UNIVERSITÀ DEGLI STUDI DI NAPOLI FEDERICO II

SOFTWARE ENGINEERING – LECTURE 23

SOFTWARE DESIGN: DESIGN PATTERNS

Proff. Luigi Libero Lucio Starace, Sergio Di Martino

luigiliberolucio.starace@unina.it, sergio.dimartino@unina.it

https://www.docenti.unina.it/luigiliberolucio.starace, https://www.docenti.unina.it/sergio.dimartino

Software Development Lifecycle

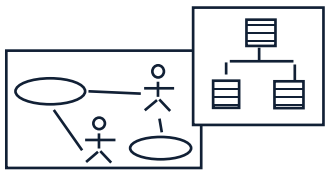
Requirements Engineering → **System Design** → **Software and UI Design** → **Implementation** → **Testing** → **Operation and Maintenance**

Requirements collected via:

- Interviews with Stakeholders
- Personas
- Stories and Scenarios

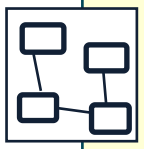
Specified using:

- 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-use of Design

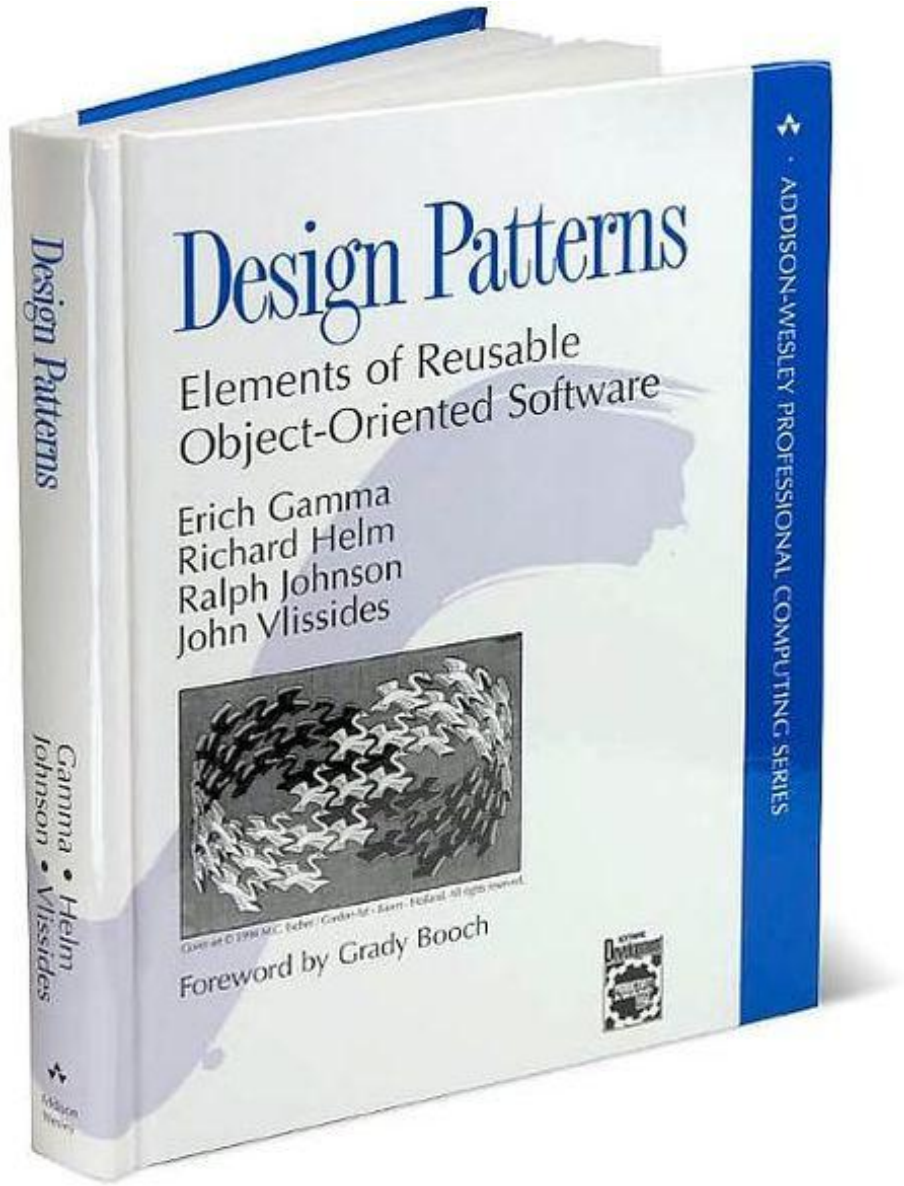
- Code re-use
  - Don’t reinvent the wheel
  - Requires clean, elegant, understandable, general, stable code
  - Leverage previous work
- Design re-use
  - Don’t reinvent the wheel
  - Requires a precise understanding of common, recurring designs
  - Leverage previous work

