UNIVERSITÀ DEGLI STUDI DI NAPOLI FEDERICO II

SOFTWARE ENGINEERING – LECTURES 18-19

Software Design: Design Patterns

Prof. Sergio Di Martino

THE WATERFALL SOFTWARE PROCESS MODEL

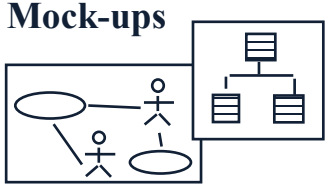
Requirements Engineering → **System Design** → **Software and UI Design** → **Implemen-tation** → **Testing** → **Operation and Maintenance**

Requirements collected via:

- Interviews with Stakeholders
- Personas
- Stories and Scenarios

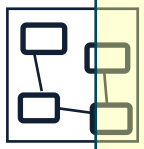
Specified using:

- Use Cases
- Natural Language
- Domain Models
- Mock-ups



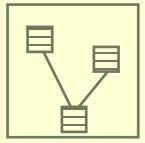
Define System Architecture

- Requirements are allocated to software sub-systems
- Sub-systems are allocated to hardware resources
- Architectural Patterns



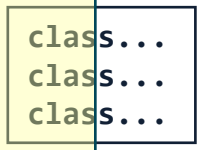
Define Subsystems

- Objects required to realize each subsystem are defined.
- Software Design Patterns
- Usability Engineering
- High-fidelity Wireframing



Each Subsystem is implemented

- Source code and other artifacts
- Clean Code
- Frameworks and ORMs
- Focus on Software Quality



Ensure the Software satisfies customers

- Code inspections
- Functional Testing (unit, integration, system testing)
- Usability Testing

System is put into practical use

Maintance will be required at some point

- To fix errors that were not discovered in previous phases
- To adapt the software to changes in requirements on in its environment

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

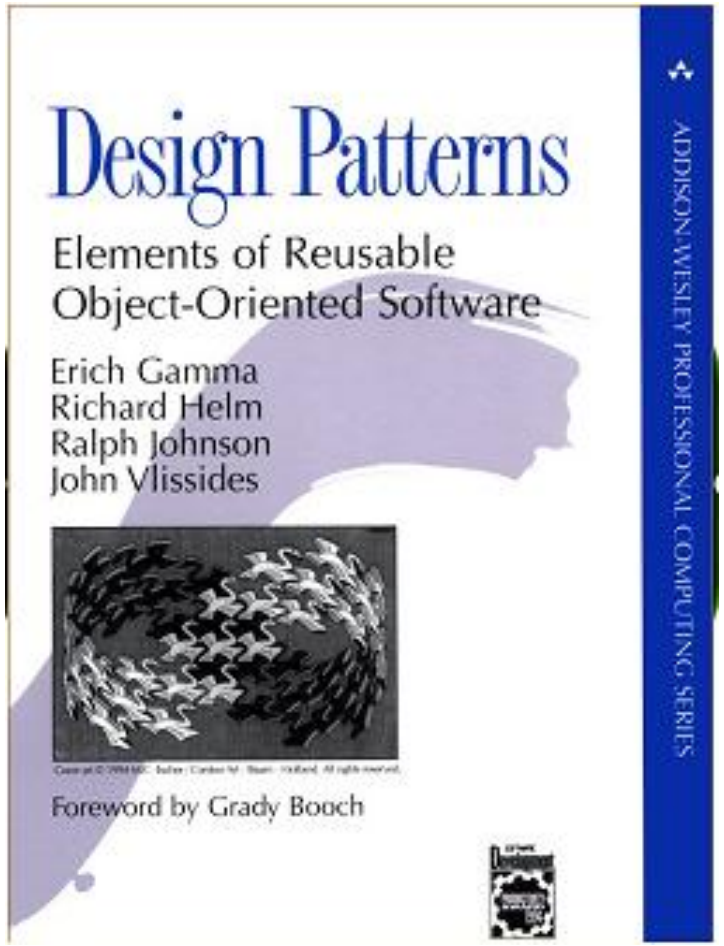
- Code re-use
  - Don’t reinvent the wheel
  - Requires clean, elegant, understandable, general, stable code
  - leverage previous work
- Design re-use
  - Don’t reinvent the wheel
  - Requires a precise understanding of common, recurring designs
  - leverage previous work

