Design Patterns

A brief introduction to what they are, why they are useful, and some examples of those that are commonly used

What are Design Patterns?

- They capture Design expertise and are abstracted from existing Design examples
- Using Design Patterns is reuse of Design Expertise
- Each pattern describes a problem which occurs over and over again in the software development design environment, and then describes the core of the solution to that problem in such a way that the solution can be used again and again, without ever doing it the same way twice

What are Design Patterns (cont'd)

- They are class- and method-level solutions to common problems in object-oriented design
- Should have a good working knowledge of OO principles (inheritance, polymorphism, abstraction, interfaces, etc.), an OO language (such as Java), and UML to use them productively
- If you want to go from being a good Java developer to a great one, study Design Patterns

Important Texts

- Definitive text: Gamma, Helm, Johnson, and Vlissides (the “Gang of Four”) – Design Patterns, Elements of Reusable Object-Oriented Software
- This book solidified thinking about patterns and became the seminal Design Patterns text
- Design Patterns in Java by Metsker and Wake is also a good text

Elements of Design Patterns – 4 Essential Parts

- Name
  - A name for a pattern adds to the design vocabulary
  - It allows design at a higher level of abstraction
- Problem
  - Description of a problem and its context
  - Sometimes includes enumeration of typical design flaws
- Solution
  - Elements that make up the design and their relationships
  - Includes responsibilities and collaborations
- Consequences
  - Time and space trade-offs
  - Possibly language and implementation concerns

Classification of Design Patterns – 3 Categories

- Creational Patterns (typically involved with object construction)
  - Factory, Abstract Factory, Builder, Prototype, Singleton
- Structural Patterns
  - Adapter, Bridge, Composite, Decorator, Façade, Flyweight, Proxy
- Behavioral Patterns
  - Interpreter, Template, Chain of Responsibility, Command, Iterator, Mediator, Memento, Observer, State, Strategy, Visitor

Where do Design Patterns sit in Software Architecture Hierarchy?

- Between Classes and Objects (the innermost layer) and Frameworks (Java API, .NET Framework, etc.)
- Layers from outermost to innermost are
  - Global
  - Enterprise
  - System
  - Application: Often a collection of Frameworks and subsystems
  - Macro: Frameworks
  - Micro: Design Patterns live here
  - Objects

Designing for Change – Causes for Redesign (I)

- Creating an object by specifying a class explicitly
  - Commits to a particular implementation instead of an interface
  - Can complicate future changes
  - Create objects indirectly
  - Patterns: Abstract Factory, Factory Method, Prototype
- Dependence on specific operations
  - Commits to one way of satisfying a request
  - Compile-time and runtime modifications to request handling can be simplified by avoiding hard-coded requests
  - Patterns: Chain of Responsibility, Command

Causes for Redesign (II)

- Dependence on hardware and software platform
  - External OS-APIs vary
  - Design system to limit platform dependencies
  - Patterns: Abstract Factory, Bridge
- Dependence on object representations or implementations
  - Clients that know how an object is represented, stored, located, or implemented might need to be changed when object changes
  - Hide information from clients to avoid cascading changes
  - Patterns: Abstract factory, Bridge, Memento, Proxy

Causes for Redesign (III)

- Algorithmic dependencies
  - Algorithms are often extended, optimized, and replaced during development and reuses
  - Algorithms that are likely to change should be isolated
  - Patterns: Builder, Iterator, Strategy, Template Method, Visitor
- Tight coupling
  - Leads to monolithic systems
  - Tightly coupled classes are hard to reuse in isolation
  - Patterns: Abstract Factory, Bridge, Chain of Responsibility, Command, Facade, Mediator, Observer

Causes for Redesign (IV)

- Extending functionality by subclassing
  - Requires in-depth understanding of the parent class
  - Overriding one operation might require overriding another
  - Can lead to an explosion of classes (for simple extensions)
  - Patterns: Bridge, Chain of Responsibility, Composite, Decorator, Observer, Strategy
- Inability to alter classes conveniently
  - Sources not available
  - Change might require modifying lots of existing classes
  - Patterns: Adapter, Decorator, Visitor