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's a Design Pattern?

- A **common** solution to a **recurring** problem in design
- Abstracts a recurring design structure
  - Comprises classes and/or objects
    - Dependencies
    - Structures
    - Interactions
    - Conventions
  - Explicitly specifies the design structure
  - 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

Why design patterns matter

- **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
- …

Classification of Patterns in SWE

- Design patterns differ by complexity, level of detail, scale of applicability
- High-level patterns are often called **architectural patterns**
  - Their scope is the high-level structure of the entire software system
  - They can be implemented in any programming language
  - **REST** can be considered an architectural pattern
- Mid/low level patterns are often referred to simply as **design patterns**
  - They apply to design choices within components or modules
  - Solve recurring problems in **object-oriented** design and implementation

Classification of Design Patterns

Design patterns are often categorized based on their **goal** or **intent**

- **Creational patterns** involve mechanisms to create object and aim at increasing flexibility, reuse of existing code, maintainability.
- **Structural patterns** involve ways to assemble objects and classes into larger structures, while keeping these structures flexible and efficient.
- **Behavioral patterns** take care of effective communication and the assignment of responsibilities between objects.

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

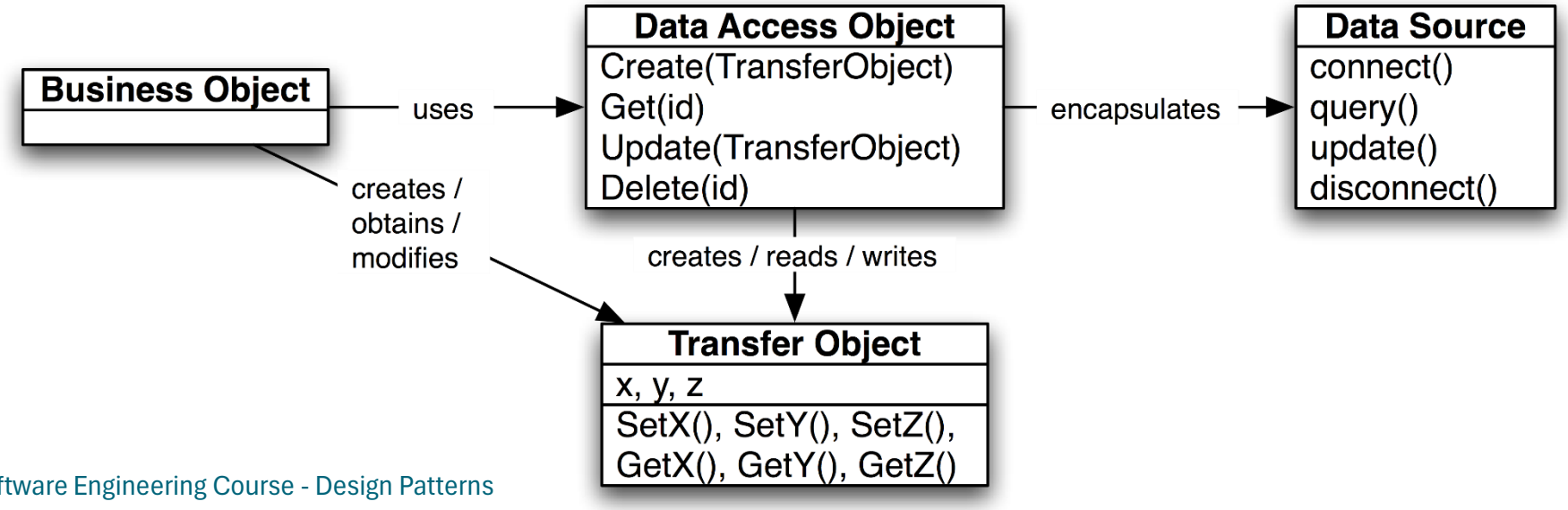
Data Access Object

DAO Pattern

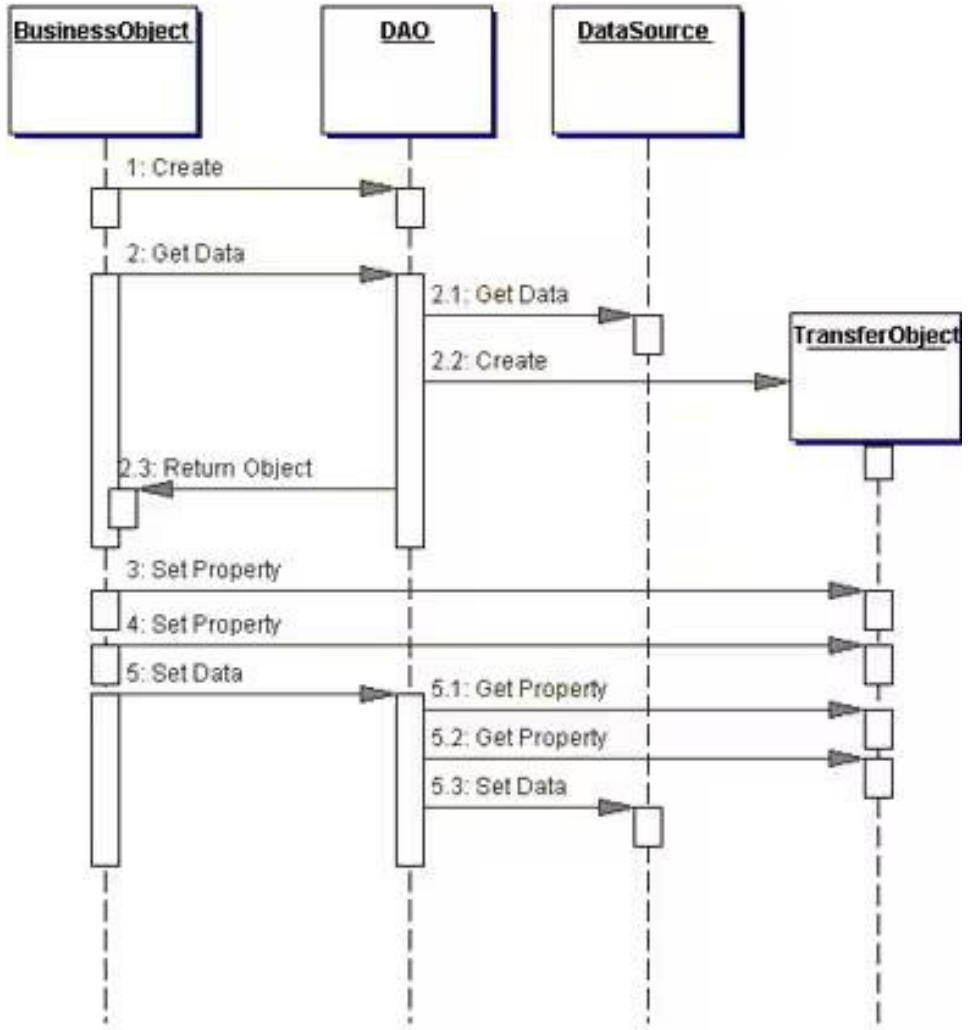
- **Problem**
  - Access to data varies greatly depending on the type of storage (relational DBMS, NoSQL DBMS, flat files, and so forth) and the vendor implementation.
- **Solution**
  - Use the Data Access Object (DAO) pattern to abstract and encapsulate all access to the data source. Each DAO manages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