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
- …

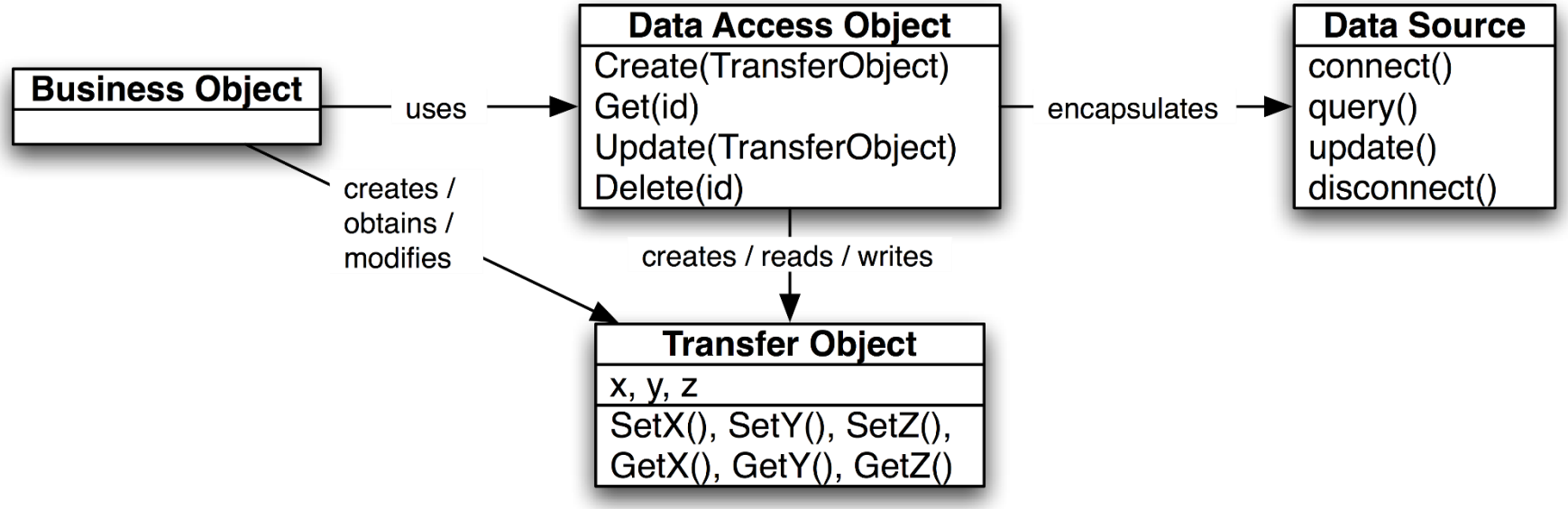
Data Access Object

DAO Pattern

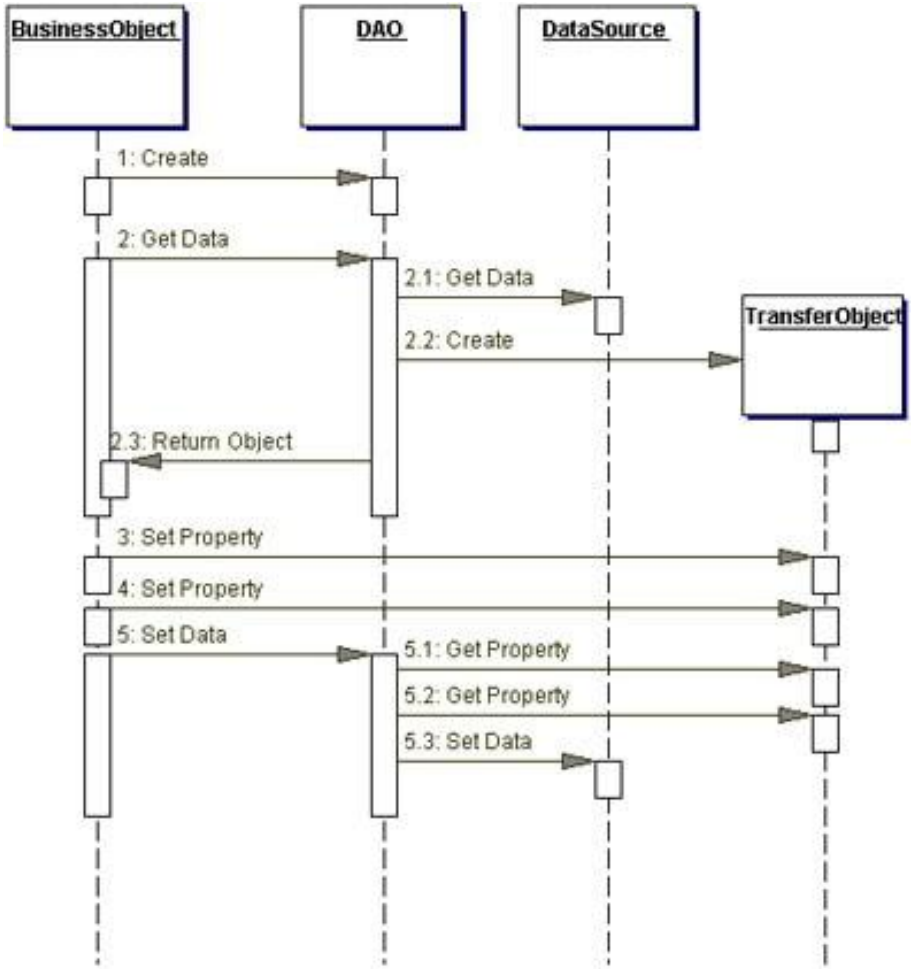
- **Problem**
  - Access to data varies greatly depending on the type of storage (relational DBMS, NoSQL DBMS, flat files, and so forth) and the vendor implementation.
- **Solution**
  - Use the Data Access Object (DAO) pattern to abstract and encapsulate all access to the data source. Each DAO manages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

