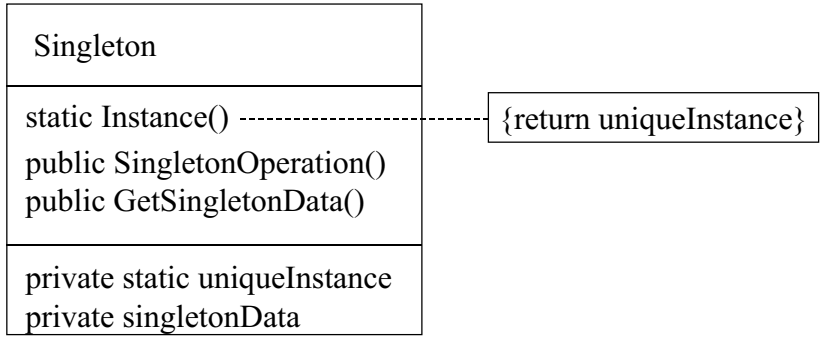
Singleton, continued

- Applicability
  - Use the Singleton pattern when
    - there must be exactly one instance of a class, and it must be accessible to clients from a well-known access point
    - when the sole instance should be extensible by subclassing, and clients should be able to use an extended instance without modifying their code

Singleton, continued

- Participants
  - Just the Singleton class
- Collaborations
  - Clients access a Singleton instance solely through Singleton's Instance operation
- Consequences
  - Controlled access to sole instance
  - Reduced name space (versus global variables)
  - Permits a variable number of instances (if desired)

Singleton Structure



Factory Method

- Intent
  - Define an interface for creating an object, but let subclasses decide which class to instantiate
- Also Known As
  - Virtual Constructor
- Motivation
  - Frameworks define abstract classes, but any particular domain needs to use specific subclasses; how can the framework create these subclasses?

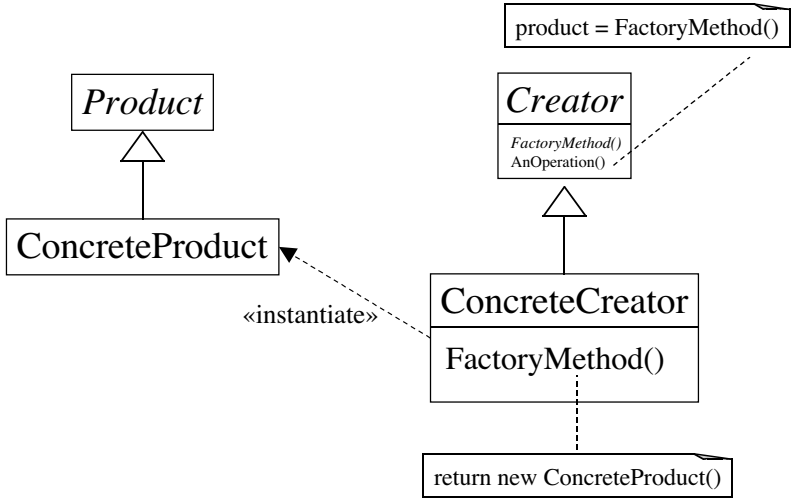
Factory Method, continued

- Applicability
  - Use the Factory Method pattern when
    - a class can't anticipate the class of objects it must create
    - a class wants its subclasses to specify the objects it creates
    - classes delegate responsibility to one of several helper subclasses, and you want to localize the knowledge of which helper subclass is the delegate

Factory Method, continued

- Participants
  - Product
    - Defines the interface of objects the factory method creates
  - Concrete Product
    - Implements the Product Interface
  - Creator
    - declares the Factory method which returns an object of type Product
  - Concrete Creator
    - overrides the factory method to return an instance of a Concrete Product

Factory Method Structure



Factory Method Consequences

- Factory methods eliminate the need to bind application-specific classes into your code
- Potential disadvantage is that clients must use subclassing in order to create a particular ConcreteProduct
  - In single-inherited systems, this constrains your partitioning choices
- Provides hooks for subclasses
- Connects parallel class hierarchies

Adapter

- Intent
  - Convert the interface of a class into another interface clients expect. Adapter lets classes work together that could not otherwise because of incompatible interfaces
- Also Known As
  - Wrapper
- Motivation
  - Sometimes a toolkit class that is designed for reuse is not reusable because its interface does not match the domain-specific interface an application requires
    - Page 139-140 of Design Patterns provides an example

