Modern Methods in Software Engineering

Object Design

Literature used

- Text book

  Chapters 8, 9

- Recommended literature:

  E. Gamma et al. Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995

  S. Holzner. Design Patterns for Dummies, Wiley Publishing, 2006

Introduction Content

- Object design activities
- Re-use
- Design patterns

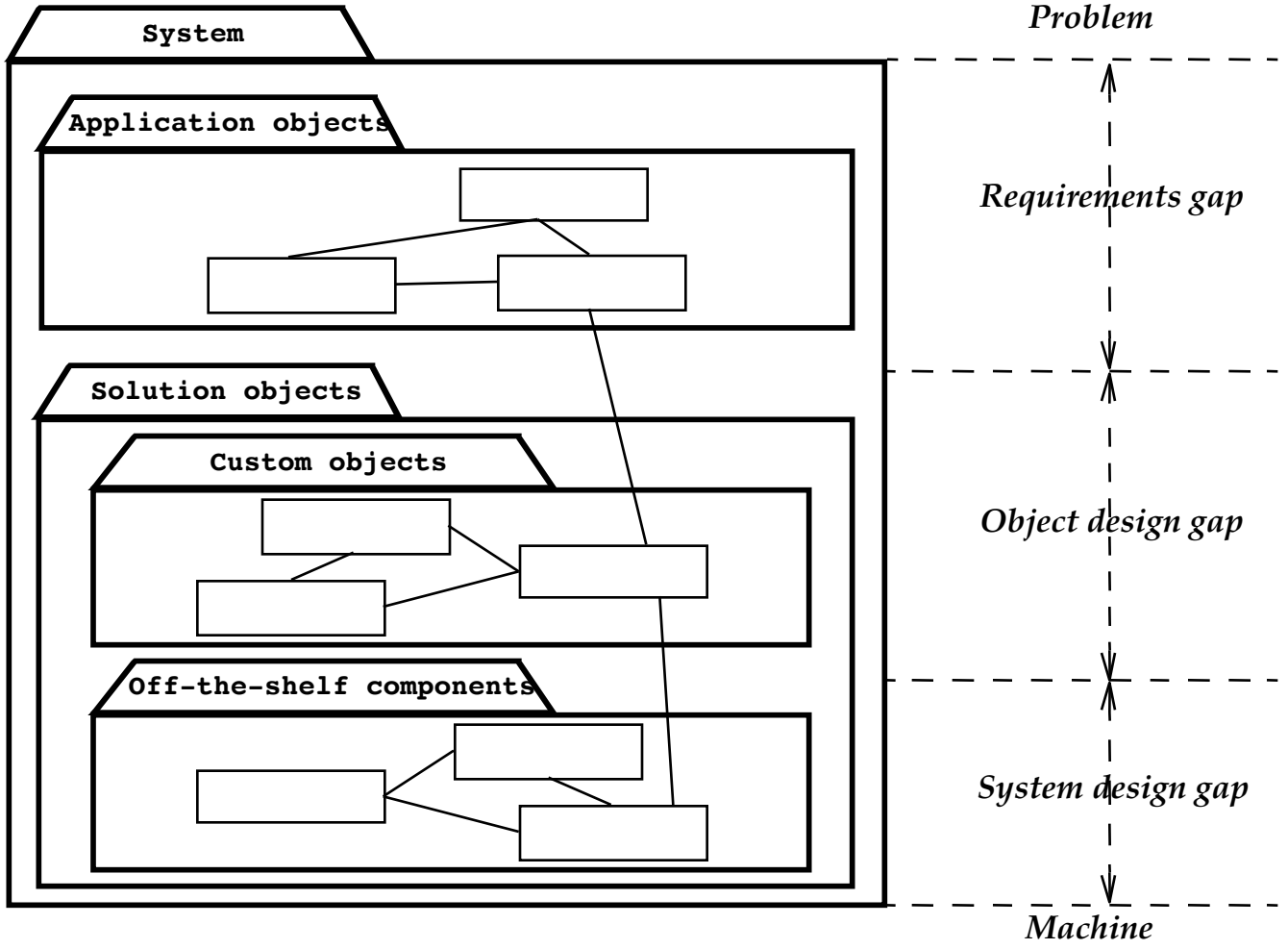
Next lectures:

- interface design
- moving to code

Object Design

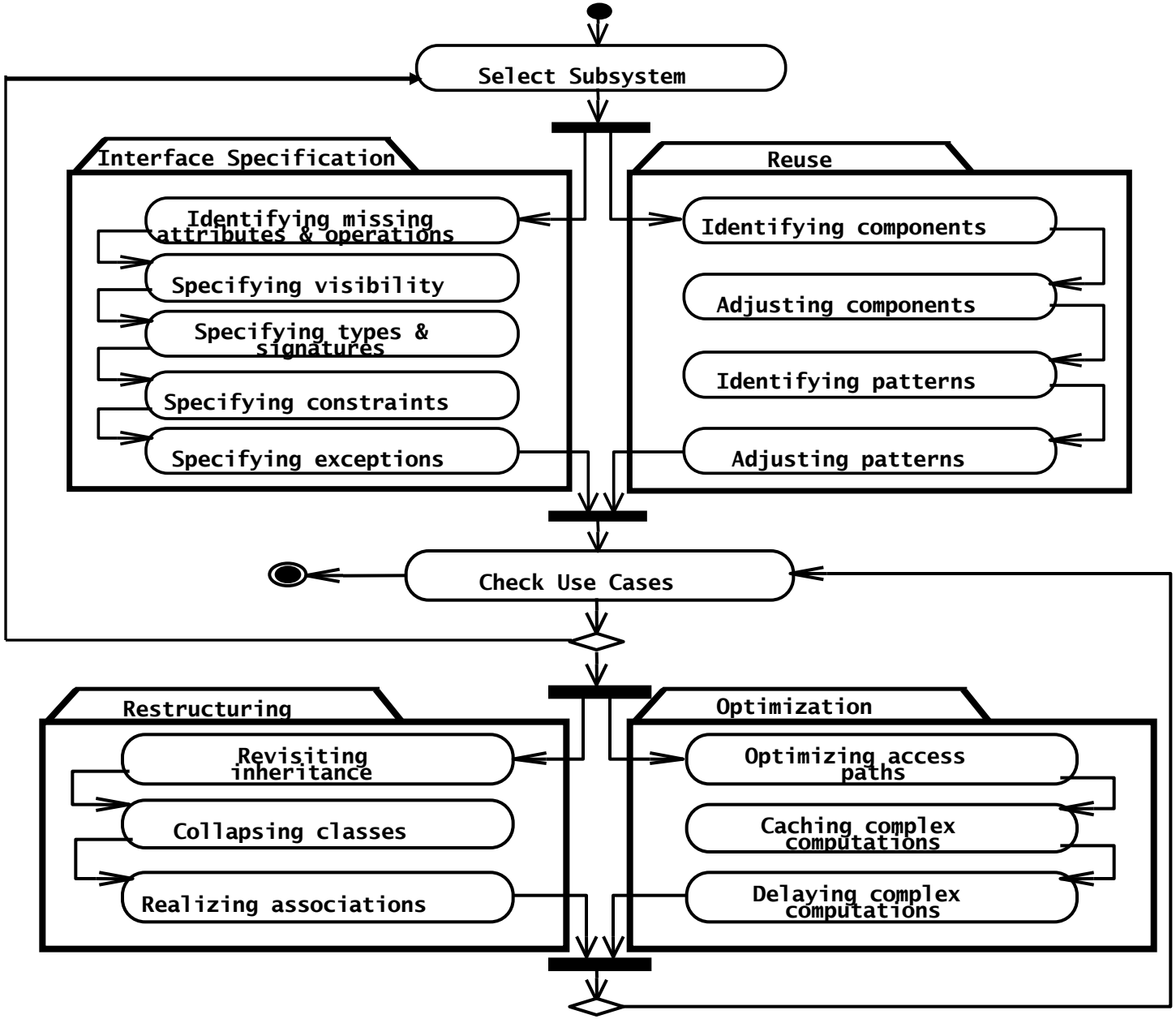
- Object design is the process of adding details to the requirements analysis and making implementation decisions
- The object designer must choose among different ways to implement the analysis model with the goal to satisfy design goals.

Object Design: Closing the Gap



Adopted from Bernd Bruegge & Allen H. Dutoit Object-Oriented Software Engineering: Using UML, Patterns, and Java

Object Design Activities



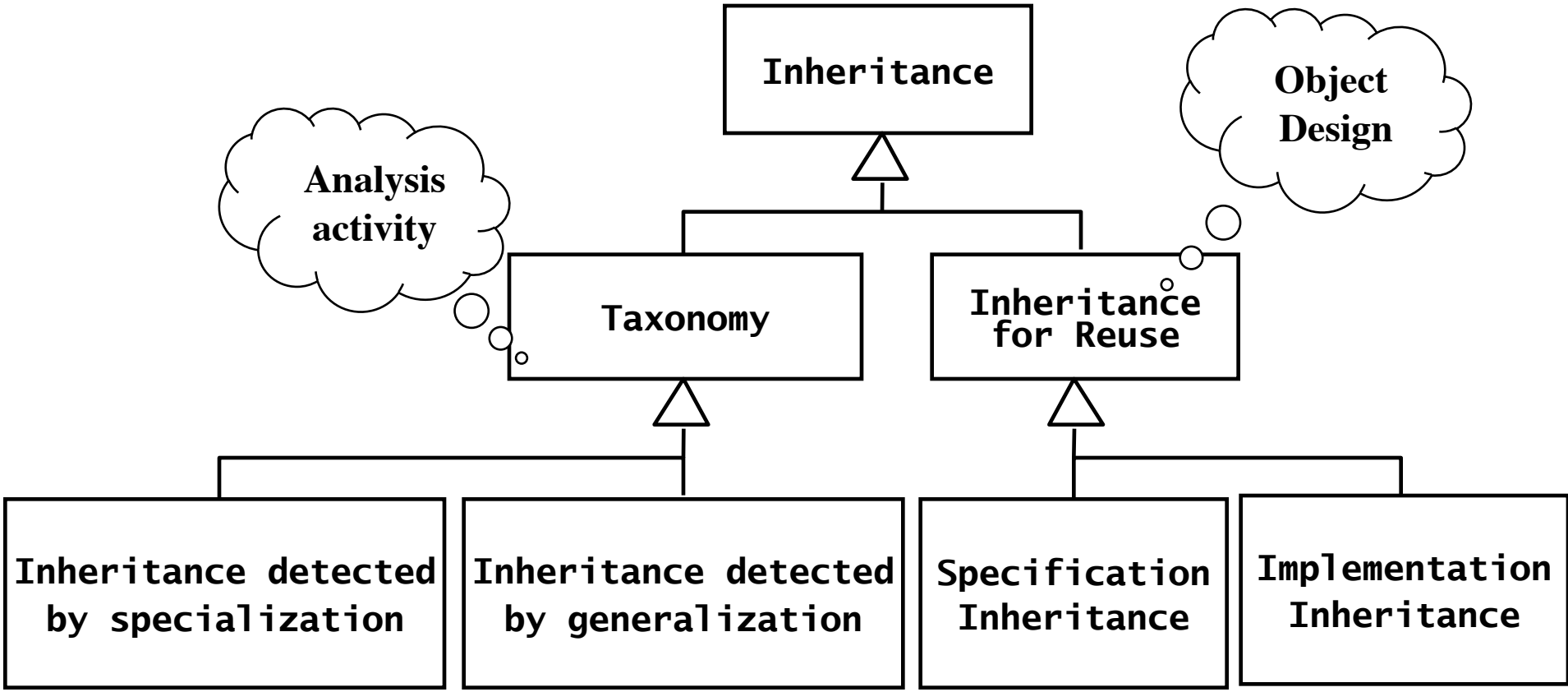
Reuse Concepts

- Specification inheritance and implementation inheritance
- Delegation
- The Liskov Substitution Principle
- Delegation and inheritance in design patterns

Inheritance

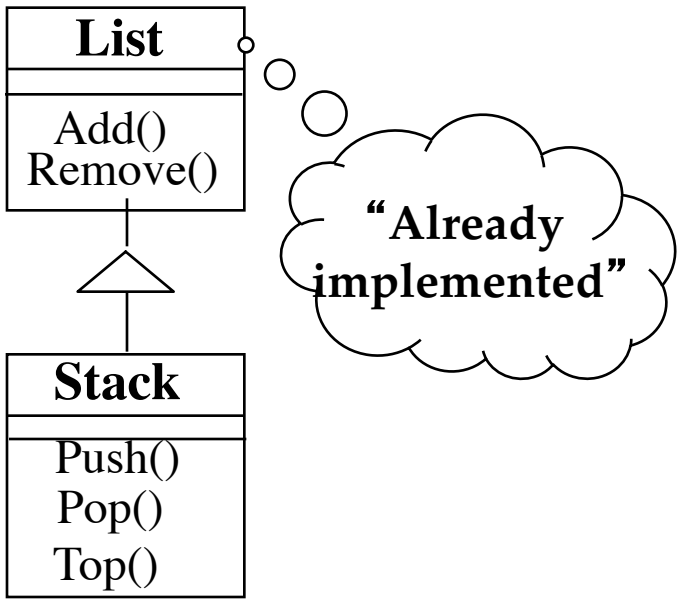
- Inheritance is used to achieve two different goals
  - Description of Taxonomies
  - Interface Specification
- Identification of taxonomies
  - Used during requirements analysis.
  - Activity: identify application domain objects that are hierarchically related
  - Goal: make the analysis model more understandable
- Interface specification
  - Used during object design
  - Activity: identify interfaces and type hierarchies
  - Goal: increase reusability, enhance modifiability and extensibility

Metamodel for Inheritance



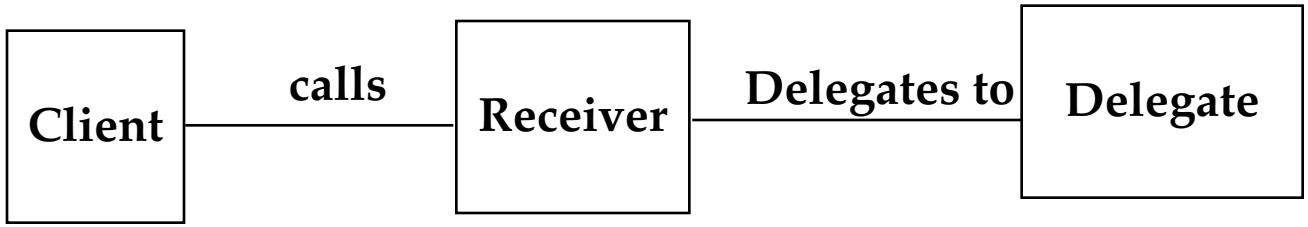
Implementation Inheritance

- A very similar class is already implemented that does almost the same as the desired class implementation.
  - Example: I have a **List** class, I need a **Stack** class. How about subclassing the **Stack** class from the **List** class and providing three methods, **Push()** and **Pop(), Top()**?



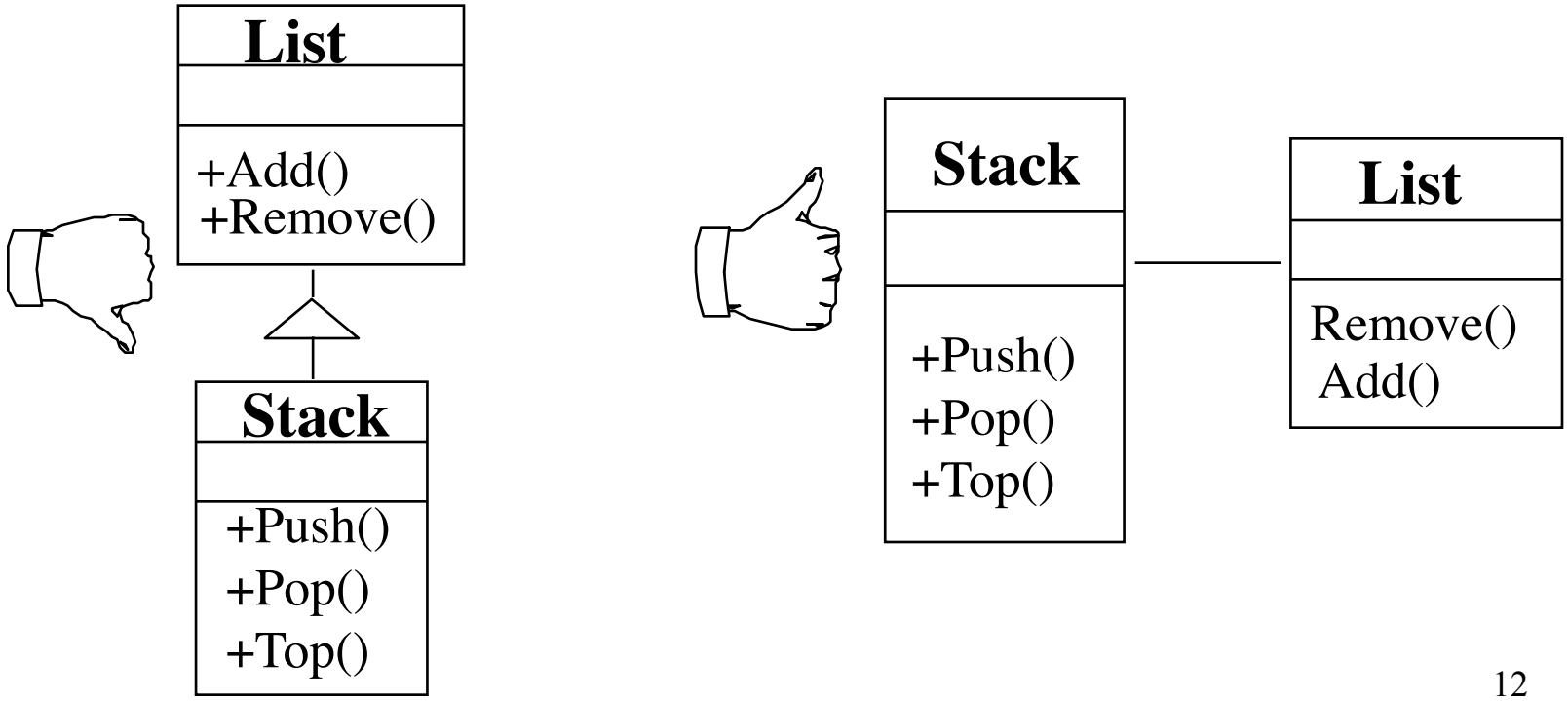
Delegation as alternative to Implementation Inheritance

