Purpose of this Lecture

- Introduce concept of “design patterns”
- Discuss the concepts of flexible object-oriented design
- Examine how to apply patterns to form good object-oriented designs

What are Patterns?

- Many people say: "A pattern is a proven solution to a problem in a context.“
- Pattern must:
  - solve a problem
  - be related to a context
  - be general to fit in many situations
  - impart knowledge (information how the pattern solves the problem)
  - have a unique name

What are Design Patterns?

"Descriptions of communicating objects and classes that are customized to solve a general design problem"

-- Gamma, et. al.

What are Design Patterns?

- A standardized solution to a problem commonly encountered during object-oriented software development.
- It's not a piece of reusable code, but an overall approach that has proven to be useful in several different systems already.

Essential Elements of a Design Pattern

- The Pattern Name
- A statement of the problem solved by the pattern
- A description of the Solution it provides
- A list of advantages and liabilities - the Consequences (good and bad) of using it

Design Pattern Examples

- Creational Patterns:
  - E.g. Abstract Factory, Factory Method, etc
- Structural Patterns:
  - E.g. Composite, Proxy, etc.
- Behavioral Patterns:
  - E.g. Command, Visitor, etc.
- There are many other pattern collections

Name & Classification

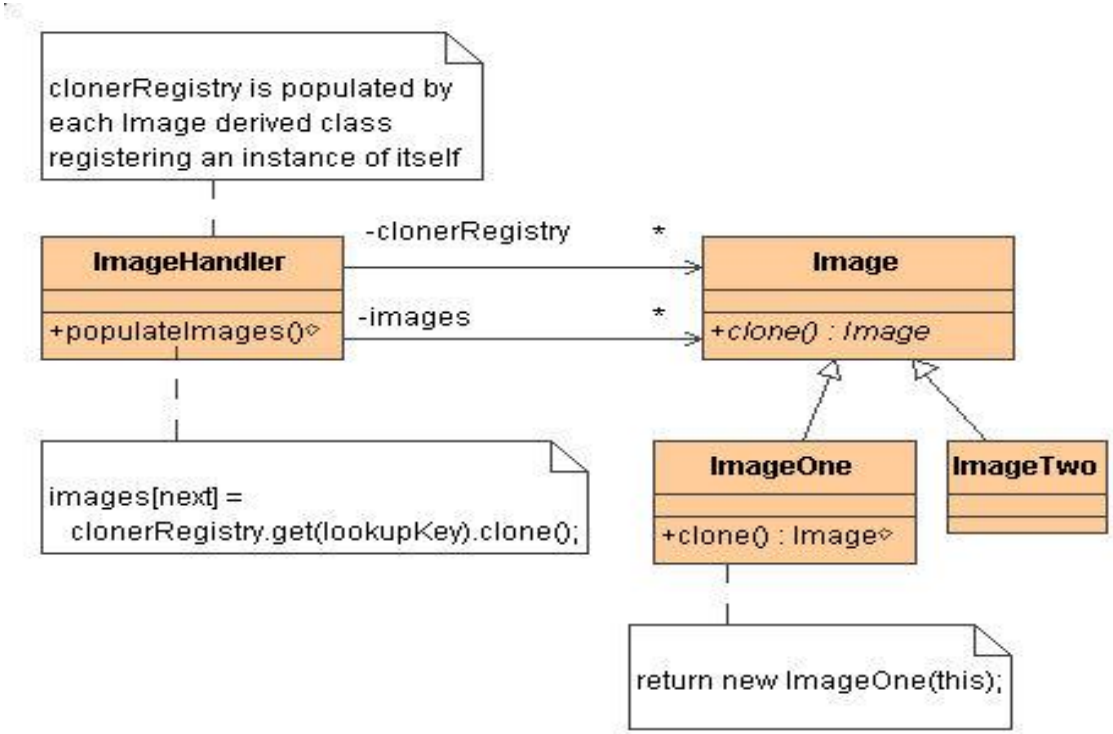
Prototype, a Creation design pattern

Intent

- The Prototype design pattern creates new instances of classes by having initialized prototype instances copy or clone themselves
- Useful when creating an instance - is usually:
  - Time-consuming
  - Very complex
- Class copies itself

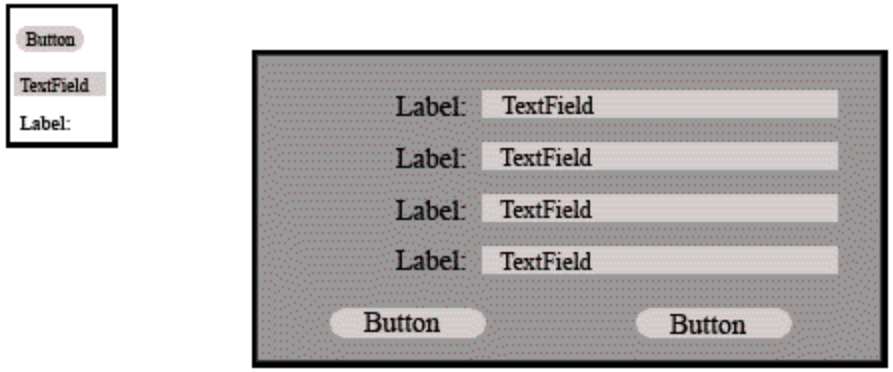
Prototype

- Factory Method: creation through inheritance.
- Prototype: creation through delegation.



Motivation

- An example of the Prototype design pattern would be in a GUI building tool



- For each graphic element which the user may customize and add to the application interface (buttons, text boxes, etc.)

Motivation

Keeping a single prototype of each GUI element on a palette for the user to choose from allows the user to customize each type once and deploy them onto the interface project as desired.

Produce a consistent look and feel across the application, and the GUI building tool need only call the instance's clone() method each time it is dragged from the palette to the GUI

Applicability

- Useful to design a system independent of how its objects are created, composed and represented
- Dynamic object specification

- No class hierarchies of "factory" classes are needed
  - Can avoid building a class hierarchy of factories paralleling the class hierarchy of products

Applicability

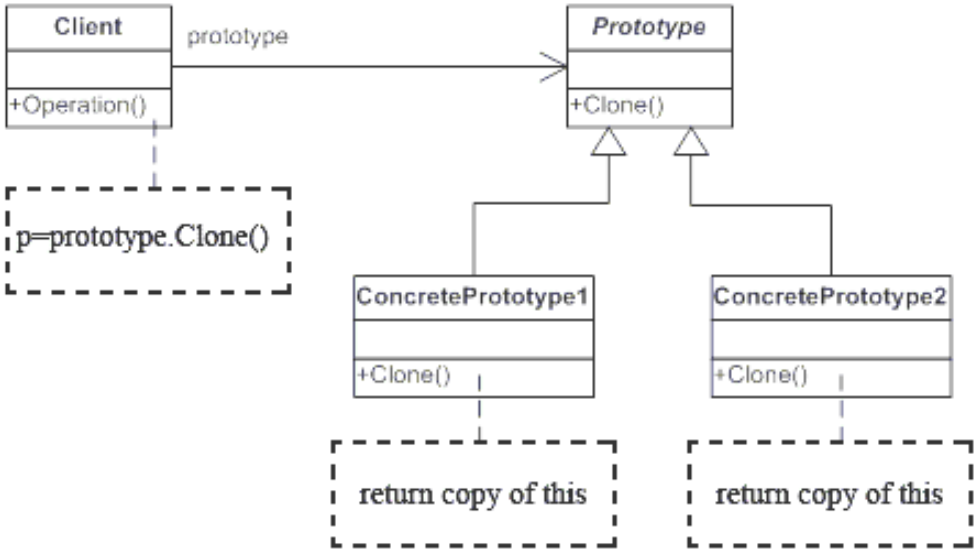
- Few different combinations of state in class instances
  - May be more convenient to install a prototype for each state and clone them, rather than instantiating the class manually, each time
  - Useful when construction and initialization of an object is more complex or time consuming than simply cloning an existing object
- Dynamic attribute specification for concrete class objects
  - Cloned concrete product object will have a complete copy of the prototype instance's state
  - Client can also specify attribute changes at creation time
  - Initialization method parameters: "copy with changes capability"

Applicability

- Allow automatic extension of the class inheritance tree for product classes
  - Automatically accommodate all subclasses of concrete classes
  - Creation logic is always attached to the concrete product class in clone() methods
  - Not in a separate concrete "factory" class

Structure

UML class diagram



Participants

Participants in the prototype pattern

- Client -
- Prototype -
  - A "clone" function, returning a copy of the original object
- ConcretePrototype - Implements the cloning operation defined in the Prototype class

Collaborations

- Collaborations between participants are initiated by the Client
- Prototypical instance of any Concrete Prototypes to be used are created

- Prototype is cloned whenever a new instance of this type is desired

Consequences

- Consequences of using the Prototype design pattern
- Alternative to Abstract Factory (and Builder, somewhat)
  - Instantiation code not in a separate concrete "factory" class
- Isolates product (data) class details from the client for instantiation
  - Streamlines client-side code for product instantiation process
  - Eliminates case-by-class handling of instantiation (same method)
  - Requires a clone (or getClone) method in each Prototype subclass
    - Difficult to reuse prexisting classes or those from external sources
    - A deepClone() method virtually requires all the member values of the Prototype subclass be declared to implement Serializable
    - Classes with circular references to other classes can’t really be cloned

Consequences

- Every Prototype subclass instantiated has a minimum of two instances
  - All used by client must be created as a prototype first
  - Possible performance drawback with large or many subclasses
- Client can be configured with Prototype subclasses dynamically
  - Can allow loading external classes into an application dynamically
  - Works well with offline or distributed object registries
- Prototype subclass member values may require more access methods
  - Any value that may differ from the prototype must be changed through access methods rather than constructors

Implementation

- Implementation issues of the Prototype design pattern:
  - Determine mechanism for generation, management, and access of initial prototype instances by client:
    - Generated externally, passed to client process
    - Managed by a prototype manager, including prototype creation

Sample Code

Client - The client object asks the prototype to clone itself

```
public class Client {
    private ConcretePrototype cpPrototype =
            new ConcretePrototype();
    …
            current = cpPrototype.getClone();
    …
}
```

Sample Code

Prototype - Data object defining an interface for creating clones of its self

```
public class Prototype implements Cloneable {
    //protected ClassMemberValue memberValue;
    public Object clone() {
      try {
          Prototype cloned = null;
          cloned = (Prototype) super.clone();
            //cloned.memberValue = memberValue;
                                        // Or use deep copy
          return cloned;
      }
```

More Information

- If Prototype Pattern may help you! Read more:
- **Solution:** Some more detailed view of what the Pattern is good for.
- **Applicability:** Tells you in which environment the Pattern makes sense.
- **Consequences:** This very important part talks about **disadvantages** and **pitfalls**.

Make Your Decision

- Ask yourself: “Is this pattern useful in my case?”
- The description of the pattern can help you to make a decision: You know…
  - …what this pattern intends to solve
  - …how it can be implemented
  - …the positive and the negative aspects

  This you know in advance, before implementation and testing.

No Magic!

- The Prototype Pattern contains nothing magical
  - Easy to understand
  - Easy to implement
  - For sure you already used Patterns without knowing it

Design Patterns are nothing mystic !!

Object Oriented Design

- Object-Oriented design and programming is widely spread and well known by programmers.

- Thinking in objects and classes are good methods in software development.

Patterns and Objects

- The implementation of Patterns were usually described as class-diagrams.
- Some Patterns (like Prototype) are about "creating" objects which only makes sense in an object-oriented environment.
- OO-programming is **not mandatory** for Patterns.
  - Patterns describe a solution for specific problems.

Pattern Catalogues

- Patterns (like Prototype) were collected in Catalogues.
- In the Internet e.g.: http://www.patterndigest.com
- In books like: “Design Patterns”
- Catalogues contain complete Pattern-Description: Name, Intent, Diagram, Solution, Consequences, …
  - Catalogues usually contain references to related Patterns, related problems, related environments, …

Pattern Catalogues (contd)

- Catalogues (usually) sort Design Patterns by categories:
  - Structural Patterns
  - Creational Patterns
  - Behavioural Patterns
  - Relationship Patterns
  - Downcasting Patterns
  - Property Patterns

Pattern Catalogues (contd)

- Usage of catalogues:
  - Browse through the catalogues for Patterns matching your problem.
  - Study class-diagrams before implementing
  - Some catalogues also contain example code (usually in Java or C++)

The GOF's Design Patterns

Highly verbose with good direction and suggestions for use, including interactions with other patterns. Each pattern contains:

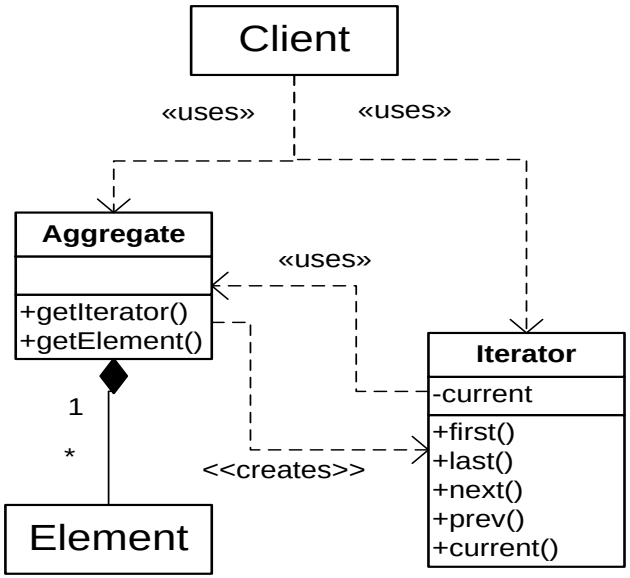
- Pattern Name
- Jurisdiction
- Intent
- Motivation
- Applicability
- Participants
- Collaborations
- Structure Diagram
- Consequences
- Implementation
- Examples
- See Also

Some Other Patterns

- Some other example Patterns:
  - Iterator
  - Command

The Iterator Pattern

- **Name:** Iterator
- **Intent:** Provide a way to access the elements of an aggregate object without exposing its underlying representation.
- **Diagram:**

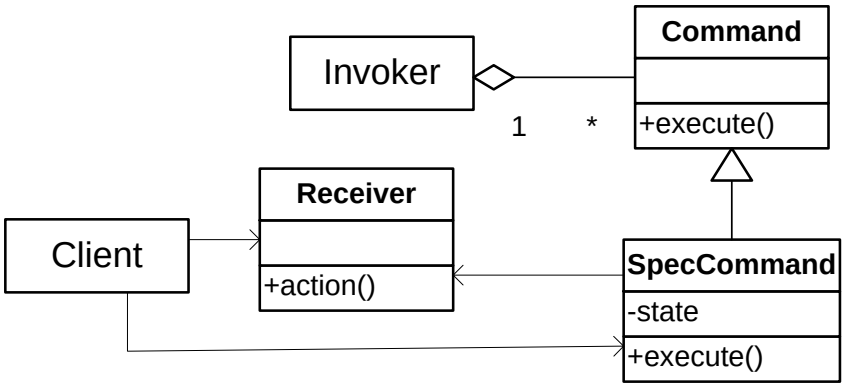


The Iterator Pattern (contd)

- The Client uses Iterator-Objects to walk through any aggregate.
- Replaces loops (for, while, etc.)
- Easy to implement
- Abstraction for accessing data
- Simple interface for all kind of aggregates
- Use more than one iterator at the same time

Command Pattern

- **Name:** Command (Behavioural Pattern)
- **Intent:** Encapsulate a request as a parameterized object; allow different requests, queue or log requests and support undoable operations
- **Diagram:**