How Design Patterns Solve Design Problems

- Finding Appropriate Objects
  - Decomposing a system into objects is the hard part
  - OO-designs often end up with classes with no counterparts in real world (low-level classes like arrays)
  - Strict modeling of the real world leads to a system that reflects today's realities but not necessarily tomorrows
  - Design patterns identify less-obvious abstractions
- Determining Object Granularity
  - Objects can vary tremendously in size and number
  - **Facade pattern** describes how to represent subsystems as objects
  - **Flyweight pattern** describes how to support huge numbers of objects

Specifying Object Interfaces

- Interface:
  - Set of all signatures defined by an object's operations
  - Any request matching a signature in the objects interface may be sent to the object
  - Interfaces may contain other interfaces as subsets
- Type:
  - Denotes a particular interface
  - An object may have many types
  - Widely different objects may share a type
  - Objects of the same type need only share parts of their interfaces
  - A **subtype** contains the interface of its **supertype**
- Dynamic binding, polymorphism

Program to an interface, not an implementation

- Manipulate objects solely in terms of interfaces defined by abstract classes!

- Benefits:
  1. Clients remain unaware of the specific types of objects they use.
  2. Clients remain unaware of the classes that implement the objects.
     Clients only know about abstract class(es) defining the interfaces

  - Do not declare variables to be instances of particular concrete classes
  - Use creational patterns to create actual objects.

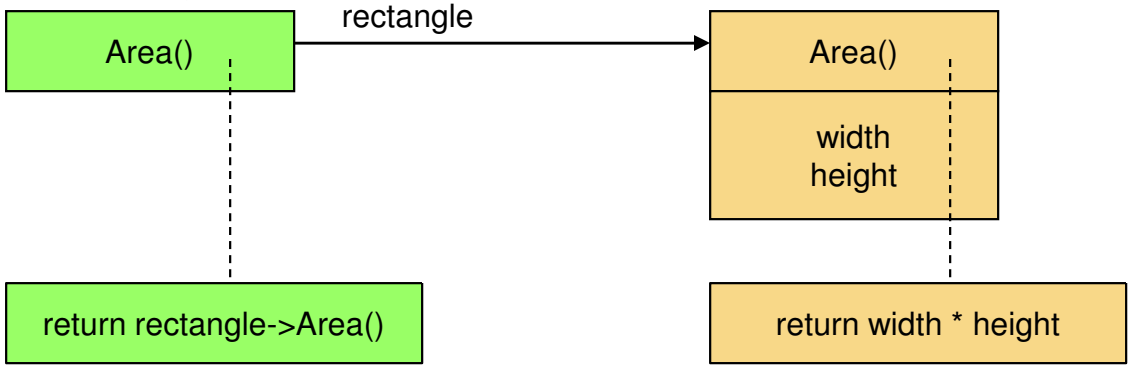
Favor object composition over class inheritance

- **White-box** reuse:
  - Reuse by subclassing (class inheritance)
  - Internals of parent classes are often visible to subclasses
  - works statically, compile-time approach
  - Inheritance breaks encapsulation
- **Black-box** reuse:
  - Reuse by object composition
  - Requires objects to have well-defined interfaces
  - No internal details of objects are visible

Delegation

Makes composition as powerful for reuse as inheritance

- Two objects involved in handling requests
- Explicit object references, no this-pointer
- Extreme example of object composition to achieve code reuse
- Drawback: dynamic, hard to understand, run-time inefficiencies



Singleton Pattern

- 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.
  - the sole instance should be extensible by sub-classing, and clients should be able to use an extended instance without modifying their code.
- Singleton classes
  - Define a method that delivers one instance (checks to see if reference is null, if so, it creates the instance)
  - Are themselves responsible for creating that instance
- We have seen an example of the Singleton Pattern in our introduction to Fitnesse
- Other examples where you want only one instance in a class: print spooler, file system, window manager