- Delegation is a way of making composition (for example aggregation) as powerful for reuse as inheritance
- A class is said to delegate to another class if it implements an operation by resending a message to another class
- In Delegation two objects are involved in handling a request
  - A receiving object delegates operations to its delegate.
  - The developer can make sure that the receiving object does not allow the client to misuse the delegate object

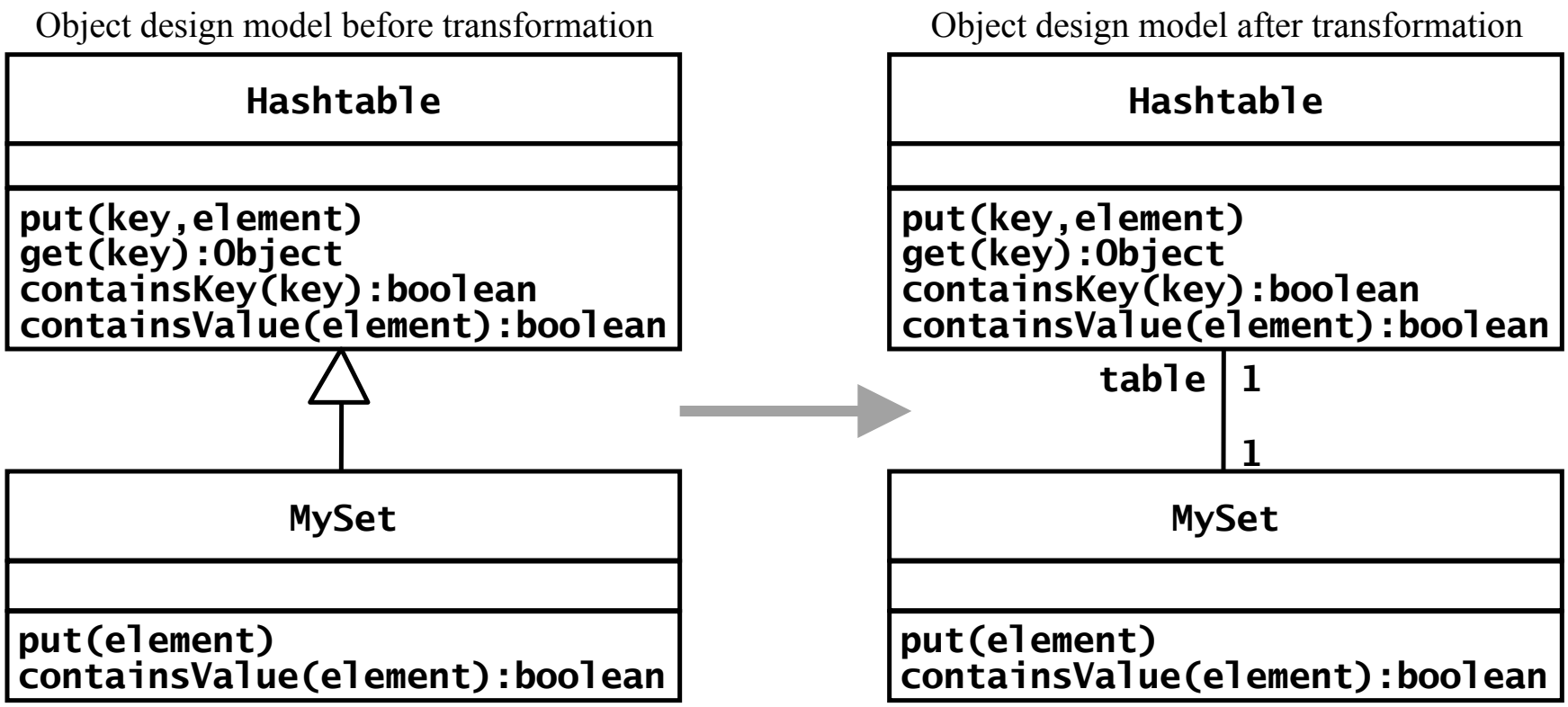


Delegation instead of Implementation Inheritance

- **Inheritance**: Extending a Base class by a new operation or overwriting an operation.
- **Delegation**: Catching an operation and sending it to another object.
- Which of the following models is better for implementing a stack?



An example of implementation inheritance



An example of implementation inheritance

```
/* Implementation of MySet using
    inheritance */
class MySet extends Hashtable {
    /* Constructor omitted */
    MySet() {
    }

    void put(Object element) {
        if (!containsKey(element)){
                put(element, this);
        }
    }
    boolean containsValue(Object
                element){
        return containsKey(element);
    }
    /* Other methods omitted */
}
```

```
/* Implementation of MySet using delegation */
class MySet {
    private Hashtable table;
    MySet() {
        table = new Hashtable();
    }
    void put(Object element) {
        if (!containsValue(element)){
                table.put(element,this);
        }
    }
    boolean containsValue(Object
                element) {
        return
        (table.containsKey(element));
    }
    /* Other methods omitted */
}
```

Comparison: Delegation vs Implementation Inheritance

- Delegation
  - Pro:
    - Flexibility: Any object can be replaced at run time by another one (as long as it has the same type)
  - Con:
    - Inefficiency: Objects are encapsulated.
- Inheritance
  - Pro:
    - Straightforward to use
    - Supported by many programming languages
    - Easy to implement new functionality
  - Con:
    - Inheritance exposes to a subclass the details of its parent class
    - Any change in the parent class implementation forces the subclass to change (which requires recompilation of both)

Implementation Inheritance vs Interface Inheritance

- Implementation inheritance
  - Also called class inheritance
  - Goal: Extend an applications' functionality by reusing functionality in parent class
  - Inherit from an existing class with some or all operations already implemented

- Interface inheritance (specification inheritance)
  - Also called subtyping
  - Inherit from an abstract class with all operations specified, but not yet implemented

The Liskov Substitution Principle

- In class hierarchies, it should be possible to treat a specialized object as if it were a base class object.
- If an object of type S can be substituted in all the places where an object of type T is expected, then S is a subtype of T.
- An inheritance relationship that complies with the Liskov Substitution Principle is called *strict inheritance*.

Inheritance

- Delegation is a preferable mechanism to implementation inheritance as it does not interfere with existing components and leads to more robust code.
- Specification inheritance is preferable to delegation in subtyping situations as it leads to a more extensible design.

A Game: Get-15

- Start with the nine numbers 1,2,3,4, 5, 6, 7, 8 and 9.
- You and your opponent take alternate turns, each taking a number
- Each number can be taken only once: If your opponent has selected a number, you cannot also take it.
- The first person to have any three numbers that total 15 wins the game.
- Example:

You:	**1**	**5**	**3**	**8**	
Opponent:		**9**			**Opponent Wins!**

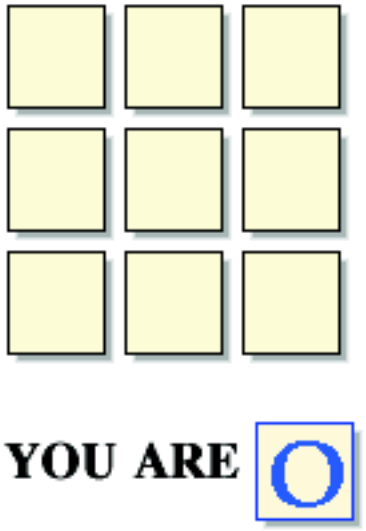
Characteristics of Get-15

- Hard to play
- The game is especially hard, if you are not allowed to write anything done.

- Why?
  - All the numbers need to be scanned to see if you have won/lost
  - It is hard to see what the opponent will take if you take a certain number
  - The choice of the number depends on all the previous numbers

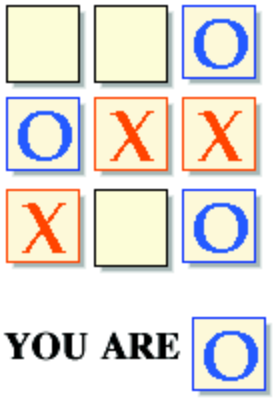
  - Not easy to devise an simple strategy

Another Game: Tic-Tac-Toe

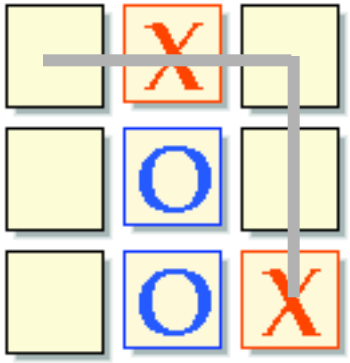


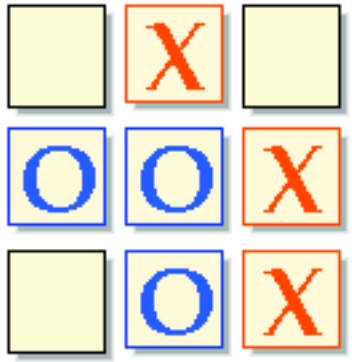
Source: http://boulter.com/ttt/index.cgi

A Draw Situation

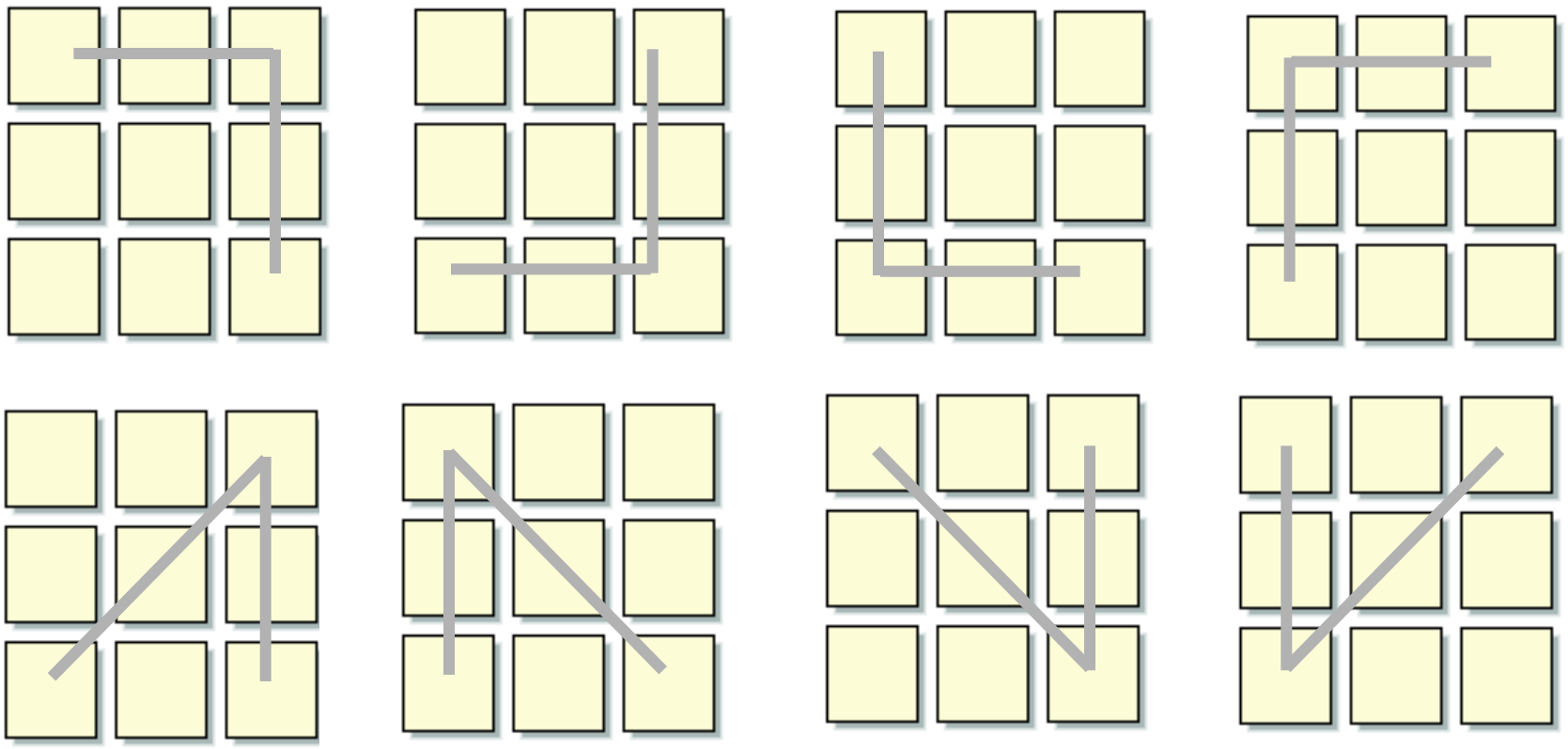


Strategy for determining a winning move



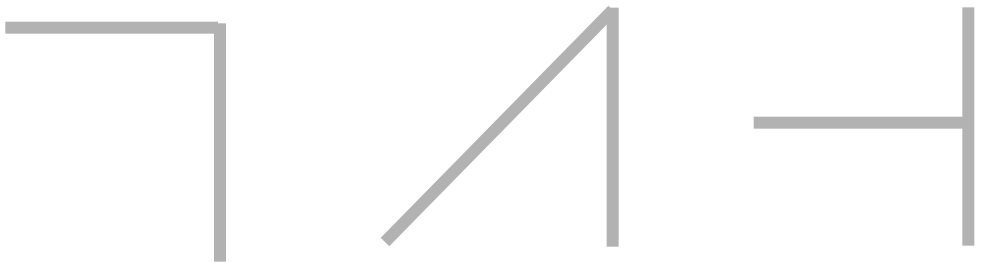


Winning patterns



Tic-Tac-Toe is "Easy"

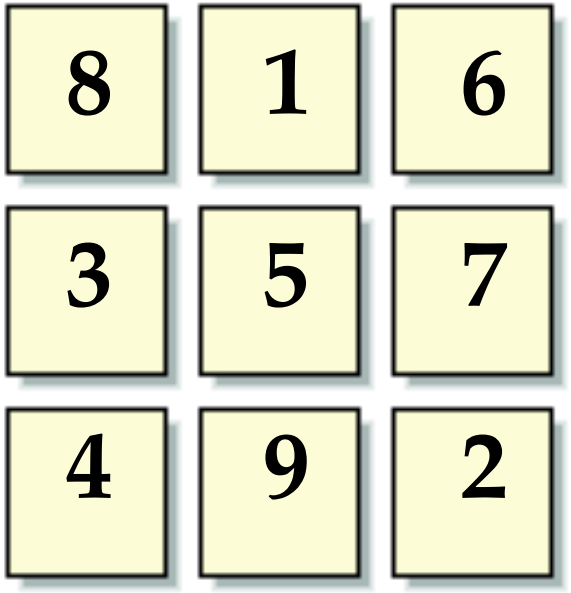
- Why? Reduction of complexity through patterns and symmetries
- **Patterns**: Knowing the following patterns, the player can anticipate the opponents move.

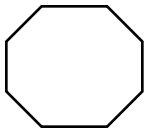


- **Symmetries**:
  - The player needs to remember only these three patterns to deal with 8 different game situations
  - The player needs to memorize only 3 opening moves and their responses

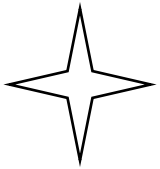
Get-15 and Tic-Tac-Toe are identical problems

- Any three numbers that solve the 15 problem also solve tic-tac-toe.
- Any tic-tac-toe solution is also a solution the 15 problem
- To see the relationship between the two games, we simply arrange the 9 digits into the following pattern