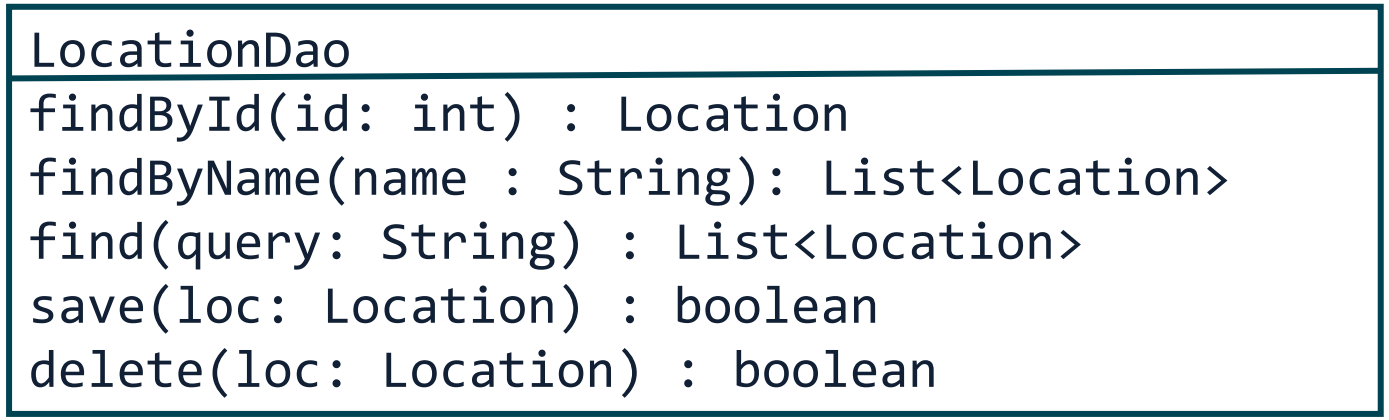


DAO Design Pattern



A DAO for a Location class

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



Singleton

(Creational)

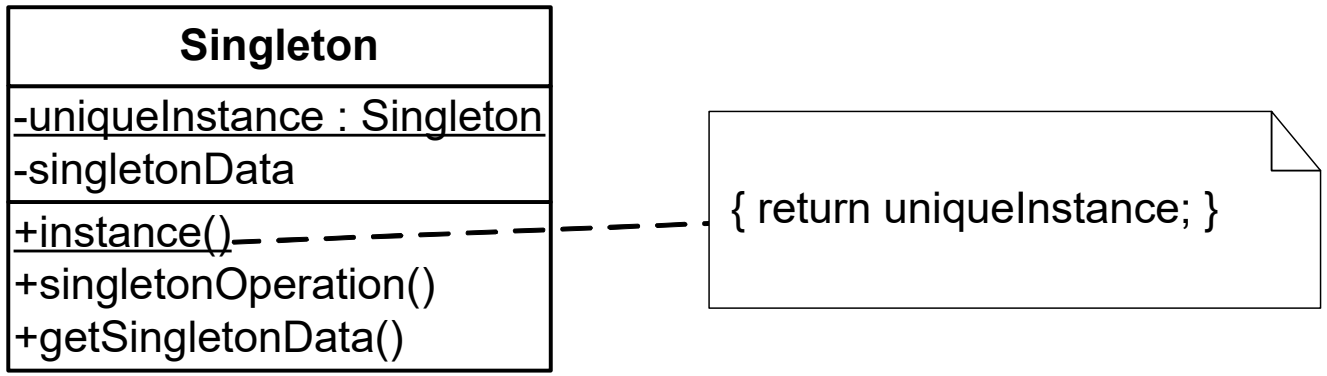
		Purpose		
		Creational	**Structural**	**Behavioral**
Scope	**Class**	Factory Method	Adapter (class)	Interpreter Template Method
	Object	Abstract Factory Builder Prototype Singleton	Adapter (object) Bridge Composite Decorator Flyweight Facade Proxy	Chain of Responsibility Command Iterator Mediator Memento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

Singleton (Creational)

- 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.