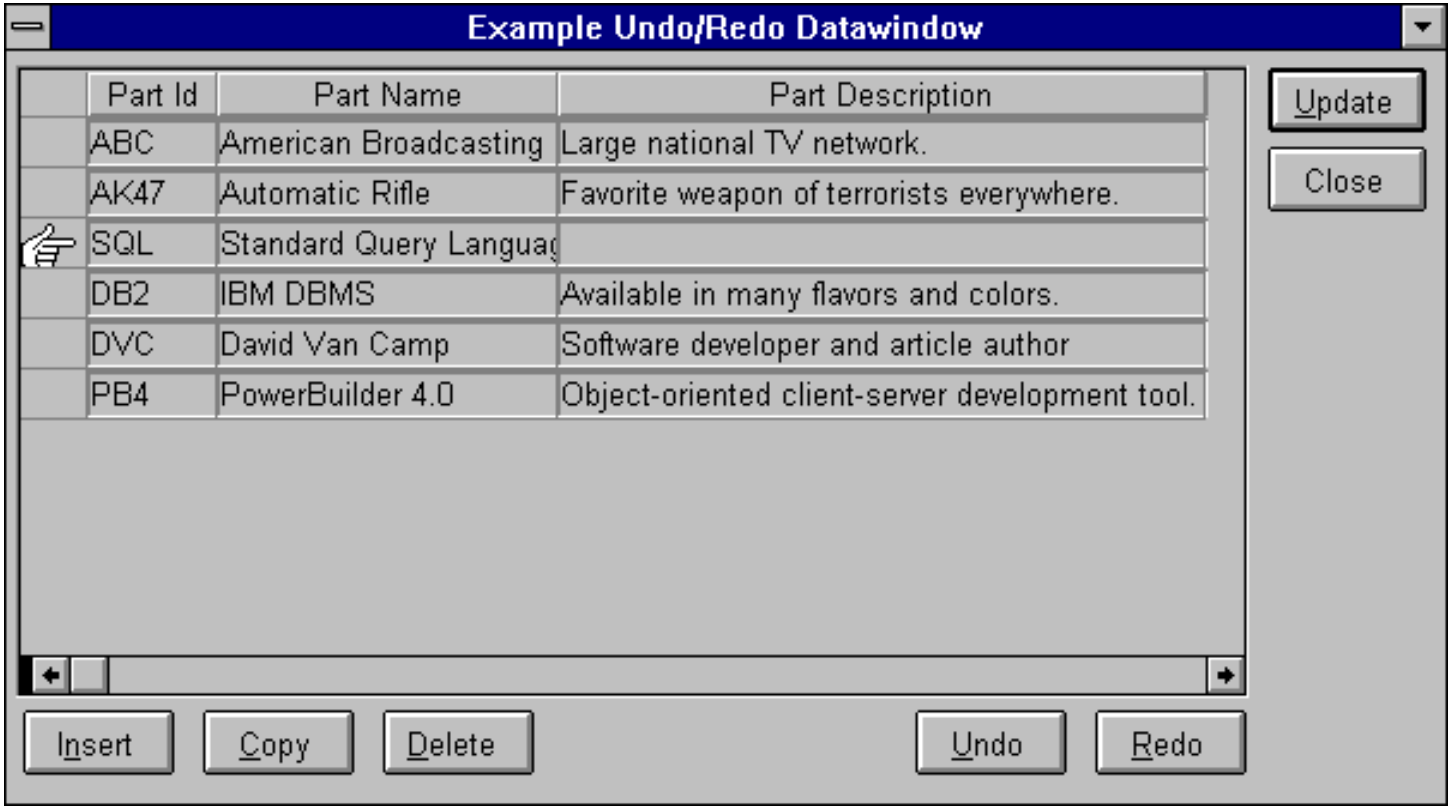
Command Pattern (contd)

- Some benefits:
  - Decouples objects from operations
  - Allows logging, “do”, “undo”
  - Easy to implement Macro-Commands

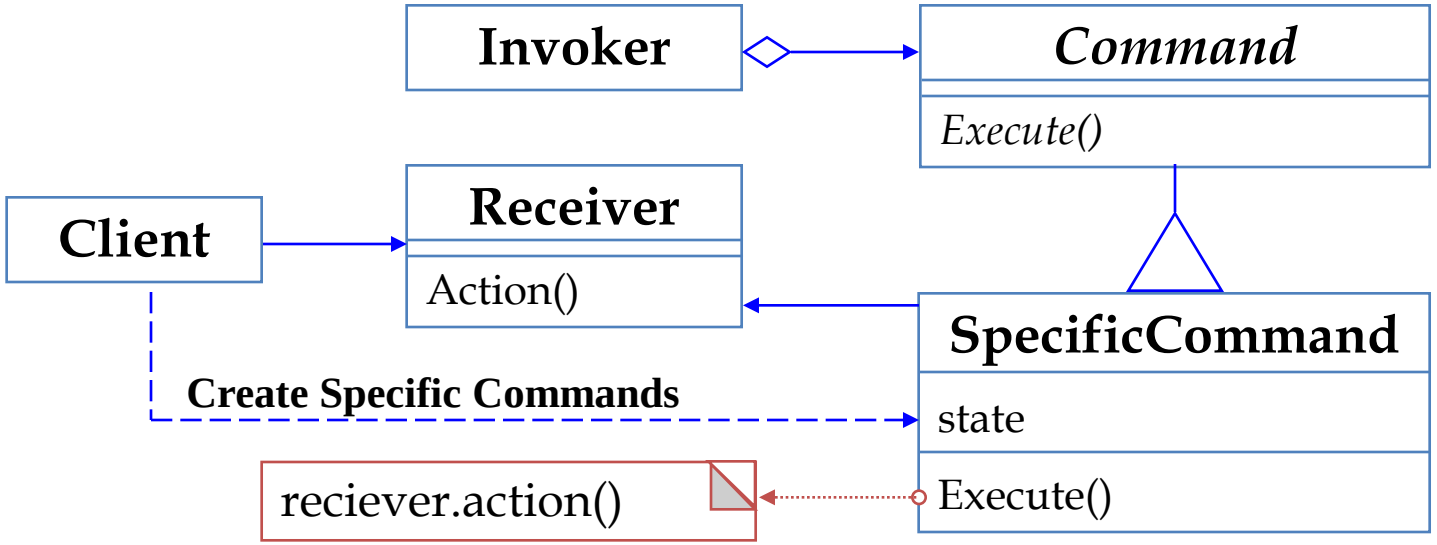
Summarize

- Patterns names may ease communication
- Patterns ease understanding software
- Patterns are proven solutions
- Patterns are well documented
- Benefits and disadvantages are known in advance

A Simple Undo/Redo Example

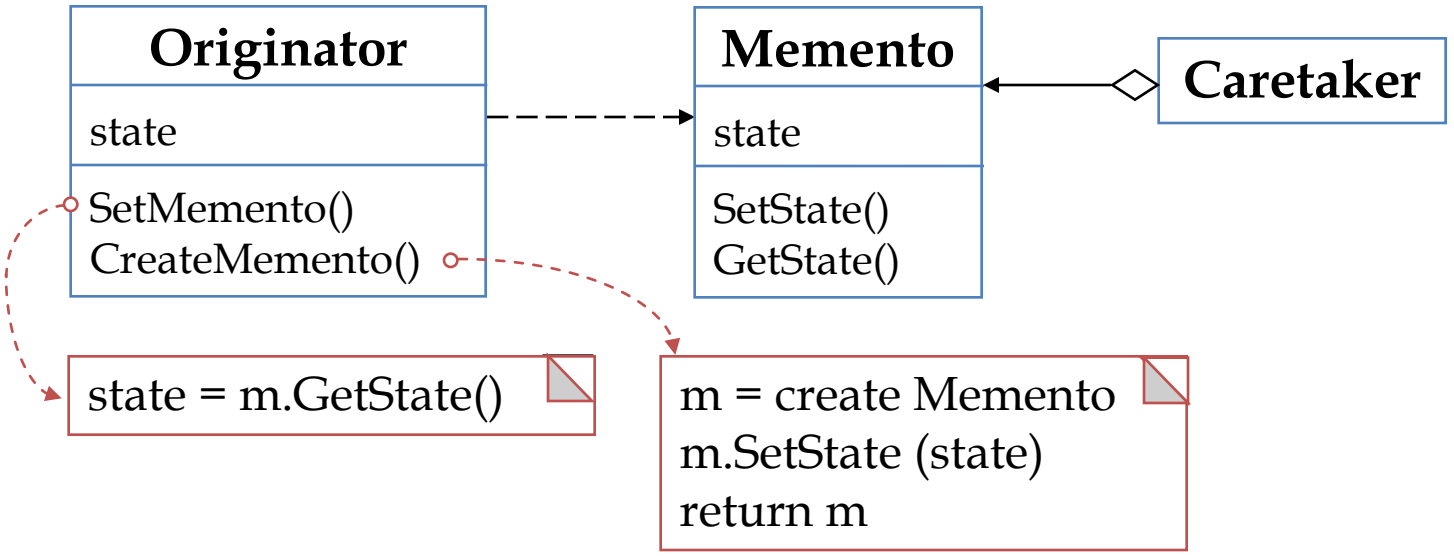


GOF Command Pattern



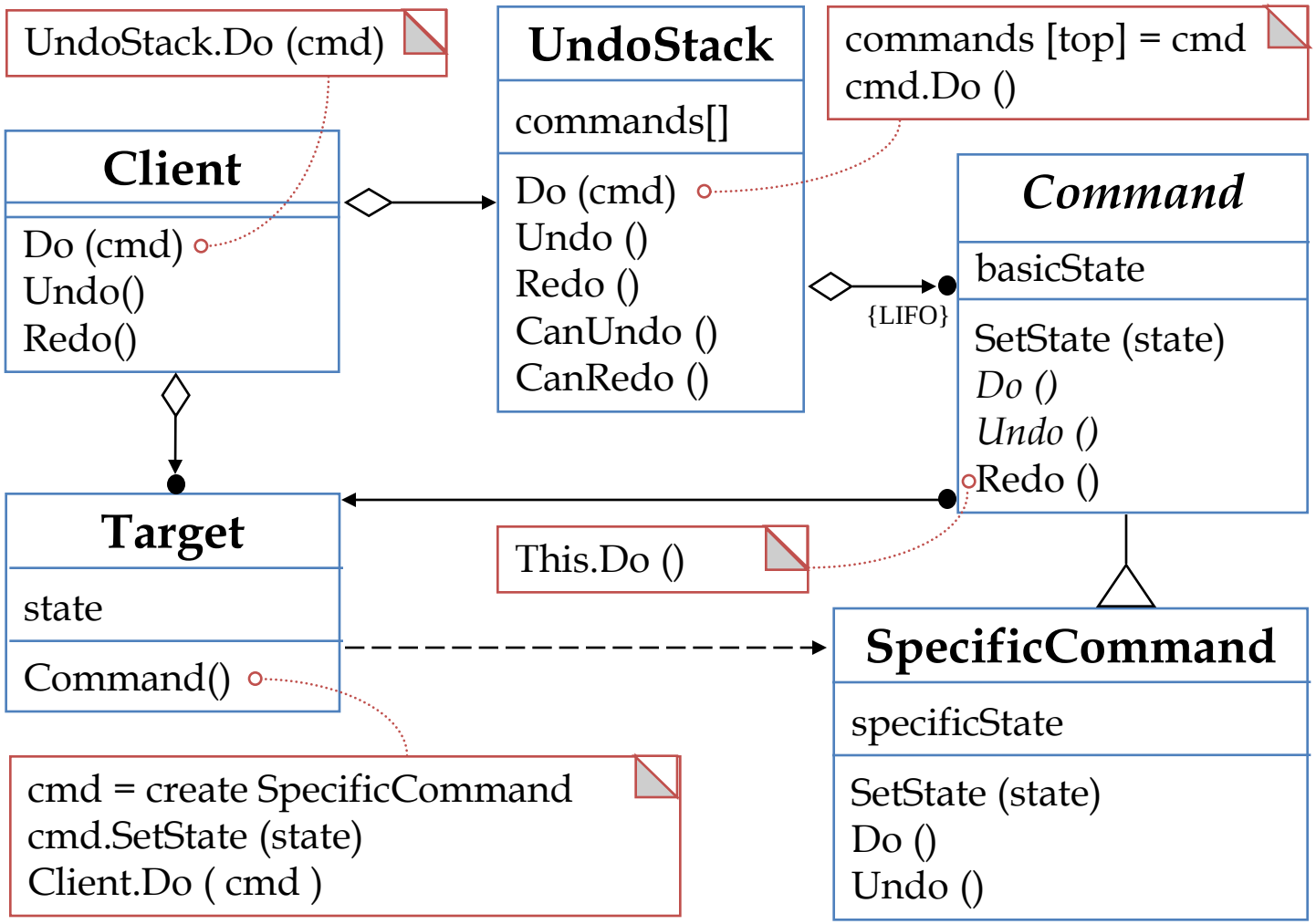
Source: *Design Patterns: Elements of Reusable Object-Oriented Software*, Gamma, Helm, Johnson & Vlissides

Memento Pattern



Source: *Design Patterns: Elements of Reusable Object-Oriented Software*, Gamma, Helm, Johnson & Vlissides

Basic Undo/Redo Pattern



Some Design Pattern Sources

- Gamma, Helm, Vlissides, Johnson: Design Patterns, 1994
- *Design Patterns for Object-Oriented Software Development*, W. Pree, Addison-Wesley, 1995
- *Pattern Languages of Program Design*, Coplien & Schmidt, eds. Addison-Wesley, 1995
- Graig Larman: Applying UML and Patterns, 2001
- Van Duyne, Hong, Landay: Design of Sites, 2002
- Pattern Homepage http://www.hillside.net
- Pattern Digest http://www.patterndigest.com
- *and many other books and articles!*

Design Pattern

- During Object Modeling we do many transformations and changes to the object model
- It’s important to make sure the object design model stays simple!

- Design patterns help keep system models simple.

Finding Objects

- The hardest problems in object-oriented system development are:
  - Identifying objects
  - Decomposing the system into objects
- Requirements Analysis focuses on application domain:
  - Object identification
- System Design addresses both, application and implementation domain:
  - Subsystem Identification
- Object Design focuses on implementation domain:
  - Additional solution objects

Techniques for Finding Objects

- Requirements Analysis
  - Start with Use Cases. Identify participating objects
  - Textual analysis of flow of events
  - Extract application domain objects by interviewing client
  - Find objects by using general knowledge
  - Extract objects from Use Case scenarios
- System Design

Another Source for Finding Objects : Design Patterns

- What are Design Patterns?
  - The recurring aspects of designs are called *design patterns* [Gamma et al 1995].
  - A *pattern* is the outline of a reusable solution to a general problem encountered in a particular context.
  - It describes the core of the solution to that problem, in such a way that you can use the solution a million times over, without ever doing it the same way twice. Many of them have been systematically documented for all software developers to use.

Studying patterns is an effective way to learn from the experience of others

Recap: Design Patterns

- What are Design Patterns?
  - The recurring aspects of designs are called *design patterns* .
  - A *pattern* is the outline of a reusable solution to a general problem encountered in a particular context.
  - It describes the core of the solution to that problem, in such a way that you can use the solution a million times over, without ever doing it the same way twice.

What is common between these definitions?

- Definition Software System
  - A software system consists of subsystems which are either other subsystems or collection of classes

- Definition Software Lifecycle:
  - The software lifecycle consists of a set of development activities which are either other actitivies or collection of tasks

Design Patterns contd.

- During Object Modeling we do many transformations and changes to the object model
- It is important to make sure the object design model stays simple!

- Design patterns helps keep system models simple.

