UNIVERSITÀ DEGLI STUDI DI NAPOLI
FEDERICO II

Ingegneria del Software – Design Patterns e Object Design

Prof. Sergio Di Martino

Re-use of Code vs. Re-us of Design

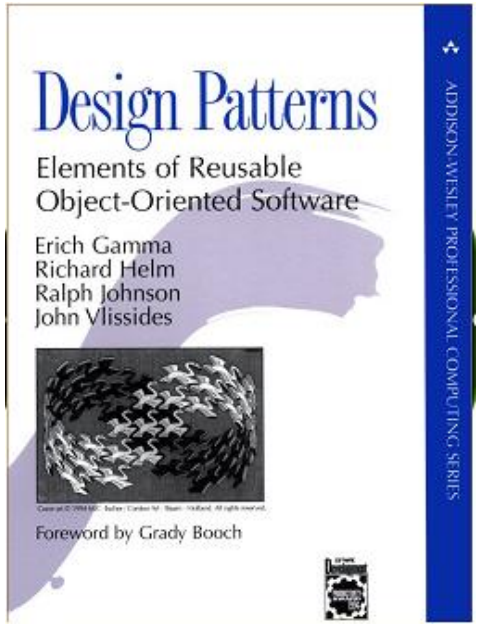
- Code re-use
  - Don’t reinvent the wheel
  - Requires clean, elegant, understandable, general, stable code
  - leverage previous work
- Design re-use
  - Don’t reinvent the wheel
  - Requires a precise understanding of common, recurring designs
  - leverage previous work

Motivation and Concept

- O-O systems exploit recurring design structures that promote
  - Abstraction
  - Flexibility
  - Modularity
  - Elegance
- Therein lies valuable design knowledge
- Problem: capturing, communicating, and applying this knowledge for re-use

What Is a Design Pattern?

- A design pattern
  - Is a common solution to a recurring problem in design
  - Abstracts a recurring design structure
  - Comprises class and/or object
    - Dependencies
    - Structures
    - Interactions
    - Conventions
  - Names & specifies the design structure explicitly
  - Distils design experience



History of Design Patterns

- Architect Christopher Alexander
  - *A Pattern Language: Towns, Buildings, Construction* (1977)
- “Gang of four”
    - Erich Gamma
    - Richard Helm
    - Ralph Johnson
    - John Vlissides
  - Design Patterns: Elements of Reusable Object-Oriented Software (1995)
- Many since
- Conferences, symposia, books

What Is a Design Pattern?

- A design pattern has 4 basic parts:
  - 1. Name
  - 2. Problem
  - 3. Solution
  - 4. Consequences and trade-offs of application
- Language- and implementation-independent
- A “micro-architecture”
- No mechanical application
  - The solution needs to be translated into concrete terms in the application context by the developer

Goals

- Codify good design
  - Distil and disseminate experience
  - Aid to novices and experts alike
  - Abstract how to think about design
- Give design structures explicit names
  - Common vocabulary
  - Reduced complexity
  - Greater expressiveness
- Capture and preserve design information
  - Articulate design decisions succinctly
  - Improve documentation
- Facilitate restructuring/refactoring
  - Patterns are interrelated
  - Additional flexibility

Design Pattern Catalogues

- GoF (“the Gang of Four”) catalogue
  - “Design Patterns: Elements of Reusable Object-Oriented Software,” Gamma, Helm, Johnson, Vlissides, Addison-Wesley, 1995
- POSA catalogue
  - Pattern-Oriented Software Architecture, Buschmann, et al.; Wiley, 1996
- …

		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

Singleton

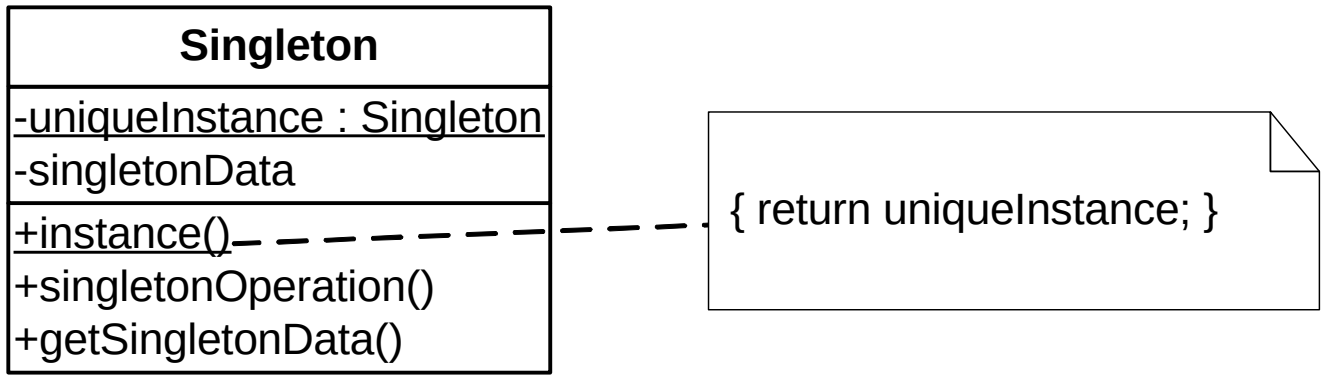
(Creational)

Singleton (Creational)

- Intent
  - Ensure a class only ever has one instance, and provide a global point of access to it.
- Applicability
  - When there must be exactly one instance of a class, and it must be accessible from a well-known access point
  - When the sole instance should be extensible by subclassing, and clients should be able to use an extended instance without modifying their code

Singleton (cont'd)

- Structure



Singleton (cont'd)

- Consequences
  - Reduces namespace pollution
  - Makes it easy to change your mind and allow more than one instance
  - Same drawbacks of a global if misused
  - Implementation may be less efficient than a global
  - Concurrency pitfalls
- Implementation
  - Static instance operation

Singleton - Example

```
public class Singleton {

   private static Singleton istanza = null;

   private Singleton () {}

   public static Singleton getSingleton() {
    if (istanza == null) {
        istanza = new Singleton();
    }
    return istanza;
   }

   public void foo(){dosmthg…}
}
```

- Although the above example uses a single instance, modifications to the function Instance() may permit a variable number of instances.
  - For example, you can design a class that allows up to three instances.

		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

Iterator

(Behavioral)

Your Situation

- You have a group of objects, and you want to iterate through them without any need to know the underlying representation.
- Also, you may want to iterate through them in multiple ways (forwards, backwards, skipping, or depending on values in the structures).
- You might even want to iterate through the list of objects simultaneously using your two or more of your multiple ways.
- Iterator pattern is very commonly used design pattern in Java and .Net programming environment.

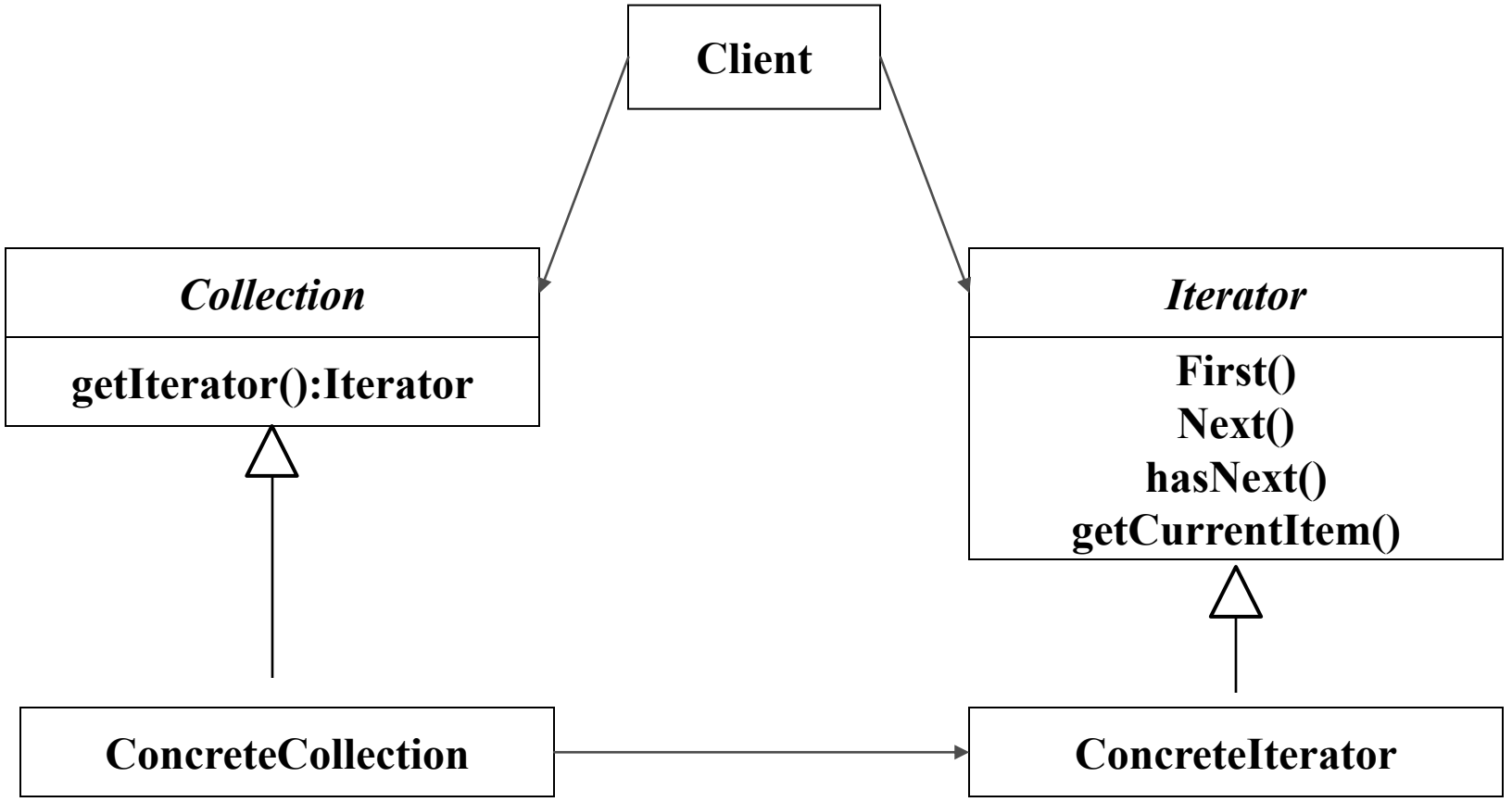
How can we do this?

- *In object-oriented programming, an iterator is an object allowing one to sequence through all of the elements or parts contained in some other object, typically a container or list. An iterator is sometimes called a cursor, especially within the context of a database.*
- The iterator pattern gives us a way to do this by separating the implementation of the iteration from the list itself.
- The Iterator is a known interface, so we don’t have to expose any underlying details about the list data if we don’t need to.

Who is involved in this?

- Iterator Interface
  - Simple methods for traversing elements in the list
- Concrete Iterator
  - A class that implements Iterator
- Iteratee Interface
  - Defines an interface for creating the list that will be iterated
- Concrete Iteratee
  - Creates a concrete iterator for its data type.

How do they work together?



Consequences

- Who controls the iteration?
  - Internal and external iterators
- Who defines the algorithm?
  - Can be stored in the iterator or iteratee
- Robustness
- Additional Iterator Operators
  - Previous, SkipTo
- Iterators have privileged access to data?

Implementation

```
C#:
 class DemoIterator
    {
// stuff to make the demo running
        private ArrayList thelist;

        DemoIterator() {
            thelist = new ArrayList();
            thelist.Add(1);
            thelist.Add(2);
        }

//key method, independent from container and iterator
        public String printListContent(IList thelist) {
            IEnumerator en= thelist.GetEnumerator();

            String retValue= "";
            while (en.MoveNext())
            {
                retValue += en.Current;
            }
            return retValue;
        }
    }
```

Related Patterns

- Composite
  - Iterators can be applied to recursive structures
  - Factory Method
    - To instantiate specific iterator subclass
  - Memento
    - Used with the iterator, captures the state of an iteration