Builder

(Creational)

		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

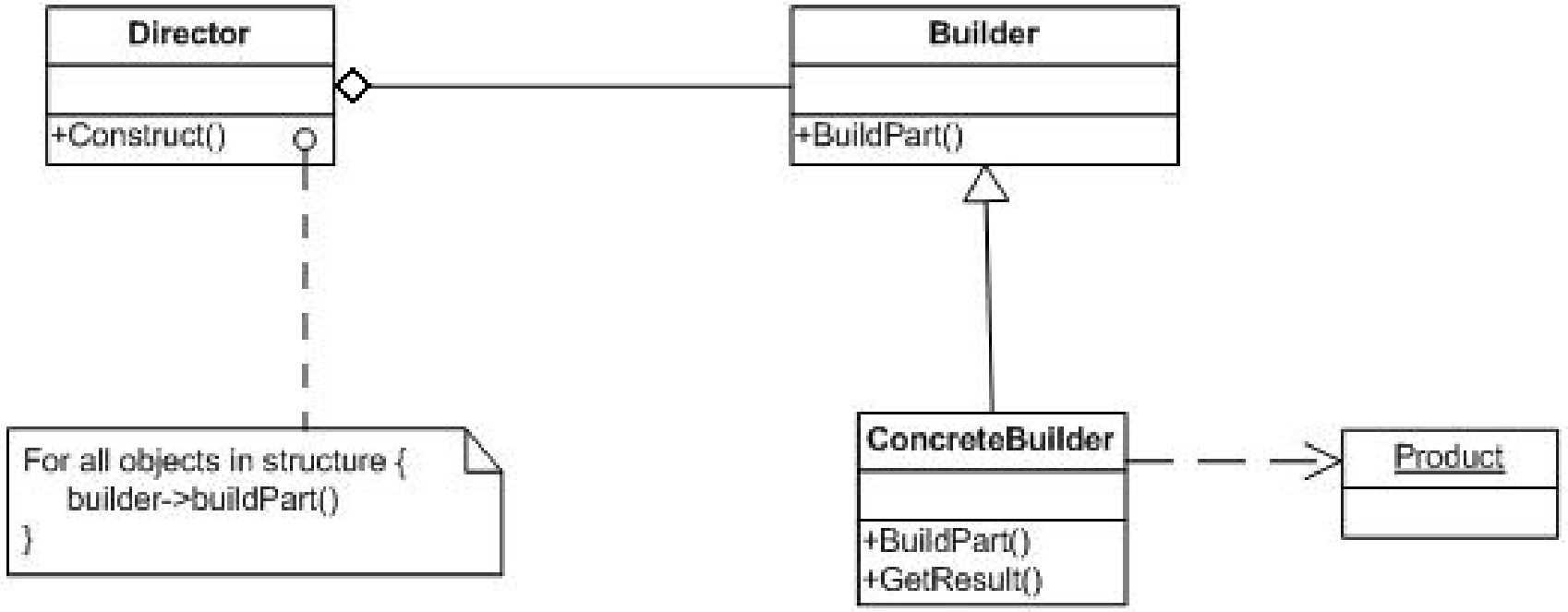
Builder

- There is a whole family of patterns (Builder and Factory) that all serve to address the problem of complex classes for which the basic object-oriented mechanism of constructors is not sufficient to guarantee sufficient flexibility and efficiency

- **Name** : Builder
- **Purpose** : It makes the construction of an object flexible and decomposable, separating it from the actual constructor of the object

Solution

- The single object Product is built by ConcreteBuilder
- The Director class can manage the construction of a set of Product objects



When do you use the Builder Pattern?