Finding Objects

- The hardest problems in object-oriented system development are:
  - Identifying objects
  - Decomposing the system into objects
- Requirements Analysis focuses on application domain:
  - Object identification
- System Design addresses both, application and implementation domain:
  - Subsystem Identification

Techniques for Finding Objects

- Requirements Analysis
  - Start with Use Cases. Identify participating objects
  - Textual analysis of flow of events (find nouns, verbs, ...)
  - Extract application domain objects by interviewing client (application domain knowledge)
  - Find objects by using general knowledge
  - Extract objects from Use Case scenarios (dynamic model)

Another Source for Finding Objects : Design Patterns

- What are Design Patterns?
  - The recurring aspects of designs are called *design patterns* [Gamma et al 1995].
  - A *pattern* is the outline of a reusable solution to a general problem encountered in a particular context.
  - It describes the core of the solution to that problem, in such a way that you can use the solution a million times over, without ever doing it the same way twice. Many of them systematically documented for all software developers to use.

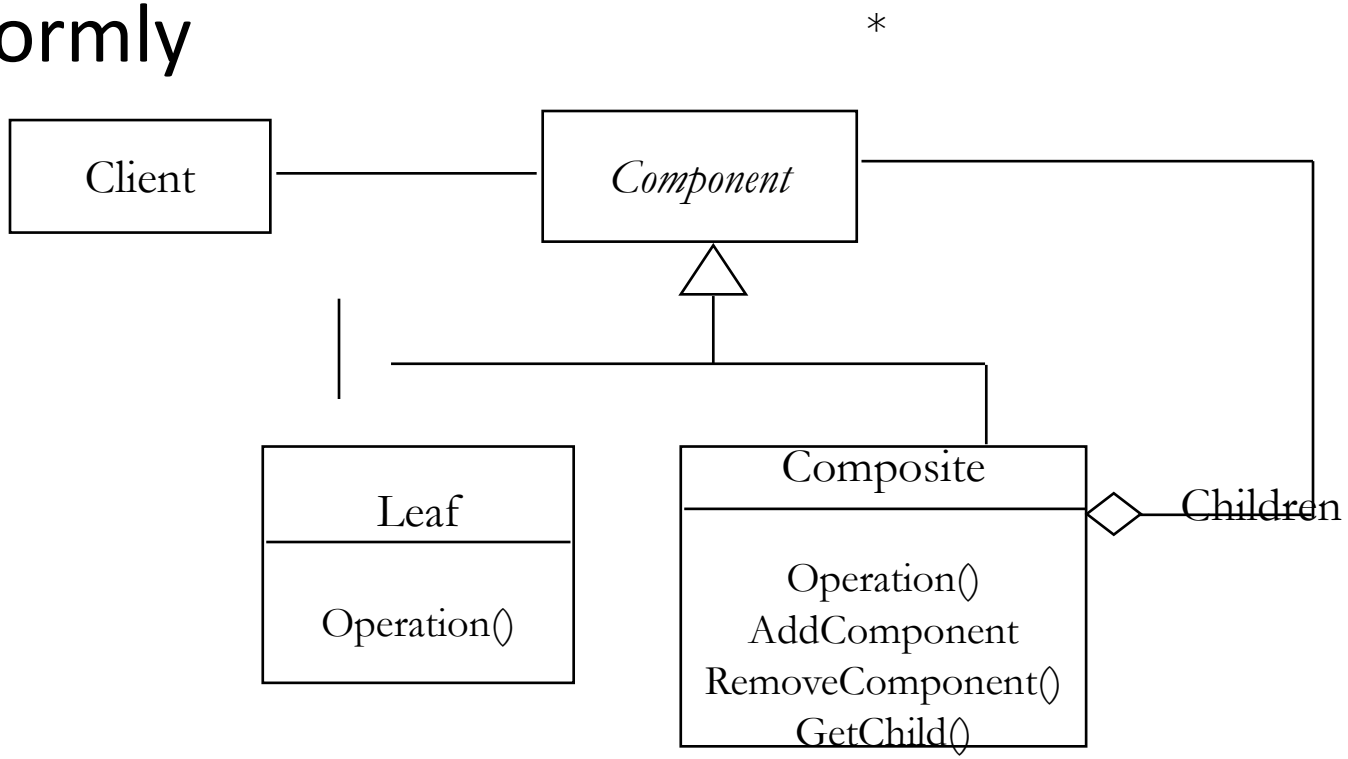
What is common between these definitions?

- Definition Software System
  - A software system consists of subsystems which are either other subsystems or collection of classes

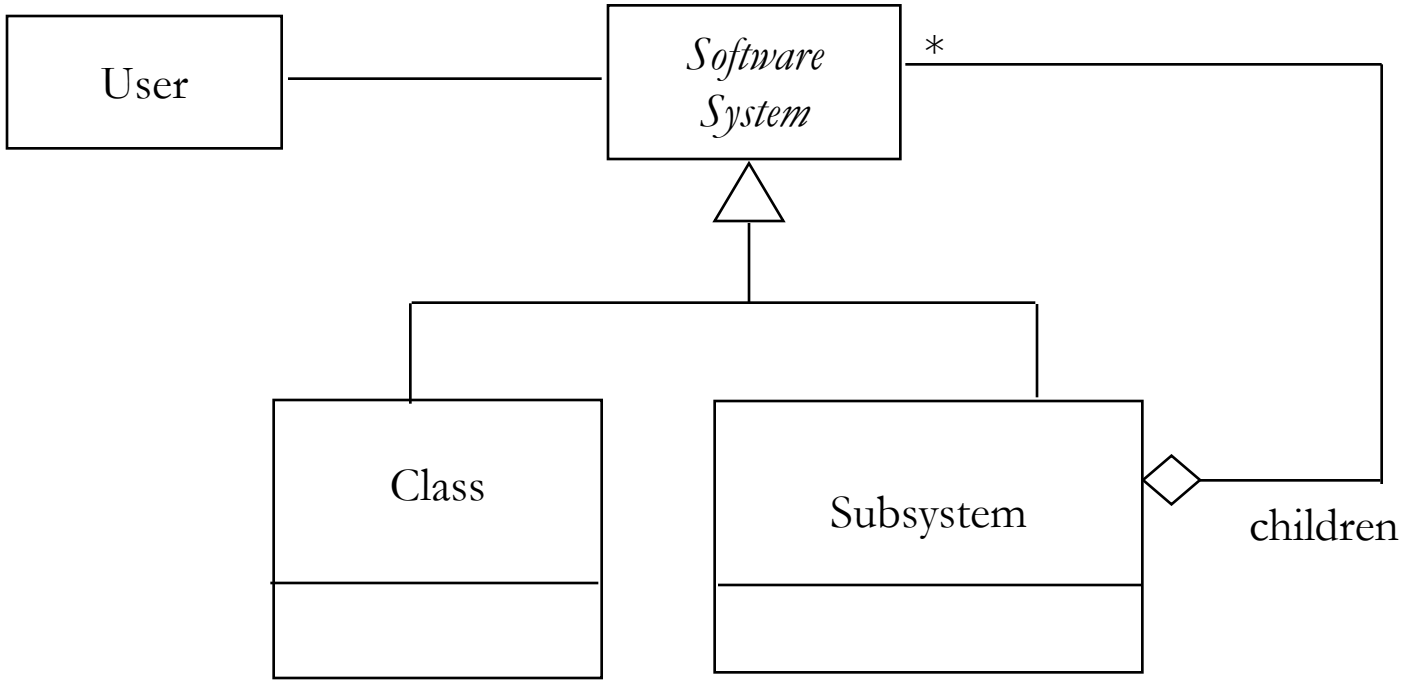
- Definition Software Lifecycle:
  - The software lifecycle consists of a set of development activities which are either other actitivies or collection of tasks

Introducing the Composite Pattern

- Models tree structures that represent part-whole hierarchies with arbitrary depth and width.
- The Composite Pattern lets client treat individual objects and compositions of these objects uniformly

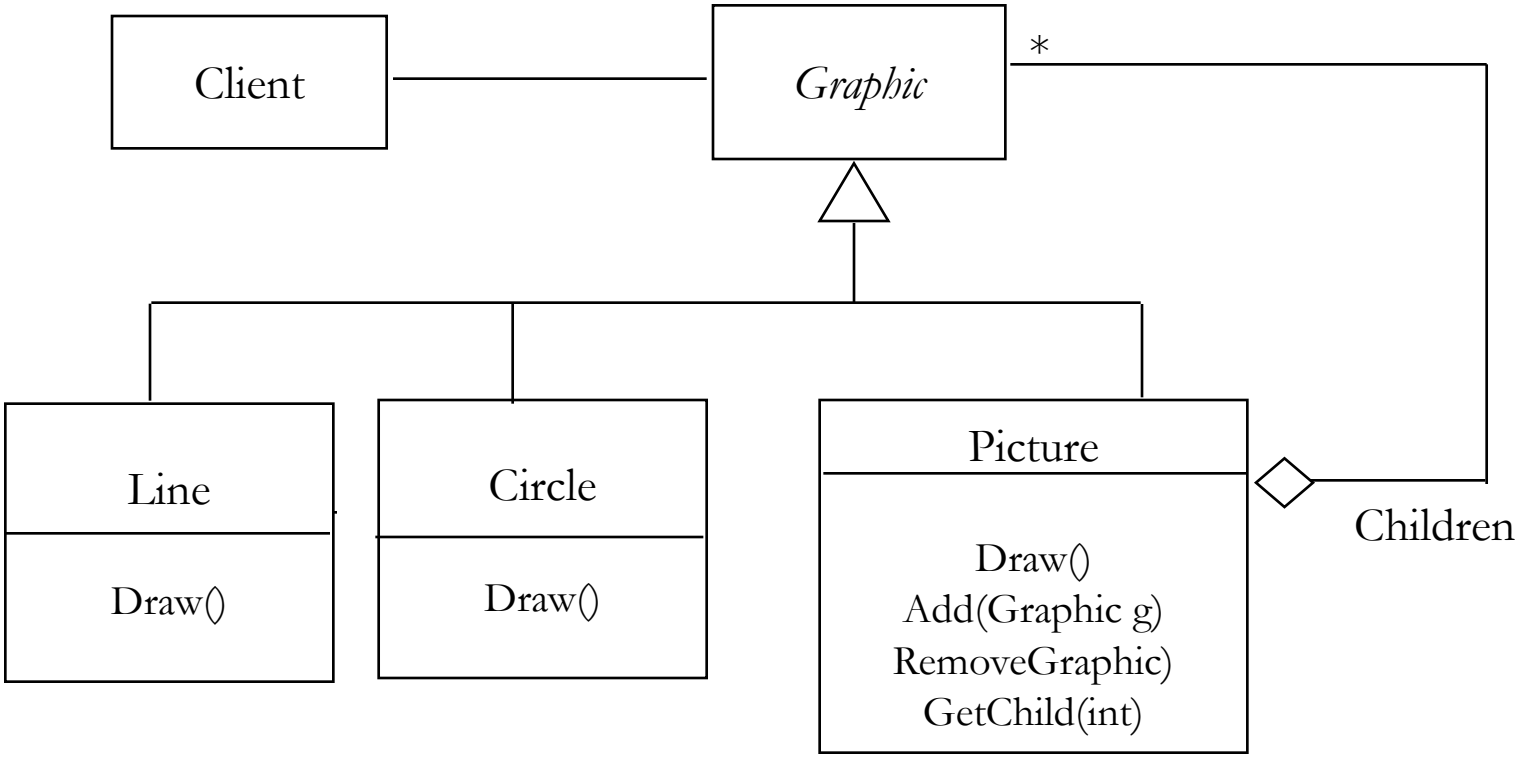


Modeling a Software System with a Composite Pattern



Graphic Applications also use Composite Patterns

• The *Graphic* Class represents both primitives (Line, Circle) and their containers (Picture)

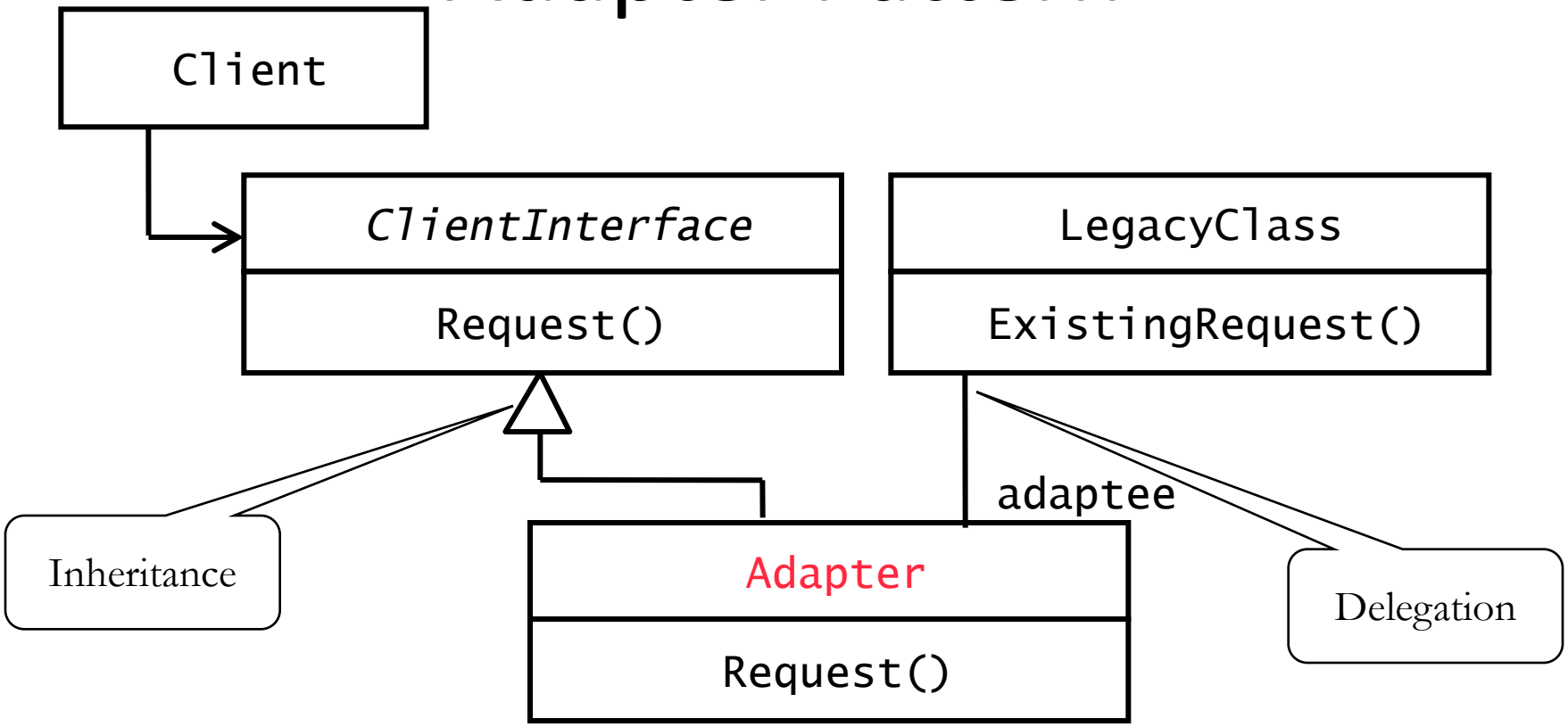


Reducing the Complexity of Models

- To communicate a complex model, use navigation and reduction of complexity
  - Do not simply use a picture from the CASE tool and dump it in front of the user
  - The key is, navigate through the model so the user can follow it
- Start with a very simple model
  - Start with the key abstractions
  - Then decorate the model with additional classes
- To reduce the complexity of the model further, look for inheritance (taxonomies)
  - Now identify or introduce patterns in the model

Many design patterns use a combination of inheritance and delegation

Adapter Pattern



The adapter pattern uses inheritance as well as delegation:

- Interface inheritance is used to specify the interface of the Adapter class.
- Delegation is used to bind the Adapter and the Adaptee

Adapter Pattern

- The adapter pattern lets classes work together that couldn't otherwise because of incompatible interfaces
  - "Convert the interface of a class into another interface expected by a client class."
  - Used to provide a new interface to existing legacy components (Interface engineering, reengineering).
- Two adapter patterns:
  - Class adapter:
    - Uses multiple inheritance to adapt one interface to another