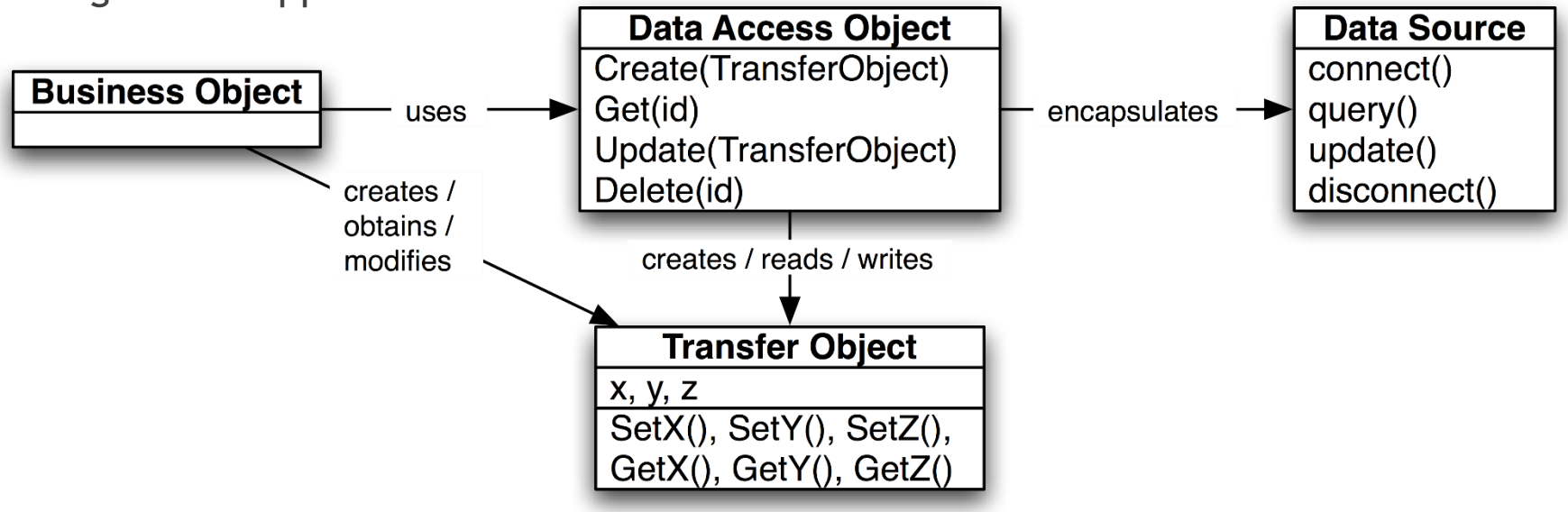
Data Access Object

DAO Pattern

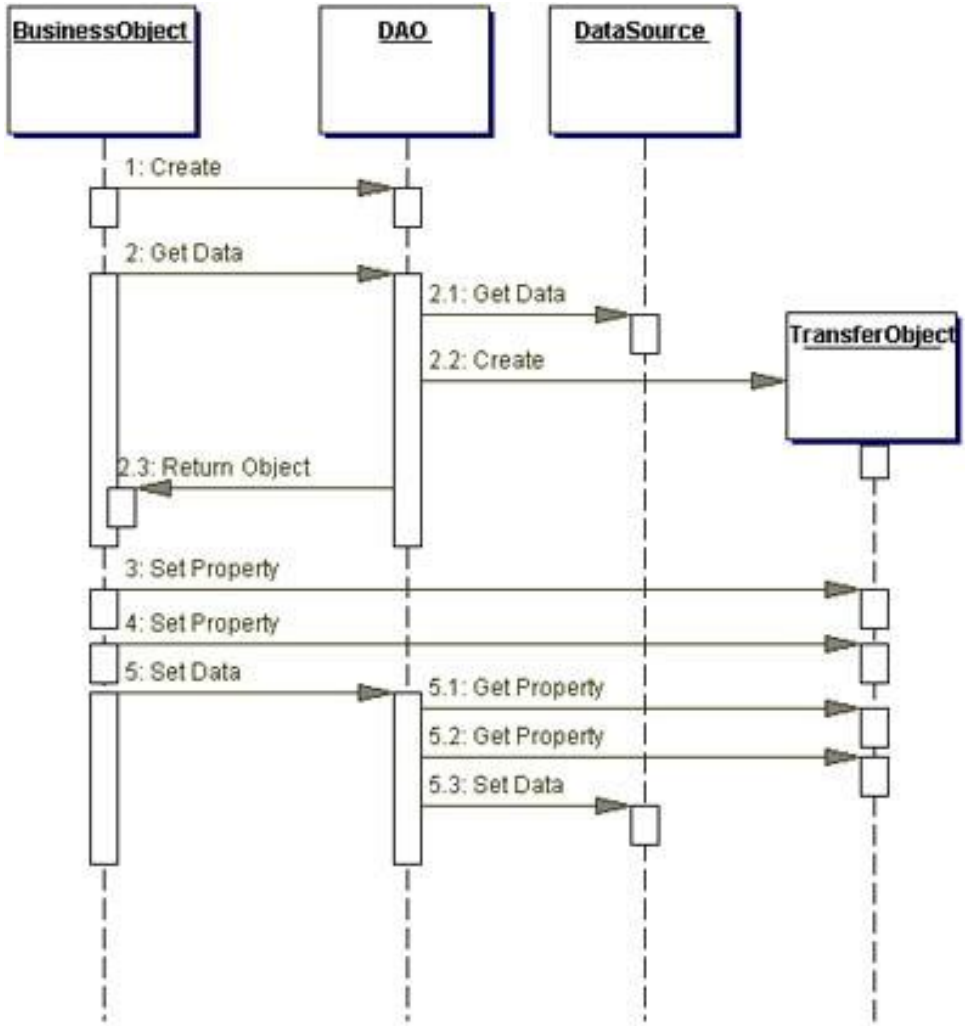
- **Problem**
  - Access to data varies greatly depending on the type of storage (relational DBMS, NoSQL DBMS, flat files, and so forth) and the vendor implementation.
- **Solution**
  - Use the Data Access Object (DAO) pattern to abstract and encapsulate all access to the data source. Each DAO manages the connection with the data source to obtain and store data.
  - The DAO implements the access mechanism required to work with the data source

DAO Design Pattern

- One DAO Class for each Entity Class
  - Abstracts CRUD (Create, Retrieve, Update, Delete) operations
- Benefits
  - Allows different storage implementations to be ‘plugged in’ with minimal impact to the rest of the system
  - Decouples persistence layer
  - Encourages and supports code reuse

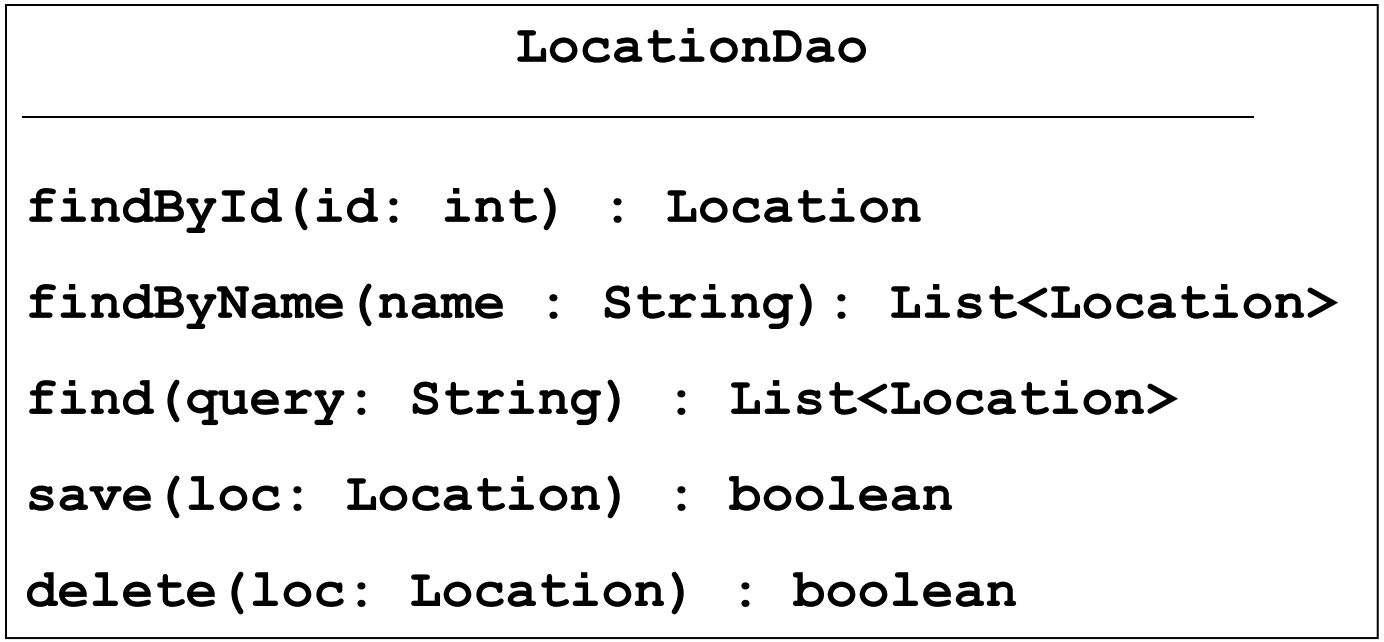


DAO Design Pattern



A DAO for a Location class

The "useful" methods depend on the domain class and the application.



Classification of GoF Design Pattern

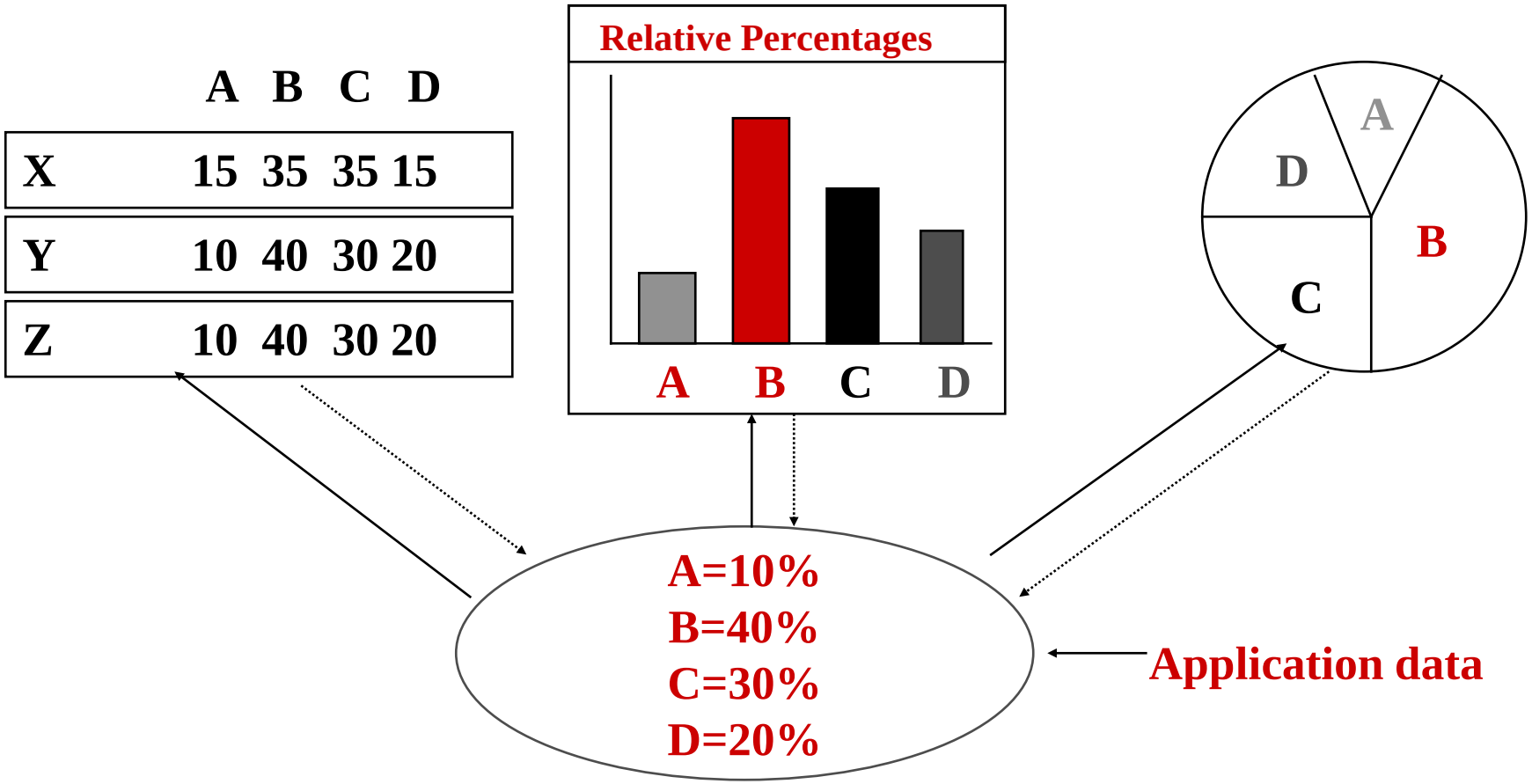
		Purpose		
		Creational	**Structural**	**Behavioral**
Scope	**Class**	Factory Method	Adapter (class)	Interpreter Template Method
	Object	Abstract Factory Builder Prototype Singleton	Adapter (object) Bridge Composite Decorator Flyweight Facade Proxy	Chain of Responsibility Command Iterator Mediator Memento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