Model / View / Controller

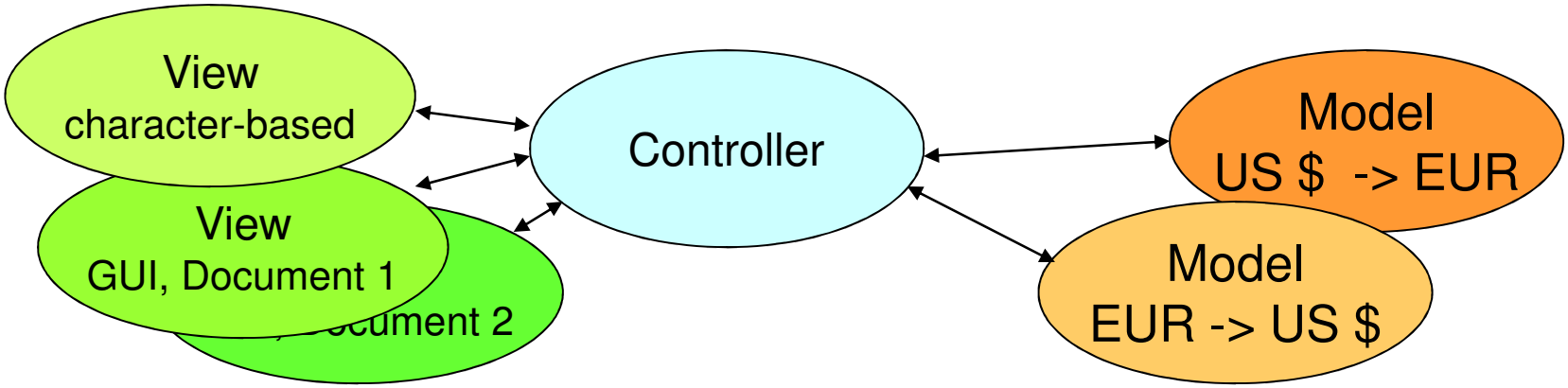
- Not listed as a pattern by Gang of Four, but recognized as one
- Model: implements algorithms and is independent of environment
- View: communicates with environment and implements I/O interface for model
- Controller: Controls data exchange (notification protocol) between model and view

MVC (cont'd)

- MVC decouples views from models – more general:
  - Decoupling objects so that changes to one can affect any number of others - without requiring the object to know details of the others
  - **Observer pattern** solves the more general problem
- MVC allows view to be nested:
  - CompositeView objects act just as View objects
  - **Composite pattern** describes the more general problem of grouping primitive and composite objects into new objects with identical interfaces
- MVC controls appearance of view by controller:
  - Example of the more general **Strategy pattern**
- MVC uses **Factory** and **Decorator patterns** as well

MVC (cont'd)

- A view of MVC



ABSTRACT FACTORY (Object Creational)

- Intent:
  - Provide an interface for creating families of related or dependent objects without specifying their concrete classes
- Motivation:
  - User interface toolkit supports multiple look-and-feel standards (Motif, Presentation Manager)
  - Different appearances and behaviors for UI widgets
  - App should not hard-code its widgets
- Solution:
  - Abstract WidgetFactory class
  - Interfaces for creating each basic kind of widget
  - Abstract class for each kind of widget,
  - Concrete classes implement specific look-and-feel