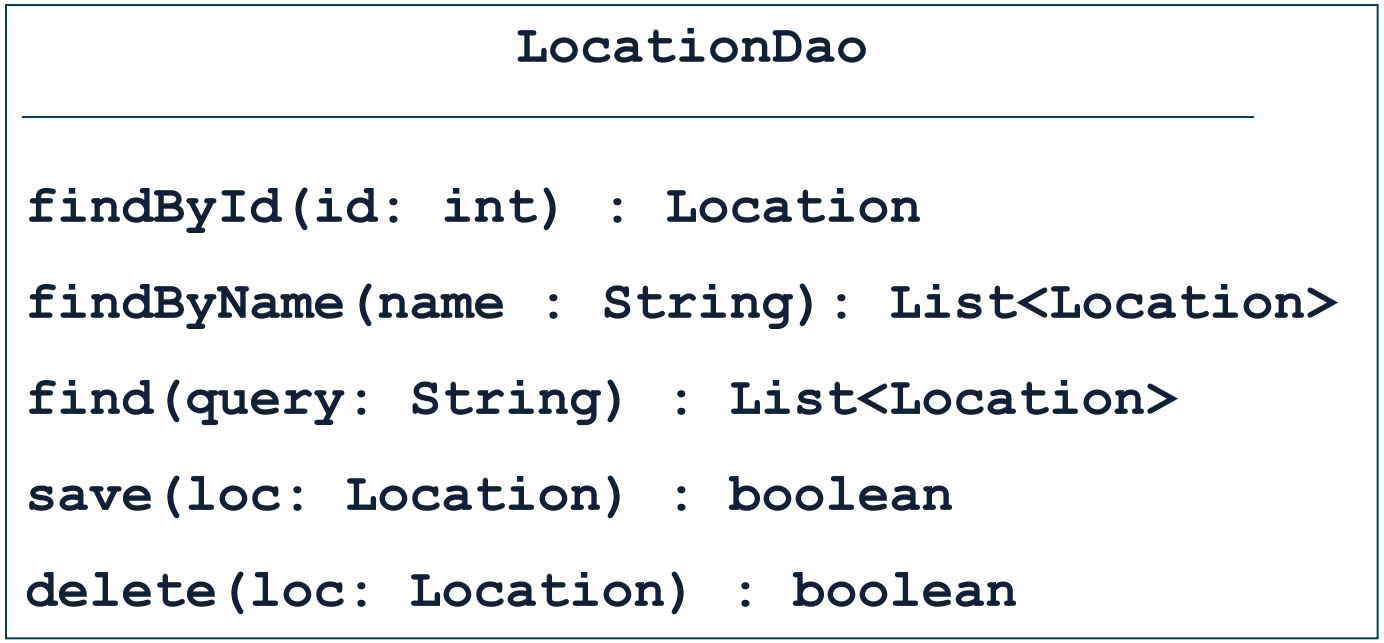


DAO Design Pattern



A DAO for a Location class

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



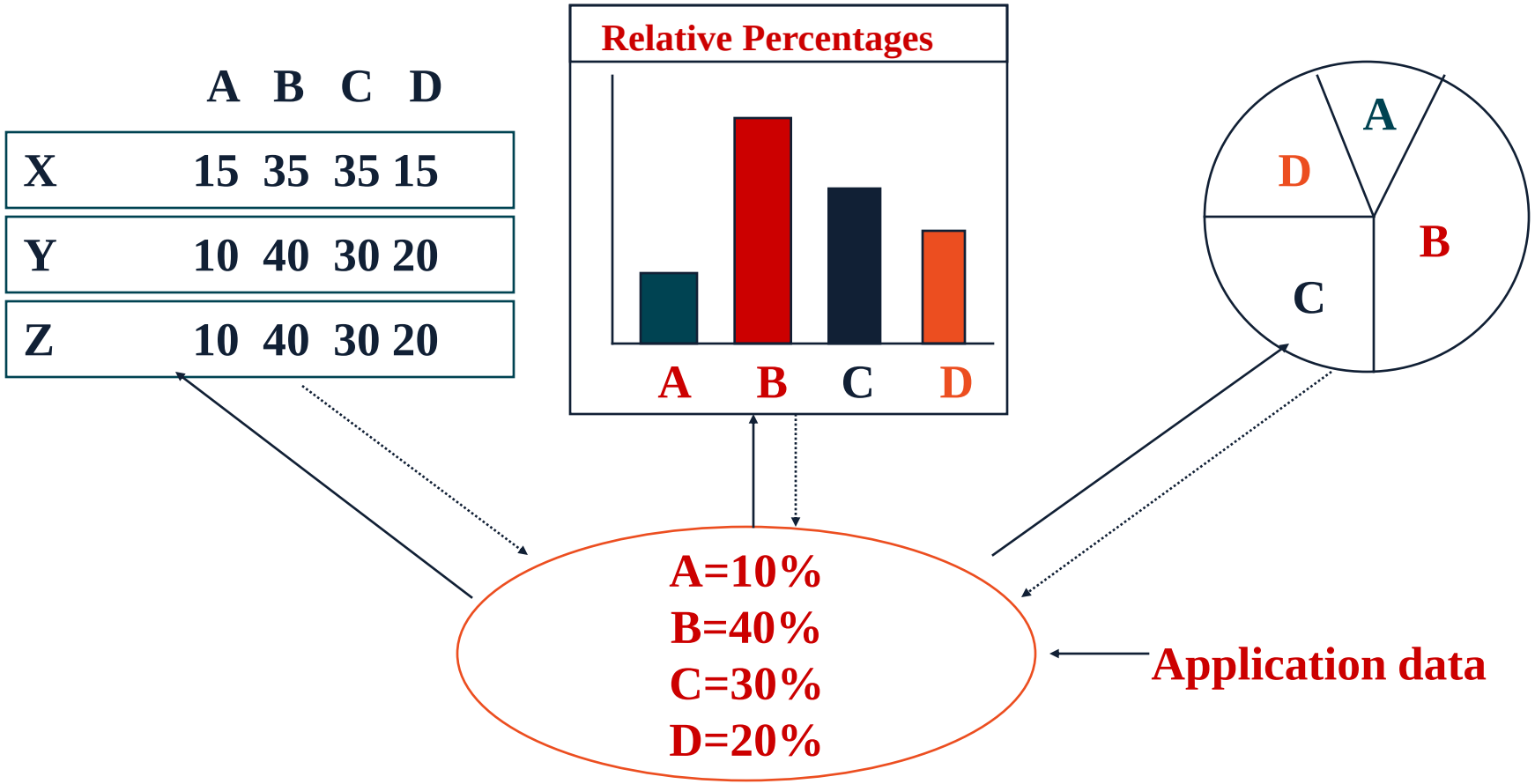
Classification of GoF Design Pattern

		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

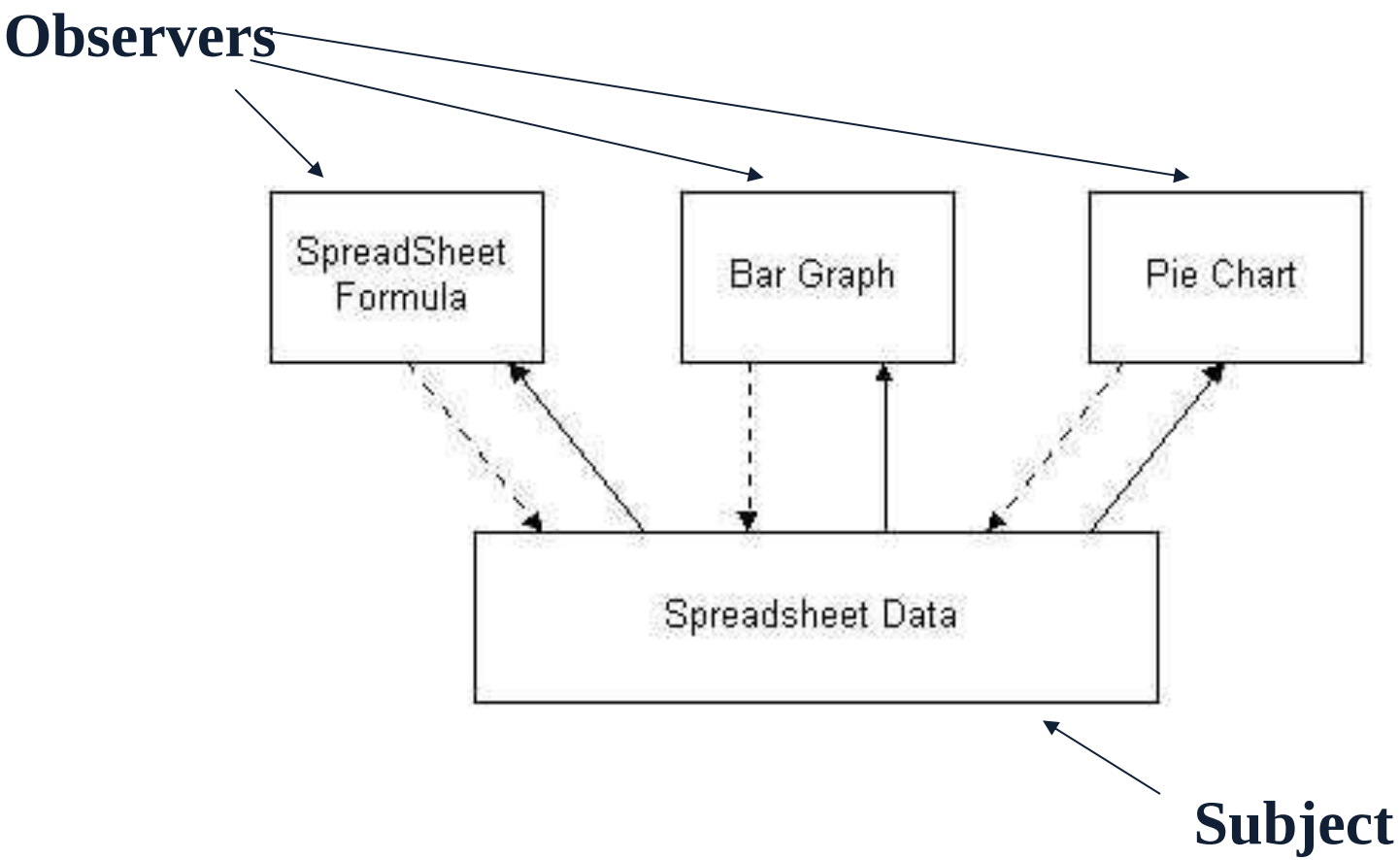
		Purpose		
		Creational	**Structural**	**Behavioral**
Scope	**Class**	Factory Method	Adapter (class)	Interpreter Template Method
	Object	Abstract Factory Builder Prototype Singleton	Adapter (object) Bridge Composite Decorator Flyweight Facade Proxy	Chain of Responsibility Command Iterator Mediator Memento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