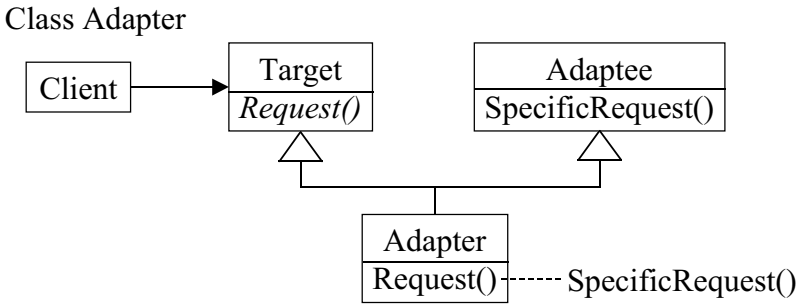
Adapter, continued

- Applicability
  - Use the Adapter pattern when
    - you want to use an existing class, and its interface does not match the one you need
    - you want to create a reusable class that cooperates with unrelated or unforeseen classes

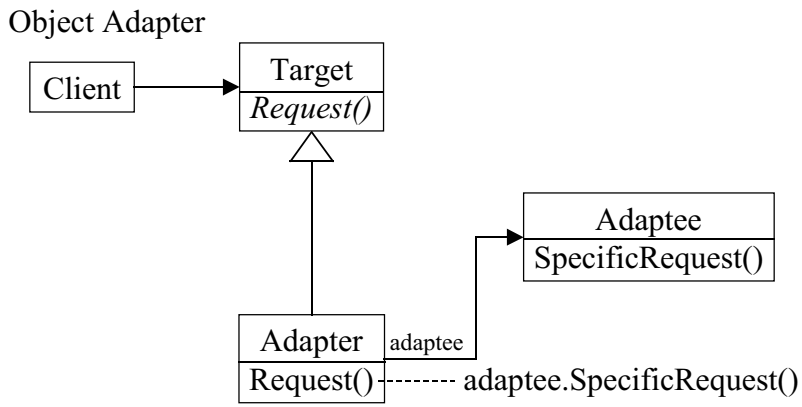
Adapter, continued

- Participants
  - Target
    - defines the domain-specific interface that Client uses
  - Client
    - collaborates with objects conforming to the Target interface
  - Adaptee
    - defines an existing interface that needs adapting
  - Adapter
    - adapts the interface of Adaptee to the Target interface

Adapter Structure



Adapter Structure



Adapter, continued

- Collaborations
  - Clients call operations on an Adapter instance. In turn, the adapter calls Adaptee operations that carry out the request
- Consequences
  - Class Adapters
    - adapts Adaptee to Target by committing to concrete Adapter class; Adapter can override Adaptee behavior
  - Object Adapters
    - lets a single Adapter work with many Adaptees; makes it harder to override Adaptee behavior

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, continued

- Structure
  - Page 177 of Design Patterns
- 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

Command

- Intent
  - Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations
- Also Known As
  - Action, Transaction
- Motivation
  - Separate details of a request from the requestor and the requestor from the receiver of the request
    - Example: Menus

Command, continued

- Applicability
  - Use the Command pattern to
    - parameterize objects by an action to perform
    - specify, queue, and execute requests
    - support undo and logging
    - structure a system around high-level operations built on primitive command

Command, continued

- Participants
  - Command
    - declares an interface for executing an operation
  - ConcreteCommand
    - defines a binding between a Receiver object and an action
    - implements Command interface
  - Client
    - creates a Concrete Command object and sets its receiver
  - Invoker
    - asks the command to carry out the request
  - Receiver
    - knows how to perform the operations of the command

Command, continued

- Structure
  - Page 236 of Design Patterns
- Collaborations
  - The client creates a ConcreteCommand object and specifies its receiver
  - An Invoker object stores the ConcreteCommand
  - The invoker issues a request by calling Execute on Command
  - The ConcreteCommand invokes operations on the Receiver
  - Page 237 of Design Patterns

Command, continued

- Consequences
  - Command decouples the object that invokes an operation from the one that implements it
  - Commands are first-class objects
  - Commands can be assembled into composite commands
  - It is easy to add new commands

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.

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, continued

- Structure
  - Page 306 of Design Patterns
- 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 of clients
  - Either Context or ConcreteState subclasses can decide which state succeeds another and under what circumstances

State, continued

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