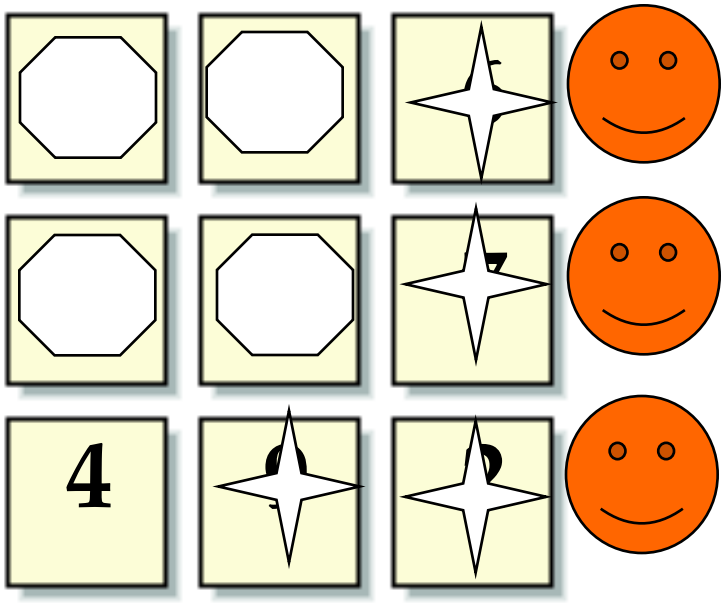


You: 1 5 3 8



Opponent: 6 9 7 2

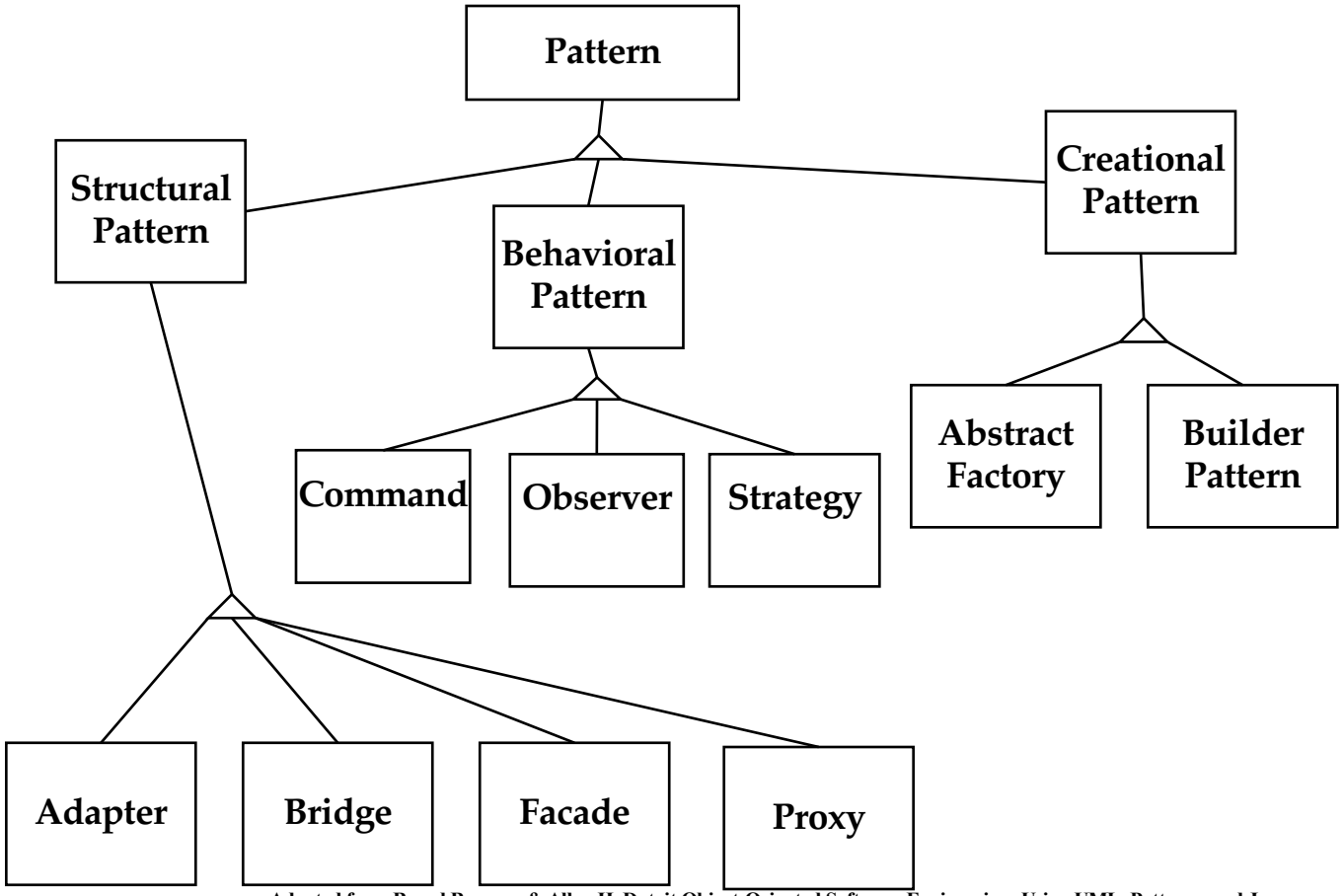
8	1	6
3	5	7
4	9	2



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 this solution a million times over, without ever doing it the same twice

A Pattern Taxonomy



Adopted from Bernd Bruegge & Allen H. Dutoit Object-Oriented Software Engineering: Using UML, Patterns, and Java

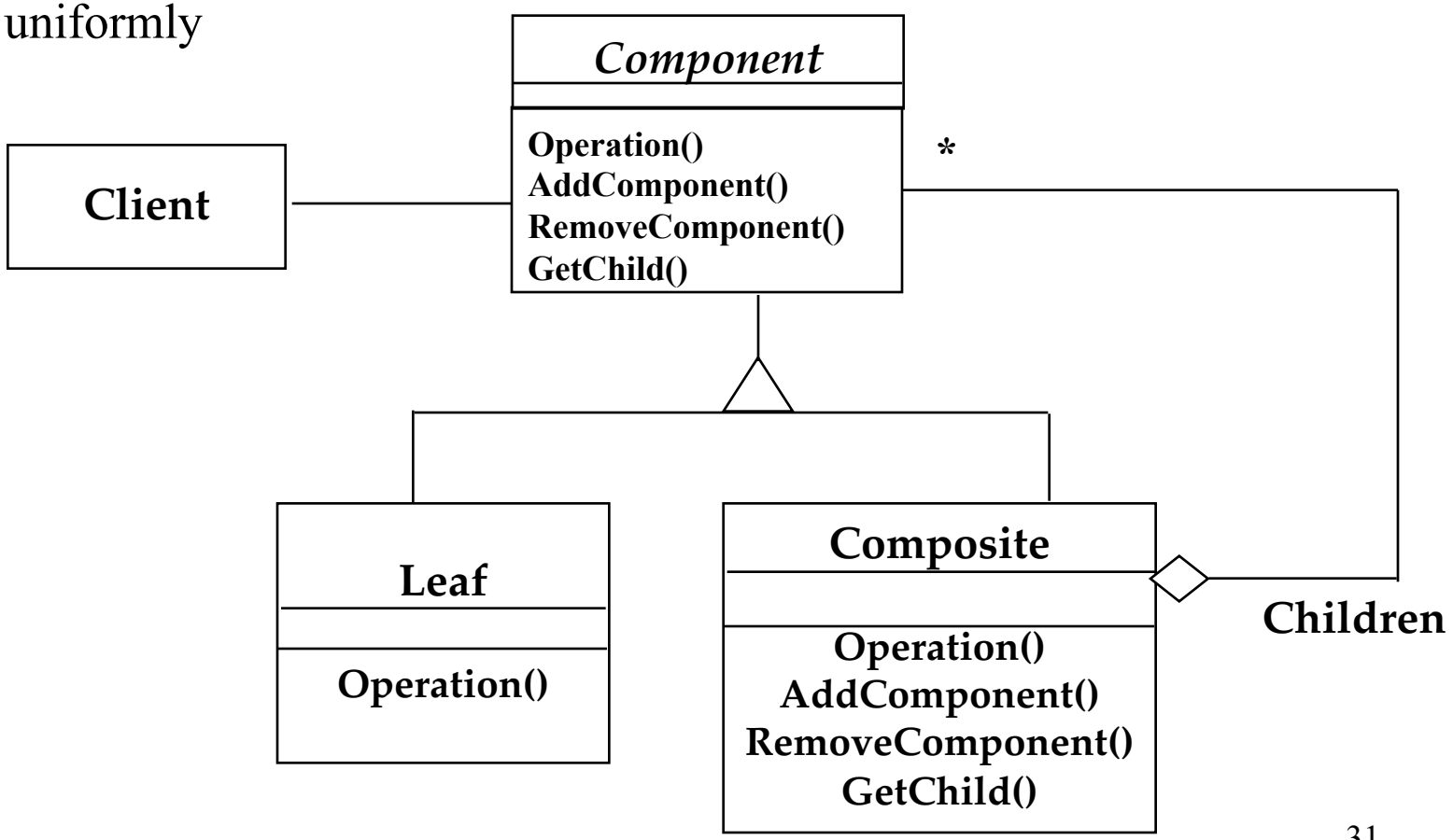
What is common between these definitions?

- 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 activities or collection of tasks

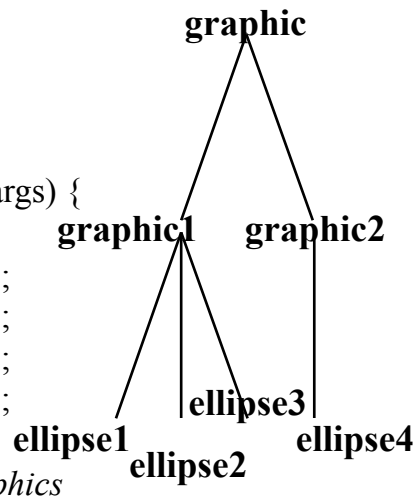
Introducing the Composite Pattern

- 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



```
/** "Component" */
interface Graphic {
    //Prints the graphic.
    public void print();
}
/** "Composite" */
import java.util.List;
import java.util.ArrayList;
class CompositeGraphic implements Graphic {
    //Collection of child graphics.
    private List<Graphic> childGraphics = new ArrayList<Graphic>();
    //Prints the graphic.
    public void print() {
        for (Graphic graphic : childGraphics) {
            graphic.print();
        }
    }
    //Adds the graphic to the composition.
    public void addComponent(Graphic graphic) {
        childGraphics.add(graphic);
    }
    //Removes the graphic from the composition.
    public void removeComponent(Graphic graphic) {
        childGraphics.remove(graphic);
    }…
}
/** "Leaf" */
class Ellipse implements Graphic {
    //Prints the graphic.
    public void print() {
        System.out.println("Ellipse");
    }
}
```

```
/** Client */
public class Program {

    public static void main(String[] args) {
        //Initialize four ellipses
        Ellipse ellipse1 = new Ellipse();
        Ellipse ellipse2 = new Ellipse();
        Ellipse ellipse3 = new Ellipse();
        Ellipse ellipse4 = new Ellipse();

        //Initialize three composite graphics
        CompositeGraphic graphic = new CompositeGraphic();
        CompositeGraphic graphic1 = new CompositeGraphic();
        CompositeGraphic graphic2 = new CompositeGraphic();

        //Composes the graphics
        graphic1.add(ellipse1);
        graphic1.add(ellipse2);
        graphic1.add(ellipse3);

        graphic2.add(ellipse4);

        graphic.add(graphic1);
        graphic.add(graphic2);

        //Prints the complete graphic (four times the string "Ellipse").
        graphic.print();
    }
}
```



Adopted from Wikipedia

Composite pattern

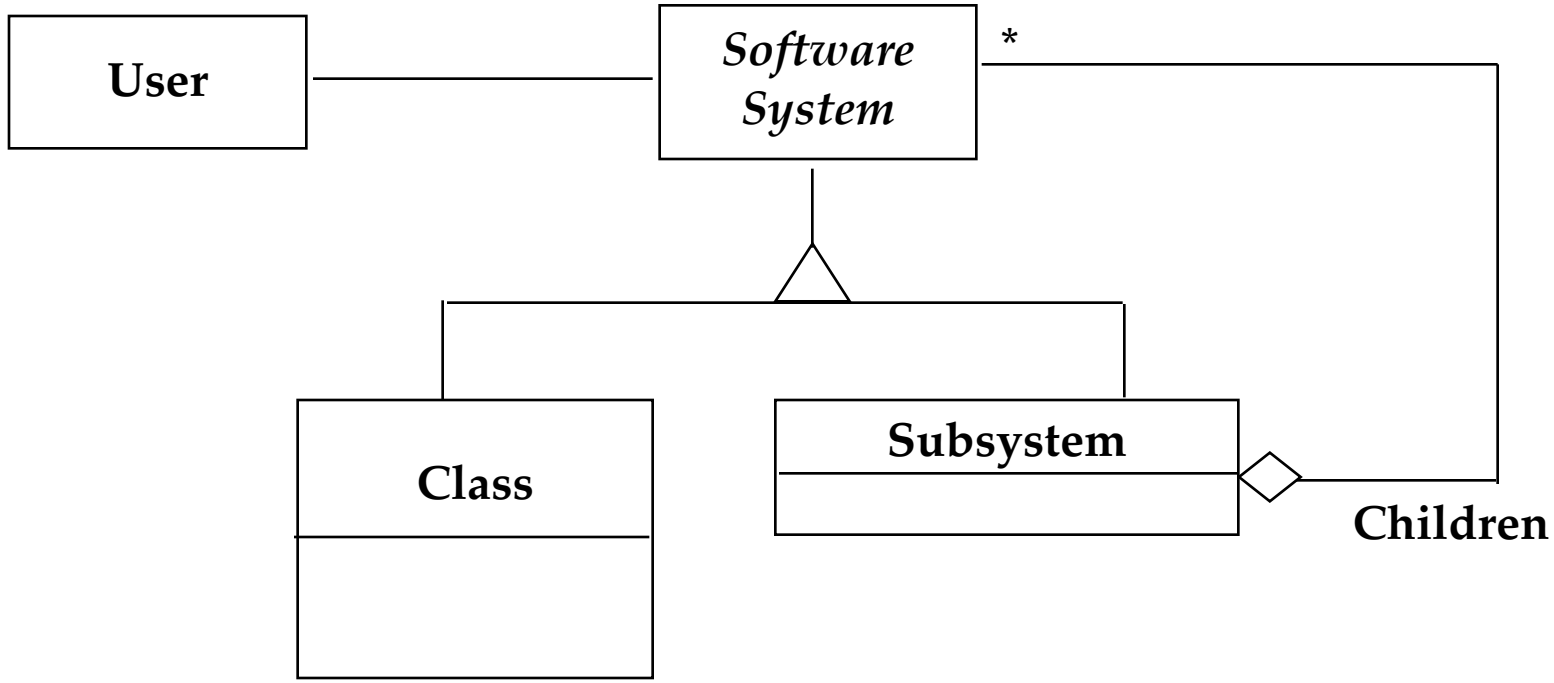
- Client uses the same code for dealing with Leaves or Composites.
- Leaf-specific behavior can be modified without changing the hierarchy.
- New classes of Leaves can be added without changing the hierarchy.

What is common between these definitions?

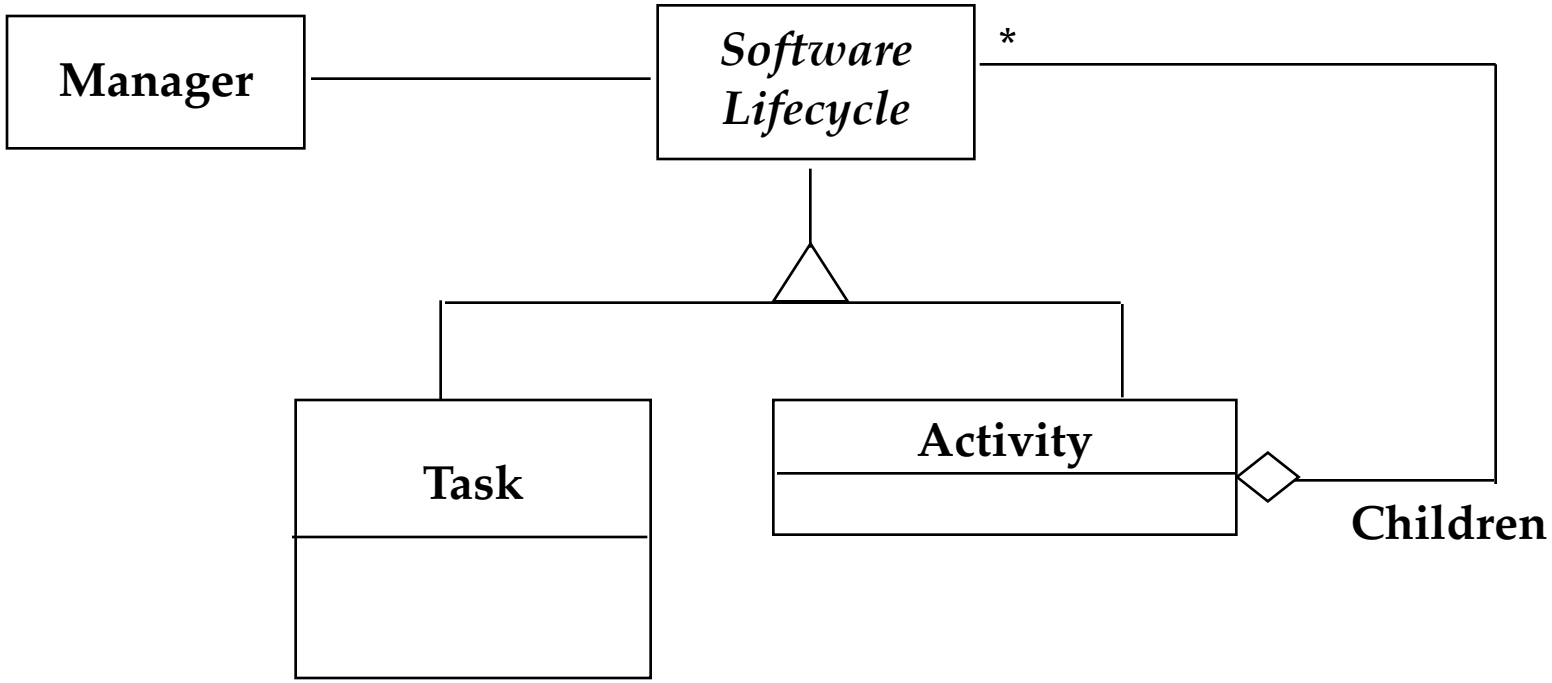
- Software System:
  - Definition: A software system consists of subsystems which are either other subsystems or collection of classes
  - Composite: Subsystem (A software system consists of subsystems which consists of subsystems , which consists of subsystems, which...)
  - Leaf node: Class

- Software Lifecycle:
  - Definition: The software lifecycle consists of a set of development activities which are either other activities or collection of tasks
  - Composite: Activity (The software lifecycle consists of activities which consist of activities, which consist of activities, which....)
  - Leaf node: Task

Modeling a Software System X with a Composite Pattern



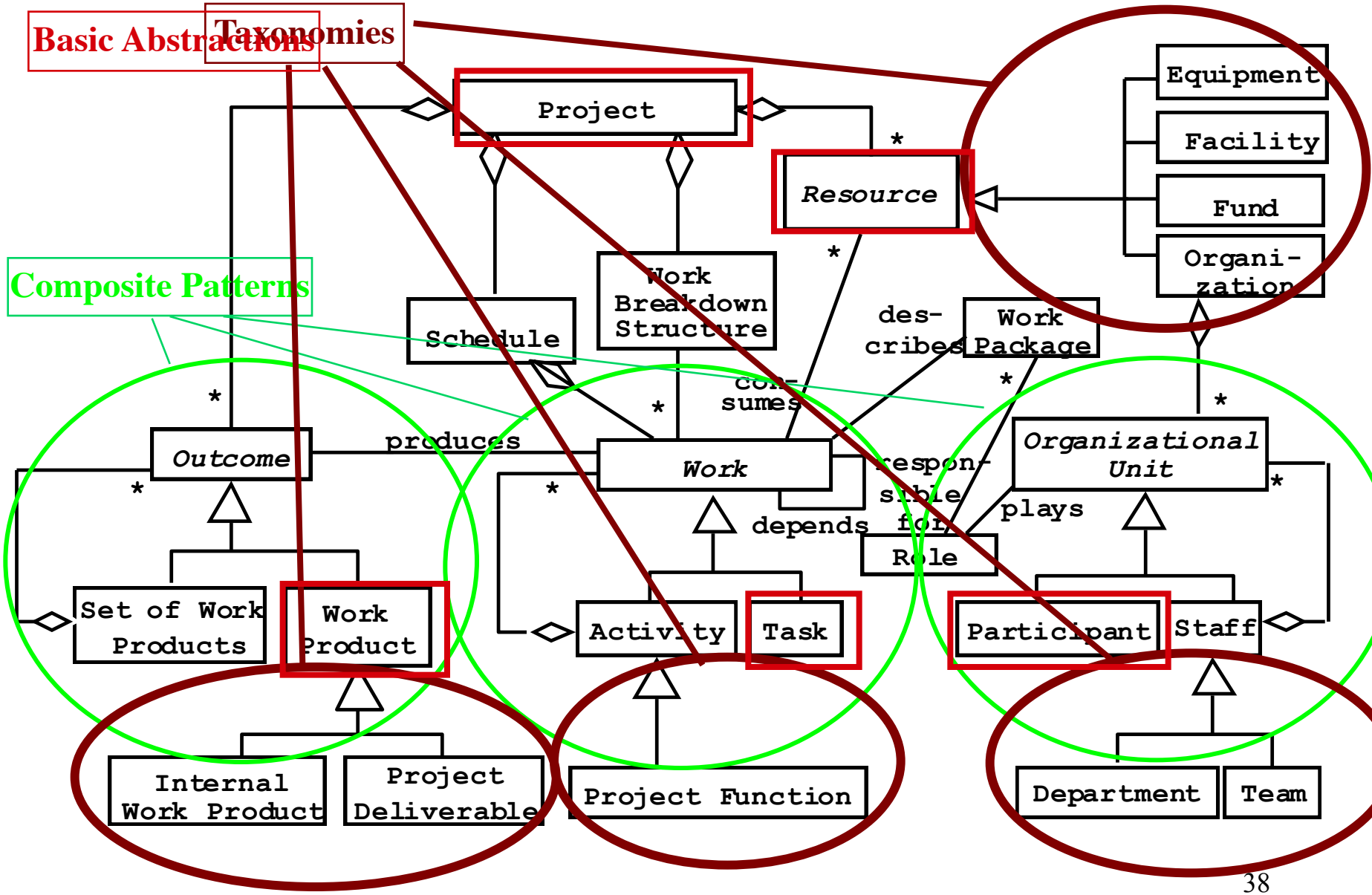
Modeling the Software Lifecycle with a Composite Pattern



Design Patterns reduce the Complexity of Models

- To communicate a complex model we use navigation and reduction of complexity
  - We do not simply use a picture of UML diagram and dump it in front of the user
  - The key is navigate through the model so that user can follow it.