Observer

Patterns by Example:
Multiple displays enabled by Observer



Schematic Observer Example

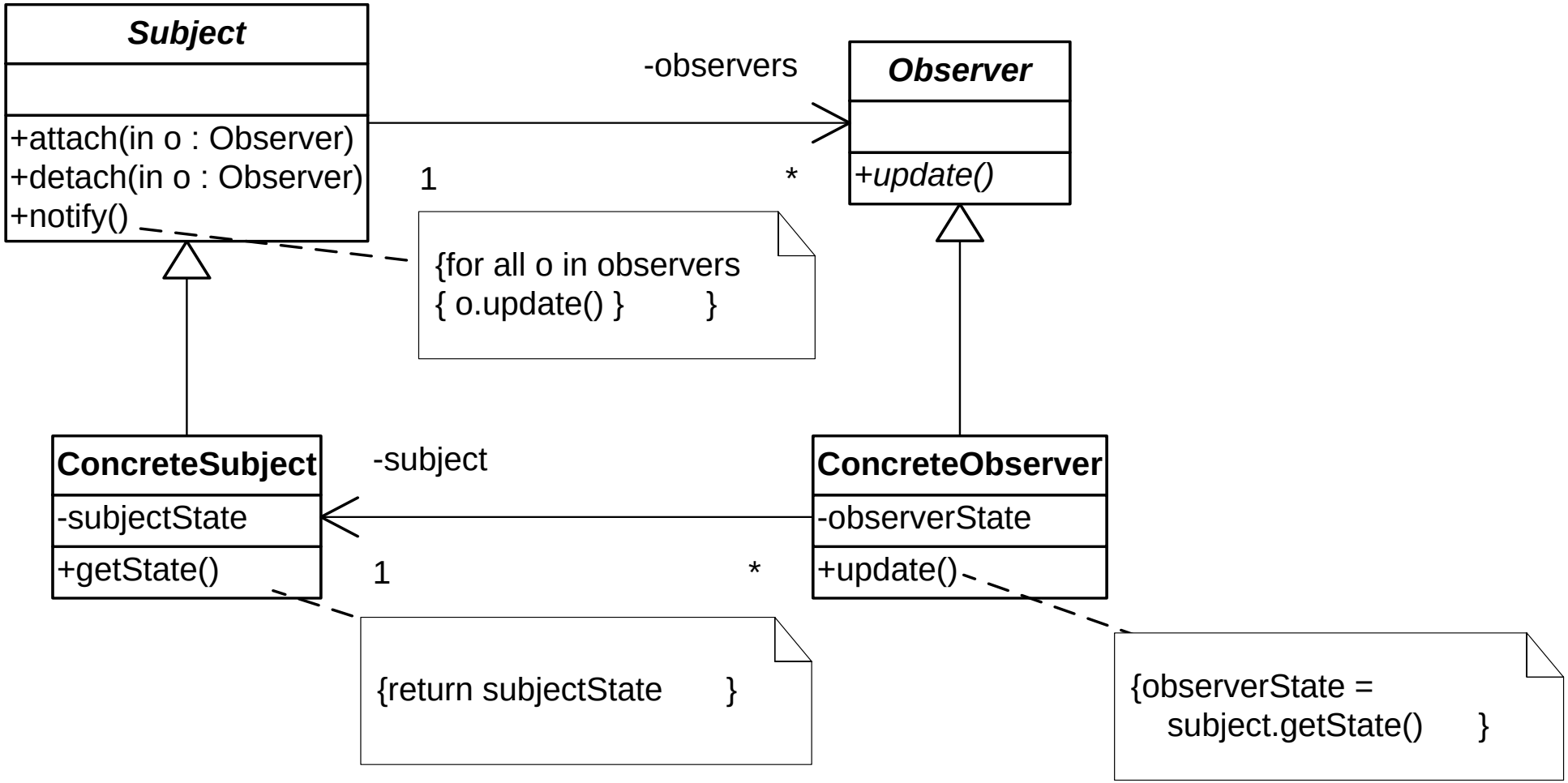


Observer (Behavioral)