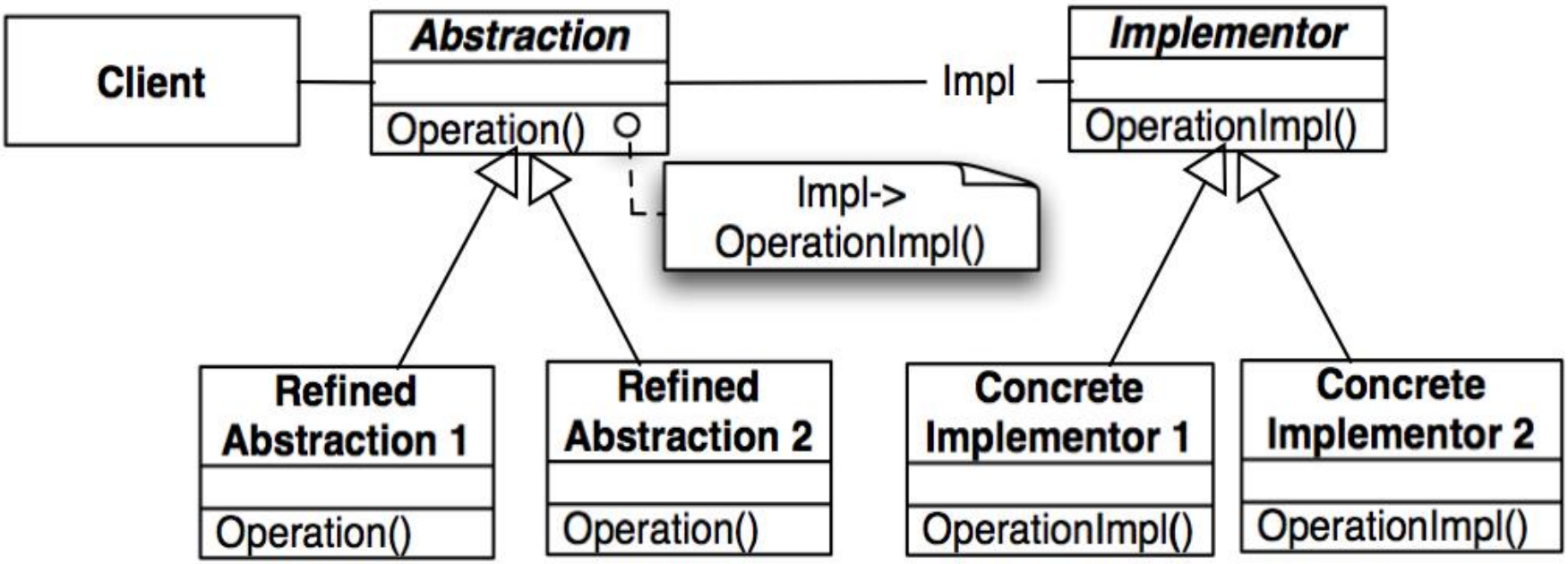
Adapter Pattern contd.

  - Object adapter:
    - Uses single inheritance and delegation
- Object adapters are much more frequent.
- We cover only object adapters (and call them adapters).

Bridge Pattern

- Use a bridge to “decouple an abstraction from its implementation so that the two can vary independently”
  - Publish interface in an inheritance hierarchy, and bury implementation in its own inheritance hierarchy.
  - Beyond encapsulation, to insulation
- Also know as a Handle/Body pattern
- Allows different implementations of an interface to be decided upon dynamically.

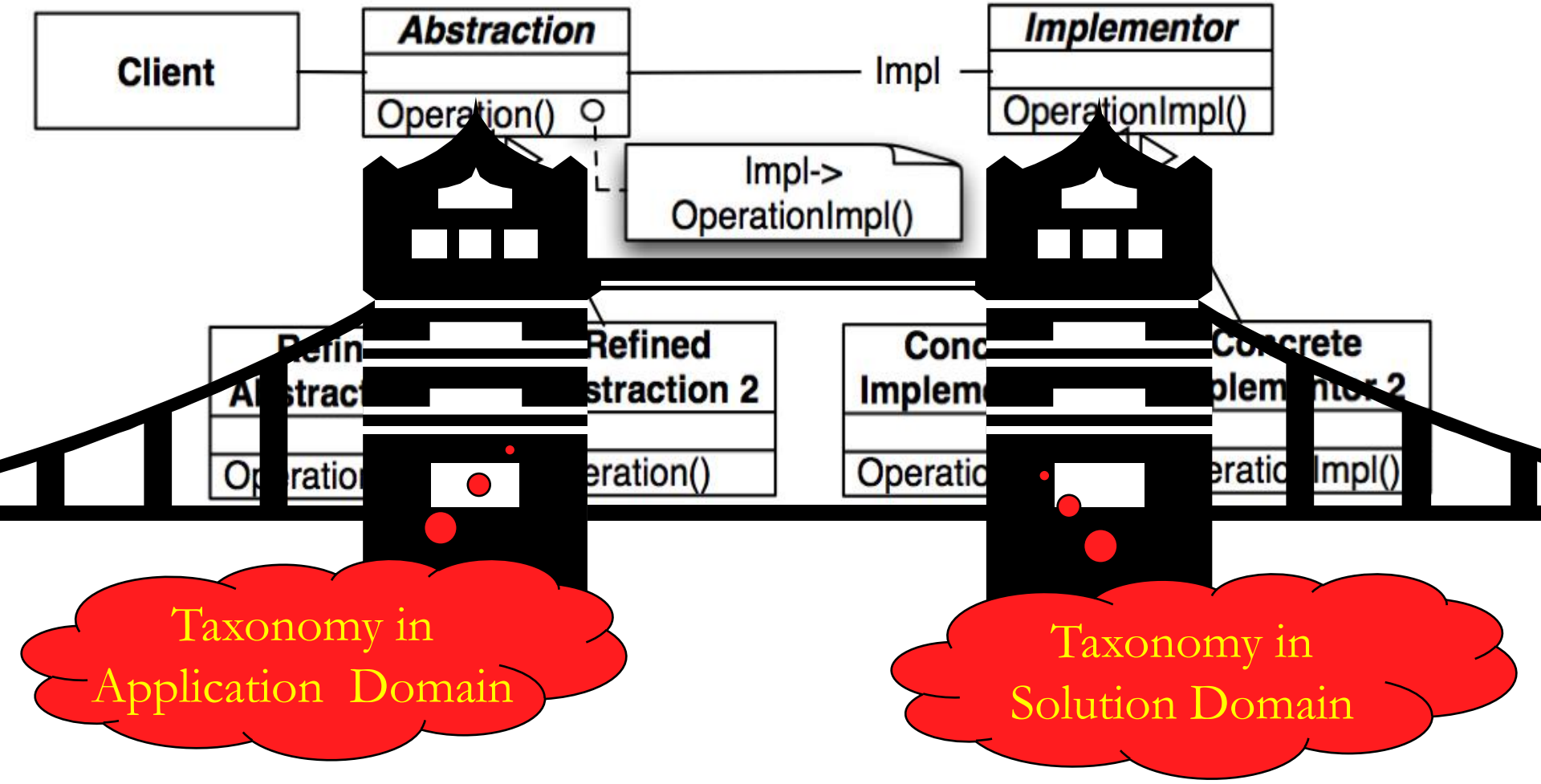
Bridge Pattern



Taxonomy in Application Domain

Taxonomy in Solution Domain

Why the Name Bridge Pattern?



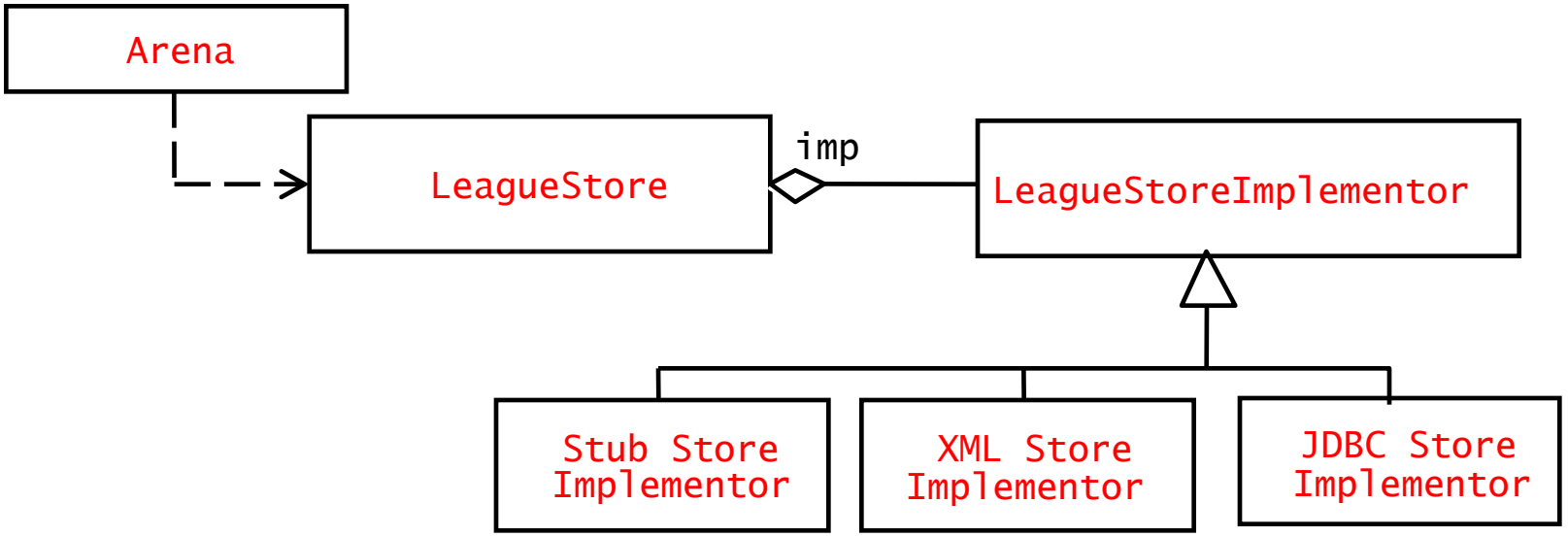
Motivation for the Bridge Pattern

- Decouples an abstraction from its implementation so that the two can vary independently
- This allows to bind one from many different implementations of an interface to a client dynamically
- Design decision that can be realized any time during the runtime of the system
  - However, usually the binding occurs at start up time of the system (e.g. in the constructor of the interface class)

Using a Bridge

- The bridge pattern can be used to provide multiple implementations under the same interface

Example use of the Bridge Pattern:
Support Multiple Database Vendors

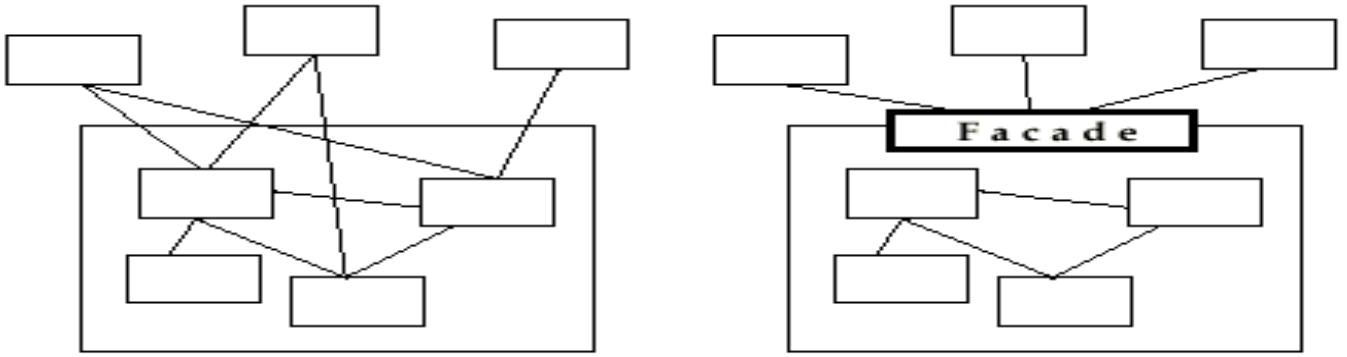


Adapter vs Bridge

- Similarities:
  - Both are used to hide the details of the underlying implementation.
- Difference:
  - The adapter pattern is geared towards making unrelated components work together
    - Applied to systems after they're designed (reengineering, interface engineering).
  - A bridge, on the other hand, is used up-front in a design to let abstractions and implementations vary independently.

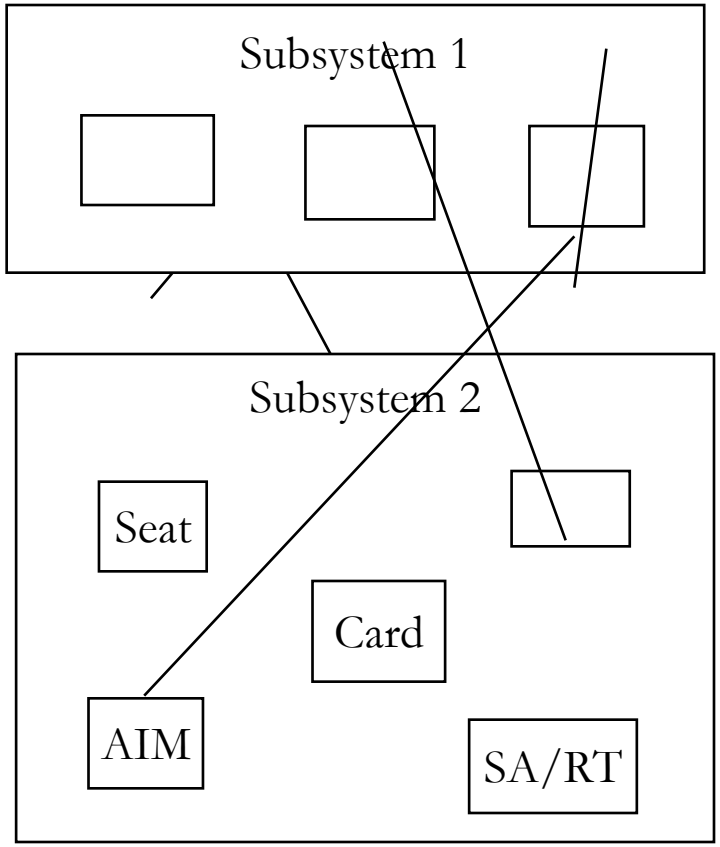
Facade Pattern

- Provides a unified interface to a set of objects in a subsystem.
- A facade defines a higher-level interface that makes the subsystem easier to use (i.e. it abstracts out the gory details)
- Facades allow us to provide a closed architecture



Design Example

- Subsystem 1 can look into the Subsystem 2 (vehicle subsystem) and call on any component or class operation at will.
- This is “Ravioli Design”
- Why is this good?
  - Efficiency
- Why is this bad?
  - Can’t expect the caller to understand how the subsystem works or the complex relationships within the subsystem.
  - We can be assured that the subsystem will be misused, leading to non-portable code



Subsystem Design with Façade, Adapter, Bridge

- The ideal structure of a subsystem consists of
  - an interface object
  - a set of application domain objects (entity objects) modeling real entities or existing systems
    - Some of the application domain objects are interfaces to existing systems
  - one or more control objects
- We can use design patterns to realize this subsystem structure
- Realization of the Interface Object: Facade
  - Provides the interface to the subsystem
- Interface to existing systems: Adapter or Bridge
  -

When should you use these Design Patterns?

- A façade should be offered by all subsystems in a software system offering services
  - The façade delegates requests to the appropriate components within the subsystem. The façade usually does not have to be changed, when the components are changed
- The adapter design pattern should be used to interface to existing components
  - Example: A smart card software system should use an adapter for a smart card reader from a specific manufacturer

When should you use these Design Patterns (2)?