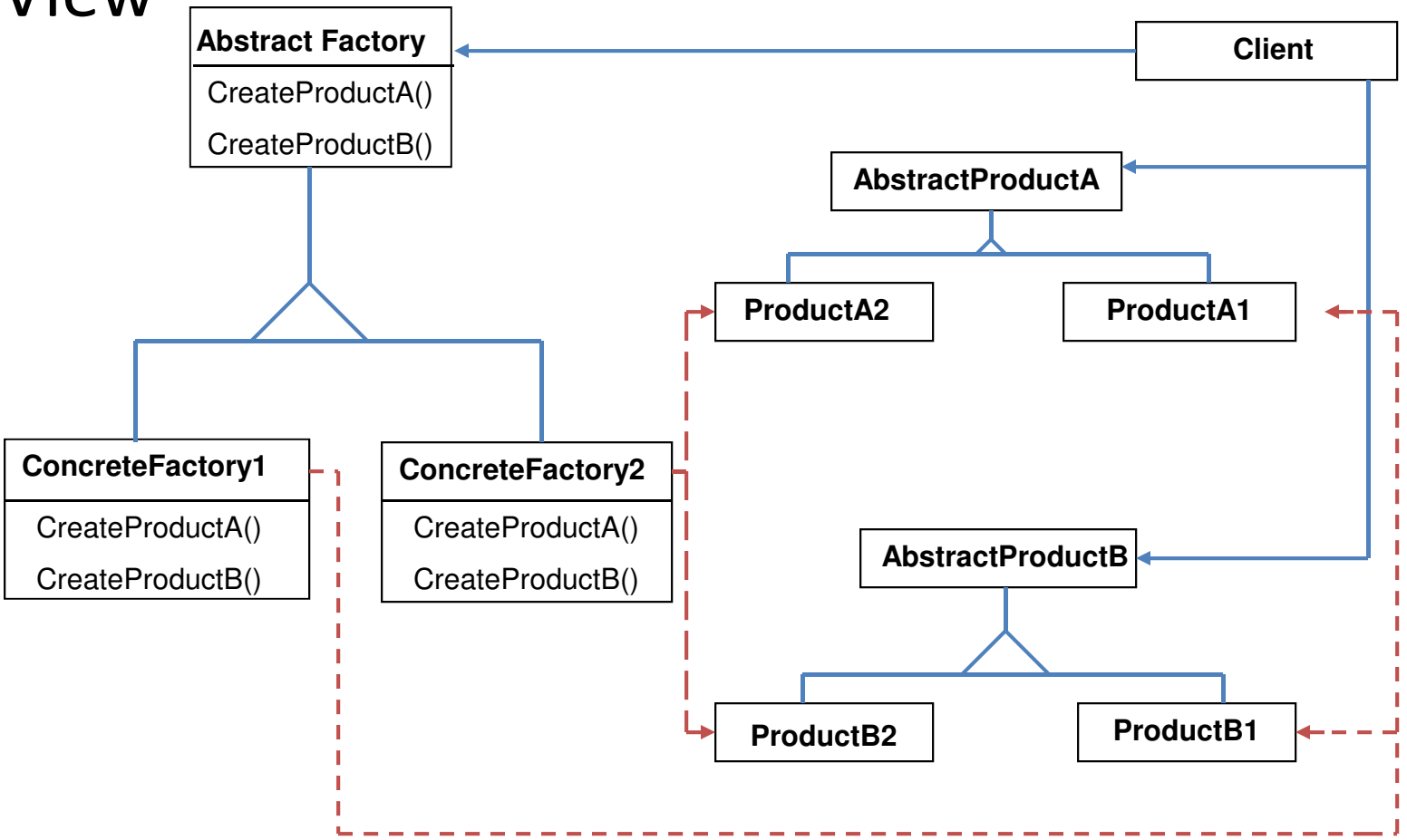
Applicability

Use the Abstract Factory pattern when

- A system should be independent of how its products are created, composed, and represented
- A system should be configured with one of multiple families of products
- A family of related product objects is designed to be used together, and you need to enforce this constraint
- You want to provide a class library of products, and you want to reveal just their interfaces, not their implementations