- 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;
  }
}
```

The Observer Pattern in JDK

- Sometimes there are mistakes also in the JDK design.
- From JDK 1 to JDK 8, Java offered two elements to support programmers in implementing the Observer Design Pattern:
  - java.util.Observer and
  - java.util.Observable

Definition of Observer, from JDK 6

java.util

Interface Observer

`public interface` **`Observer`**

A class can implement the `Observer` interface when it wants to be informed of changes in observable objects.

Since:

JDK1.0

See Also:

`Observable`

Method Summary	
`void`	**`update`**`(Observable o, Object arg)` This method is called whenever the observed object is changed.

Definition of Observable, from JDK 6

java.util

Class Observable

java.lang.Object
  └ **java.util.Observable**

---

public class **Observable**
extends Object

Method Summary	
void	**addObserver**(Observer o) Adds an observer to the set of observers for this object, provided that it is not the same as some observer already in the set.
protected void	**clearChanged**() Indicates that this object has no longer changed, or that it has already notified all of its observers of its most recent change, so that the hasChanged method will now return false.
int	**countObservers**() Returns the number of observers of this Observable object.
void	**deleteObserver**(Observer o) Deletes an observer from the set of observers of this object.
void	**deleteObservers**() Clears the observer list so that this object no longer has any observers.
boolean	**hasChanged**() Tests if this object has changed.
void	**notifyObservers**() If this object has changed, as indicated by the hasChanged method, then notify all of its observers and then call the clearChanged method to indicate that this object has no longer changed.
void	**notifyObservers**(Object arg) If this object has changed, as indicated by the hasChanged method, then notify all of its observers and then call the clearChanged method to indicate that this object has no longer changed.
protected void	**setChanged**() Marks this Observable object as having been changed; the hasChanged method will now return true.

What are the limitations?

- These two elements have been deprecated since Java 9. Why?
  - Observable is a class
  - The proposed notification mechanism can be a significant limitation in some context. It supports only the notion that something has changed, but there is no way to convey any information about what has changed.

		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

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