		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

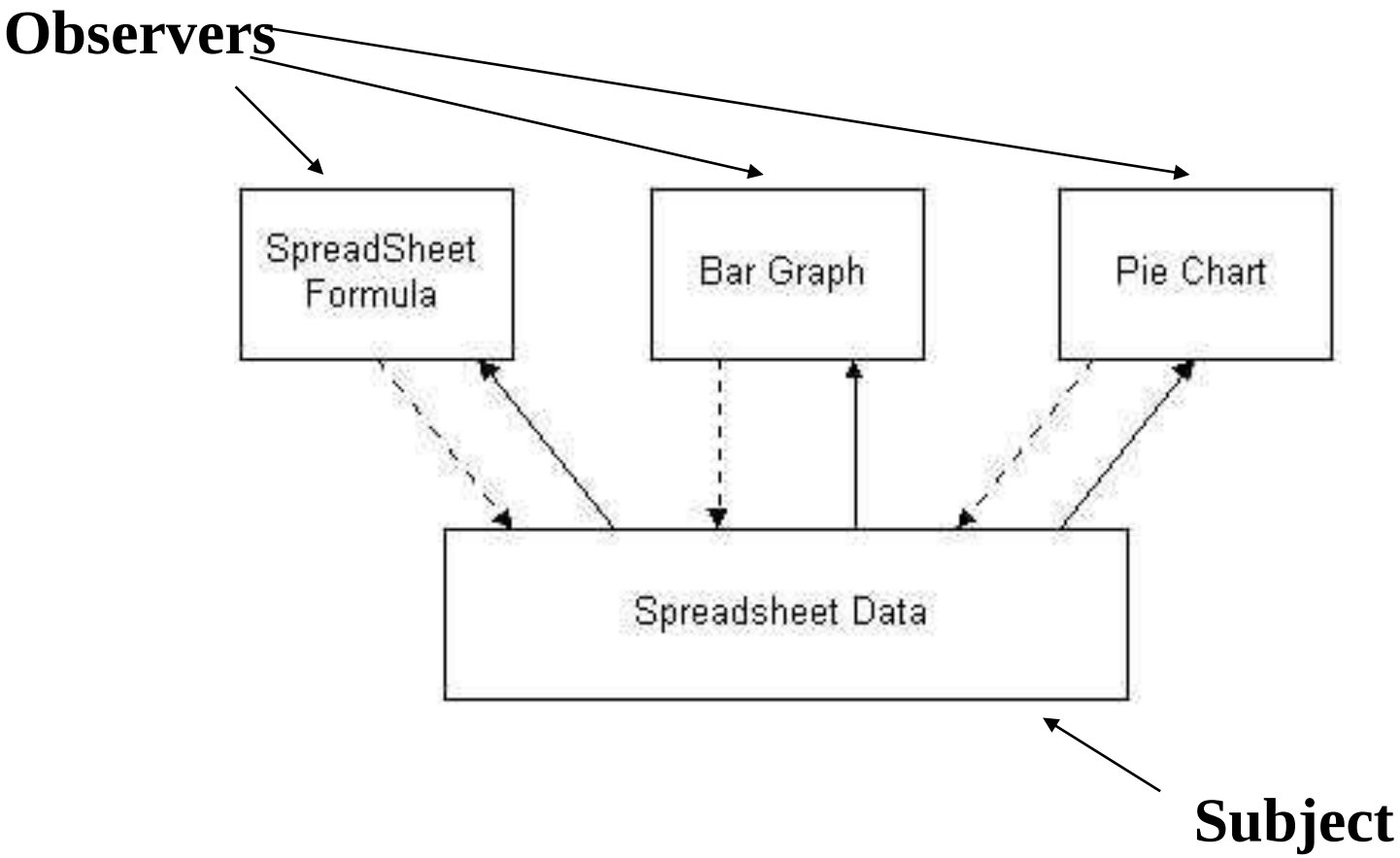
Observer

(Behavioral)

Patterns by Example:
Multiple displays enabled by Observer



Schematic Observer Example

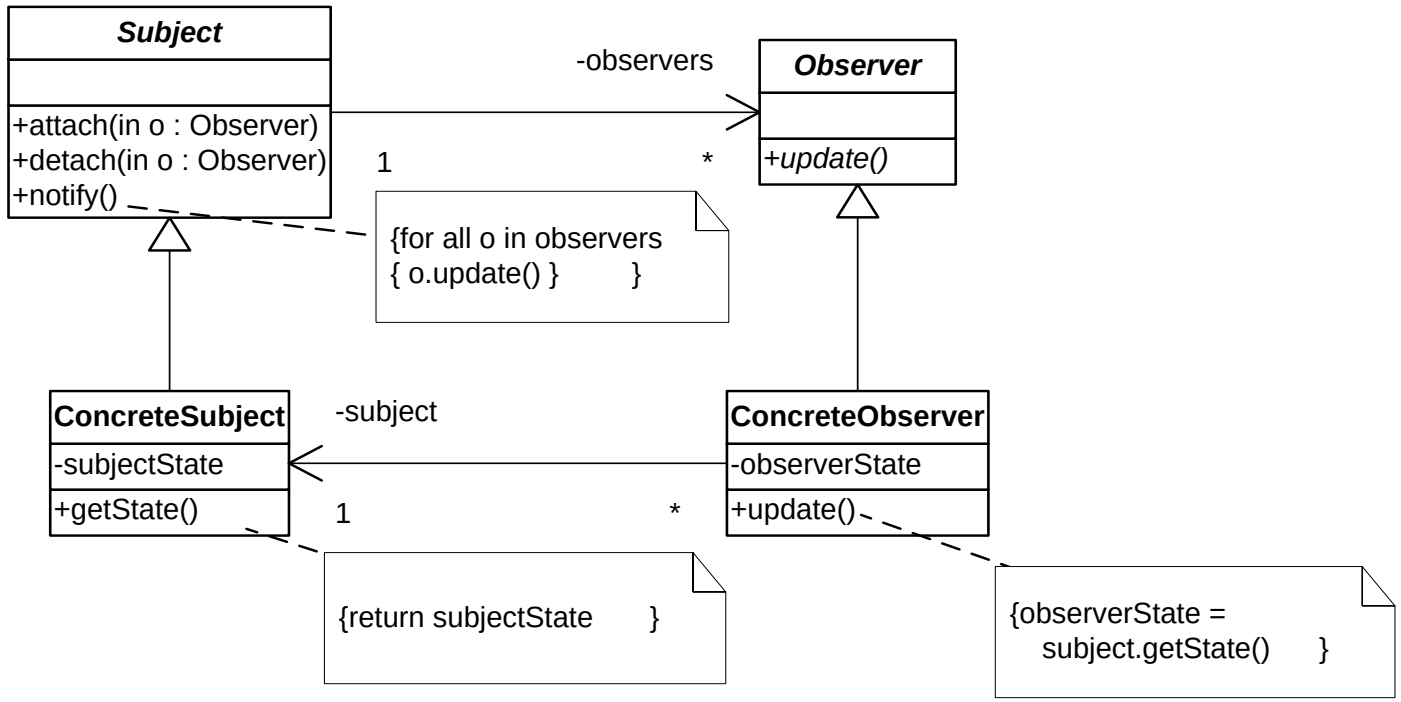


Observer (Behavioral)

- Intent
  - Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically
- Applicability
  - When an abstraction has two aspects, one dependent on the other
  - 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 notify other objects without making assumptions about who these objects are

Observer (Cont'd)

- Structure



Observer (Cont'd)

- Consequences
  - Modularity: subject and observers may vary independently
  - Extensibility: can define and add any number of observers
  - Customizability: different observers provide different views of subject
  - Unexpected updates: observers don't know about each other
  - Update overhead: might need hints
- Implementation
  - Subject-observer mapping
  - Dangling references
  - Avoiding observer-specific update protocols: the push and push/pull models
  - Registering modifications of interest explicitly

Observer - Subject Interface

```
// ISubject --> interface for the subject
public interface ISubject {

  // Registers an observer to the subject's notification list
  void RegisterObserver(IObserver observer);

  // Removes a registered observer from the subject's notification list
  void UnregisterObserver(IObserver observer);

  // Notifies the observers in the notification list of any change that occurred in
    the subject
  void NotifyObservers();
}
```

Observer - Observer Interface

```
// IObserver --> interface for the observer
public interface IObserver {

   /* Called by the subject to update the observer of any change. The method
   parameters can be modified to fit certain criteria */
   void Update();
}
```

Observer - Subject Impl (1)

```
// Subject --> class that implements the ISubject interface

using System.Collections;

public class Subject : ISubject {
    // use array list implementation for collection of observers
    private ArrayList observers;
    // decoy item to use as counter
    private int counter;
    // constructor
    public Subject() {
    observers = new ArrayList();
    counter = 0;
    }
public void RegisterObserver(IObserver observer)
{
   // if list does not contain observer, add
   if(!observers.Contains(observer))
        { observers.Add(observer); }
}
```

Observer - Subject Impl (2)

```
public void UnregisterObserver(IObserver observer) {
    // if observer is in the list, remove
    if(observers.Contains(observer))
      { observers.Remove(observer); }
}

public void NotifyObservers() {
    // call update method for every observer
    foreach(IObserver observer in observers)
      { observer.Update(); }
}

// use function to illustrate observer function
// the subject will notify only when the counter value is divisible by 5
public void Operate() {
    for(counter = 0; counter < 25; counter++)
        { if(counter % 5 == 0)
          { NotifyObservers(); }
        }
   }

}
```

Observer - Observer Implementation

```
// Observer --> Implements the IObserver

public class Observer : IObserver {
/* this will count the times the subject changed evidenced by the number of
   times it notifies this observer */

   private int counter;
   // a getter for counter
   public int Counter {
    get { return counter; }
   }

   public Observer() {
   counter = 0;
   }

   // counter is incremented with every notification
   public void Update() {
    counter += 1;
   }
}
```

		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

Factory Method

(Creational)

Da slides su System Design

```
class Traveler
{
    Car c=new Car();
    void startJourney()
    {
       c.move();
    }
}

class Car
{
  void move()
  {
     // logic...
  }
}
```

- Cosa succede se vogliamo riusare la nostra classe Traveler con un altro mezzo di trasporto che non sia Car?
- Cosa succede se vogliamo cambiare i metodi di Car?
- Traveler ha un attributo di tipo Car, e quindi un forte accoppiamento.

Esempio (cont.)

```
class Traveler
{
    Vehicle v;
    public void setV(Vehicle v)
    {
      this.v = v;
    }

    void startJourney()
    {
       v.move();
    }
}

Interface Vehicle
{
   void move();
}
```

```
class Car implements Vehicle
{
    public void move()
    {
         // logic
    }
}
```

```
class Bike implements Vehicle
{
    public void move()
    {
         // logic
    }
}
```

Factory Method
(Class Creational)

- Intent:
  - In Factory pattern, we instantiate objects without exposing the creation logic to the client, and we refer to newly created object casting it to a common interface.
- Motivation:
  - Framework use abstract classes to define and maintain relationships between objects
  - Framework has to create objects as well - must instantiate classes but only knows about abstract classes - which it cannot instantiate
  - Factory method encapsulates knowledge of which subclass to create - moves this knowledge out of the framework

Applicability

- Use the Factory Method pattern when
  - a class can´t anticipate the class of objects it must create.
  - classes delegate responsibility to one of several helper subclasses, and you want to localize the knowledge of which helper subclass is the delegate.

Participants

- Product
  - Defines the interface of objects the factory method creates
- ConcreteProduct
  - Implements the product interface
- Creator
  - Declares the factory method which returns object of type *Product*

Factory Pattern

- Example in Eclipse on Traveler
- We are going to create a *Shape* interface and concrete classes implementing the *Shape* interface. A factory class *ShapeFactory* is defined as a next step.
- *FactoryPatternDemo*, our demo class will use S*hapeFactory* to get a *Shape* object. It will pass information (*CIRCLE / RECTANGLE / SQUARE*) to *ShapeFactory* to get the type of object it needs.

Factory Example

```
public class VehicleFactory {

    public Vehicle instantiateVehicle(){
        // Logic to choose the instance

        if (weather.rains())
            return new Car();
        else
            return new Bike();
    }

}
```

Factory Example (2)

```
public class BusinessLogic {

    public static void main(String[] args) {

        VehicleFactory vehicleFactory = new
VehicleFactory();

        Traveler t = new Traveler();
        t.setV(vehicleFactory.instantiateVehicle());
        t.startJourney();
    }

}
```

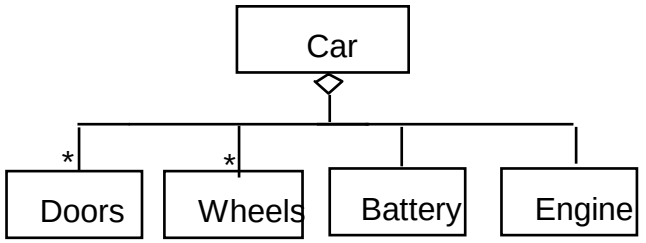
		Purpose		
		Creational	**Structural**	**Behavioral**
Scope	**Class**	Factory Method	Adapter (class)	Interpreter Template Method
	Object	Abstract Factory Builder Prototype Singleton	Adapter (object) Bridge Composite Decorator Flyweight Facade Proxy	Chain of Responsibility Command Iterator Mediator Memento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

Composite

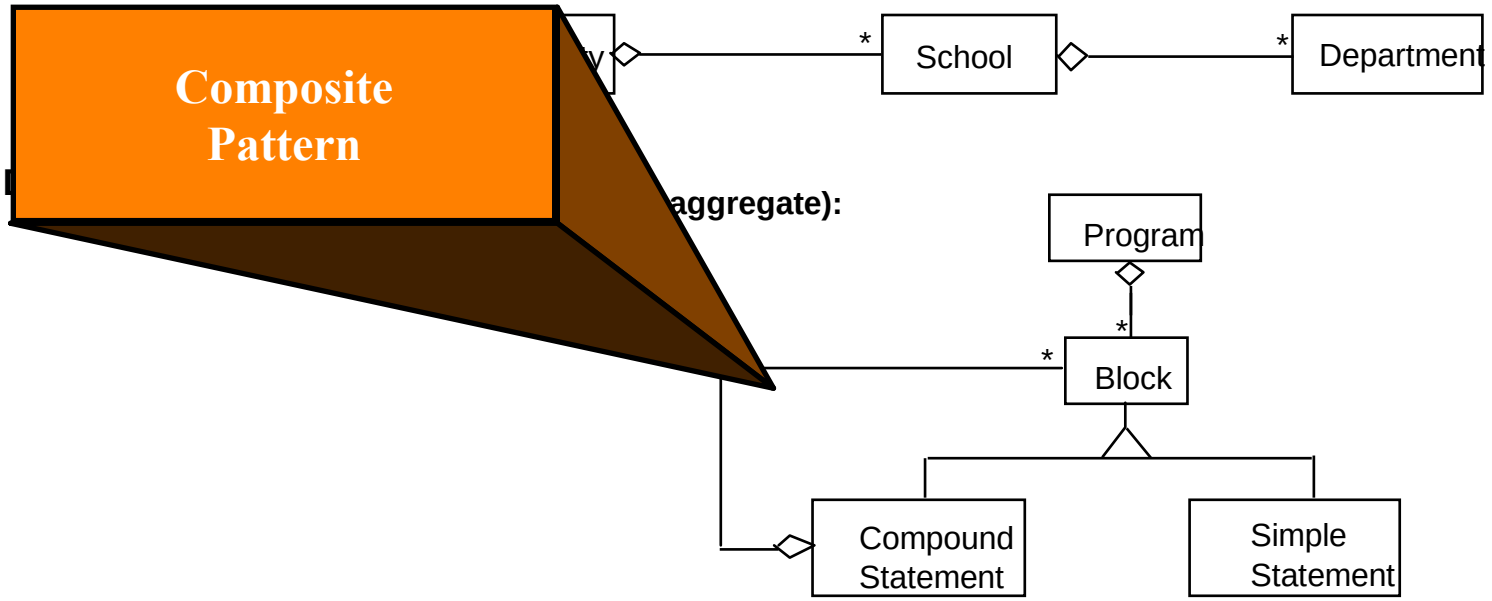
(Structural)