Example: A More Complex Model

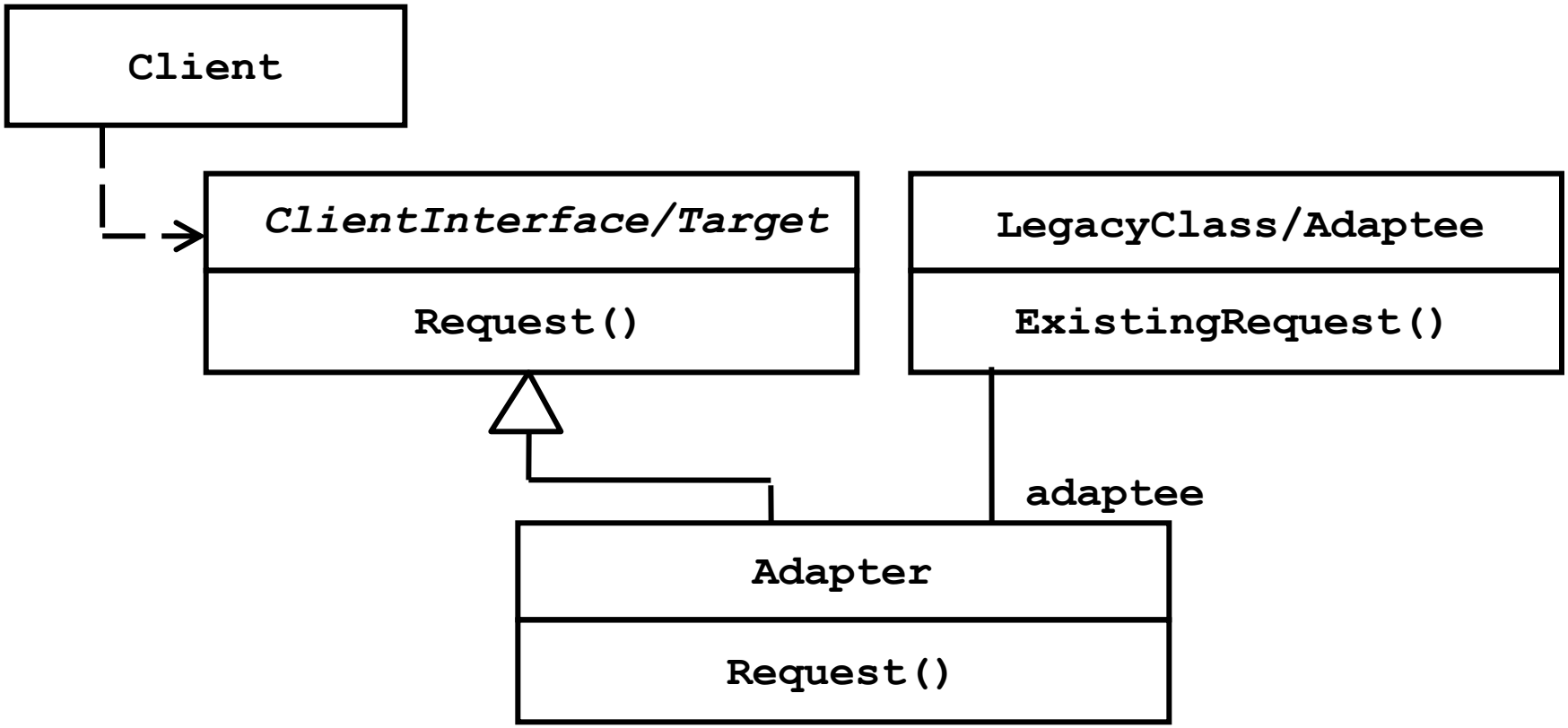


Adopted from Bernd Bruegge & Allen H. Dutoit Object-Oriented Software Engineering: Using UML, Patterns, and Java

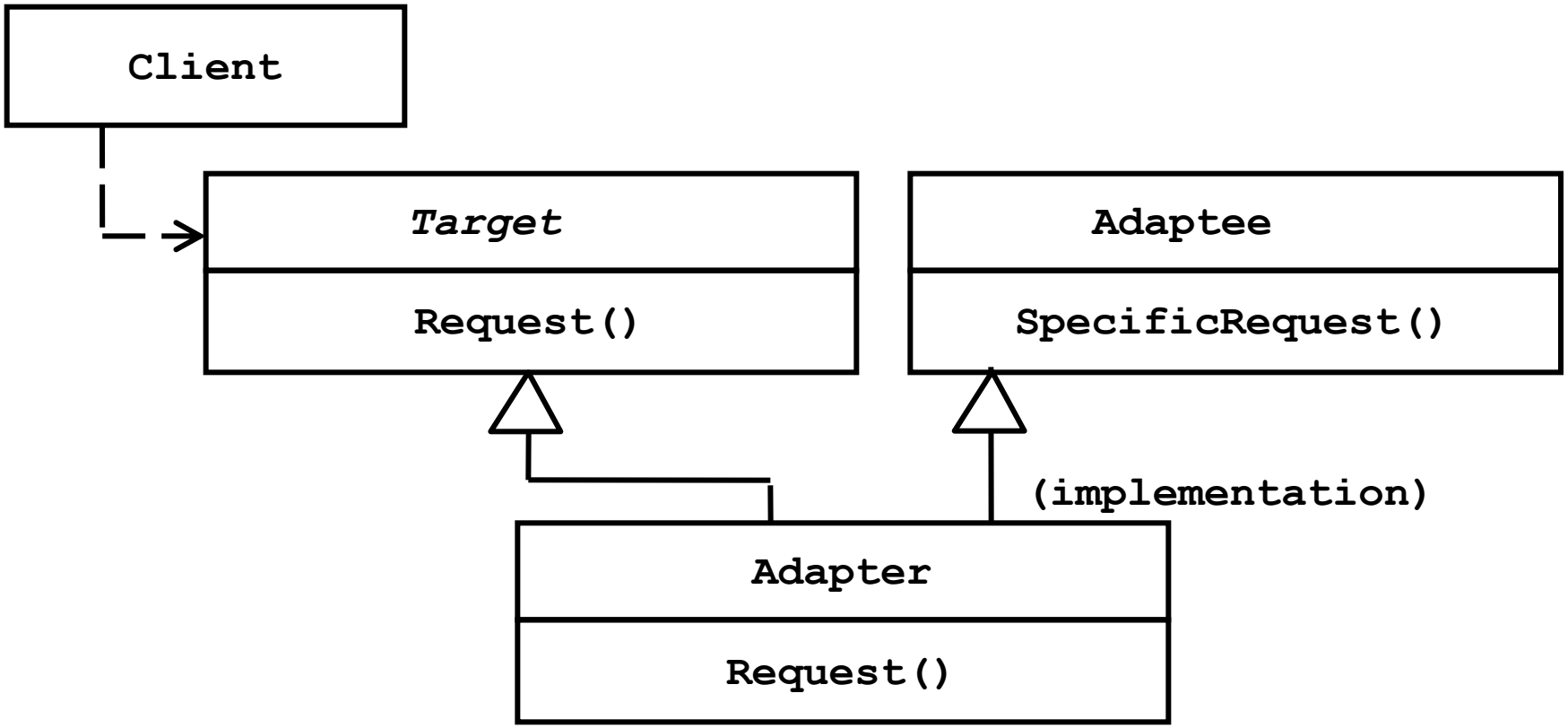
Adapter pattern

- "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:
    - Uses multiple inheritance to adapt one interface to another
  - Object adapter:
    - Uses single inheritance and delegation
- Object adapters are much more frequent. We will only cover object adapters (and call them therefore simply adapters)

Adapter pattern (object)



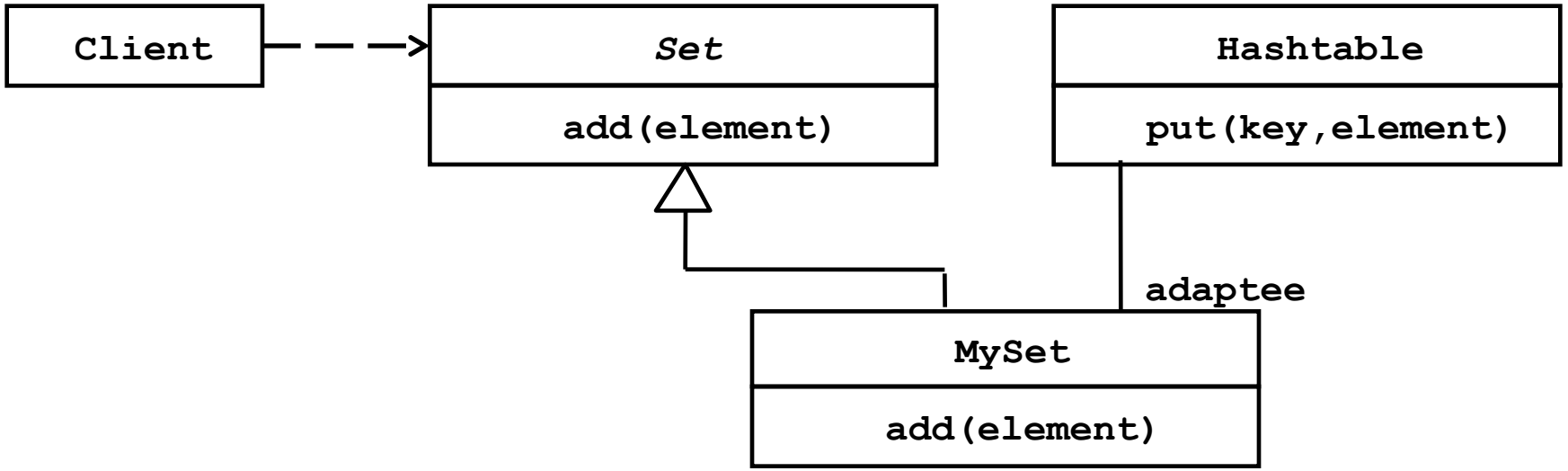
Adapter pattern (class)



Adapter pattern

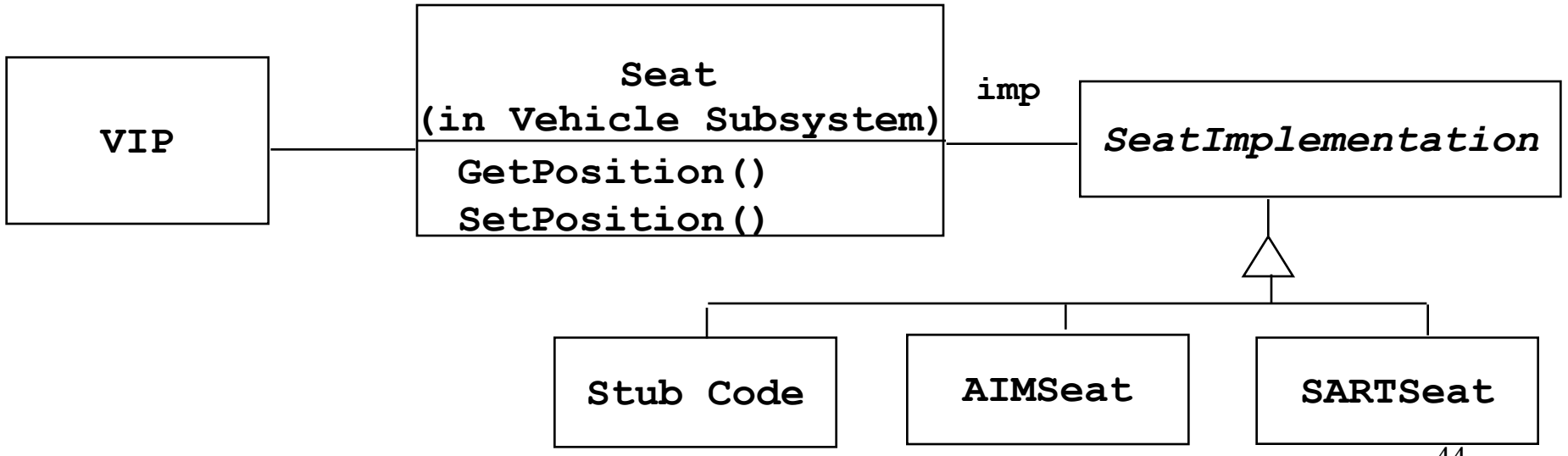
- Client and LegacyClass work together without modification of either Client or LegacyClass.
- Adapter works with LegacyClass and all of its subclasses.
- A new Adapter needs to be written for each specialization (e.g., subclass) of ClientInterface.

Adapter pattern and the Set problem (object)

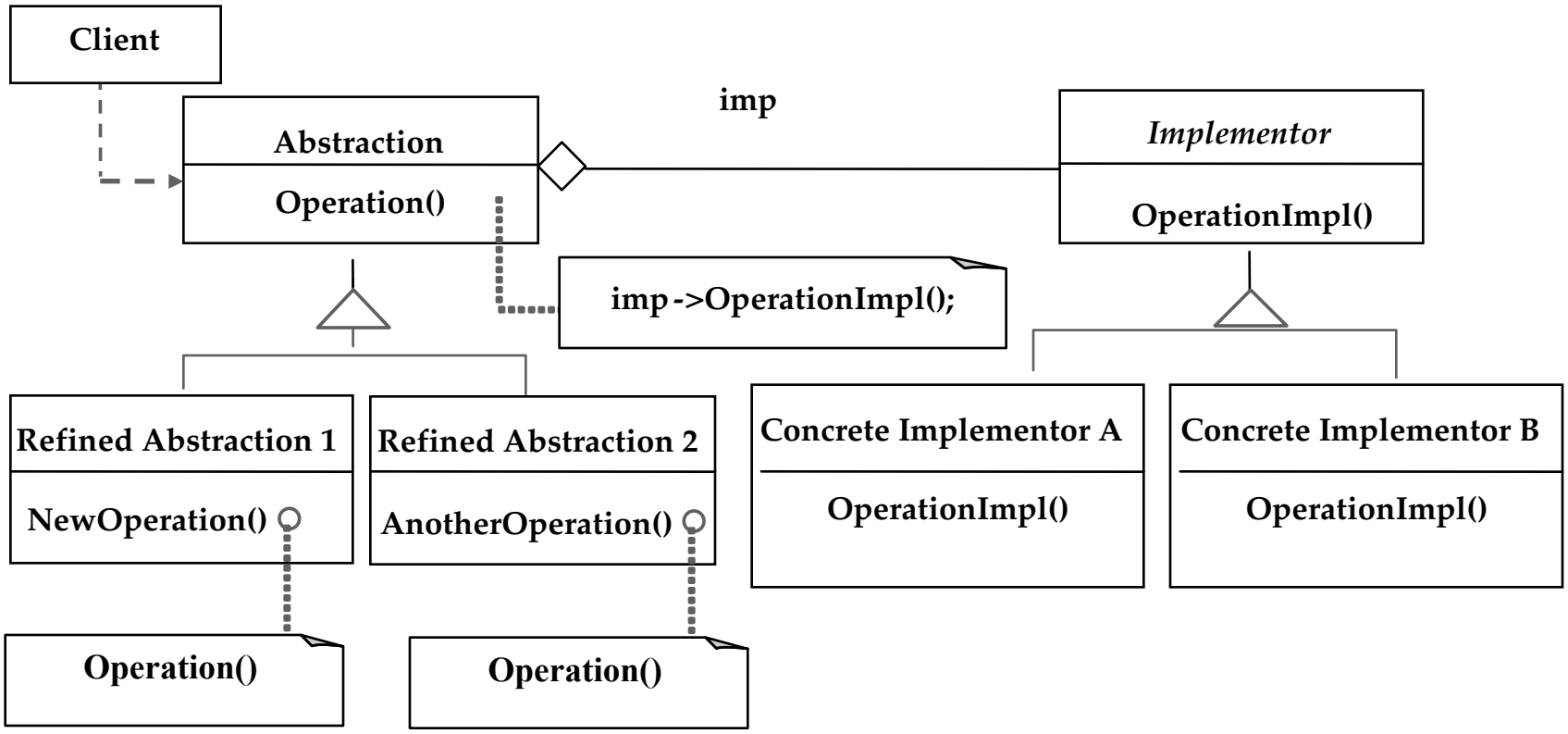


Bridge Pattern

- The bridge pattern is used to provide multiple implementations under the same interface.
- Examples: Interface to a component that is incomplete, not yet known or unavailable during testing
- Example: if seat data is required to be read, but the seat is not yet implemented, known, or only available by a simulation, provide a bridge:



Bridge Pattern

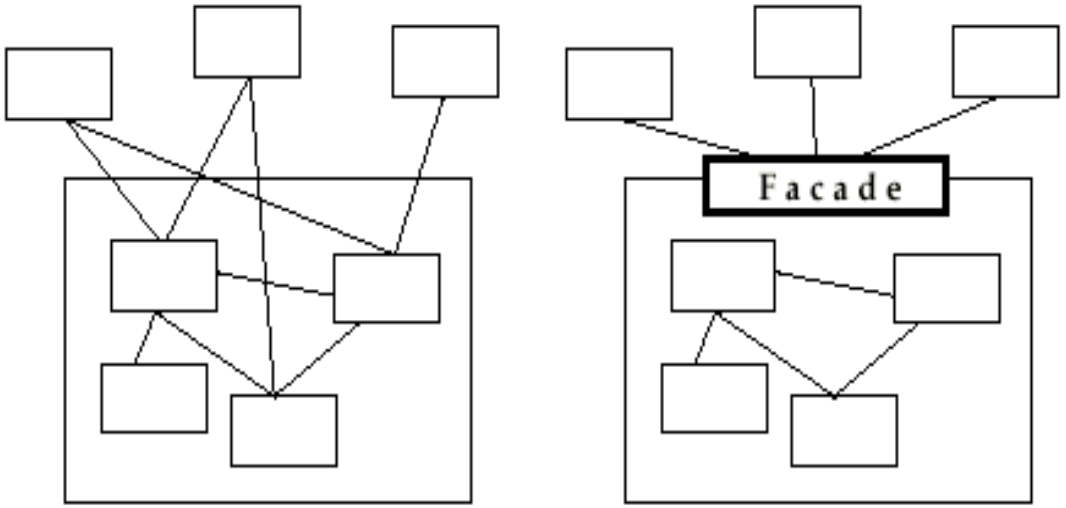


Adapter vs Bridge

- 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.
    - Greenfield engineering of an "extensible system"

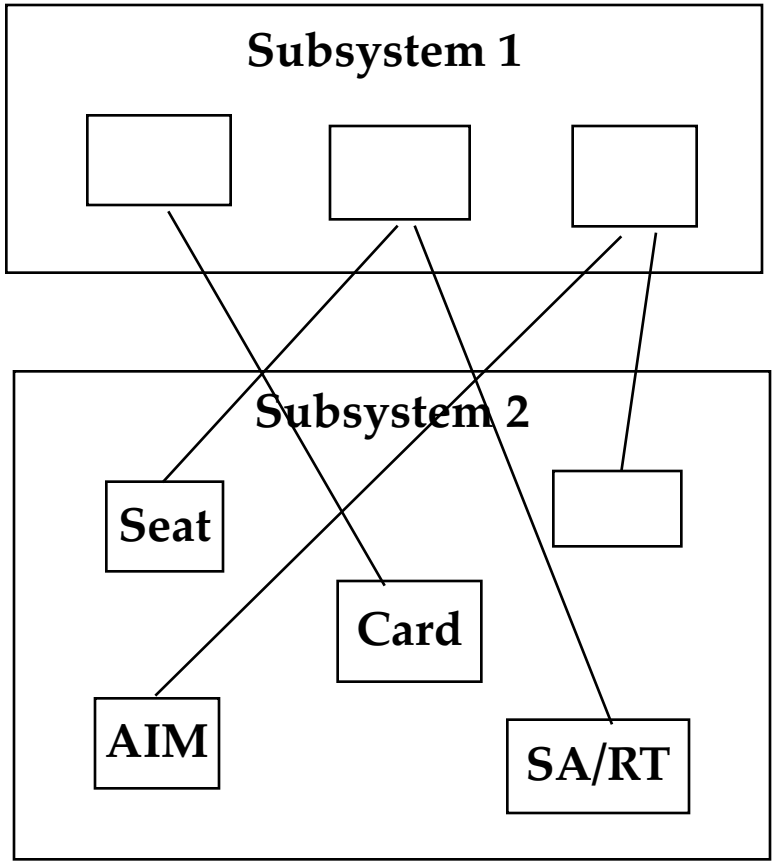
Facade Pattern

- 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 details)
- Facades allow us to provide a closed architecture



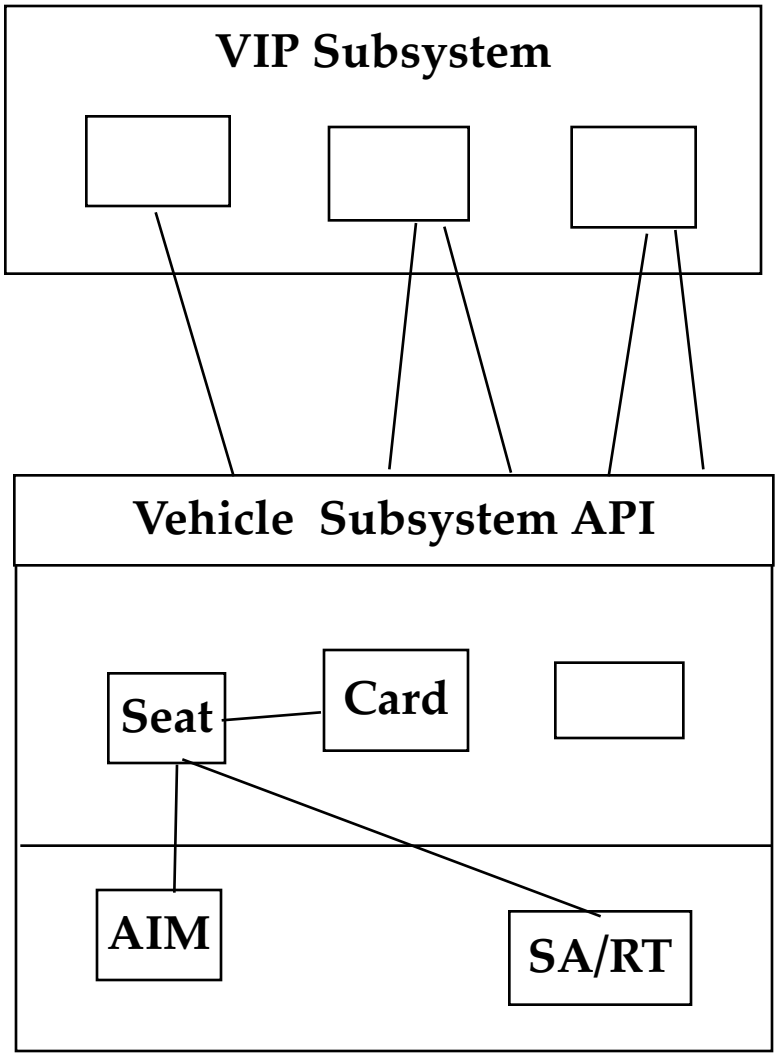
Adopted from Gamma

Design Example



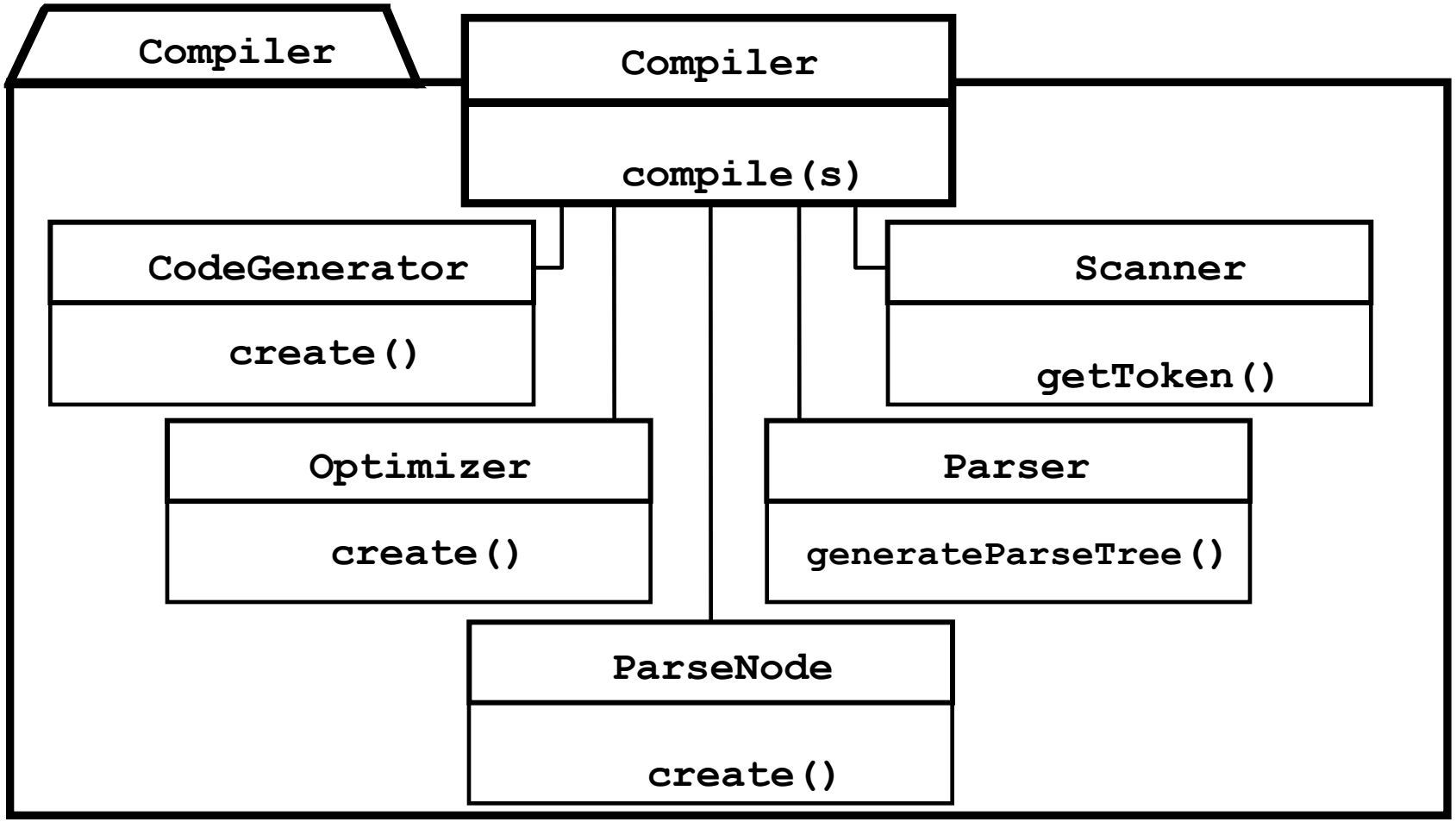
- Subsystem 1 can look into the Subsystem 2 (vehicle subsystem) and call on any component or class operation at will.
- This is "Ravioli Design"

Realizing an Opaque Architecture with a Facade



- The subsystem decides exactly how it is accessed.
- No need to worry about misuse by callers
- If a façade is used the subsystem can be used in an early integration test
  - We need to write only a driver

Example


Compiler
Compiler
compile(s)
CodeGenerator
create()
Scanner
getToken()
Optimizer
create()
Parser
generateParseTree()
ParseNode
create()

Design Patterns encourage reusable designs

- A facade pattern should be used by all subsystems in a software system. The façade defines all the services of the subsystem.
  - The facade will delegate requests to the appropriate components within the subsystem. Most of the time the façade does not need to be changed, when the component is changed,
- Adapters should be used to interface to existing components.
  - For example, a smart card software system should provide an adapter for a particular smart card reader and other hardware that it controls and queries.
- Bridges should be used to interface to a set of objects
  - where the full set is not completely known at analysis or design time.
  - when the subsystem must be extended later after the system has been deployed

Review: Design pattern

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 data structures (link lists, binary trees...)

…an example of *modifiable* design

- Learning to design starts by studying other designs

Why are modifiable designs important?

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

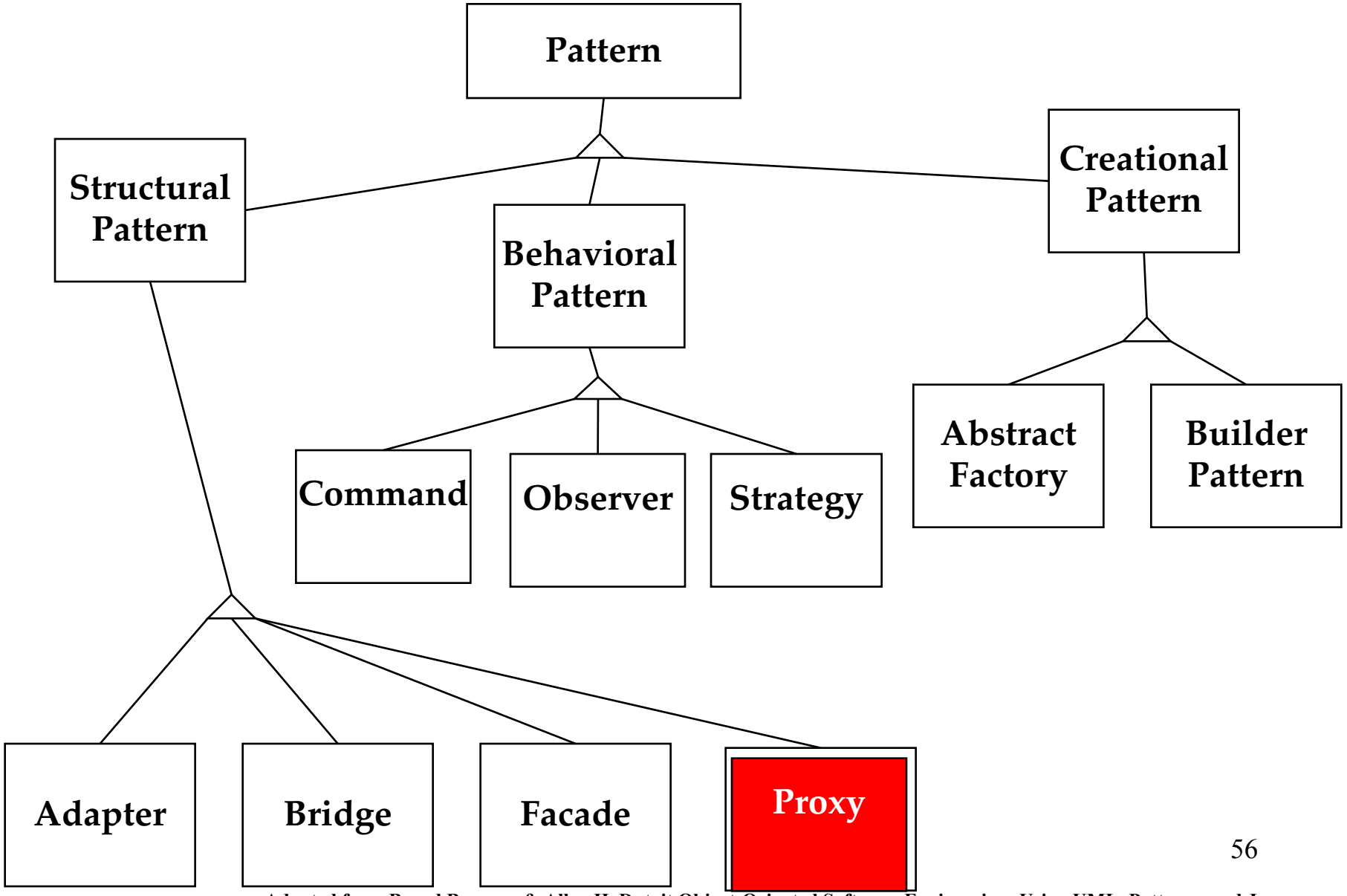
How do design patterns help?

- They are generalized from existing systems
- They provide a shared vocabulary to designers
- They provide examples of modifiable designs
  - Abstract classes
  - Delegation

Pattern Taxonomy

- Structural Patterns
  - Adapters, Bridges, Facades, and Proxies are variations on a single theme:
    - They reduce the coupling between two or more classes
    - They introduce an abstract class to enable future extensions
    - They encapsulate complex structures
- Behavioral Patterns
  - Here we are concerned with algorithms and the assignment of responsibilities between objects: Who does what?
  - Behavioral patterns allow us to characterize complex control flows that are difficult to follow at runtime.
- Creational Patterns
  - Here our goal is to provide a simple abstraction for a complex instantiation process.
  - We want to make the system independent from the way its objects are created, composed and represented.