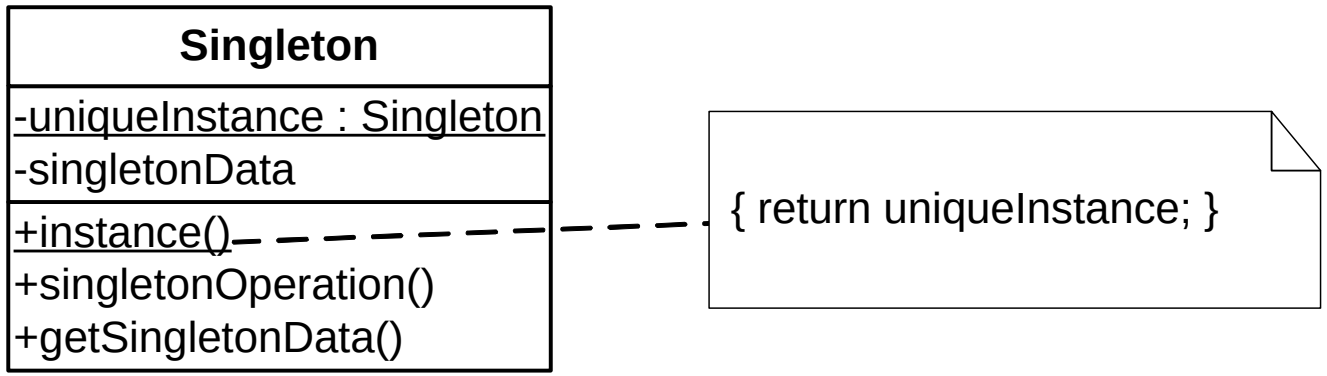
Singleton

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

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.

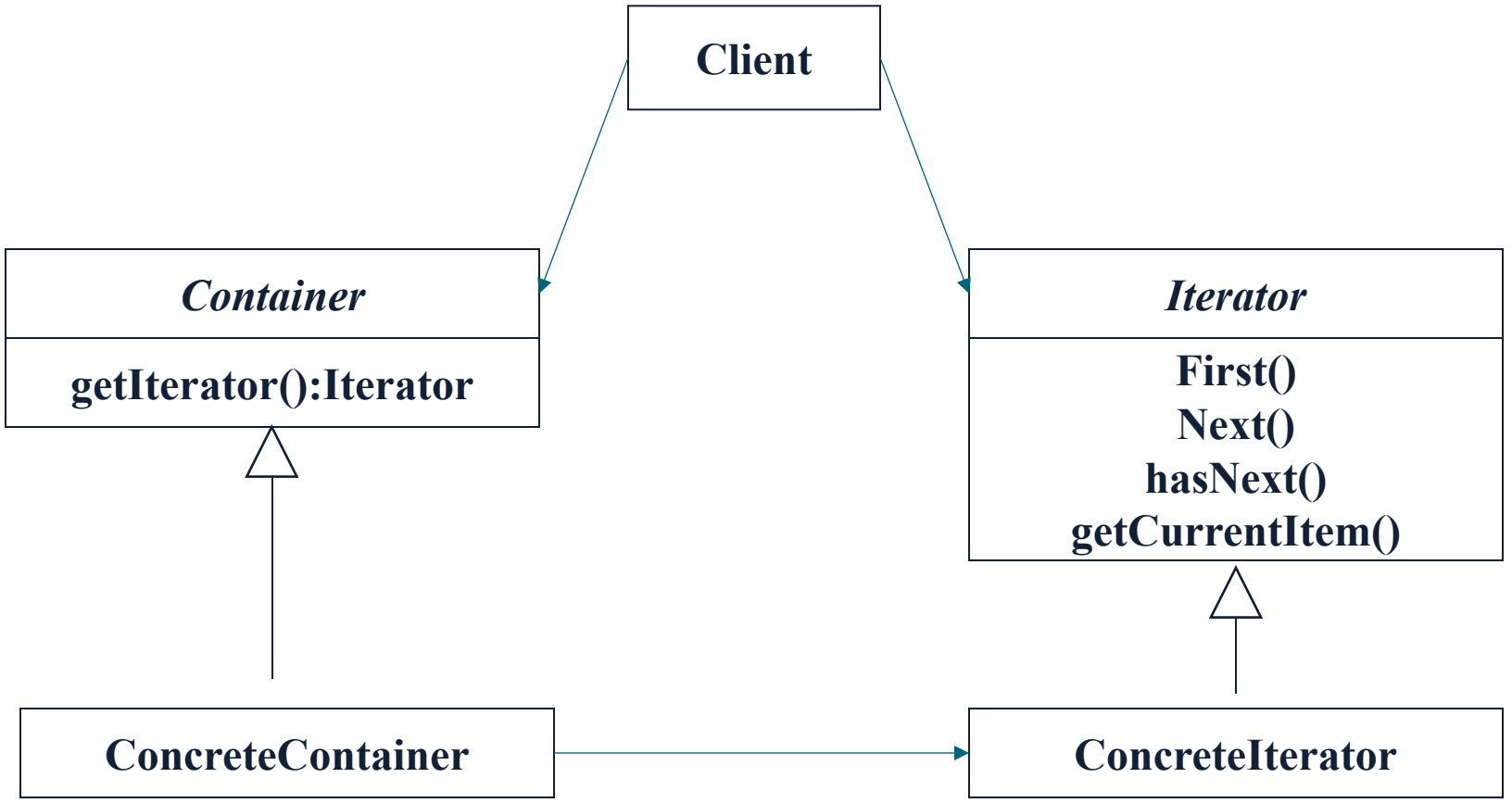
How can we do this?

- *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

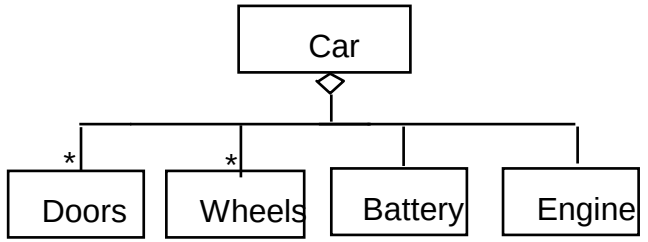
		Purpose		
		Creational	**Structural**	**Behavioral**
Scope	**Class**	Factory Method	Adapter (class)	Interpreter Template Method
	Object	Abstract Factory Builder Prototype Singleton	Adapter (object) Bridge Composite Decorator Flyweight Facade Proxy	Chain of Responsibility Command Iterator Mediator Memento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

Composite

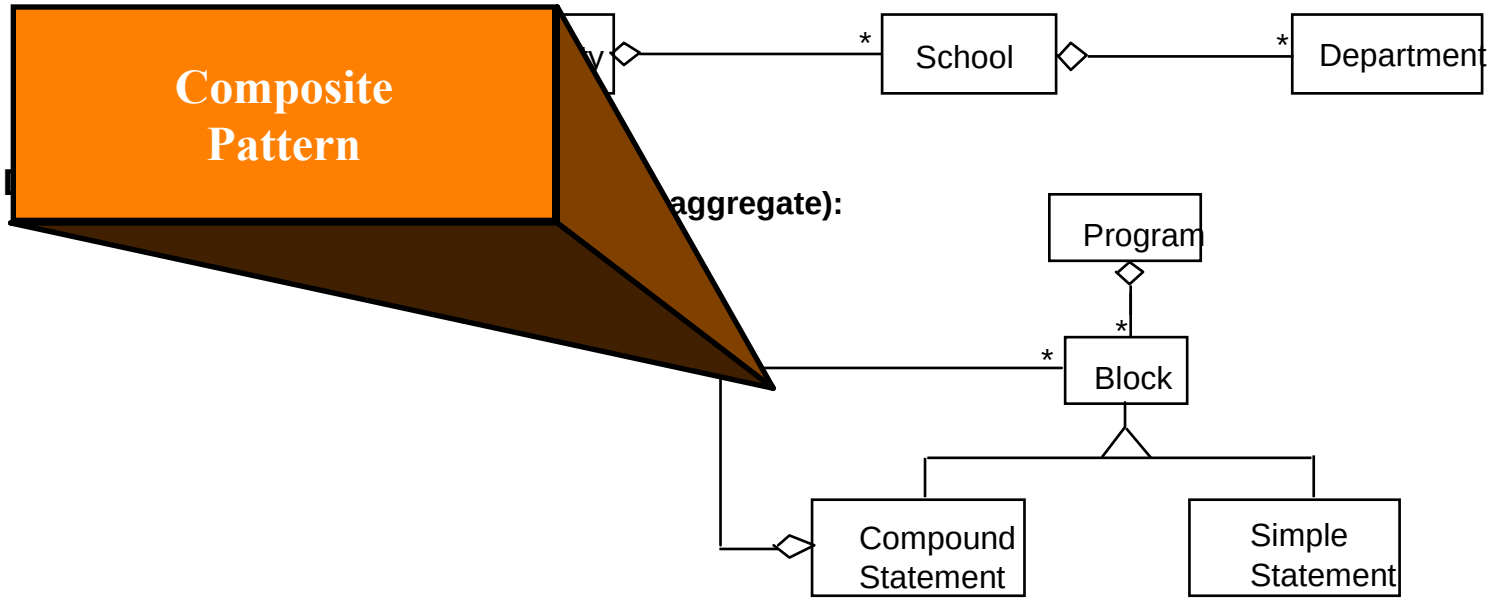
(Structural)

Review: Modeling Typical Aggregations

Fixed Structure:

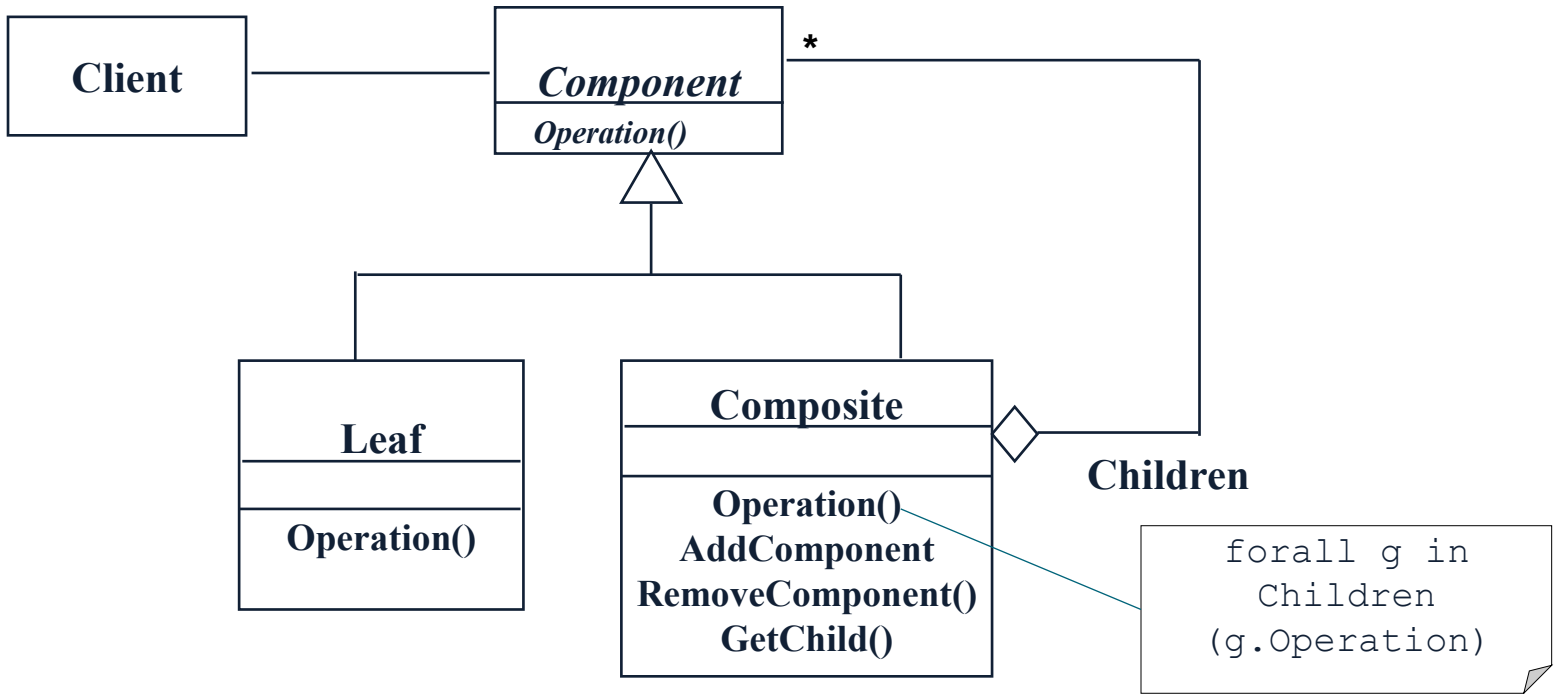


Organization Chart (variable aggregate):



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 { //Redraw the graphic component.
  public void reDraw();
}

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

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

Composite Implementation (2)

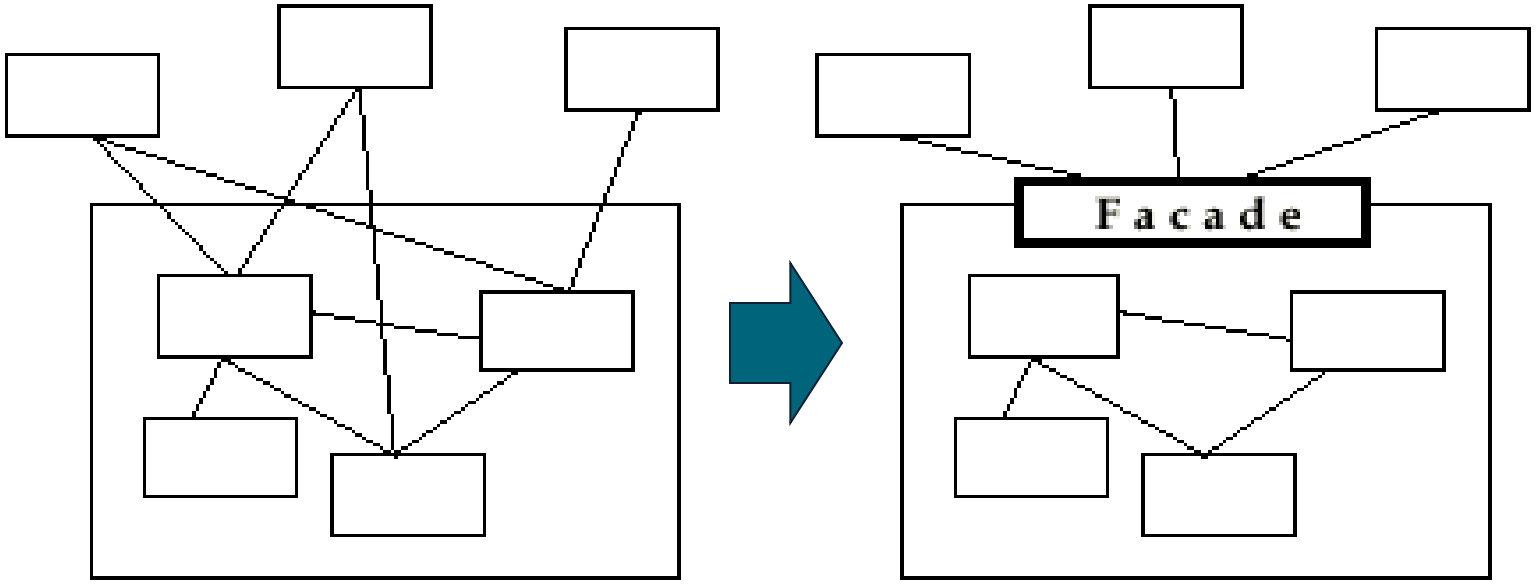
```
 //Adds the graphic to the composition.
  public void add(Graphic graphic)
  { mChildGraphics.add(graphic); }