- The bridge design pattern should be used to interface to a set of objects
  - where the full set of objects is not completely known at analysis or design time.
  - when a subsystem or component must be replaced later after the system has been deployed and client programs use it in the field.

Summary

- Design patterns are partial solutions to common problems such as
  - separating an interface from a number of alternate implementations
  - wrapping around a set of legacy classes
  - protecting a caller from changes associated with specific platforms
- A design pattern consists of a small number of classes
  - uses delegation and inheritance
  - provides a modifiable design solution
- These classes can be adapted and refined for the specific system under construction

Reuse and Patterns II

Motivation (1)

- Developing software is hard
- Designing reusable software is more challenging
  - finding good objects and abstractions
  - flexibility, modularity, elegance reuse
  - takes time for them to emerge, trial and error
- Successful designs do exist
  - exhibit recurring class and object structures

Motivation (2)

- During Object Modeling we do many transformations and changes to the object model.
- It is important to make sure the object design model stays simple!
- We examine how to use design patterns to keep system models simple.

Motivation (4)

- The hardest problems in object-oriented system development are:
  - Identifying objects
  - Decomposing the system into objects
- Requirements Analysis
  - focuses on application domain
  - Object identification
- System Design
  - addresses both, application and implementation domain

Another Source of Objects

- Design Patterns

- What are Design Patterns?
  - A design pattern describes a problem which occurs over and over again in our environment
  - Then it describes the core of the solution to that problem, in such a way that you can use the solution a million times over, without ever doing it the same twice

Design Patterns (1)

A design pattern is…

…a template solution to a recurring design problem

- Look before re-inventing the wheel just one more time

…reusable design knowledge

- Higher level than classes or datastructures (link lists,binary trees...)
- Lower level than application frameworks

Design Patterns (2)

- Describes recurring design structure
  - names, abstracts from concrete designs
  - identifies classes, collaborations, responsibilities
  - applicability, trade-offs, consequences
- Design patterns represent solutions to problems that arise when developing software within a particular context
  - "Patterns == problem/solution pairs in a context"
- Patterns capture the static and dynamic *structure* and *collaboration* among key participants in software designs
  - Especially good for describing how and why to resolve *non-functional issues*

- Patterns facilitate reuse of successful software architectures and designs.

Applications

- Wide variety of application domains:
  - drawing editors, banking, CAD, CAE, cellular network management, telecomm switches, program visualization
- Wide variety of technical areas:
  - user interface, communications, persistent objects, O/S kernels, distributed systems

Why modifiable designs?

A modifiable design enables…

…an iterative and incremental development cycle

- concurrent development
- risk management
- flexibility to change

…to minimize the introduction of new problems when fixing old ones

…to deliver more functionality after initial delivery

What makes a design modifiable?

- Low coupling and high cohesion
- Clear dependencies
- Explicit assumptions

How do design patterns help?

- They are generalized from existing systems
- They provide a ***shared vocabulary*** to designers

Design Pattern Space

- Creational patterns:
  - Deal with initializing and configuring classes and objects
- Structural patterns:
  - Deal with decoupling interface and implementation of classes and objects
  - Composition of classes or objects
- Behavioral patterns:
  - Deal with dynamic interactions among societies of classes and objects

Categorize Design Patterns

		Purpose		
		Creational	Structural	Behavioral
Scope	Class	Factory Method	Adapter (class)	Interpreter Template Method
	Object	Abstract Factory Builder Prototype Singleton	Adapter (object) Bridge Composite Decorator Flyweight Facade Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

Adapter pattern

- Delegation is used to bind an **Adapter** and an **Adaptee**
- Interface inheritance is used to specify the interface of the **Adapter** class.
- ***Target*** and **Adaptee** (usually called legacy system) pre-exist the **Adapter.**

Adapter Pattern

- “Convert the interface of a class into another interface clients expect.”
- The adapter pattern lets classes work together that couldn’t otherwise because of incompatible interfaces
- Used to provide a new interface to existing legacy components (Interface engineering, reengineering).
- Also known as a wrapper
- Two adapter patterns:
  - Class adapter:
  - Object adapter:
- Object adapters are much more frequent.

Bridge Pattern

- Use a bridge to “decouple an abstraction from its implementation so that the two can vary independently”.
- Also know as a Handle/Body pattern.
- Allows different implementations of an interface to be decided upon dynamically.

Using a Bridge

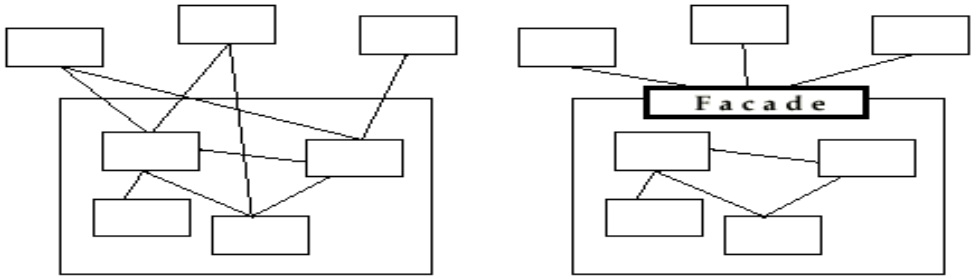
- Used to provide multiple implementations under the same interface.
- Examples: Interface to a component that is incomplete, not yet known or unavailable during testing

Adapter vs Bridge

- Similarities:
  - Both are used to hide the details of the underlying implementation.
- Difference:
  - Adapter pattern geared towards making unrelated components work together
    - Applied to systems after they're designed (reengineering, interface engineering).
  - Bridge, used up-front in a design to let abstractions and implementations vary independently.

Facade Pattern

- Provides a unified interface to a set of objects in a subsystem.
- A facade defines a higher-level interface that makes the subsystem easier to use
- Facades allow us to provide a closed architecture



Encouraging Reusable Designs

- A facade pattern should be used by all subsystems in a software system. The façade defines all the services of the subsystem.

- Adapters should be used to interface to existing components.

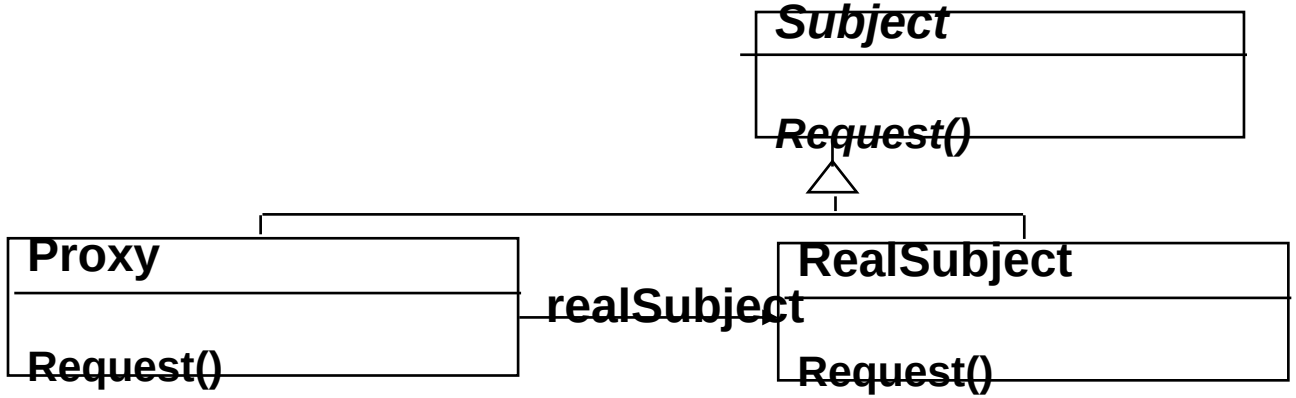
- Bridges should be used to interface to a set of objects. Use it for testing.

Proxy Pattern

- What is expensive?
  - Object Creation
  - Object Initialization
- Defer object creation and object initialization to the time you need the object
- Proxy pattern:
  - Reduces the cost of accessing objects
  - Uses another object (“the proxy”) that acts as a stand-in for the real object
  - The proxy creates the real object only if the user asks for it

Proxy pattern

- Interface inheritance is used to specify the interface shared by **Proxy** and **RealSubject.**
- Delegation is used to catch and forward any accesses to the **RealSubject** (if desired)
- Proxy patterns can be used for lazy evaluation and for remote invocation.
- Proxy patterns can be implemented with a Java interface.



Proxy Applicability

- Remote Proxy
  - Local representative for an object in a different address space
  - Caching of information: Good if information does not change too often
- Virtual Proxy
  - Object is too expensive to create or too expensive to download
  - Proxy is a stand-in
- Protection Proxy
  - Proxy provides access control to the real object
  - Useful when different objects should have different access and viewing rights for the same document.

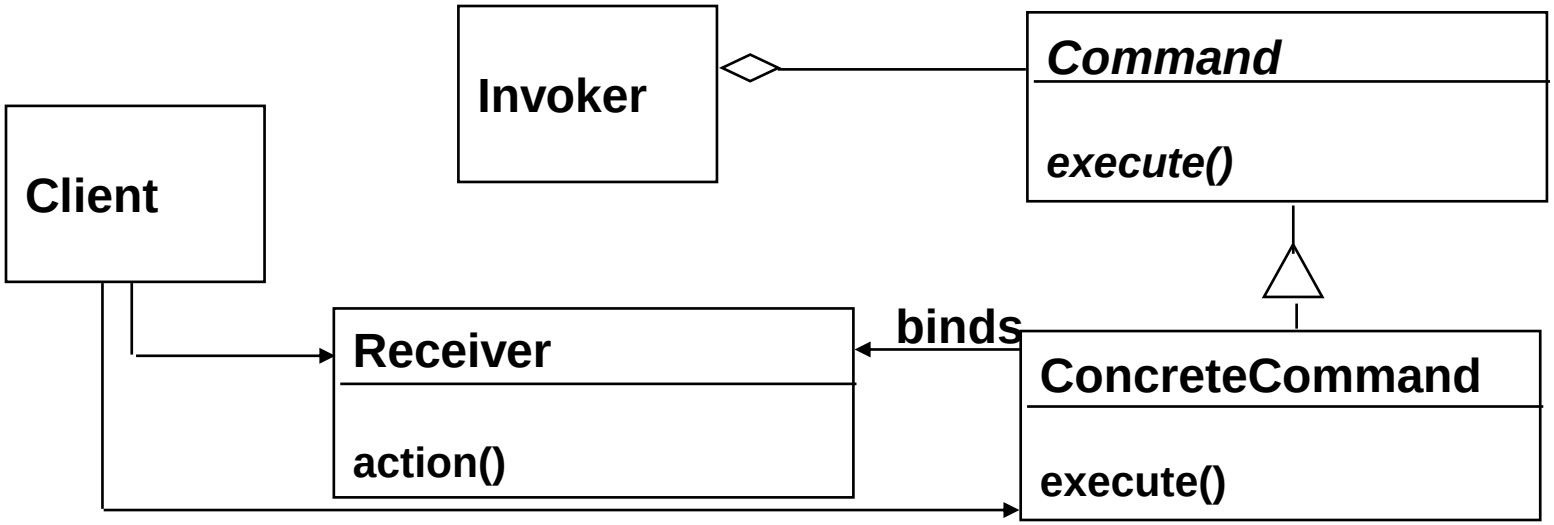
Command Pattern

- You want to build a user interface
- You want to provide menus
- You want to make the user interface reusable across many applications
  - You cannot hardcode the meanings of the menus for the various applications
  - The applications only know what has to be done when a menu is selected.
- Such a menu can easily be implemented with the Command Pattern

Command pattern

- **Client** creates a **ConcreteCommand** and binds it with a **Receiver.**
- **Client** hands the **ConcreteCommand** over to the **Invoker** which stores it.
- The **Invoker** has the responsibility to do the command ("execute" or "undo")**.**
  - See next slide

Command Pattern



Command pattern Applicability

- “Encapsulate a request as an object, thereby letting you
  - parameterize clients with different requests,
  - queue or log requests, and
  - support undoable operations.”

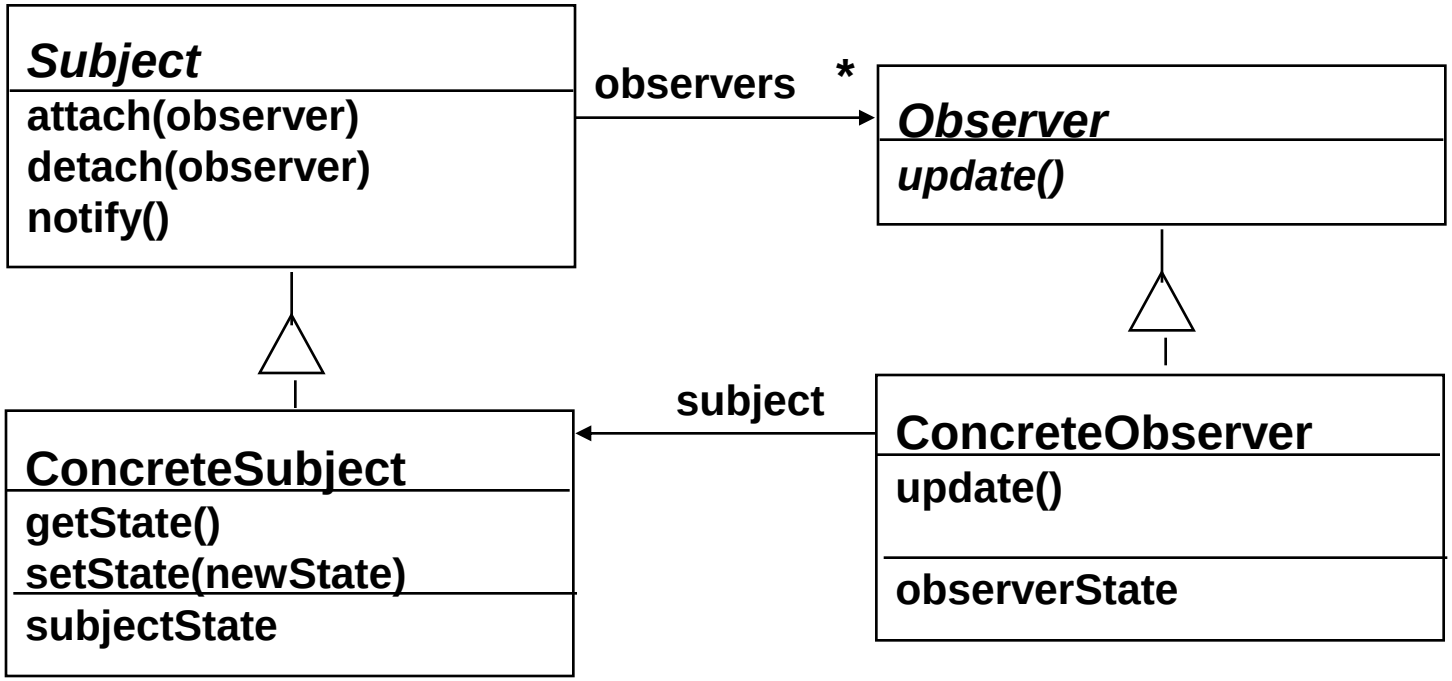
- Uses:
  - Undo queues
  - Database transaction buffering

Observer pattern

- “Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.”
- Also called “Publish and Subscribe”
- Uses:
  - Maintaining consistency across redundant state
  - Optimizing batch changes to maintain consistency

Observer pattern

- The **Subject** represents the actual state, the **Observers** represent different views of the state.
- **Observer** can be implemented as a Java interface.
- **Subject** is a super class (needs to store the observers vector) ***not*** an interface.