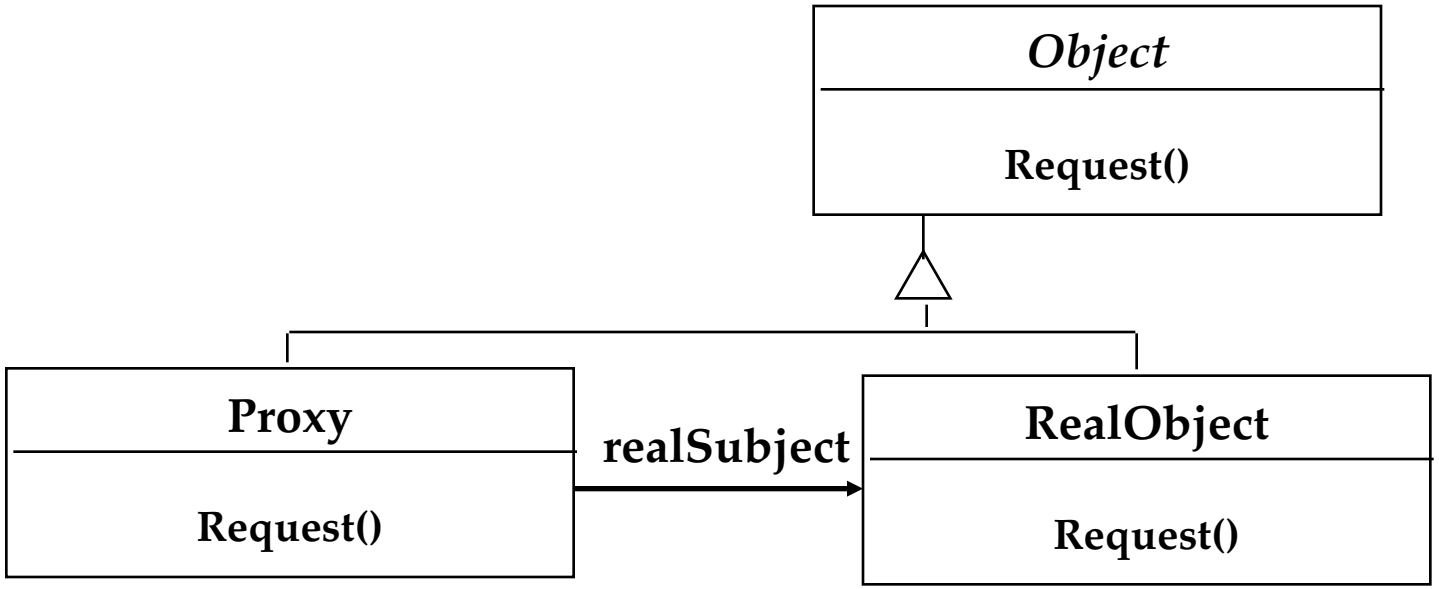
A Pattern Taxonomy



Proxy Pattern

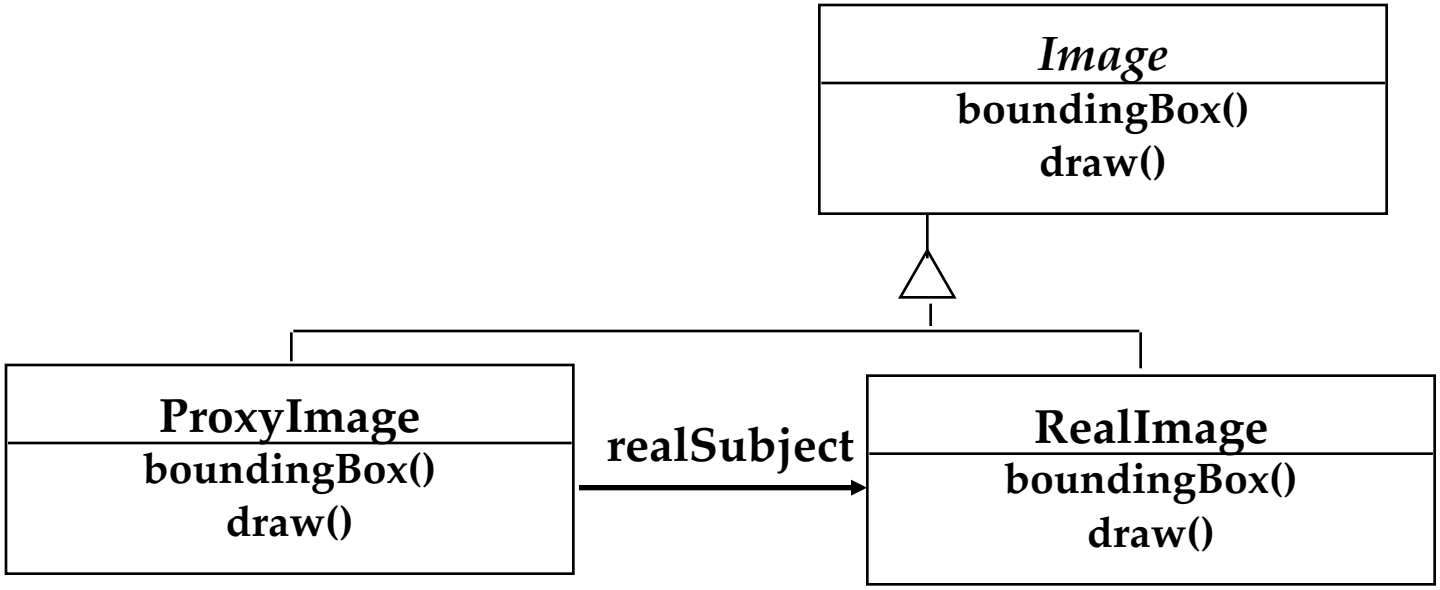
- What is expensive?
  - Object Creation
  - Object Initialization
- Defer object creation and object initialization to the time you need the object
- Proxy pattern:
  - Reduces the cost of accessing objects
  - Uses another object (“the proxy”) that acts as a stand-in for the real object
  - The proxy 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 **RealObject.**
- Delegation is used to catch and forward any accesses to the **RealObject** (if desired)
- Proxy patterns can be used for lazzy evaluation and for remote invocation.
- Proxy patterns can be implemented with a Java interface.

Virtual Proxy example

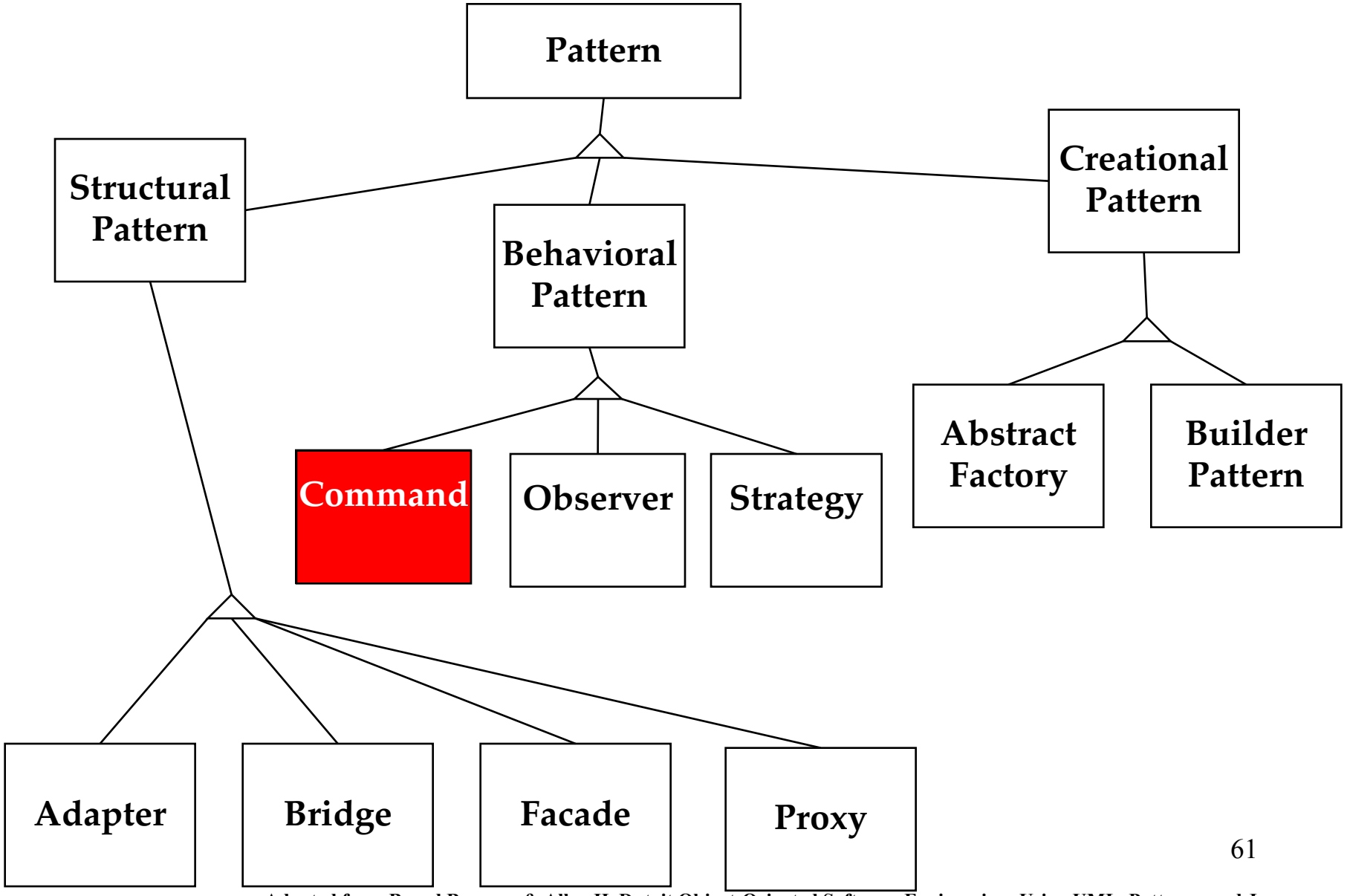


- **Images** are stored and loaded separately from text
- If a **RealImage** is not loaded a **ProxyImage** displays a grey rectangle in place of the image
- The client cannot tell that it is dealing with a **ProxyImage** instead of a **RealImage**
- A proxy pattern can be easily combined with a **Bridge**

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.
  - Example: Grade information for a student shared by administrators, teachers and students.

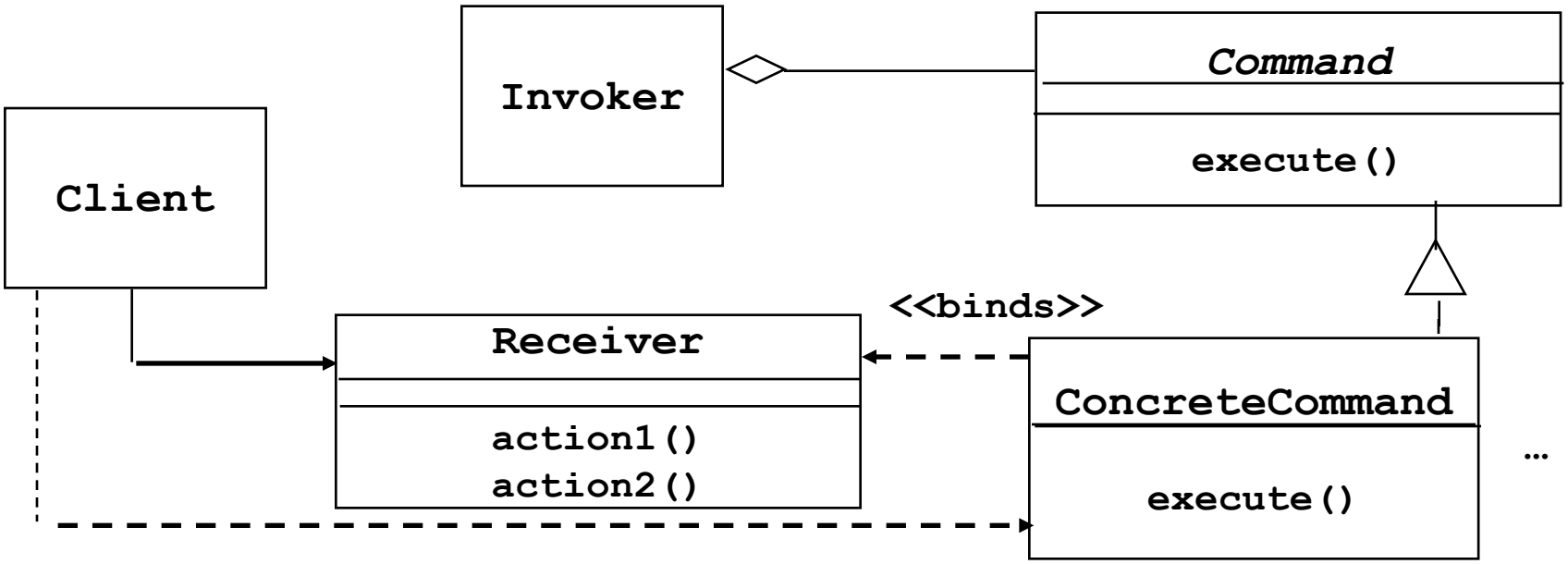
A Pattern Taxonomy



Command Pattern: Motivation

- 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”)**.**

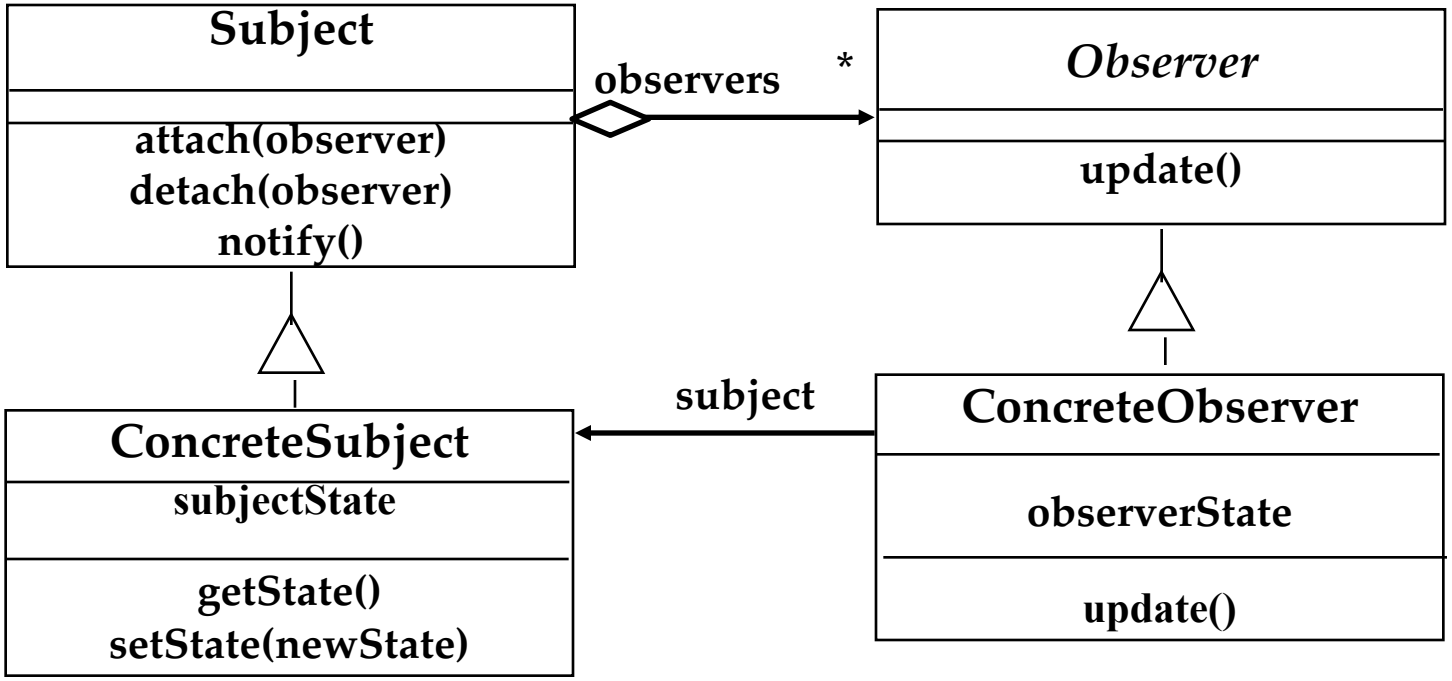
```
import java.util.List;
import java.util.ArrayList;
/* The Command interface */
public interface Command {
   void execute();
}
/* The Invoker class */
public class Switch {
   private List<Command> history = new ArrayList<Command>();
   public void storeAndExecute(Command cmd) {
      this.history.add(cmd); // optional
      cmd.execute();
   }
}
 /* The Receiver class */
public class Subject{
    public void turnOn() {
      System.out.println("The light is on");
   }
    public void turnOff() {
      System.out.println("The light is off");
   }
}
/* The Command for turning on the subject-ConcreteCommand #1 */
public class OnCommand implements Command {
   private Subject theSubject;
    public OnCommand(Subject subject) {
      this.theSubject = subject;
   }
    public void execute(){
      theSubject.turnOn();
   }
}
/* The Command for turning off the subject - ConcreteCommand
#2 */
public class OffCommand implements Command {
   private Subject theSubject;
    public OffCommand(Subject subject) {
      this.theSubject = subject;
   }
    public void execute(){
      theSubject.turnOff();
   }
}

/* The test class or client */
public class PressSwitch {
   public static void main(String[] args){
      Light lamp = new Subject();
      Command switchUp = new OnCommand(lamp);
      Command switchDown = new OffCommand(lamp);
      Switch mySwitch = new Switch();
      switch(args[0]) {
         case "ON":
            mySwitch.storeAndExecute(switchUp);
         break;
         case "OFF":
            mySwitch.storeAndExecute(switchDown);
         break;
         default:
            System.out.println("Argument \"ON\" or \"OFF\" is
                  required.");
      }
   }
}
```