ABSTRACT FACTORY Structure

- View

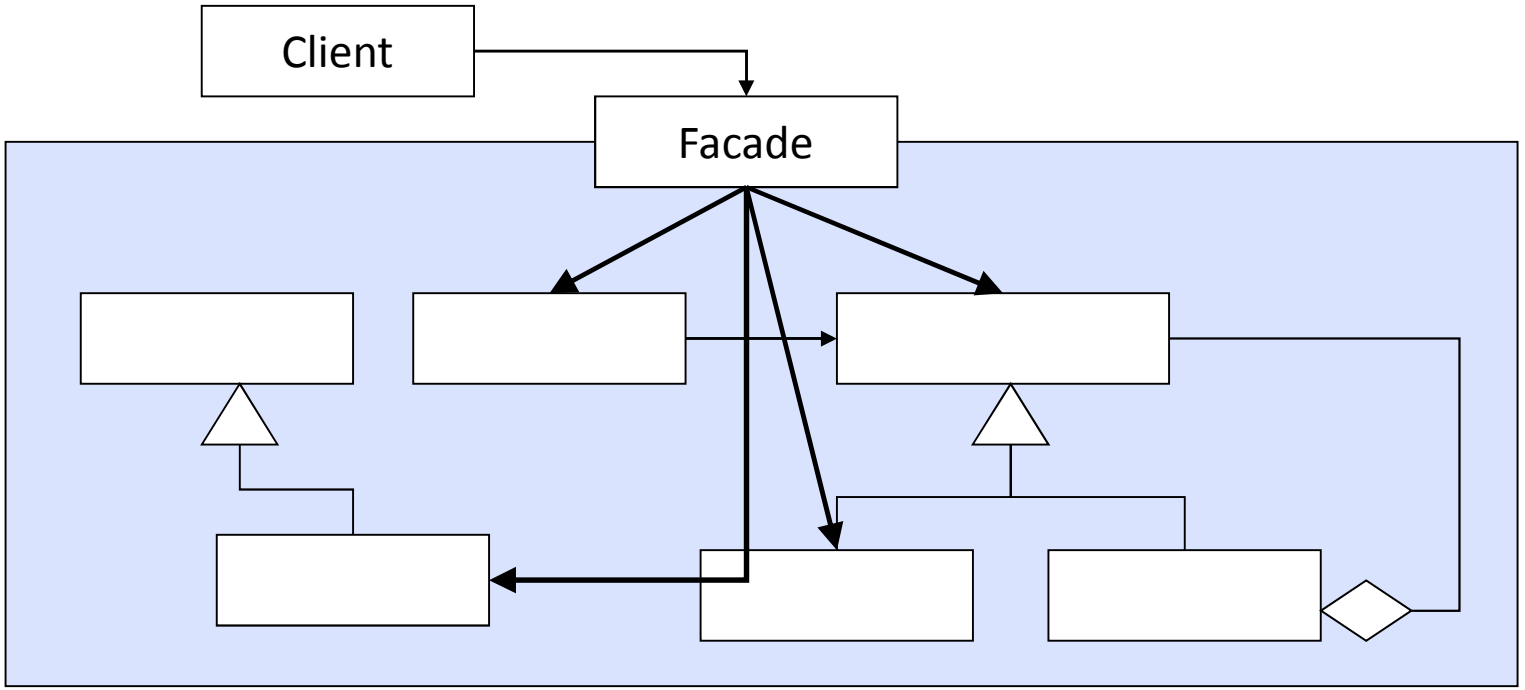


ABSTRACT FACTORY
Participants

- AbtractFactory
  - Declares interface for operations that create abstract product objects
- ConcreteFactory
  - Implements operations to create concrete product objects
- AbstractProduct
  - Declares an interface for a type of product object
- ConcreteProduct
  - Defines a product object to be created by concrete factory
  - Implements the abstract product interface
- Client
  - Uses only interfaces declared by AbstractFactory and AbstractProduct classes