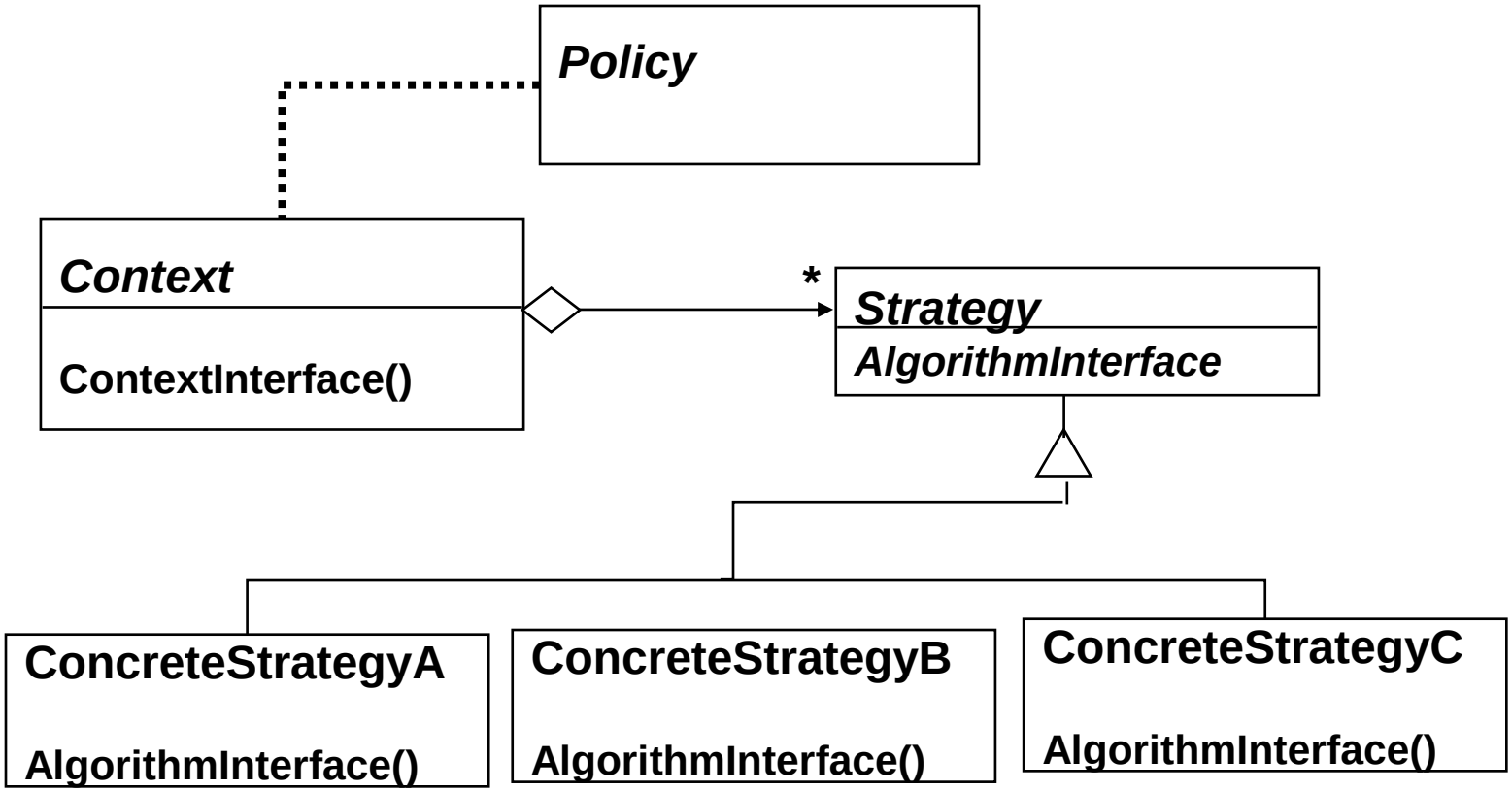


Strategy Pattern

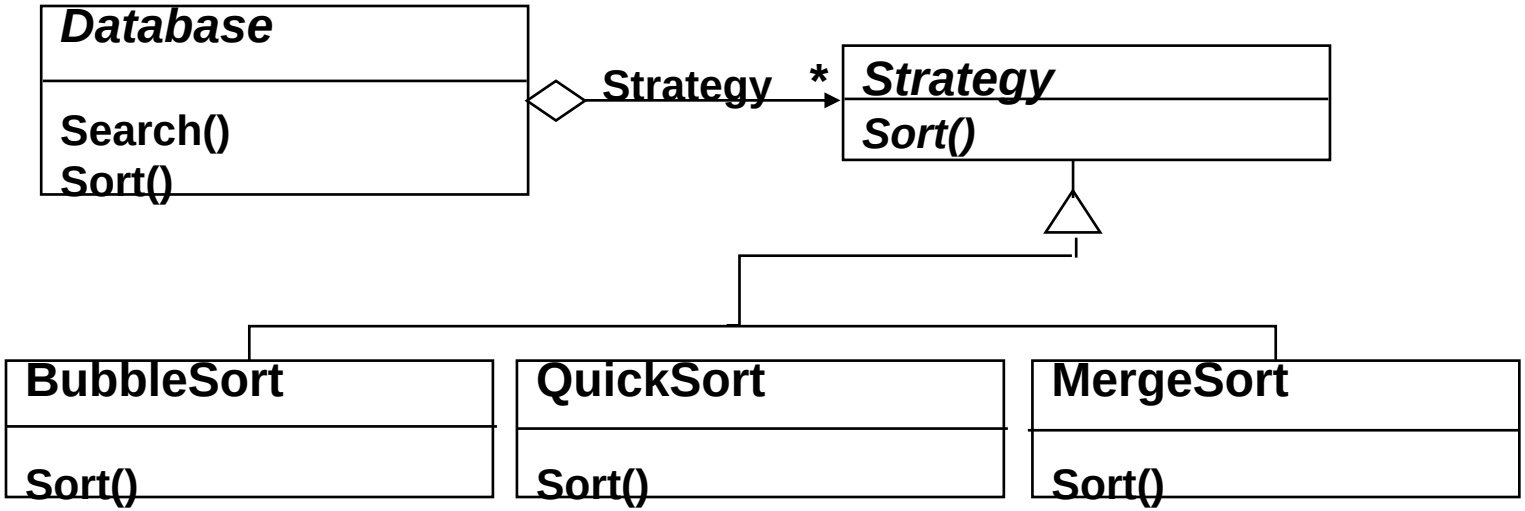
- Many different algorithms exists for the same task
- Examples:
  - Breaking a stream of text into lines
  - Parsing a set of tokens into an abstract syntax tree
  - Sorting a list of customers
- The different algorithms will be appropriate at different times
  - Rapid prototyping vs delivery of final product
- We don’t want to support all the algorithms if we don’t need them
- Need a new algorithm? We want to add it easily without disturbing the application using the algorithm

Strategy Pattern

Policy decides which Strategy is best given the current Context



Example: A Database Application



Applicability of Strategy Pattern

- Many related classes differ only in their behavior. Strategy allows us to configure a single class with one of many behaviors

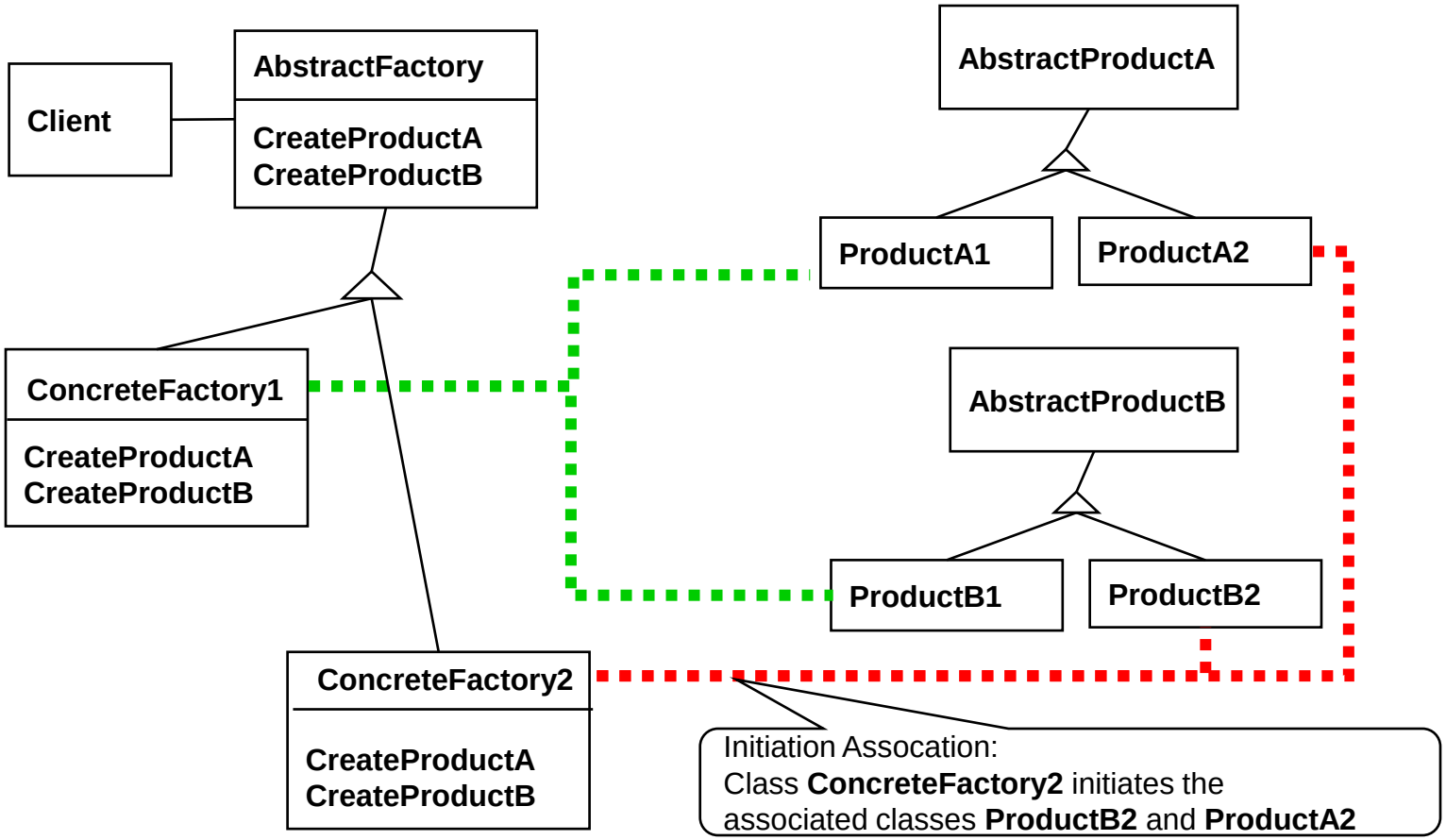
- Different variants of an algorithm are needed that trade-off space against time. All these variants can be implemented as a class hierarchy of algorithms

Abstract Factory Motivation

- 2 Examples
- Consider a user interface toolkit that supports multiple looks and feel standards such as Motif, Windows 95 or the Finder in MacOS.
  - How can you write a single user interface and make it portable across the different look and feel standards for these window managers?

- Consider a facility management system for an intelligent house that supports different control systems such as Siemens' Instabus, Johnson & Control Metasys or Zumtobe's proprietary standard.
  - How can you write a single control system that is independent from the manufacturer?

Abstract Factory



Applicability

- Independence from Initialization or Represenation:
  - The system should be independent of how its products are created, composed or represented
- Manufacturer Independence:
  - A system should be configured with one family of products, where one has a choice from many different families.
  - You want to provide a class library for a customer (“facility management library”), but you don’t want to reveal what particular product you are using.
- Constraints on related products
  - A family of related products is designed to be used together and you need to enforce this constraint
- Cope with upcoming change:
  - You use one particular product family, but you expect that the underlying technology is changing very soon, and new products will appear on the market.

Summary (2)

- Structural Patterns
  - Focus: How objects are composed to form larger structures
  - Problems solved:
    - Realize new functionality from old functionality,
    - Provide flexibility and extensibility
- Behavioral Patterns
  - Focus: Algorithms and the assignment of responsibilities to objects
  - Problem solved:
    - Too tight coupling to a particular algorithm

- Creational Patterns
  - Focus: Creation of complex objects
  - Problems solved:
    - Hide how complex objects are created and put together

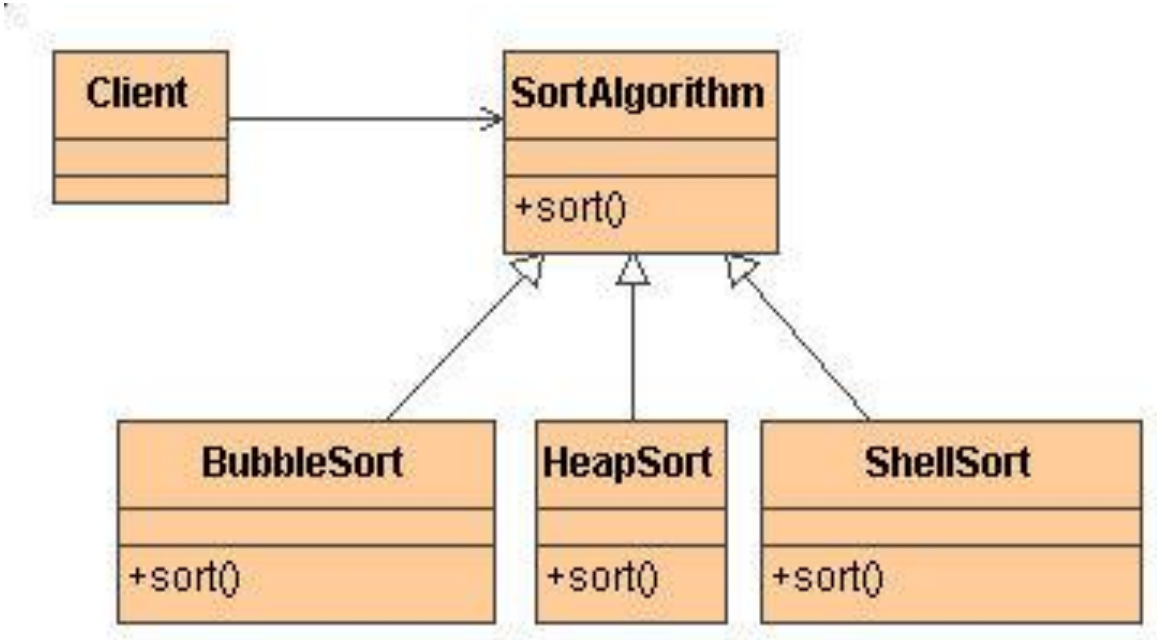
Conclusions

- Design patterns:
- Provide solutions to common problems.
- Lead to extensible models and code.
- Can be used as is or as examples of interface inheritance and delegation.
- Apply the same principles to structure and to behavior.