Review: Modeling Typical Aggregations

Fixed Structure:

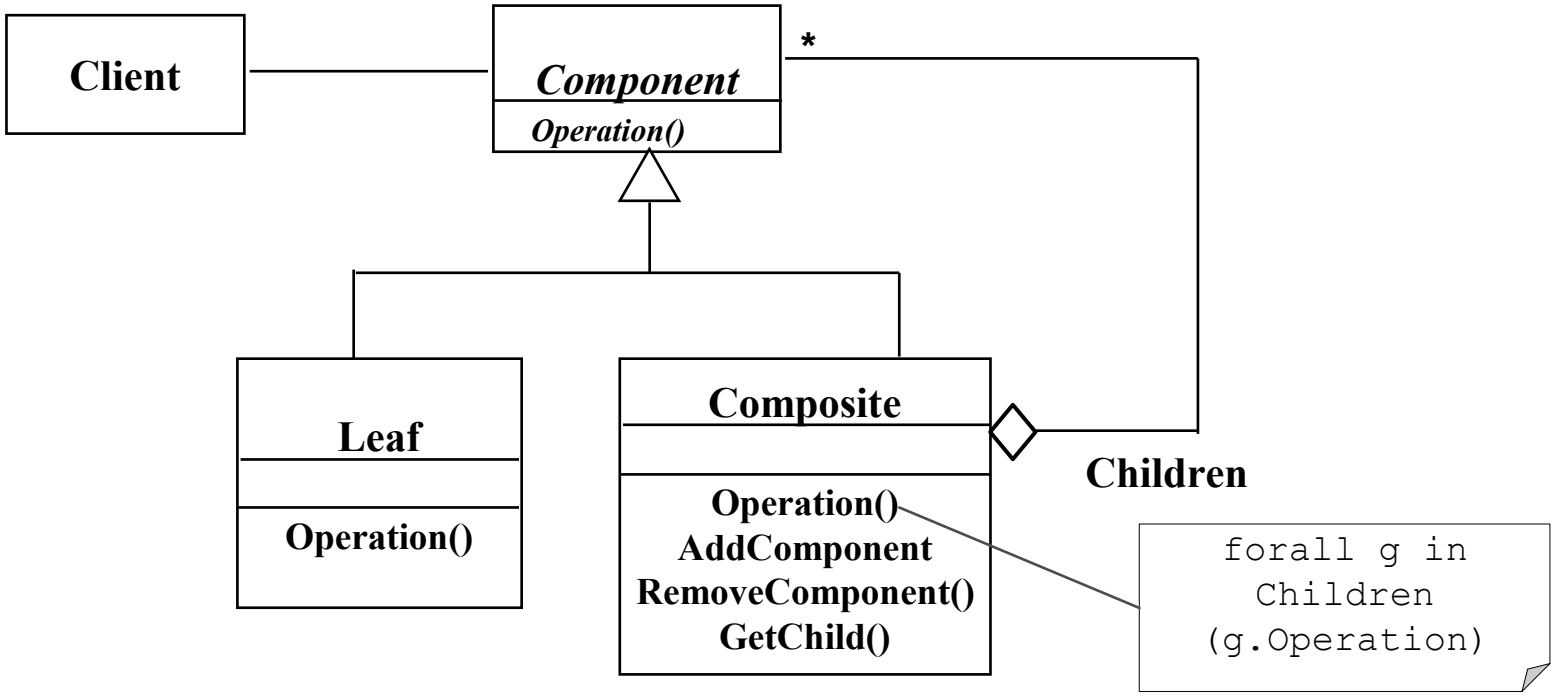


Organization Chart (variable aggregate):



Composite Pattern

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



Composite Implementation (1)

```
/** "Component" */
interface Graphic { //Prints the graphic.
   public void print();
}

/** "Composite" */
class CompositeGraphic implements Graphic {
//Collection of child graphics.
private List<Graphic> mChildGraphics = new ArrayList<Graphic>();

 //Prints the graphic.
public void print() {
   for (Graphic graphic : mChildGraphics)
    { graphic.print(); }
}
```

Composite Implementation (2)

```
 //Adds the graphic to the composition.
   public void add(Graphic graphic)
   { mChildGraphics.add(graphic); }

   //Removes the graphic from the composition.
   public void remove(Graphic graphic)
   { mChildGraphics.remove(graphic); }
}

/** "A Leaf" */
class Ellipse implements Graphic {
    //Prints the graphic.
    public void print() {
      System.out.println("Ellipse");
    }
}
```

Object Design

Object Design

- L’Object Design è la fase nel ciclo di vita del software in cui si definiscono le scelte finali prima dell’implementazione
- In questa fase, l’analista deve scegliere tra i differenti modi di implementare i modelli di analisi, rispettando requisiti non funzionali e criteri di design.
  - Si specificano quali classi implementeranno le funzionalità descritte in analisi, come comincheranno tra loro, etc…
- E’ il passo finale prima dell’implementazione

Goal dell'Object Design

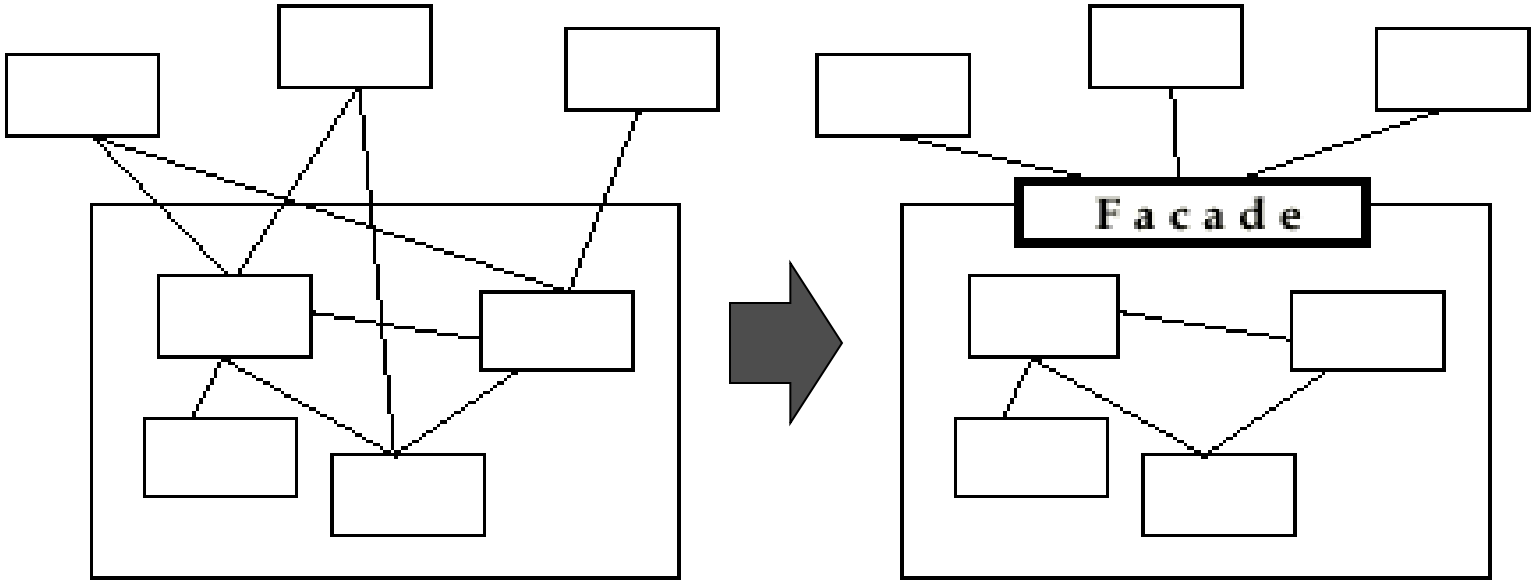
- Identification of existing components
- Full definition of relations
- Full definition of classes (System Design => Service, Object Design => API)
- Specifying the contract for each component
- Choosing algorithms and data structures
- Identifying possibilities of reuse
- Detection of solution-domain classes
- Optimization
- Packaging

		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

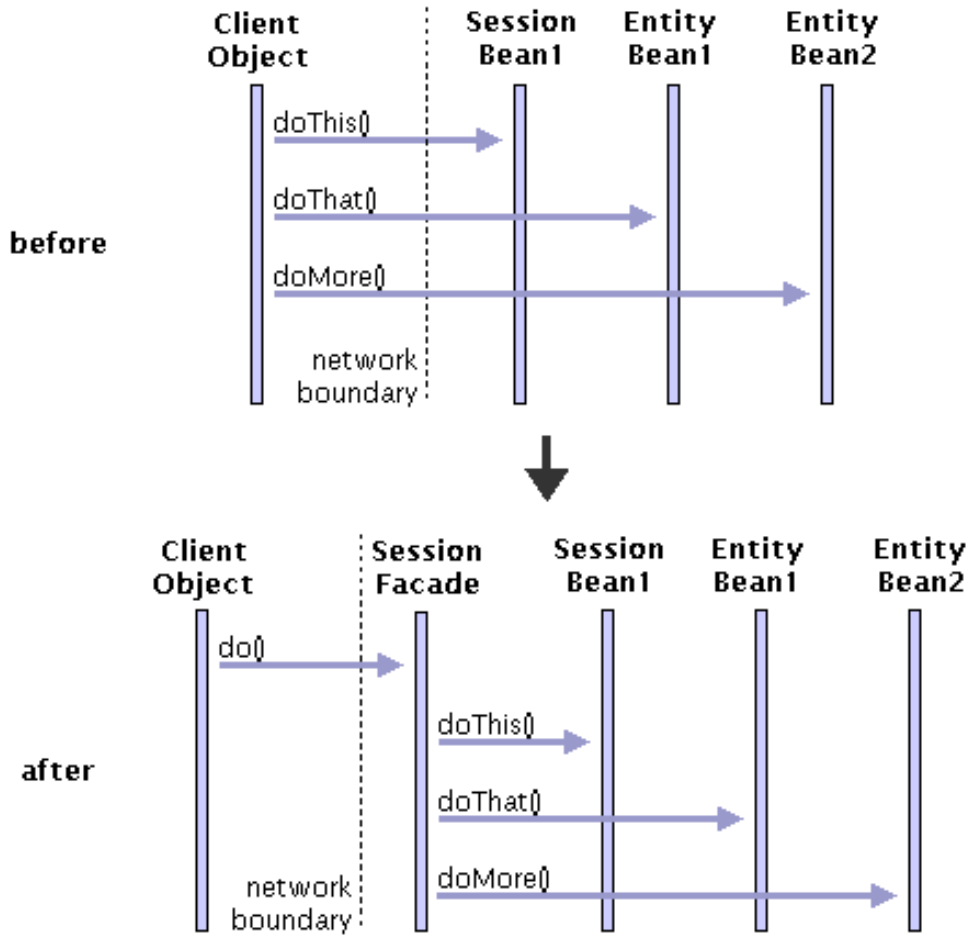
Facade

Facade Pattern

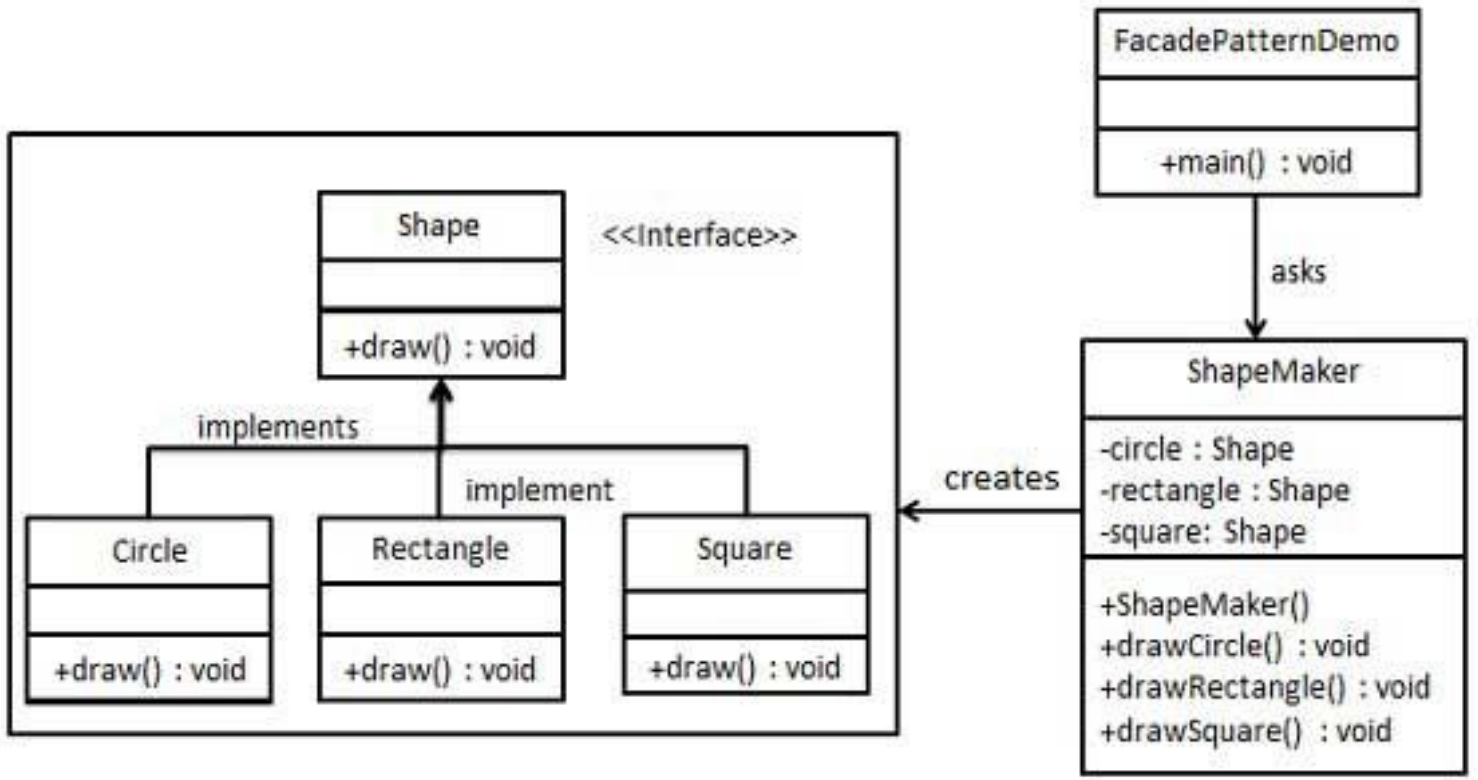
- Provides a unified interface to a set of objects in a subsystem.
- A facade defines a higher-level interface that makes the subsystem easier to use (i.e. it abstracts out the gory details)
- Facades allow us to provide a closed architecture