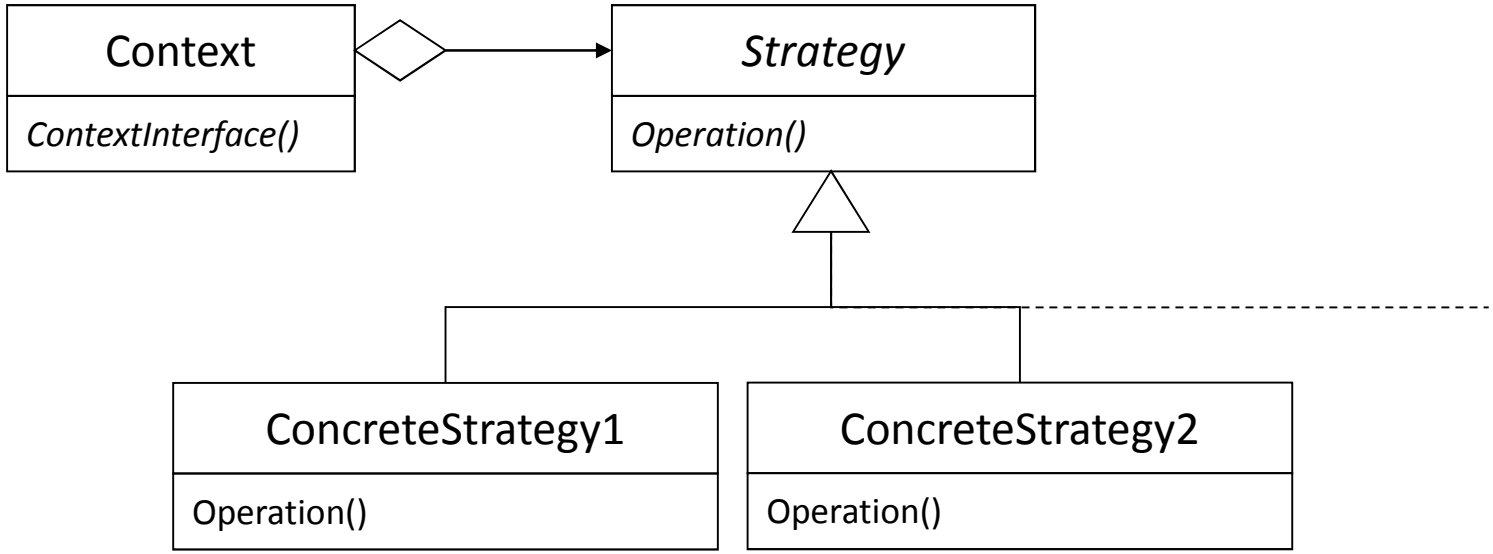
Facade

- Provide unified interface to interfaces within a subsystem
- Shield clients from subsystem components
- Promote weak coupling between client and subsystem components



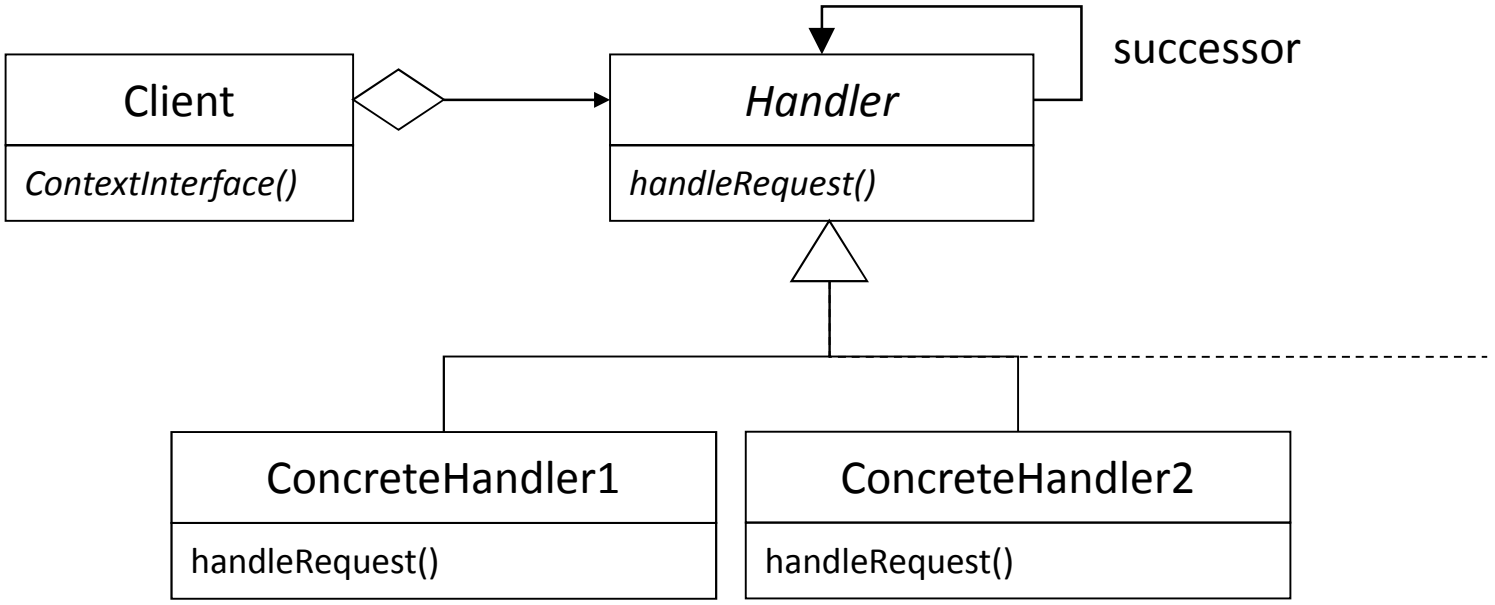
Strategy

- Make algorithms interchangeable---”changing the guts”
- Alternative to subclassing
- Choice of implementation at run-time
- Increases run-time complexity



Chain of Responsibility

- Decouple sender of a request from receiver
- Give more than one object a chance to handle
- Flexibility in assigning responsibility
- Often applied with Composite



Closing Remarks

- Learning to apply patterns is an ongoing process, just like learning to apply recursion
- The better you understand Object Oriented principles, the more able you will be to apply design patterns
- You are well served to educate yourself at least nominally on the concept of patterns – it could well be a topic of discussion at a job interview