Design Class Diagrams

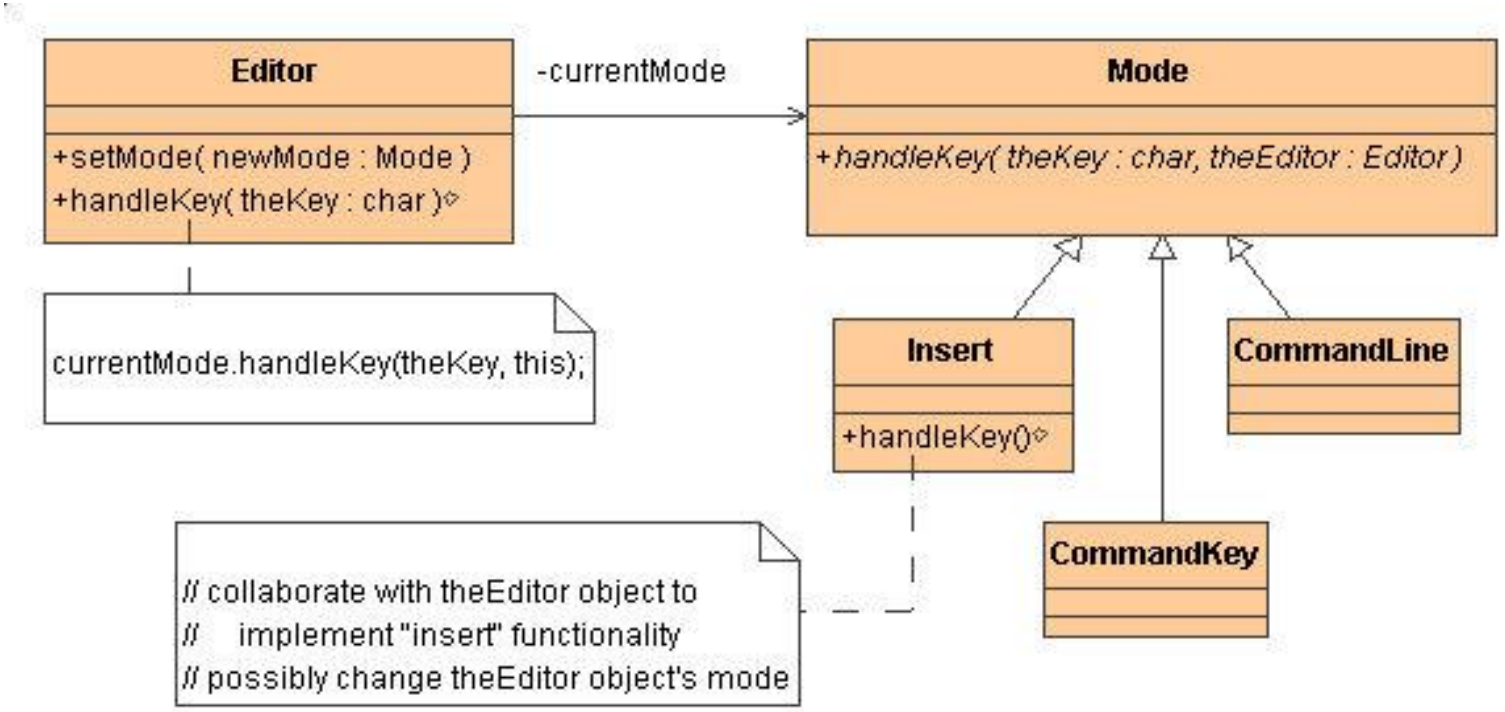
- **Strategy**
  - "Strategy, State, Bridge (and to some degree Adapter) have similar solution structures. They all share elements of the "handle/body" idiom. They differ in intent - that is, they solve different problems."
  - Most of the GoF patterns exercise the two levels of indirection demonstrated here.
    1. Promote the "interface" of a method to an abstract base class or interface, and bury the many possible implementation choices in concrete derived classes.

2. Hide the implementation hierarchy behind a "wrapper" class that can perform responsibilities like: choosing the best implementation, caching, state management, remote access.



State

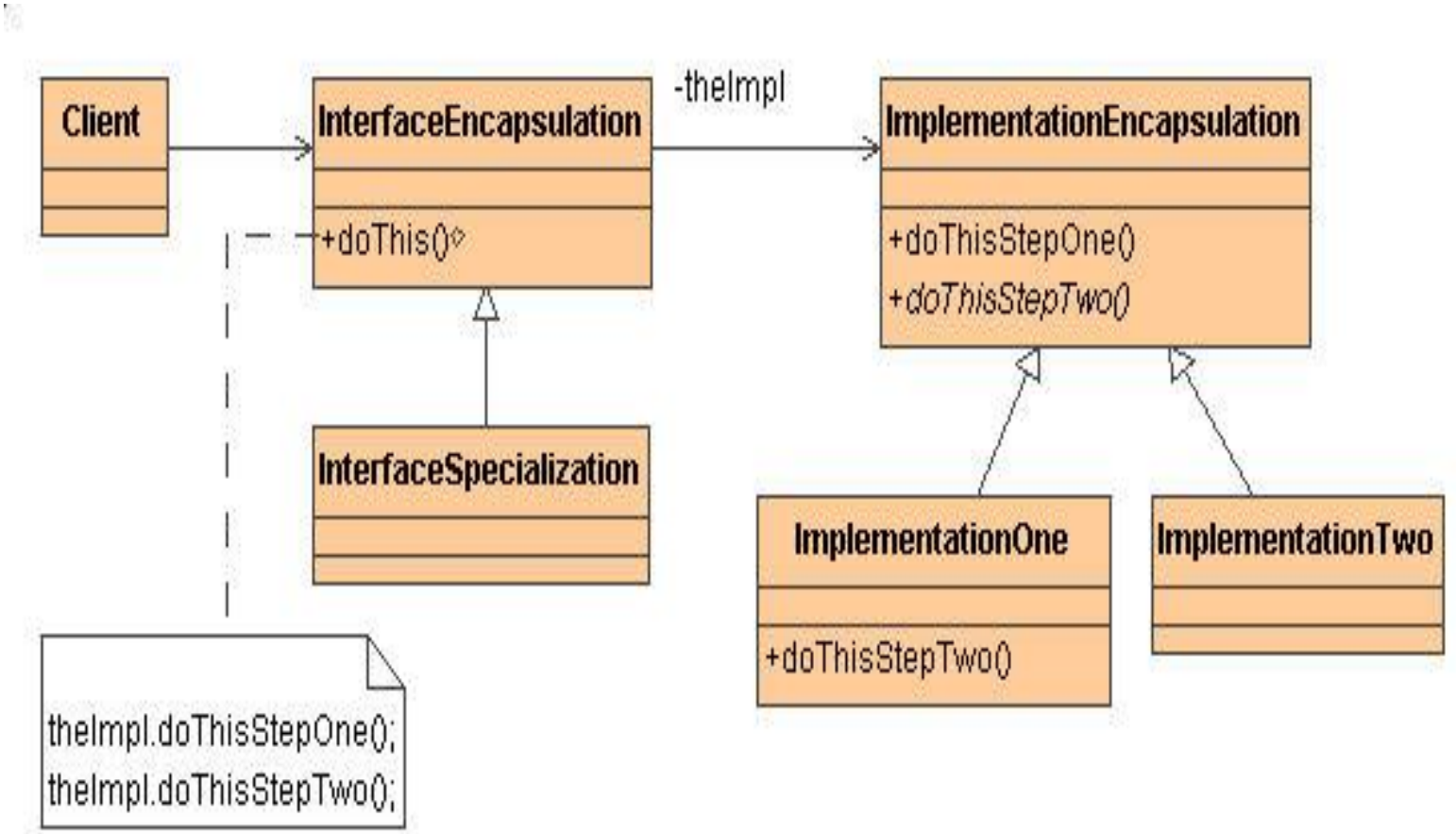
- Whereas Strategy is a bind-once pattern, State is more dynamic.



Bridge

- The structure of State and Bridge are identical (except that Bridge admits hierarchies of envelope classes, whereas State allows only one).
  - The two patterns use the same structure to solve different problems:
    - State allows an object's behavior to change along with its state, while Bridge's intent is to decouple an abstraction from its implementation so that the two can vary independently.

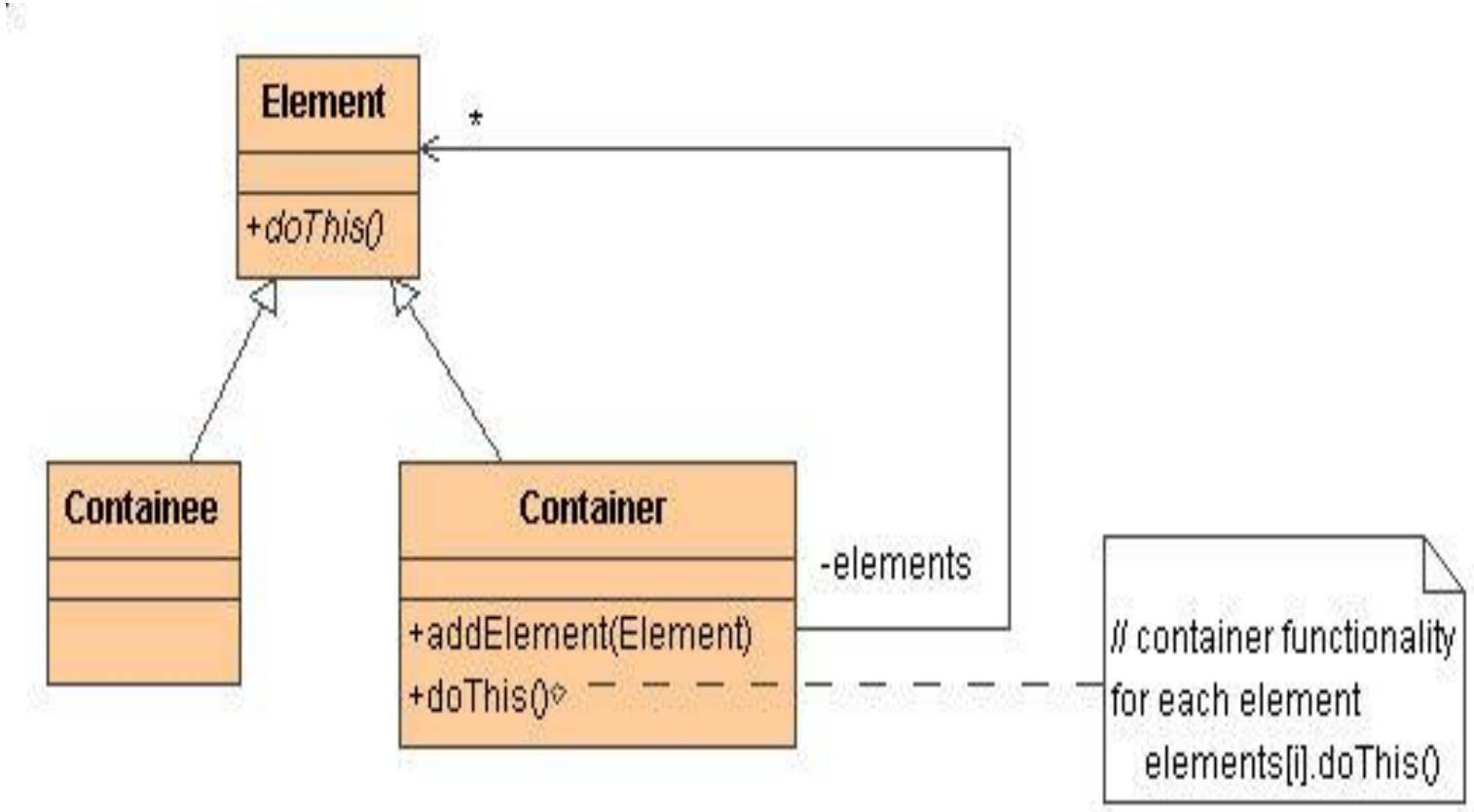
Bridge Structure



Composite

- Three GoF patterns rely on recursive composition:
  - Composite
  - Decorator and
  - Chain of Responsibility.