  //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 reDraw() {
      System.out.println("Ellipse");
    }
}
```

		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

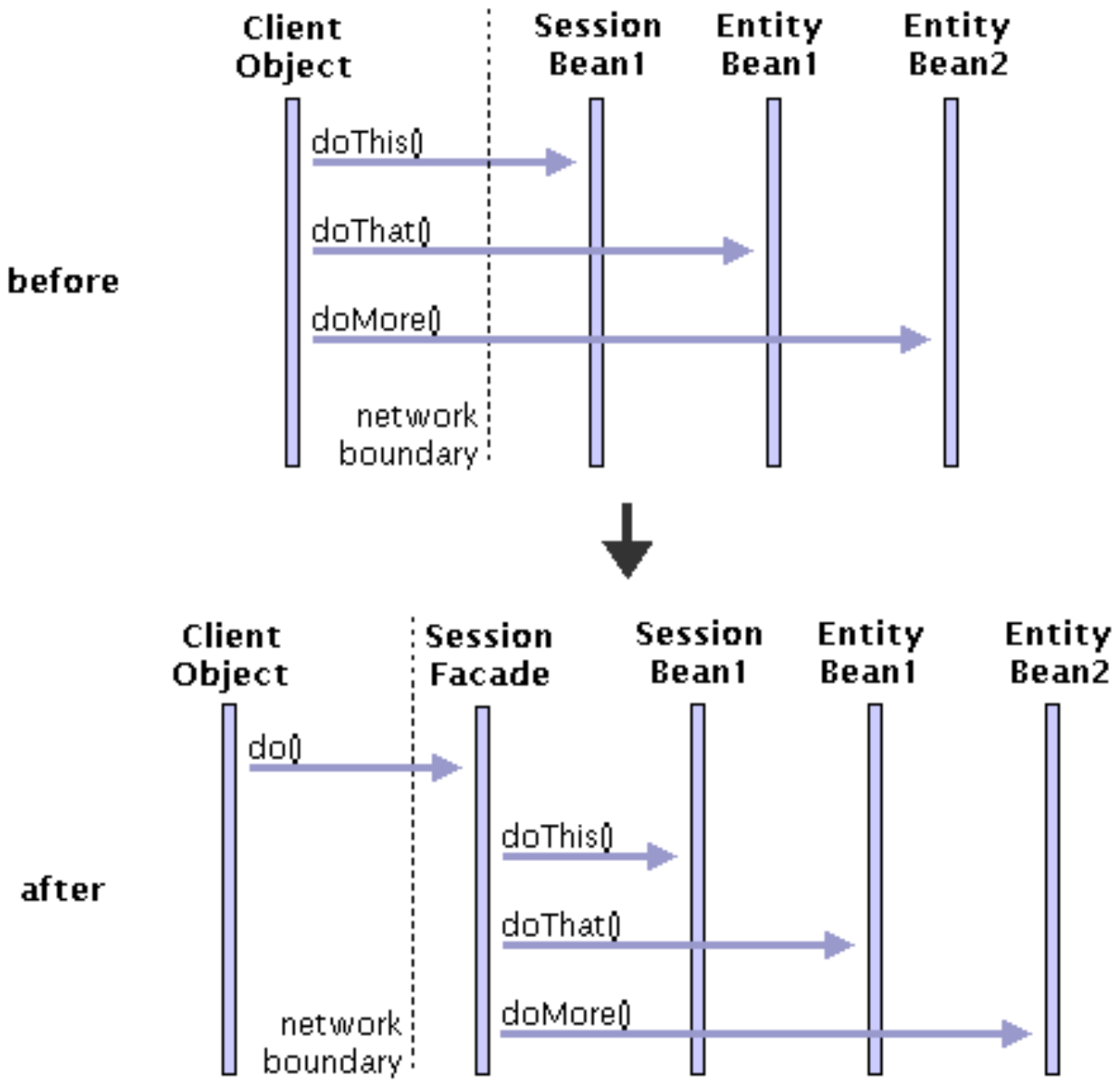
Facade

Facade Pattern

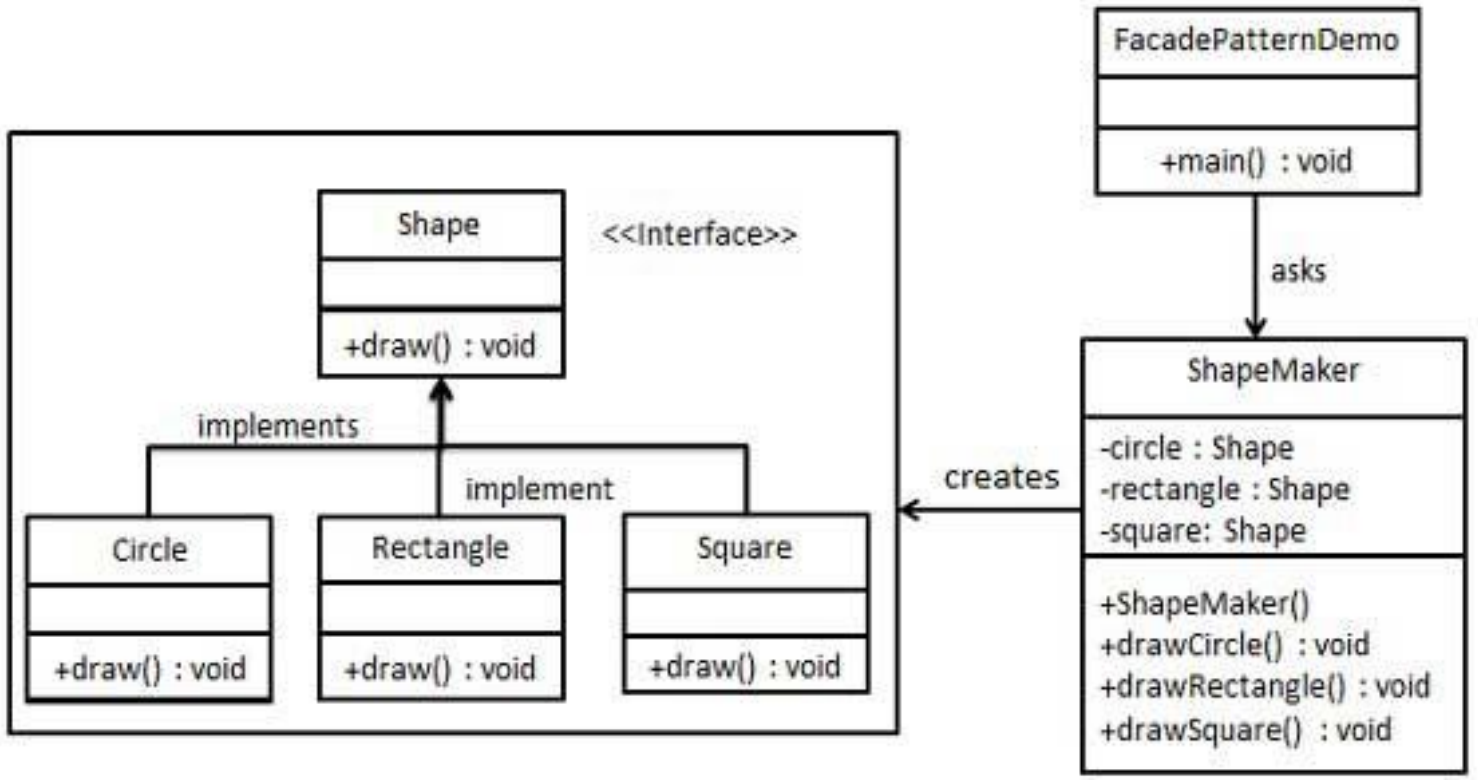
- Provides a unified interface to a set of objects in a subsystem.
- A facade defines a higher-level interface that makes the subsystem easier to use (i.e. it abstracts out the gory details)
- Facades allow us to provide a closed architecture



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

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