Façade


Client Object
Session Bean1
Entity Bean1
Entity Bean2
doThis()
doThat()
doMore()
network boundary
before
Client Object
Session Facade
Session Bean1
Entity Bean1
Entity Bean2
do()
doThis()
doThat()
doMore()
network boundary
after

Façade Pattern Example



Façade Pattern Example (2)

```
public interface Shape {
   void draw();
}

public class Rectangle implements Shape {

   @Override
   public void draw() {
      System.out.println("Rectangle::draw()");
   }
}

public class Square implements Shape {

   @Override
   public void draw() {
      System.out.println("Square::draw()");
   }
}

public class Circle implements Shape {

   @Override
   public void draw() {
      System.out.println("Circle::draw()");
   }
}
```

Façade Pattern Example (3)

```
public class ShapeMaker {
   private Shape circle;
   private Shape rectangle;
   private Shape square;

   public ShapeMaker() {
      circle = new Circle();
      rectangle = new Rectangle();
      square = new Square();
   }

   public void drawCircle(){
      circle.draw();
   }
   public void drawRectangle(){
      rectangle.draw();
   }
   public void drawSquare(){
      square.draw();
   }
}
```

```
public class FacadePatternDemo {
   public static void main(String[]
    args) {
      ShapeMaker shapeMaker = new
    ShapeMaker();

      shapeMaker.drawCircle();
      shapeMaker.drawRectangle();
      shapeMaker.drawSquare();

   }
}
```

Ideal Structure of a Subsystem: Façade + Adapter or Bridge

- 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
- Realization of Interface Object: Facade
  - Provides the interface to the subsystem
- Interface to existing systems: Adapter or Bridge
  - Provides the interface to existing system (legacy system)
  - The existing system is not necessarily object-oriented!

Approfondimenti sul Riuso, e relazioni con i DP

The use of inheritance

- Inheritance is used to achieve 2 different goals
  - Description of Taxonomies
  - Reuse Code
- Identification of taxonomies
  - Used during requirements analysis.
  - Activity: identify application domain objects that are hierarchically related
  - Goal: make the analysis model more understandable
- Service specification
  - Used during object design
  - Goal: increase reusability, enhance modifiability and extensibility
- Inheritance is found either by specialization or generalization, when dealing with Taxonomies

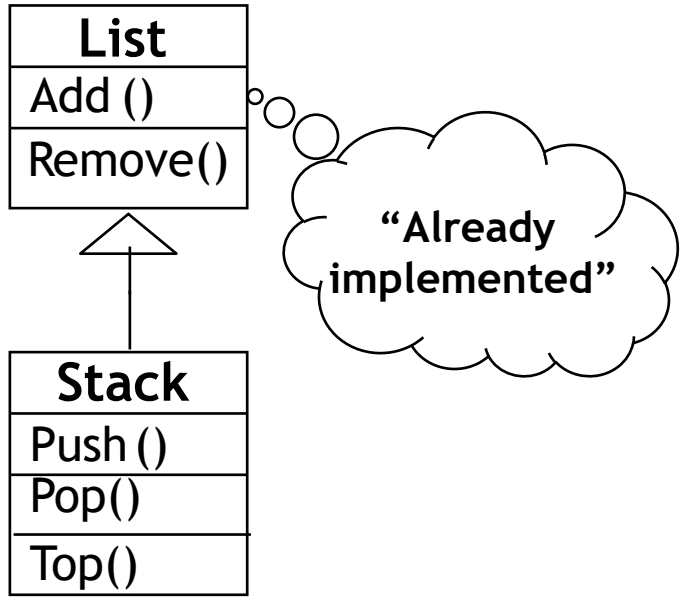
Implementation Inheritance vs Interface Inheritance

- 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
  - Also called subtyping
  - Inherit from an abstract class with all operations specified, but not yet implemented

Reuse by Implementation Inheritance

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

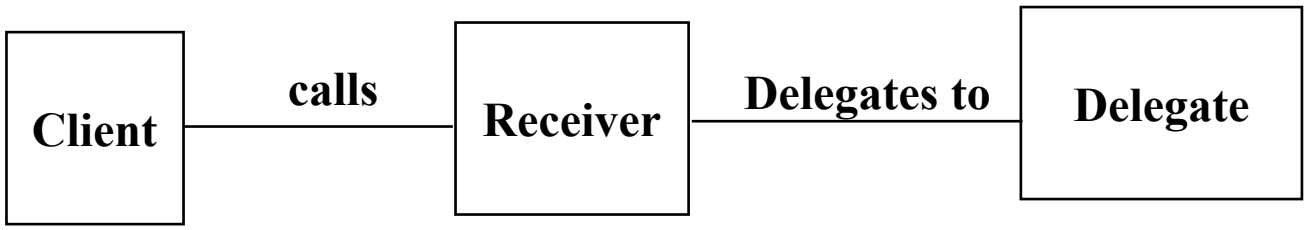


Problem with implementation inheritance

- Two main problems:
  1. Inheritance is the highest possible coupling between two classes
  2. With Inheritance, the subclass exposes also the superclass methods
- Some of the inherited operations might exhibit unwanted behavior.
  - What happens if the Stack user calls Remove() instead of Pop()?

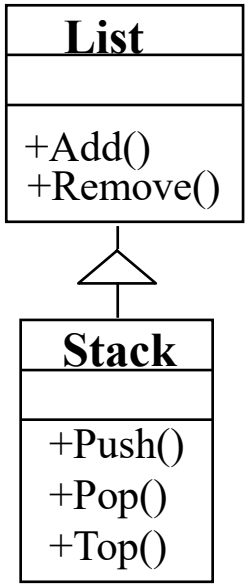
Delegation

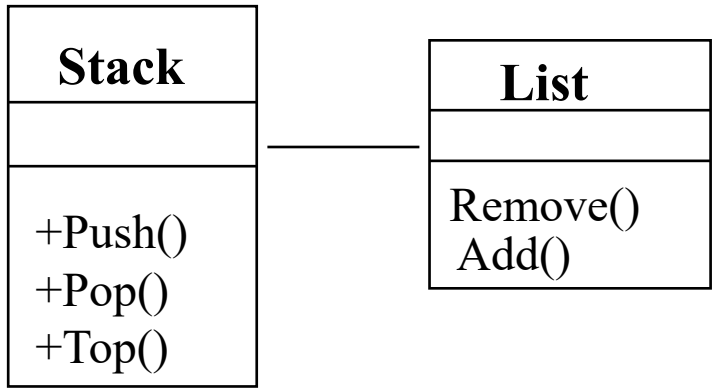
- Delegation is a way of making composition (for example aggregation) as powerful for reuse as inheritance
- In Delegation two objects are involved in handling a request
  - A receiving object delegates operations to its delegate.
  - The developer can make sure that the receiving object does not allow 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?





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 a subclass to the details of its parent class
    - Any change in the parent class implementation forces the subclass to change (which requires recompilation of both)

Design Heuristics

- Avoid to use implementation inheritance, always use interface inheritance
- A subclass should never hide operations implemented in a superclass
- If you are tempted to use implementation inheritance, use delegation instead

		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 (Method)

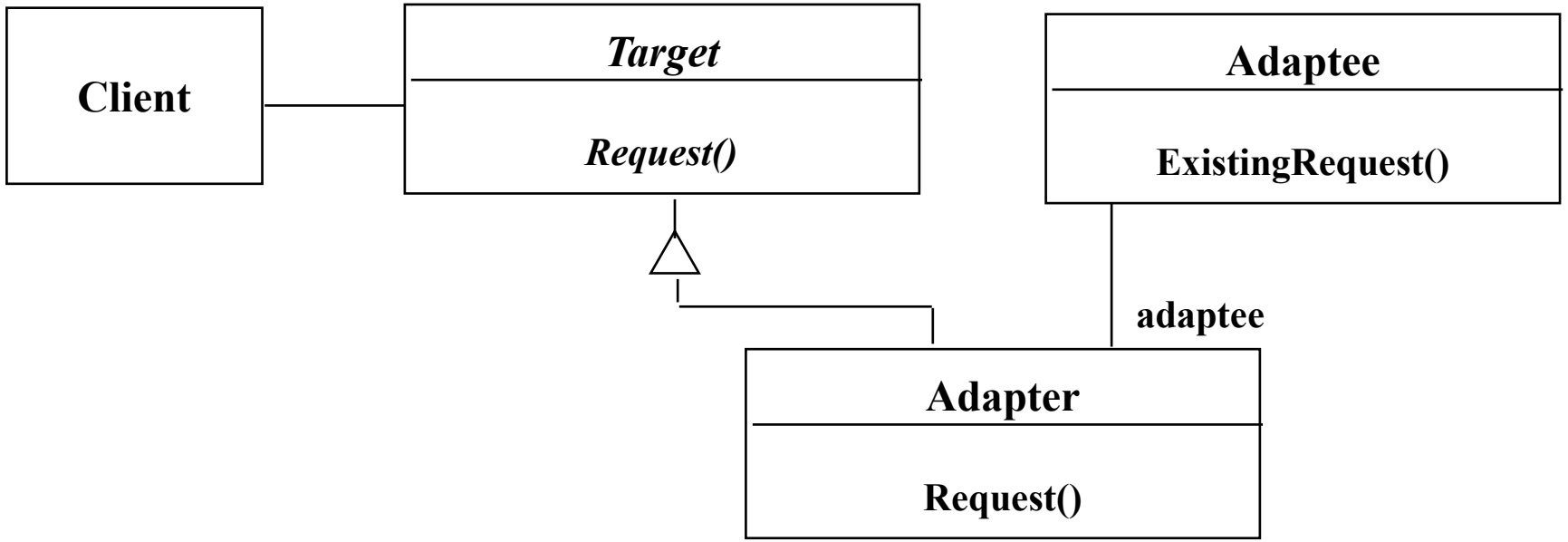
Structural

Adapter Method Pattern

- Convert the interface of a class into another interface clients expect.
  - Adapter lets classes work together that couldn’t otherwise because of incompatible interfaces
- Used often to provide a new interface to existing legacy components (Interface engineering, reengineering).
- Also known as a *wrapper*
- A real life example could be a case of card reader which acts as an adapter between memory card and a PC. You plugin the memory card into card reader and card reader into the PC so that memory card can be read via PC.