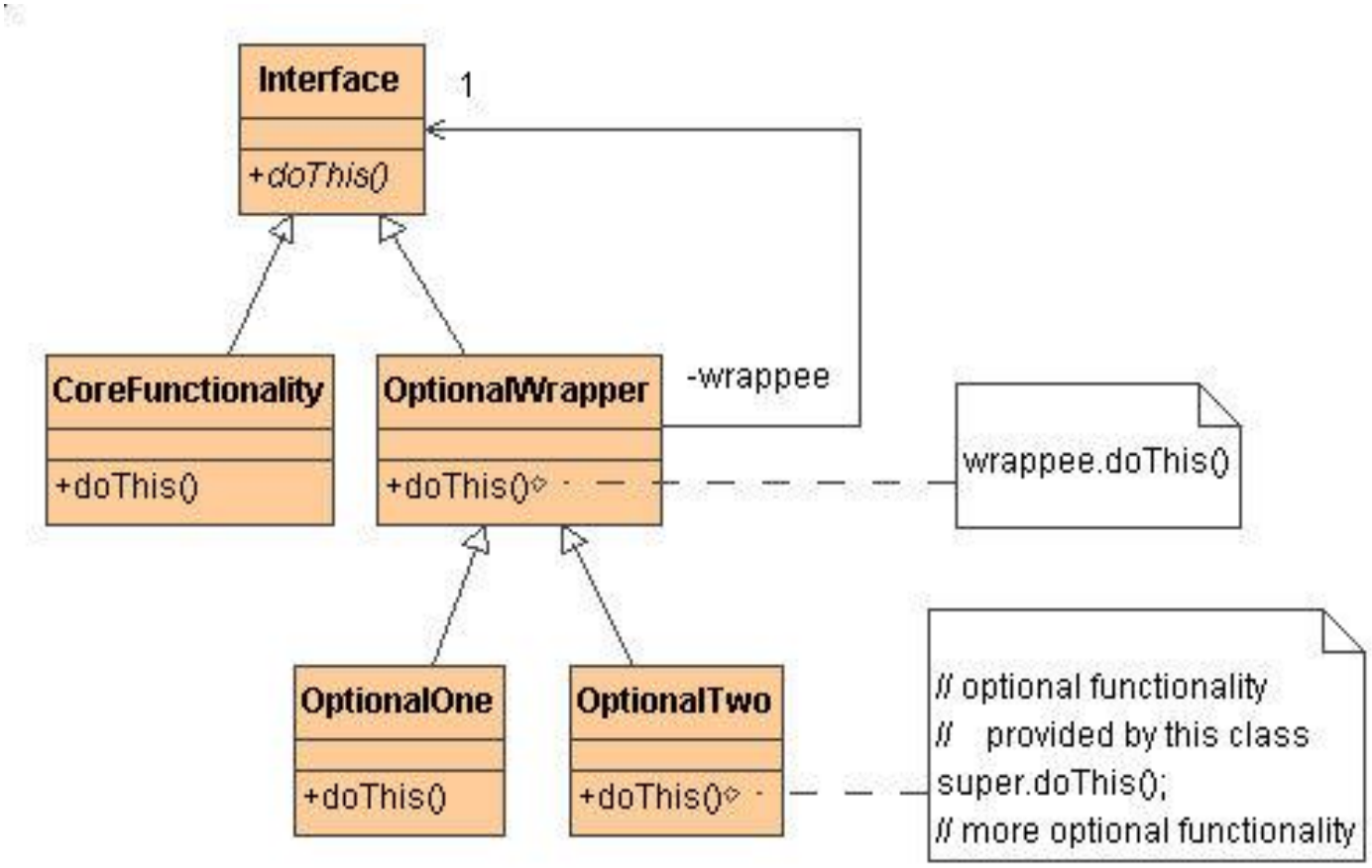
Composite Structure



Decorator

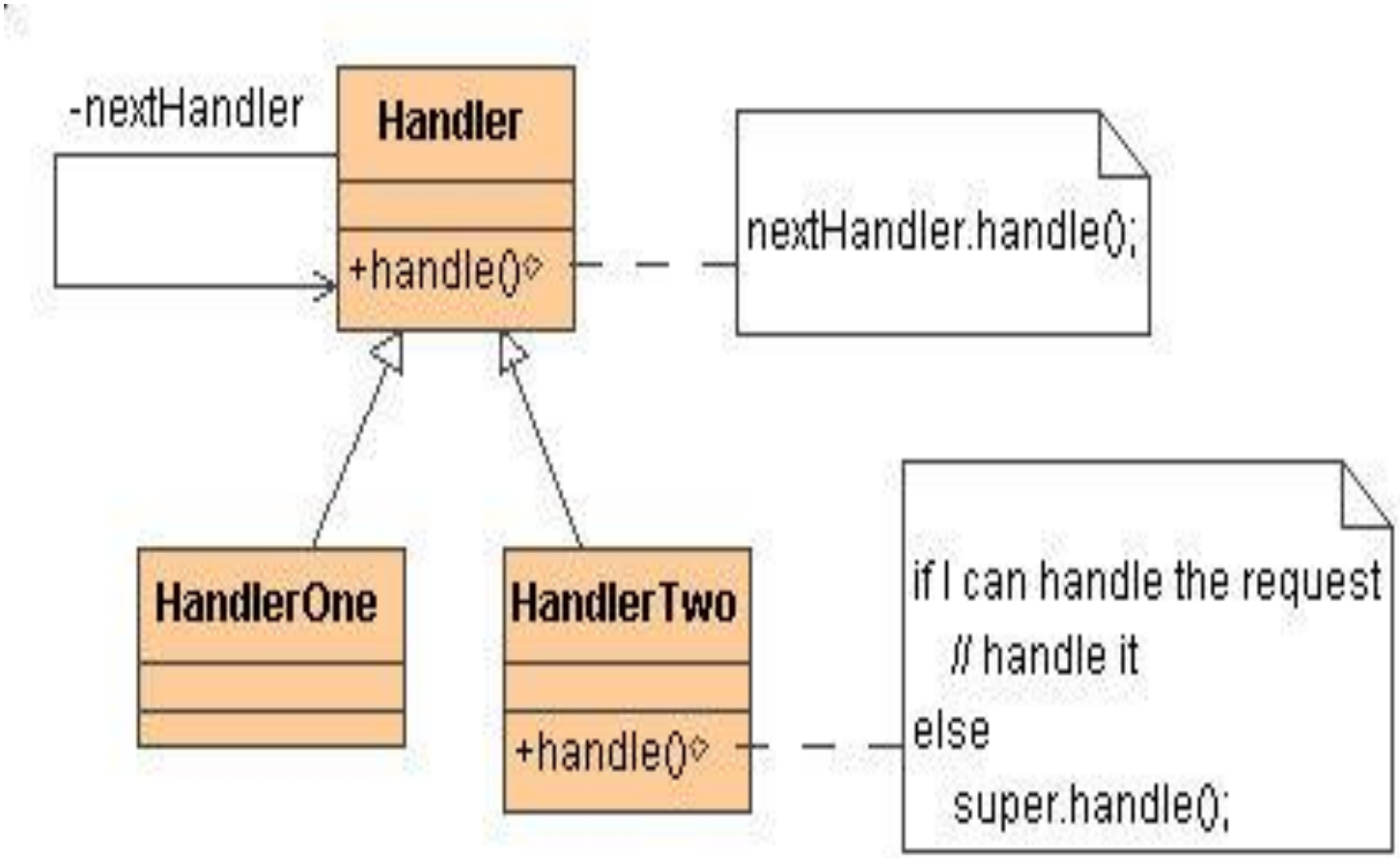
- Decorator is designed to let you add responsibilities to objects without subclassing.
- Composite's focus is not on embellishment but on representation.
  - These intents are distinct but complementary. Consequently, Composite and Decorator are often used in concert.

Decorator


Interface
1
+doThis()
CoreFunctionality
+doThis()
OptionalWrapper
+doThis()
-wrappee
wrappee.doThis()
OptionalOne
+doThis()
OptionalTwo
+doThis()
// optional functionality
// provided by this class
super.doThis();
// more optional functionality

Chain of Responsibility

- Chain of Responsibility, Command, Mediator, and Observer, address how you can decouple senders and receivers, but with different trade-offs.
- Chain of Responsibility passes a sender request along a chain of potential receivers.


-nextHandler
Handler
+handle()
nextHandler.handle();
HandlerOne
HandlerTwo
+handle()
if I can handle the request
// handle it
else
super.handle();

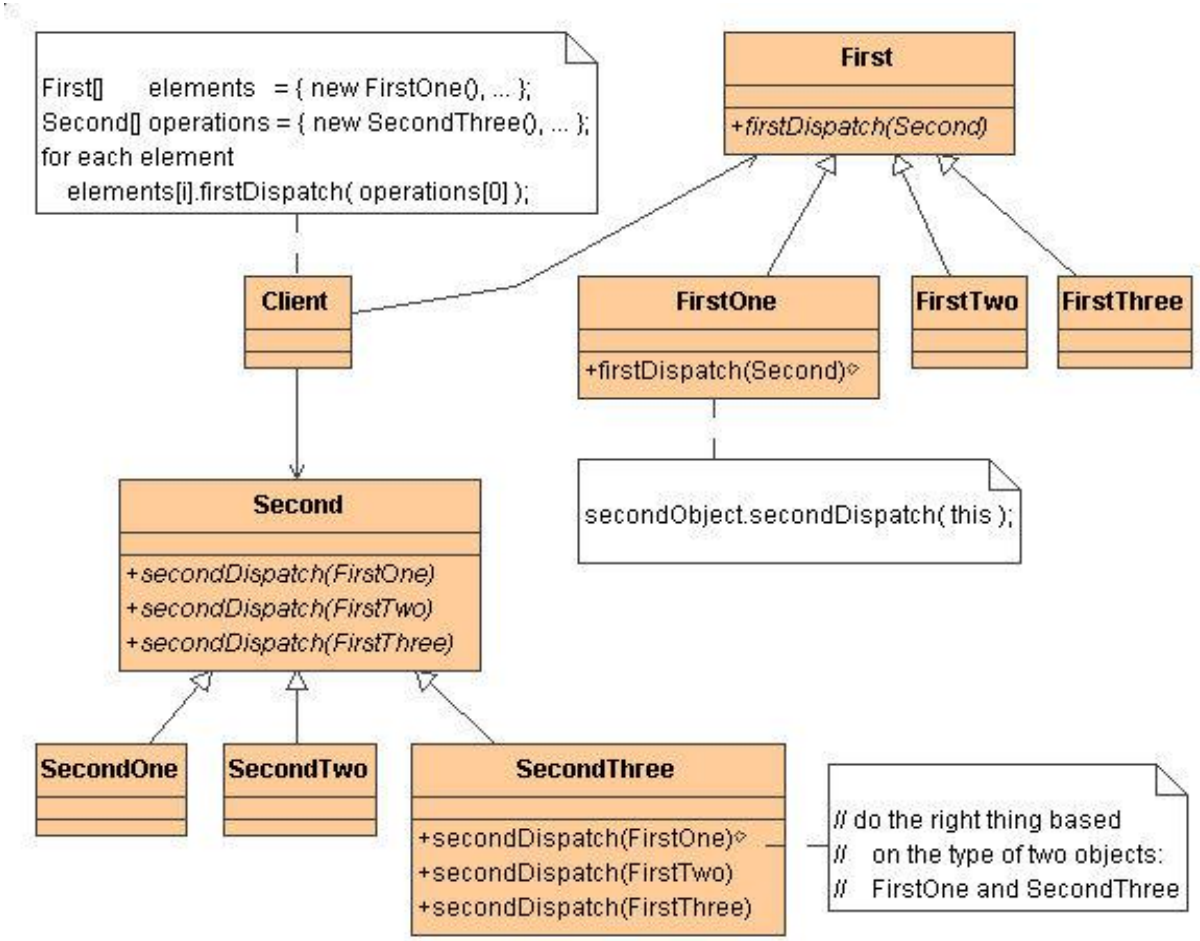
Façade

- Facade defines a new interface, whereas Adapter uses an old interface.
  - Remember that Adapter makes two existing interfaces work together as opposed to defining an entirely new one.

Visitor

- The Visitor pattern is the classic technique for recovering lost type information without resorting to dynamic casts.

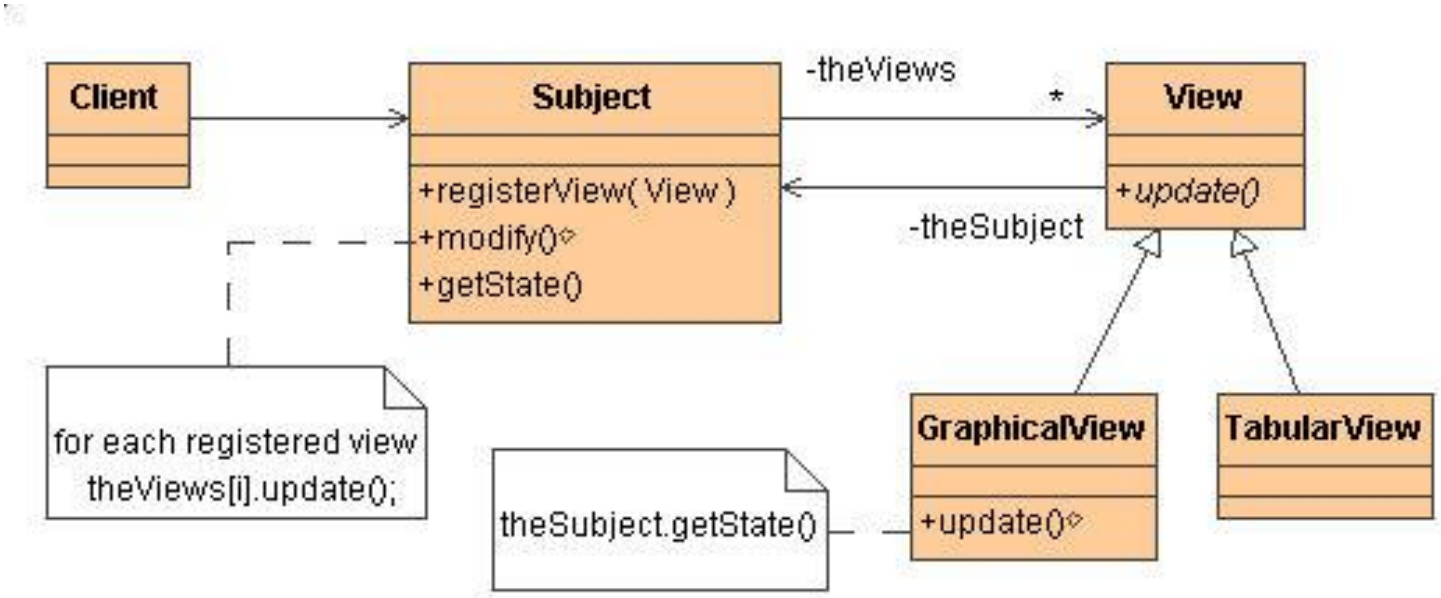
Visitor


First[] elements = { new FirstOne(), ... };
Second[] operations = { new SecondThree(), ... };
for each element
elements[i].firstDispatch(operations[0]);
First
+firstDispatch(Second)
Client
FirstOne
+firstDispatch(Second)
FirstTwo
FirstThree
secondObject.secondDispatch(this);
Second
+secondDispatch(FirstOne)
+secondDispatch(FirstTwo)
+secondDispatch(FirstThree)
SecondOne
SecondTwo
SecondThree
+secondDispatch(FirstOne)
+secondDispatch(FirstTwo)
+secondDispatch(FirstThree)
// do the right thing based
// on the type of two objects:
// FirstOne and SecondThree

Observer

- Mediator and Observer are competing patterns. The difference between them is that Observer distributes communication by introducing "observer" and "subject" objects, whereas a Mediator object encapsulates the communication between other objects.
- It’s been found easier to make reusable Observers and Subjects than to make reusable Mediators.

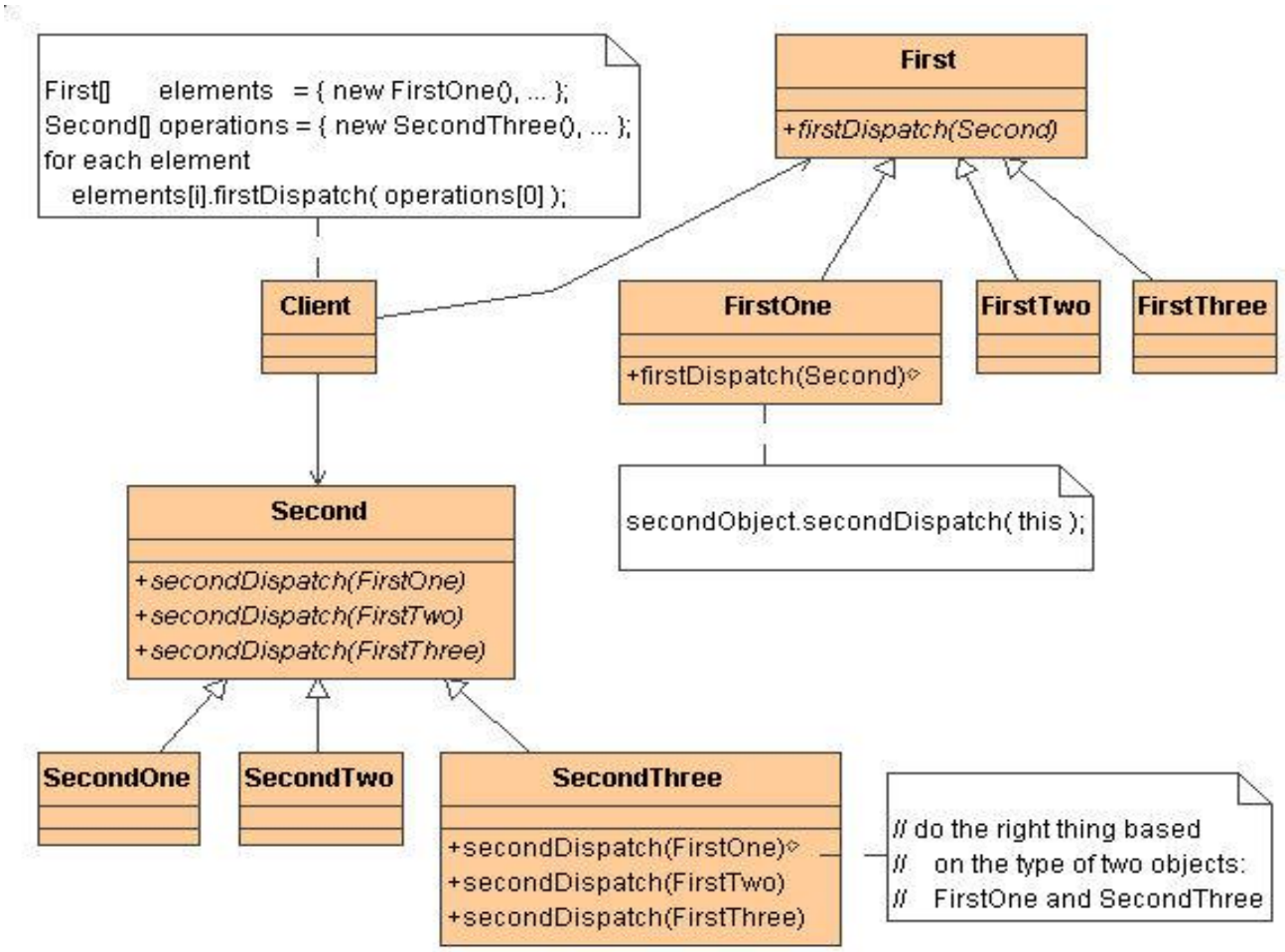
- On the other hand, Mediator can leverage Observer for dynamically registering colleagues and communicating with them.



Visitor

- The Visitor pattern is the classic technique for recovering lost type information without resorting to dynamic casts.

Visitor


First[] elements = { new FirstOne(), ... };
Second[] operations = { new SecondThree(), ... };
for each element
elements[i].firstDispatch(operations[0]);
First
+firstDispatch(Second)
Client
FirstOne
+firstDispatch(Second)
FirstTwo
FirstThree
secondObject.secondDispatch(this);
Second
+secondDispatch(FirstOne)
+secondDispatch(FirstTwo)
+secondDispatch(FirstThree)
SecondOne
SecondTwo
SecondThree
+secondDispatch(FirstOne)
+secondDispatch(FirstTwo)
+secondDispatch(FirstThree)
// do the right thing based
// on the type of two objects:
// FirstOne and SecondThree

Singleton

- Singleton should be considered only if all three of the following criteria are satisfied:
  - Ownership of the single instance cannot be reasonably assigned
  - Lazy initialization is desirable
  - Global access is not otherwise provided for