Adopted from Wikipedia

Observer pattern

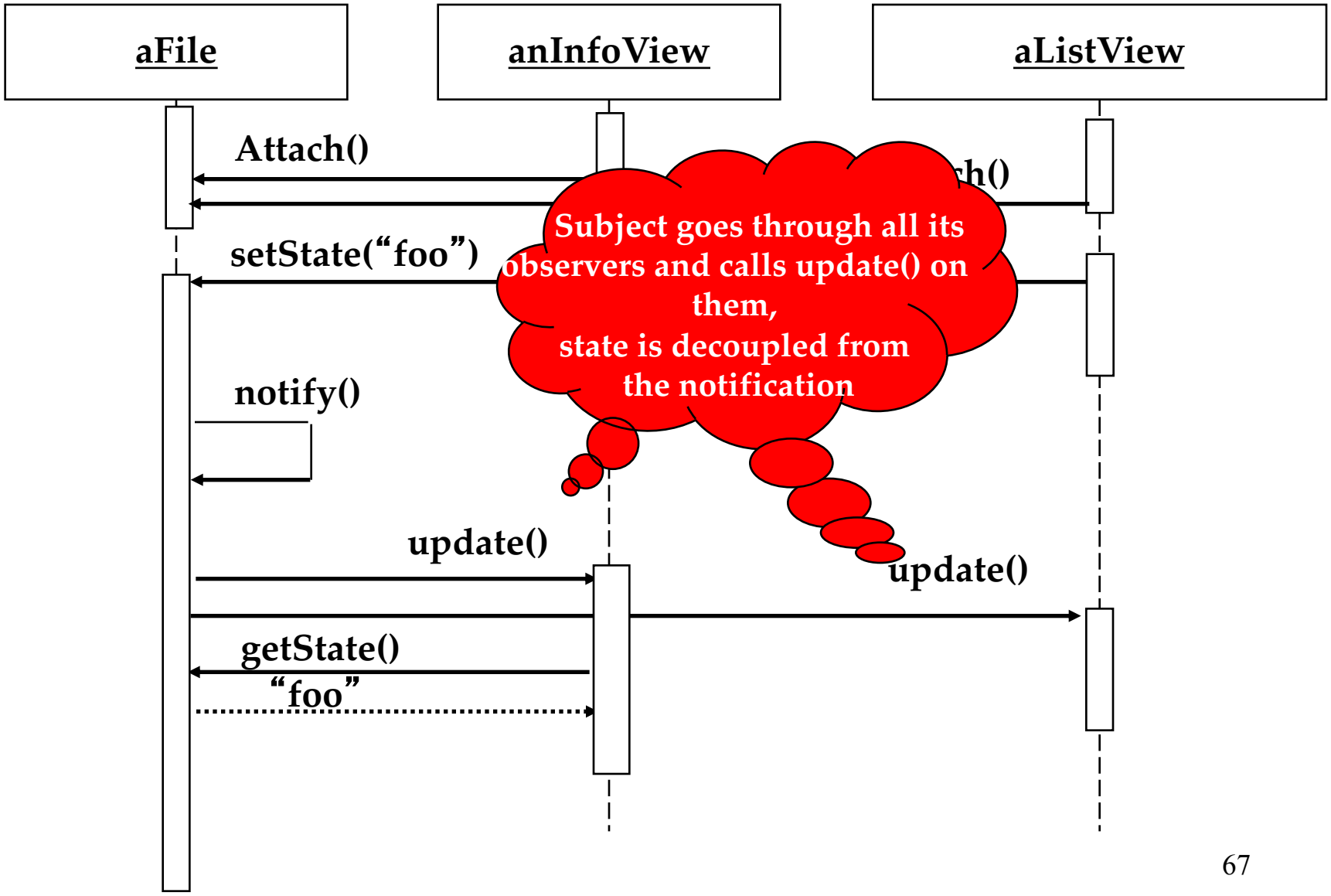
- "Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically."
- Also called "Publish and Subscribe"
- Uses: Maintaining consistency between related object
  - When an abstraction has two aspects. One dependent on the other. Encapsulating these aspects in separate objects lets you vary and reuse them independently
  - When a change to one object requires changing others, and you don't know how many objects need to be changed.
  - When an object should be able to notify other objects without making assumptions about who these objects are. In other words, you don't want these objects tightly coupled.

Observer pattern



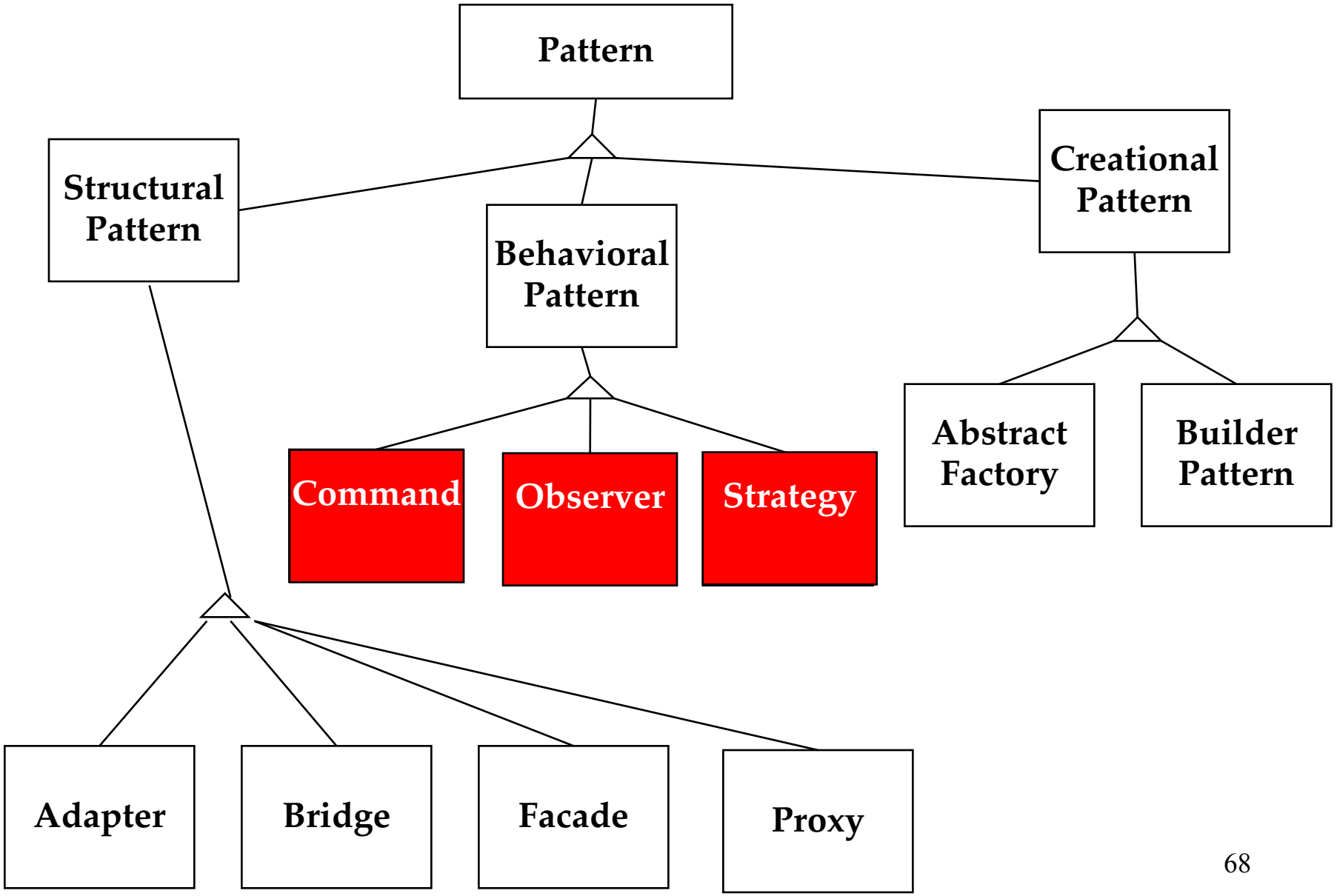
- The **Subject** represents the actual state, the **Observers** represent different views of the state.
- **Observer** can be implemented as a Java interface.
- **Subject** is a superclass (needs to store the observers vector) ***not*** an interface.

Sequence diagram for scenario: Change filename to “foo”



Adopted from Bernd Bruegge & Allen H. Dutoit Object-Oriented Software Engineering: Using UML, Patterns, and Java

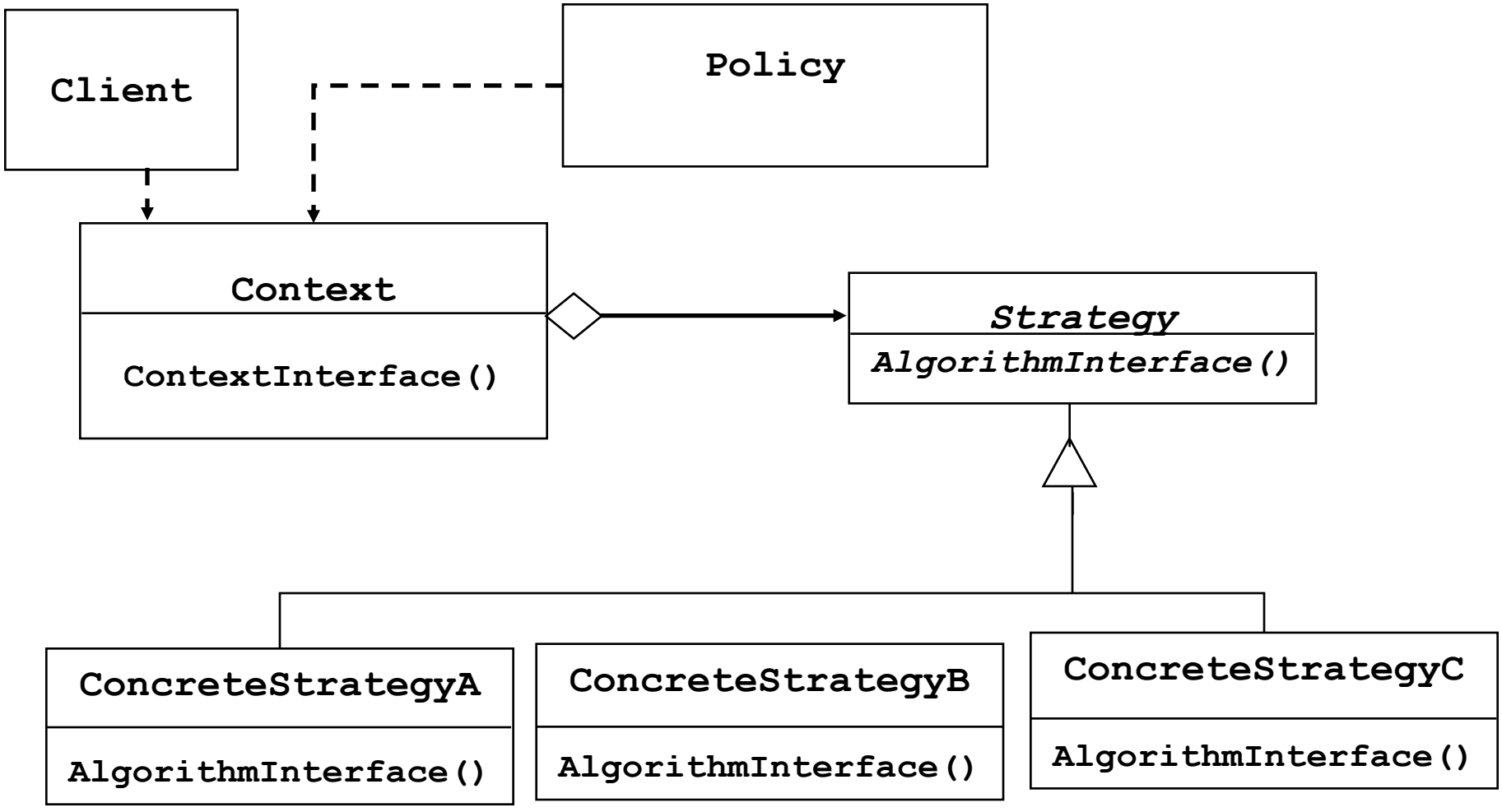
Pattern Taxonomy



Strategy Pattern

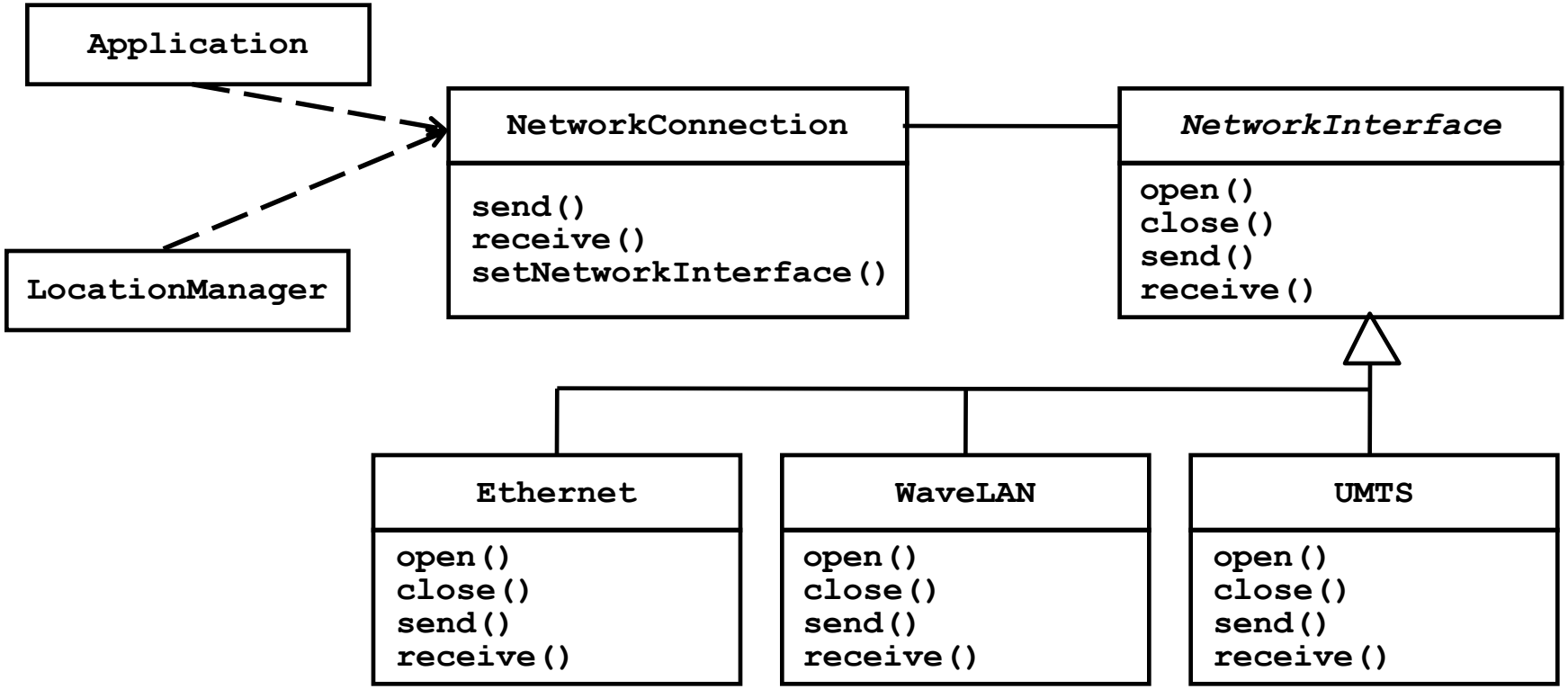
- Many different algorithms exists for the same task
- Examples:
  - Breaking a stream of text into lines
  - Parsing a set of tokens into an abstract syntax tree
  - Sorting a list of customers
- The different algorithms will be appropriate at different times
  - Rapid prototyping vs. delivery of final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

Strategy Pattern

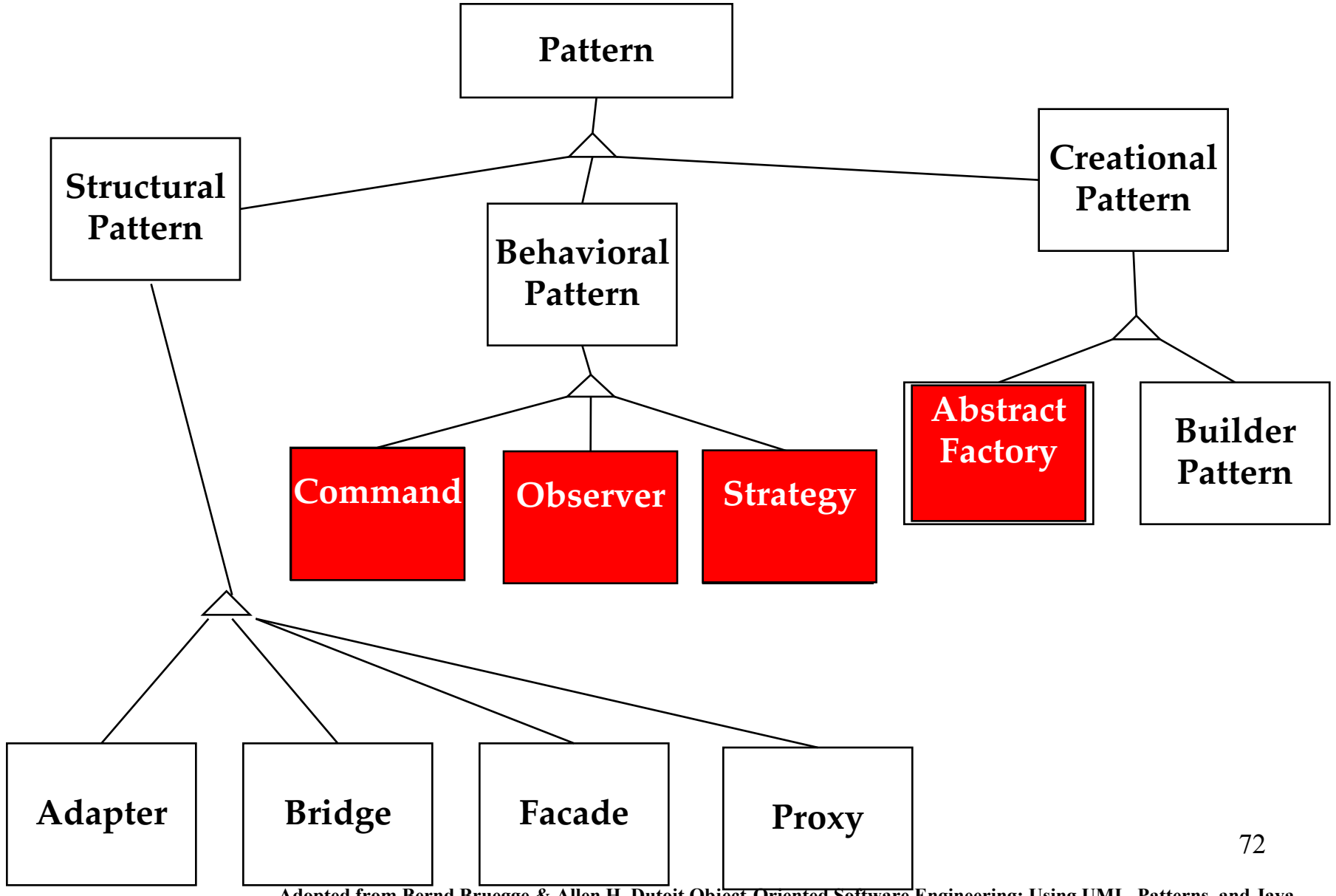


`Policy` decides which `Strategy` is best given the current `Context`

Applying the Strategy pattern for encapsulating multiple implementations of a NetworkInterface.