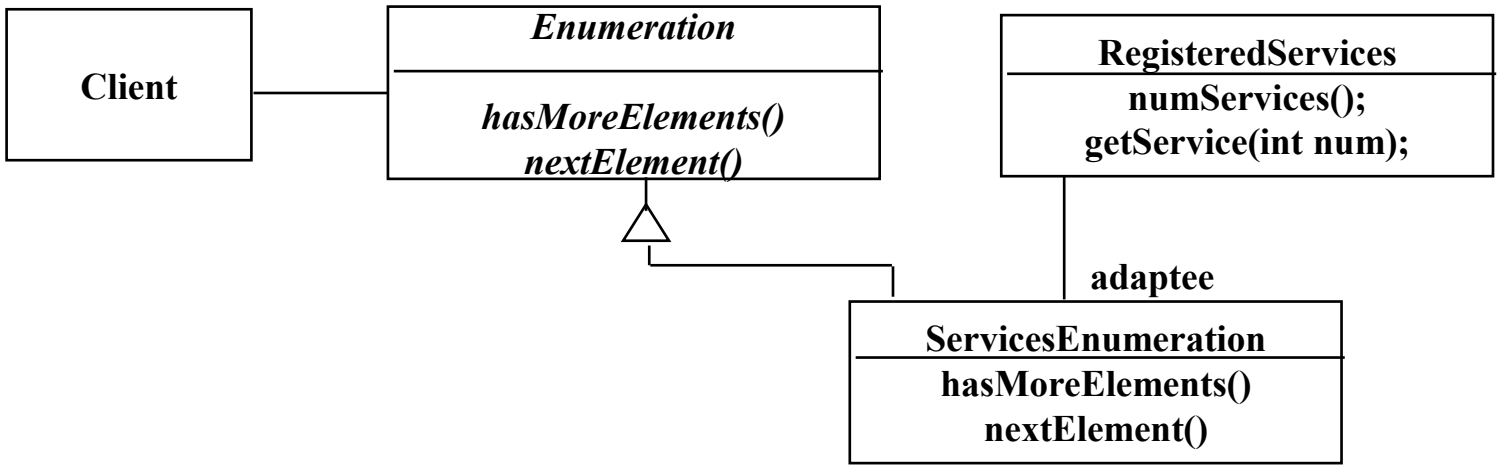
Adapter Method Pattern

- 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 Method example

```
public class ServicesEnumeration implements Enumeration {
  public boolean hasMoreElements() {
    return this.currentServiceIndex <= adaptee.numServices();
  }
  public Object nextElement() {
    if (!this.hasMoreElements()) {
       throw new NoSuchElementException();
    }
    return adaptee.getService(this. currentServiceIndex++);
  }
```



		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

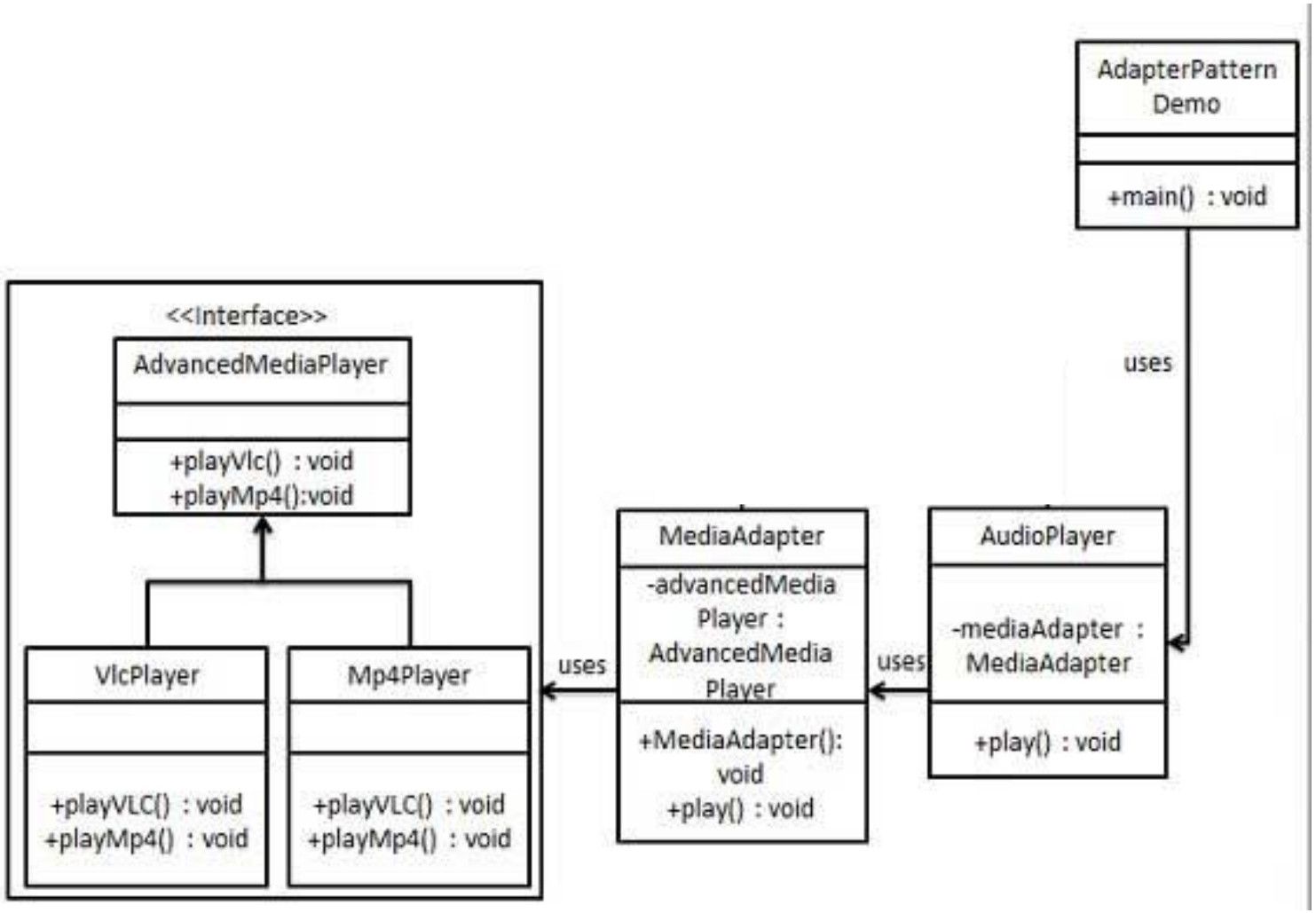
Adapter (Class)

Structural

Adapter Class Pattern

- The adapter (Class) pattern is useful in situations where one or more already existing classes provide some or all of the services you need but do not use the interface you need.
- We show the Adapter (Class) pattern via an example in which an audio player device can play mp3 files only and wants to use an already existing advanced audio player capable of playing vlc and mp4 files.

Adapter Class Pattern Example



Adapter Class Pattern Example (1)

```
public class AudioPlayer {
   MediaAdapter mediaAdapter;

   public void play(String audioType, String fileName) {

      //inbuilt support to play mp3 music files
      if(audioType.equalsIgnoreCase("mp3")){
         System.out.println("Playing mp3 file. Name: " + fileName);

      }

      //mediaAdapter is providing support to play other file formats
      else if(audioType.equalsIgnoreCase("vlc") ||
   audioType.equalsIgnoreCase("mp4")){
         mediaAdapter = new MediaAdapter(audioType);
         mediaAdapter.play(audioType, fileName);
      }

      else{
         System.out.println("Invalid media. " + audioType + " format
   not supported");
      }
   }
}
```

Adapter Class Pattern Example (2)

```
public class MediaAdapter {

   AdvancedMediaPlayer advancedMusicPlayer;

   public MediaAdapter(String audioType){

      if(audioType.equalsIgnoreCase("vlc") ){
         advancedMusicPlayer = new VlcPlayer();

      }else if (audioType.equalsIgnoreCase("mp4")){
         advancedMusicPlayer = new Mp4Player();
      }
   }

   public void play(String audioType, String fileName) {

      if(audioType.equalsIgnoreCase("vlc")){
         advancedMusicPlayer.playVlc(fileName);
      }
      else if(audioType.equalsIgnoreCase("mp4")){
         advancedMusicPlayer.playMp4(fileName);
      }
   }
}
```

		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

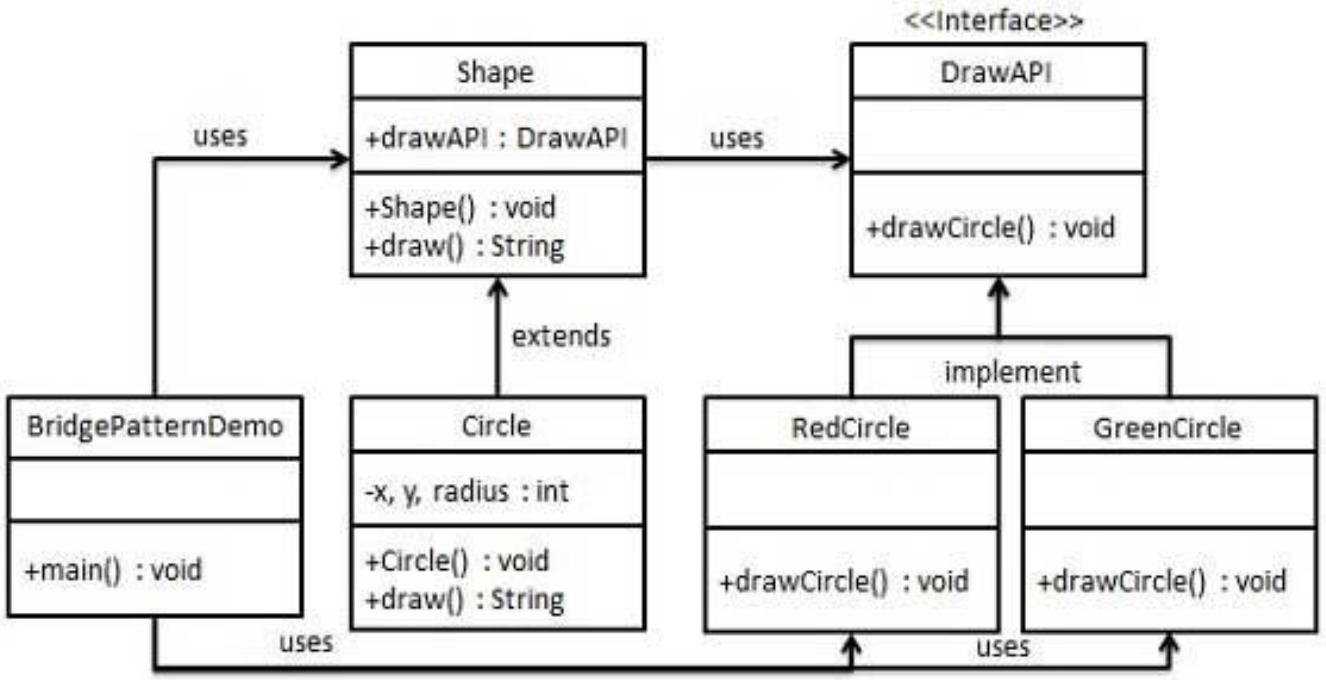
Bridge

Bridge Pattern

- Use a bridge to “decouple an abstraction from its implementation so that the two can vary independently”. (From [Gamma et al 1994])
- Allows different implementations of an interface to be decided upon dynamically.
- The bridge pattern is useful when both the class and what it does vary often. The class itself can be thought of as the implementation and what the class can do as the abstraction.
- When a class varies often, the Bridge Pattern become very useful, because changes to a program's code can be made easily with minimal prior knowledge about the program.

Bridge Pattern Example

- A circle can be drawn in different colors using same abstract class method but different bridge implementer classes.



Bridge Pattern Example (1)

```
public interface DrawAPI {
   public void drawCircle(int radius, int x, int y);
}

public class RedCircle implements DrawAPI {
   @Override
   public void drawCircle(int radius, int x, int y) {
      System.out.println("Drawing a Red Circle");
   }
}

public class GreenCircle implements DrawAPI {
   @Override
   public void drawCircle(int radius, int x, int y) {
      System.out.println("Drawing a Green Circle");
   }
}
```

Bridge Pattern Example (2)

```
public abstract class Shape {
   protected DrawAPI drawAPI;

   protected Shape(DrawAPI drawAPI){
      this.drawAPI = drawAPI;
   }
   public abstract void draw();
}

public class Circle extends Shape {
   private int x, y, radius;

   public Circle(int x, int y, int radius, DrawAPI drawAPI) {
      super(drawAPI);
      this.x = x;
      this.y = y;
      this.radius = radius;
   }

   public void draw() {
      drawAPI.drawCircle(radius,x,y);
   }
}
```

Bridge Pattern Example (3)

```
public class BridgePatternDemo {
   public static void main(String[] args) {
      Shape redCircle = new Circle(100,100, 10, new RedCircle());
      Shape greenCircle = new Circle(100,100, 10, new GreenCircle());

      redCircle.draw();
      greenCircle.draw();
   }
}
```

- Consequence: the way a circle can be drawn can vary independently from the Circle class, and the Circle class can vary without impact on the .

Adapter vs Bridge

- Similarities:
  - Both 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.
    - Green field engineering of an "extensible system"
    - New "beasts" can be added to the "object zoo", even if these are not known at analysis or system design time.

		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

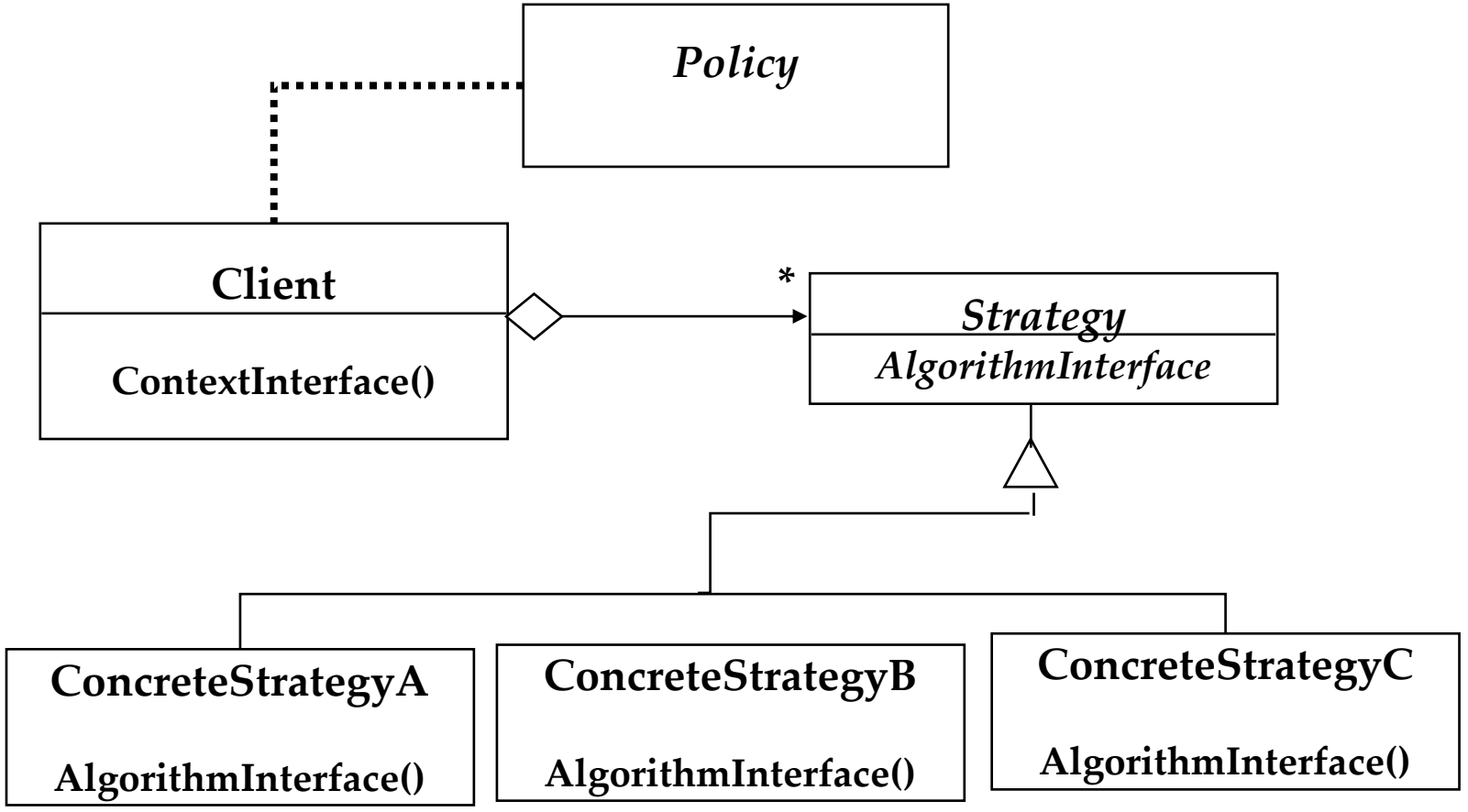
Strategy

Strategy Pattern

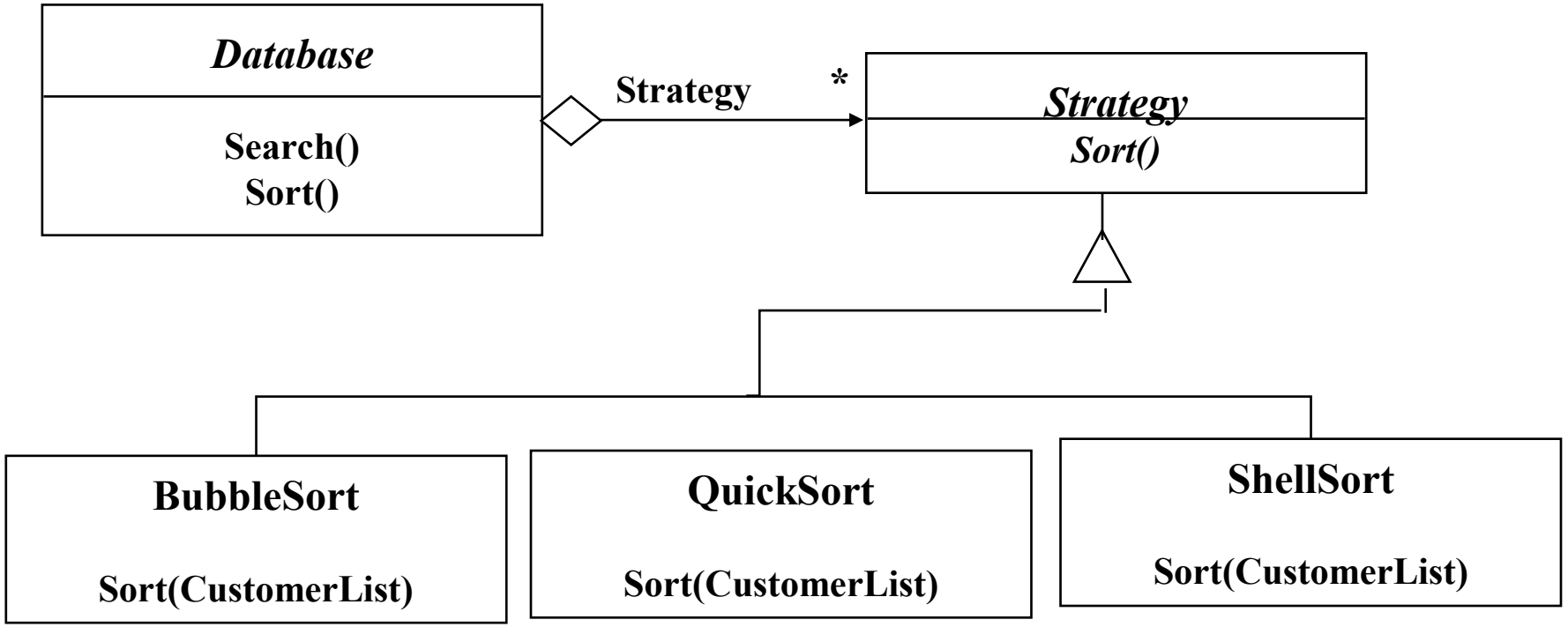
- Many different algorithms exists for the same task
- Examples:
  - Breaking a stream of text into lines
  - Parsing a set of tokens into an abstract syntax tree
  - Sorting a list of customers
- The different algorithms will be appropriate at different times
- We don’t want to support all the algorithms 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 a Strategy Pattern in a Database Application



Applicability of Strategy Pattern

- Many related classes differ only in their behavior. Strategy allows 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, and selected at runtime

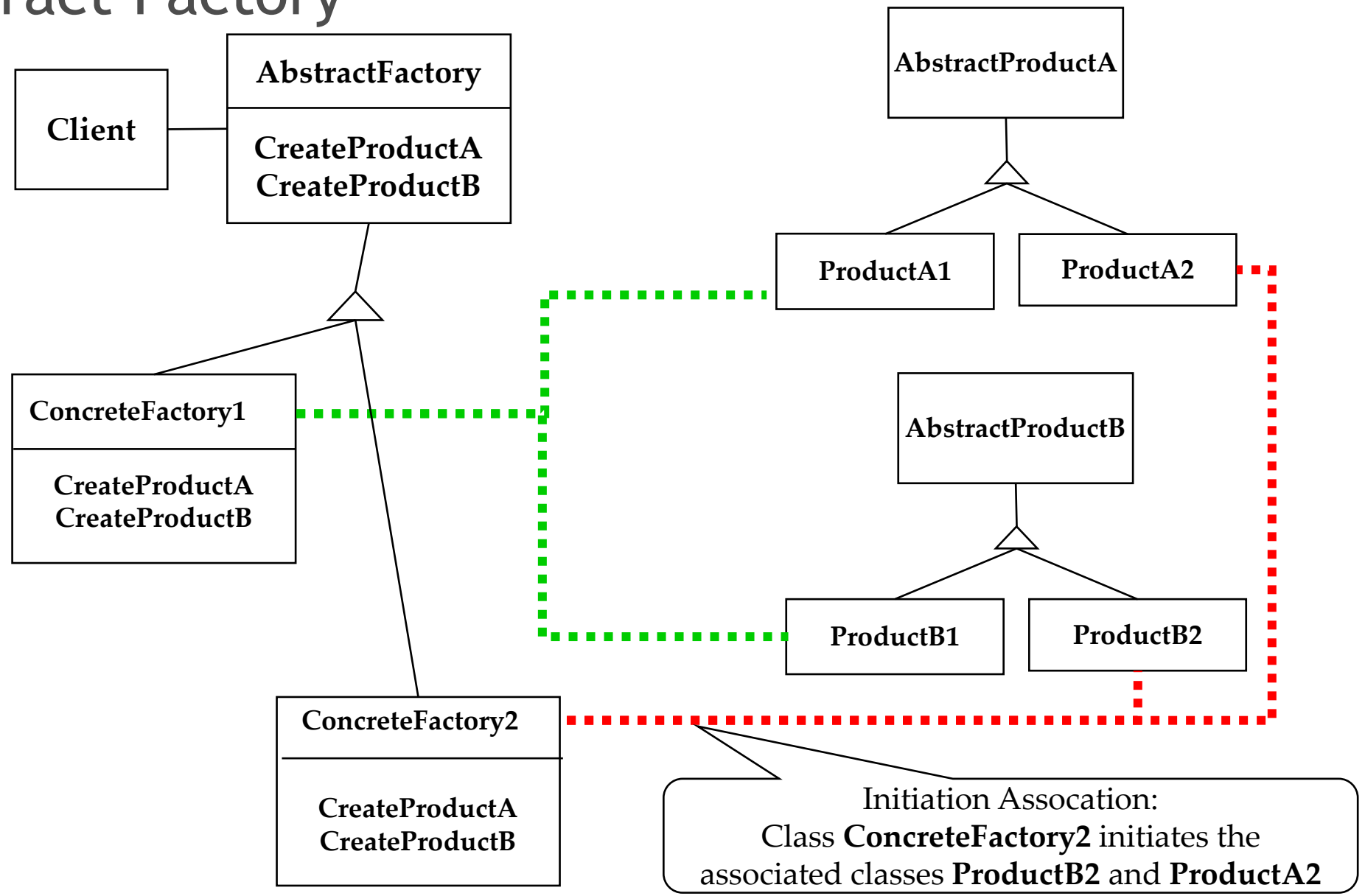
		Purpose		
		Creational	**Structural**	**Behavioral**
Scope	**Class**	Factory Method	Adapter (class)	Interpreter Template Method
	Object	Abstract Factory Builder Prototype Singleton	Adapter (object) Bridge Composite Decorator Flyweight Facade Proxy	Chain of Responsibility Command Iterator Mediator Memento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

Abstract Factory

Abstract Factory Motivation

- Consider a user interface toolkit that supports multiple looks and feel standards such as X, Windows Vista 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?

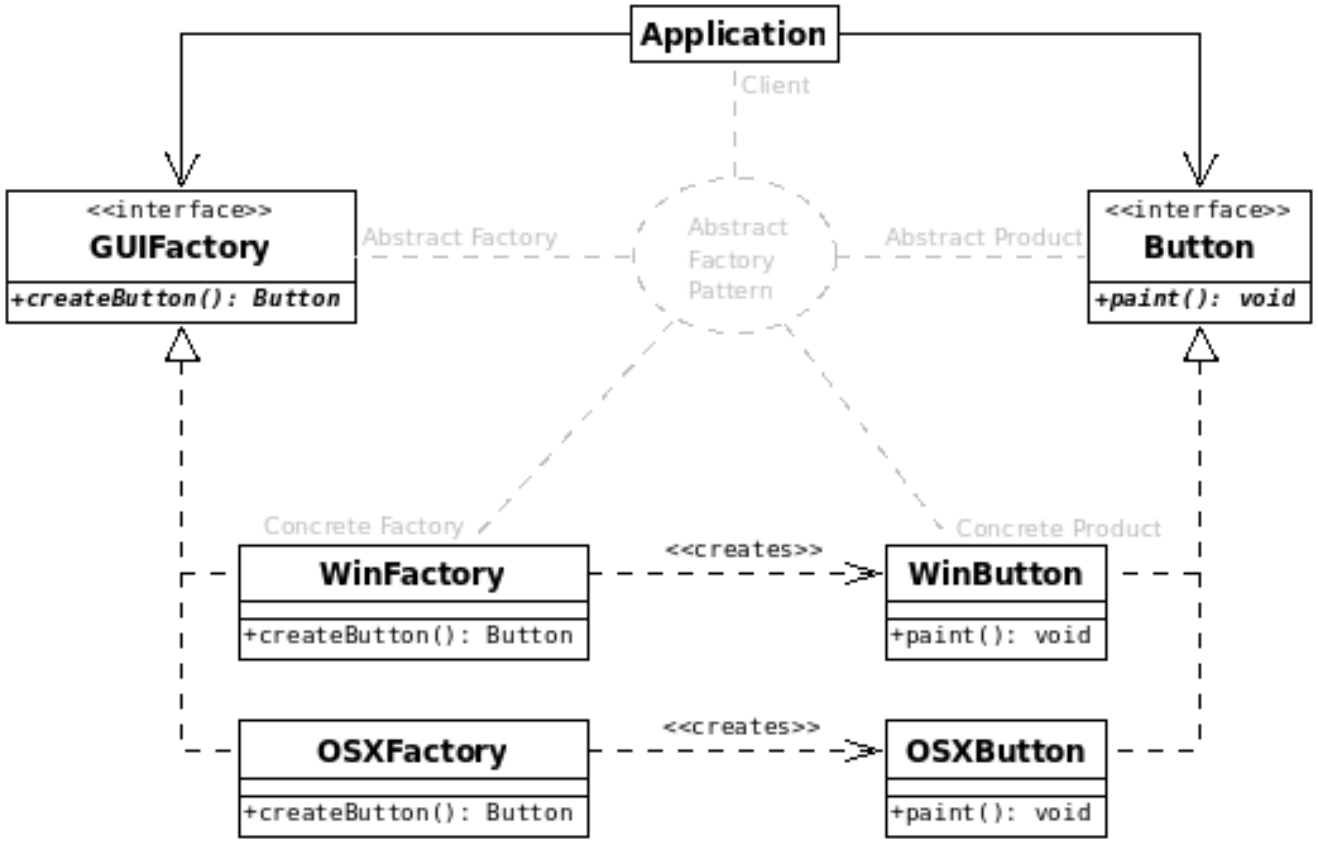
Abstract Factory



Applicability 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 of multiple family of products
  - 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.