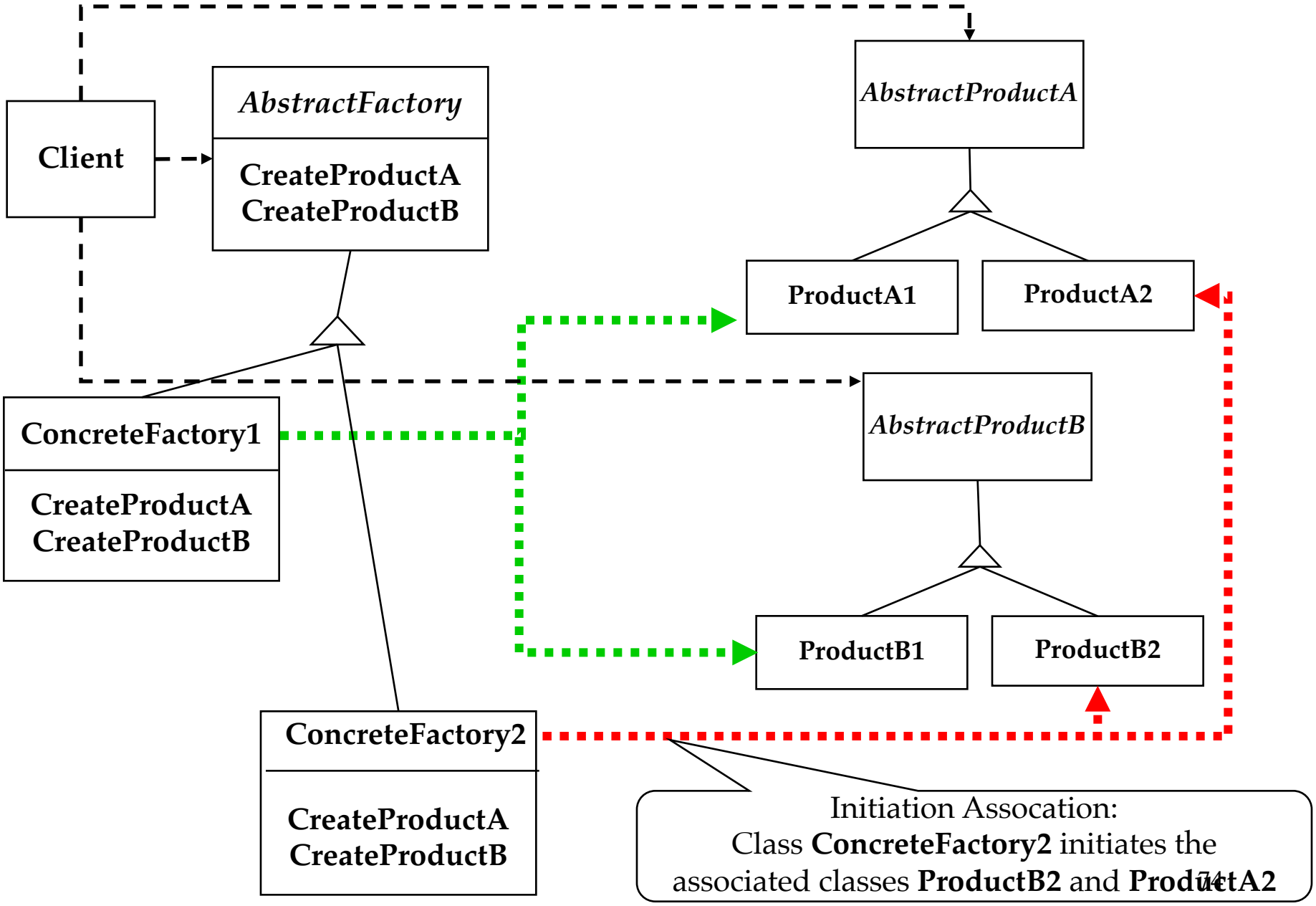
Pattern Taxonomy



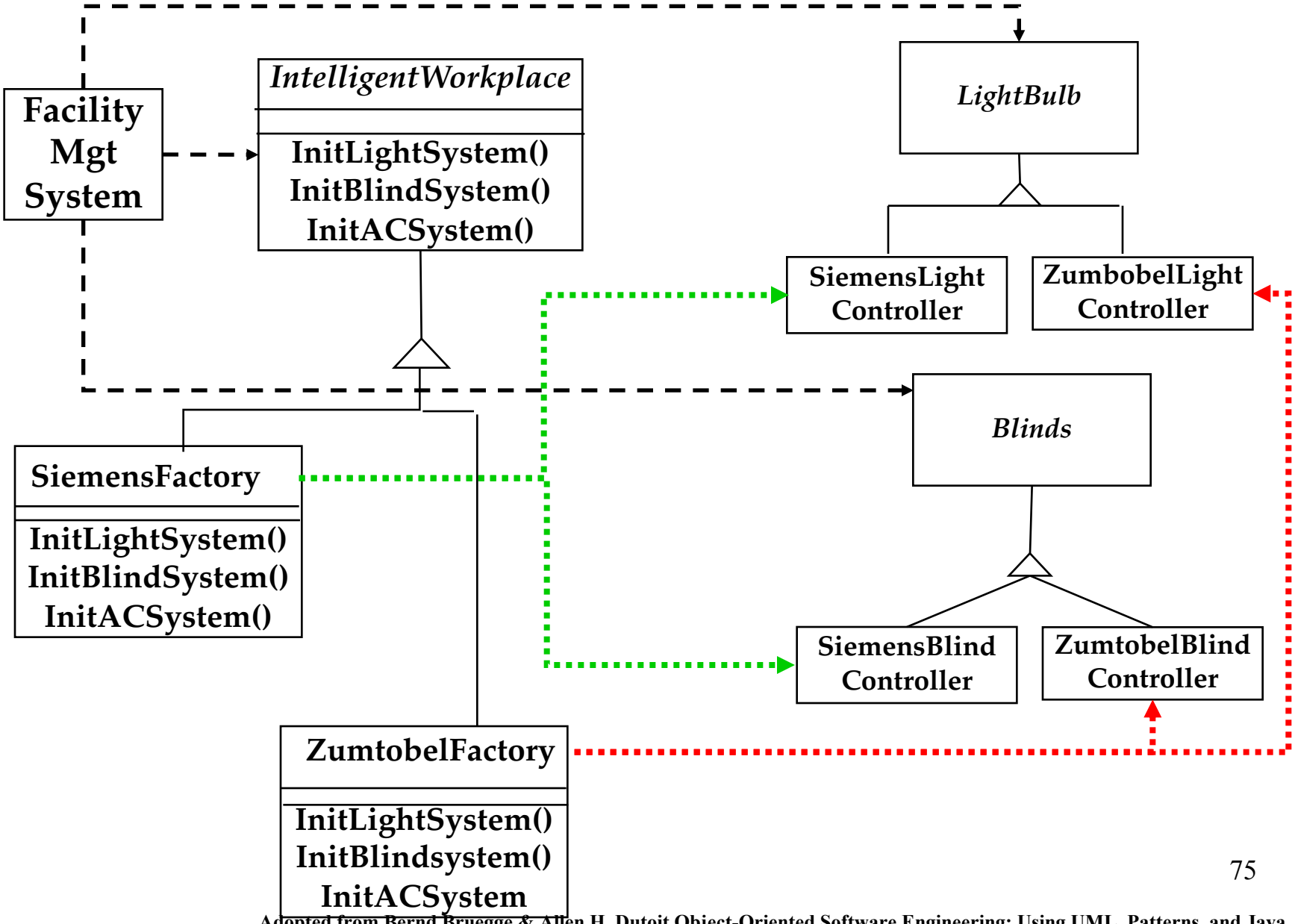
Abstract Factory Motivation

- 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 - provides an interface for creating families of related or dependent objects without specifying their concrete classes.

Abstract Factory



A Facility Management System for the Intelligent Workplace



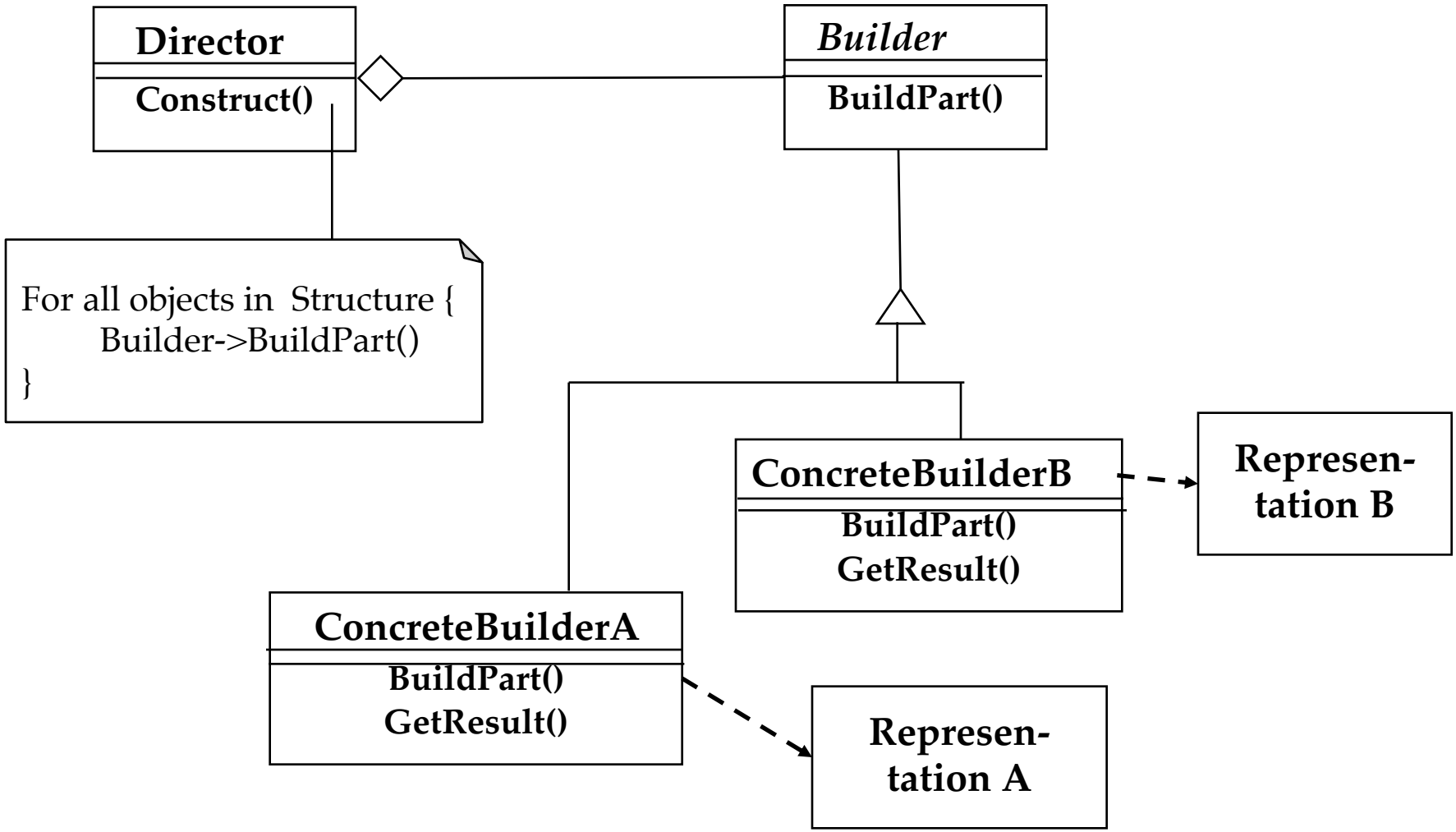
Applicability for Abstract Factory Pattern

- Independence from Initialization or Representation:
  - 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.

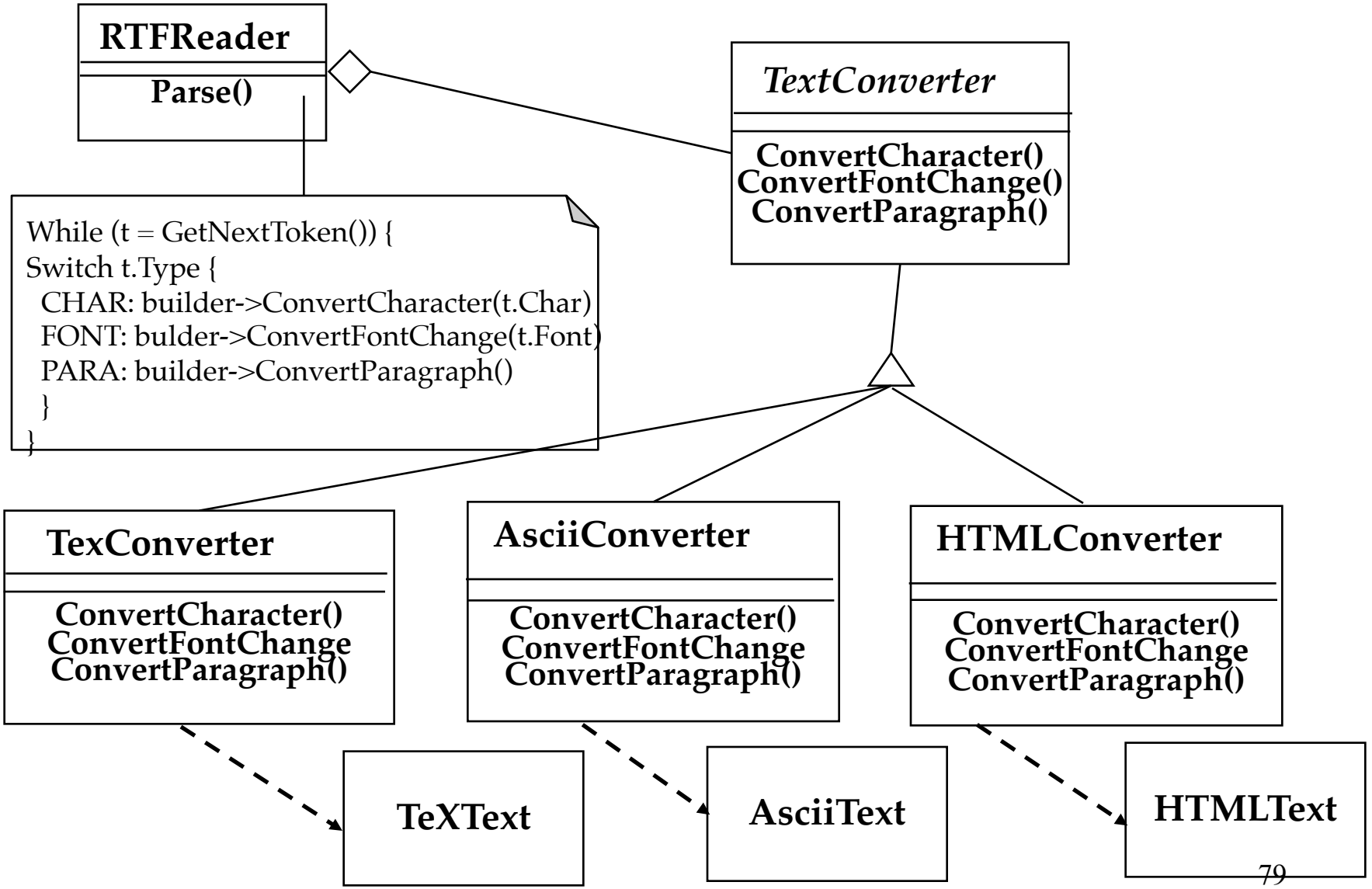
Builder Pattern Motivation

- Software companies make their money by introducing new formats, forcing users to upgrades
  - But you don't want to upgrade your software every time there is an update of the format for Word documents
- Idea: A reader for RTF format
  - Convert RTF to many text formats (EMACS, Framemaker 4.0, Framemaker 5.0, Framemaker 5.5, HTML, SGML, WordPerfect 3.5, WordPerfect 7.0, ASCII….)
    - *Problem: The number of conversions is open-ended.*
- Solution
  - Configure the RTF Reader with a "builder" object that specializes in conversions to any known format and can easily be extended to deal with any new format appearing on the market
- BUILDER - separates the construction of a complex object from its representation so that the same construction process can create different representations.

Builder Pattern



Builder Pattern - Example



Comparison: Abstract Factory vs Builder

- Abstract Factory
  - Focuses on product family
    - The products can be simple ("light bulb") or complex ("engine")
  - Does not hide the creation process
    - The product is immediately returned
- Builder
  - The underlying product needs to be constructed as part of the system, but the creation is very complex
  - The construction of the complex product changes from time to time
  - The builder patterns hides the creation process from the user:
    - The product is returned after creation as a final step
- Abstract Factory and Builder work well together for a family of multiple complex products

Nonfunctional Requirements may give a clue for the use of Design Patterns

- *Text:* “manufacturer independent”, “device independent”, “must support a family of products”
  - Abstract Factory Pattern
- *Text:* “must interface with an existing object”
  - Adapter Pattern
- *Text:* “must deal with the interface to several systems, some of them to be developed in the future”, “ an early prototype must be demonstrated”
  - Bridge Pattern

Textual Clues in Nonfunctional Requirements

- *Text:* “complex structure”, “must have variable depth and width”
  - Composite Pattern
- *Text:* “must interface to a set of existing objects”
  - Façade Pattern
- *Text:* “must be location transparent”
  - Proxy Pattern
- *Text:* “must be extensible”, “must be scalable”
  - Observer Pattern
- *Text:* “must provide a policy independent from the mechanism”
  - Strategy Pattern
- *Text*: “all commands should be undoable" , “all transactions should be logged”
  - Command Pattern

Summary

- Reuse as a part of Object Design
- 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

Next lecture

- Text book

  Chapters 9 & 10