Example: a GUI



Example: a GUI – The Code

```
abstract class GUIFactory {

    public static GUIFactory
   getFactory() {

        int sys =
   readFromConfigFile("OS_TYPE");

        if (sys == 0)

            return new WinFactory();

        else

            return new OSXFactory();

}

    public abstract Button
   createButton();

}
```

```
class WinFactory extends GUIFactory
    {

    public Button createButton() {

        return new WinButton();}

}
```

```
class OSXFactory extends GUIFactory
    {

    public Button createButton() {

        return new OSXButton(); }

}
```

Example: a GUI – The Code (2)

```
abstract class Button {               // ANCHE INTERFACCIA
    public abstract void paint();}

class WinButton extends Button {
    public void paint() {
        System.out.println("Sono un WinButton: "); }
}
class OSXButton extends Button {
    public void paint() {
        System.out.println("Sono un OSXButton: ");}
}

public class Application {
    public static void main(String[] args) {
        GUIFactory factory = GUIFactory.getFactory();
        Button button = factory.createButton();
        button.paint();
}
```

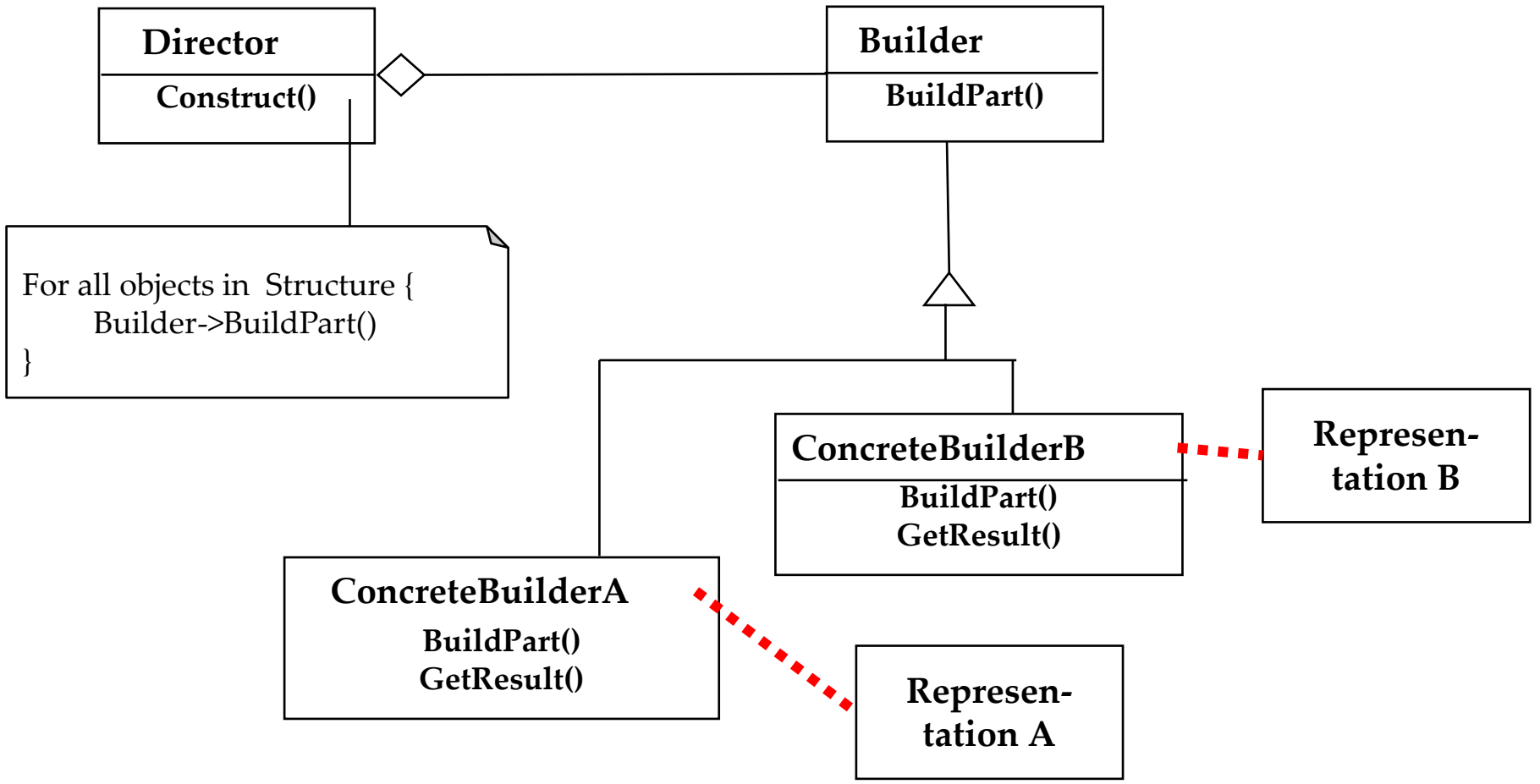
		Purpose		
		Creational	**Structural**	**Behavioral**
Scope	**Class**	Factory Method	Adapter (class)	Interpreter Template Method
	Object	Abstract Factory Builder Prototype Singleton	Adapter (object) Bridge Composite Decorator Flyweight Facade Proxy	Chain of Responsibility Command Iterator Mediator Memento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

Builder

Builder Pattern Motivation

- The intention is to separate the construction of a complex object from its representation so that the same construction process can create different representations.
- 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, ….)
    - 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 Pattern



When do you use the Builder Pattern?

- The creation of a complex product must be independent of the particular parts that make up the product
  - In particular, the creation process should not know about the assembly process (how the parts are put together to make up the product)
- The creation process must allow different representations for the object that is constructed.
- Examples:
  - A house with one floor, 3 rooms, 2 hallways, 1 garage and three doors.
  - A skyscraper with 50 floors, 15 offices and 5 hallways on each floor.

Example 1

```
class Pizza {
        string dough;
        string sauce;
        string topping;
        public Pizza() {}
        public void SetDough( string d){ dough = d;}
        public void SetSauce( string s){ sauce = s;}
        public void SetTopping( string t){ topping = t;}
}

//Abstract Builder
abstract class PizzaBuilder {
        protected Pizza pizza;
        public PizzaBuilder(){}
        public Pizza GetPizza(){ return pizza; }
        public void CreateNewPizza() { pizza = new Pizza(); }
        public abstract void BuildDough();
        public abstract void BuildSauce(); public abstract void BuildTopping();
}
```

Example 1

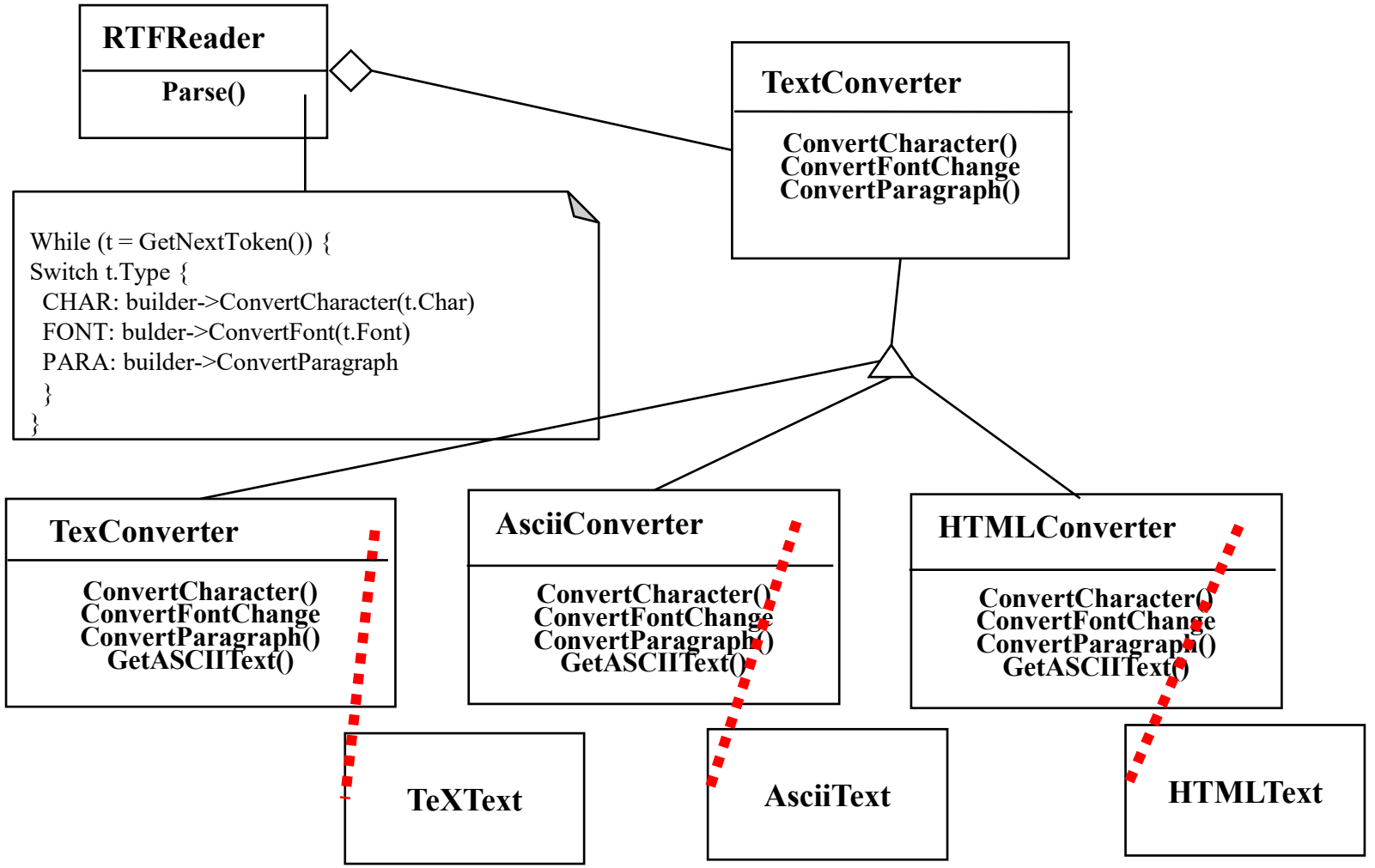
```
//Concrete Builder
class HawaiianPizzaBuilder : PizzaBuilder {
public override void BuildDough() { pizza.SetDough("cross"); }
public override void BuildSauce() { pizza.SetSauce("mild"); }
public override void BuildTopping() { pizza.SetTopping("ham+pineapple"); }
}

//Concrete Builder
class SpicyPizzaBuilder : PizzaBuilder {
public override void BuildDough() { pizza.SetDough("pan baked"); }
public override void BuildSauce() { pizza.SetSauce("hot"); }
public override void BuildTopping() { pizza.SetTopping("pepparoni+salami"); }
}
```

Example 1

```
/** "Director" */
class Waiter {
        private PizzaBuilder pizzaBuilder;
        public void SetPizzaBuilder (PizzaBuilder pb) { pizzaBuilder = pb; }
        public Pizza GetPizza() { return pizzaBuilder.GetPizza(); }
        public void ConstructPizza() {
                pizzaBuilder.CreateNewPizza();
                pizzaBuilder.BuildDough();
                pizzaBuilder.BuildSauce();
                pizzaBuilder.BuildTopping();
                }
}
/** A customer ordering a pizza. */
class BuilderExample {
        public static void Main(String[] args) {
        Waiter waiter = new Waiter();
        PizzaBuilder hawaiianPizzaBuilder = new HawaiianPizzaBuilder();
        PizzaBuilder spicyPizzaBuilder = new SpicyPizzaBuilder();
        waiter.SetPizzaBuilder ( hawaiianPizzaBuilder );
        waiter.ConstructPizza();
        Pizza pizza = waiter.GetPizza();
        }
}
```

Example 2


RTFReader
Parse()
TextConverter
ConvertCharacter()
ConvertFontChange
ConvertParagraph()
While (t = GetNextToken()) {
Switch t.Type {
CHAR: builder->ConvertCharacter(t.Char)
FONT: bulder->ConvertFont(t.Font)
PARA: builder->ConvertParagraph
}
}
TexConverter
ConvertCharacter()
ConvertFontChange
ConvertParagraph()
GetASCIIText()
AsciiConverter
ConvertCharacter()
ConvertFontChange
ConvertParagraph()
GetASCIIText()
HTMLConverter
ConvertCharacter()
ConvertFontChange
ConvertParagraph()
GetASCIIText()
TeXText
AsciiText
HTMLText

Abstract Factory vs Builder

- Abstract Factory
  - Focuses on product family
    - The products can be simple (“light bulb”) or complex
  - The abstract factory 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 is very complex
  - The construction of the complex product changes from time to time
  - The builder patterns hides the complex 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

Conclusions

Summary

- 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

Observations

- Patterns permit design at a more abstract level
  - Treat many class/object interactions as a unit
  - Often beneficial after initial design
  - Targets for class refactorings
- Variation-oriented design
  - Consider what design aspects are variable
  - Identify applicable pattern(s)
  - Vary patterns to evaluate tradeoffs
  - Repeat

Conclusion

- 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 patterns solve all your software engineering problems

Conclusions

- We could go on and on and present different patterns.
- However, at the end of the day we need to first define our problem before we come up with a solution.
  - And since patterns are all about solutions to a problem, don't look at patterns until you have already defined the problem!

(Design) Pattern References

- The Timeless Way of Building, Alexander; Oxford, 1979; ISBN 0-19-502402-8
- A Pattern Language, Alexander; Oxford, 1977; ISBN 0-19-501-919-9
- Design Patterns, Gamma, et al.; Addison-Wesley, 1995; ISBN 0-201-63361-2; CD version ISBN 0-201-63498-8
- Pattern-Oriented Software Architecture, Buschmann, et al.; Wiley, 1996; ISBN 0-471-95869-7
- Analysis Patterns, Fowler; Addison-Wesley, 1996; ISBN 0-201-89542-0
- Smalltalk Best Practice Patterns, Beck; Prentice Hall, 1997; ISBN 0-13-476904-X
- The Design Patterns Smalltalk Companion, Alpert, et al.; Addison-Wesley, 1998; ISBN 0-201-18462-1
- AntiPatterns, Brown, et al.; Wiley, 1998; ISBN 0-471-19713-0