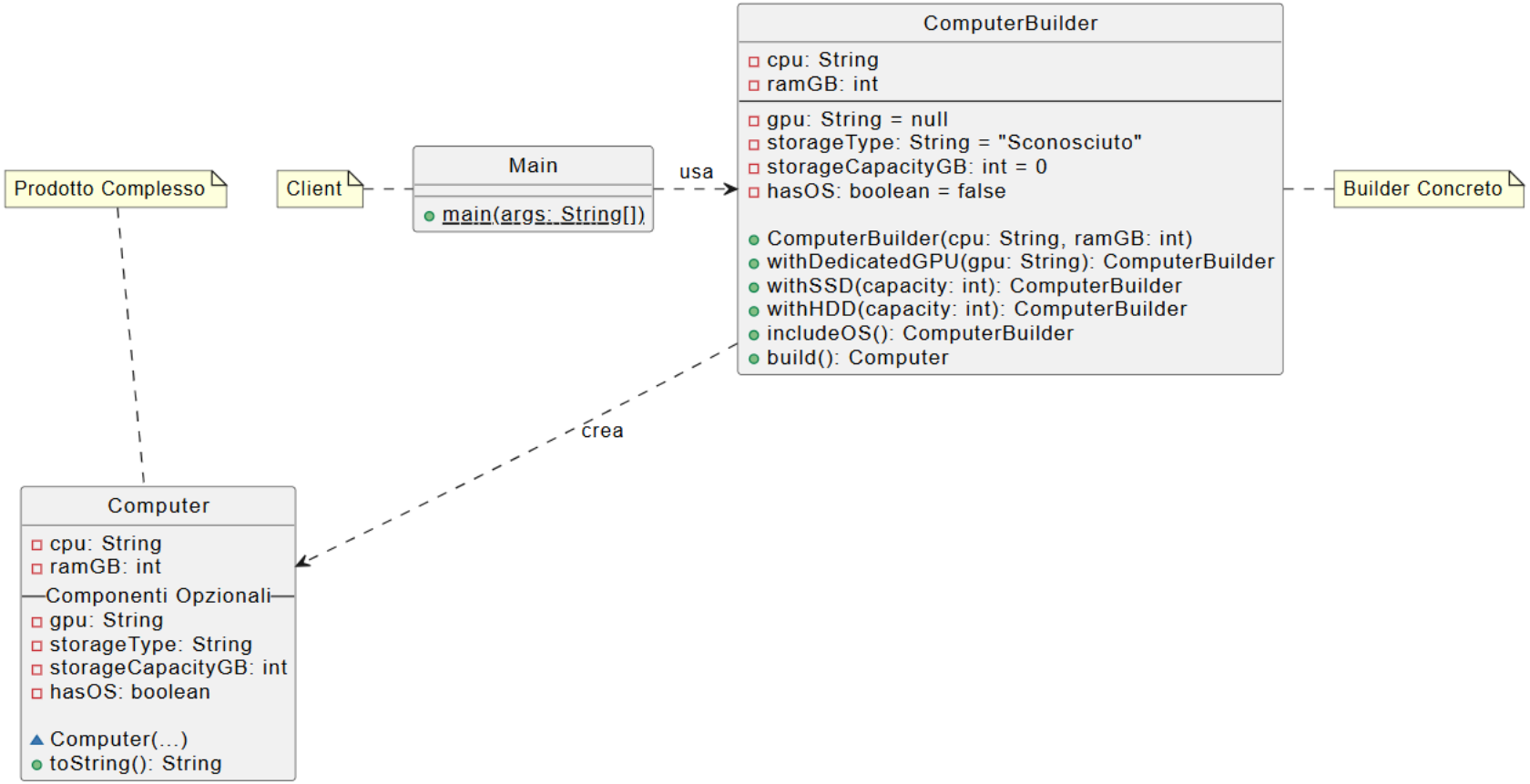
- The creation of a complex product must be independent of the particular parts that make up the product
  - In particular, the creation process should not know about the assembly process (how the parts are put together to make up the product)
- The creation process must allow different representations 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.

Simple Example : Building a Computer

- We want to build objects of a Computer class, in order to manage their advertising, sale, etc.
- A computer can have many parameters, and you may need to add more parameters in the future
- We separate the Computer object from a ComputerBuilder object that only deals with managing all the possible variants of the construction of a Computer object in a flexible way

    - This simple example shows the use of a ConcreteBuilder class but not the use of the Director class

Class Diagram



Esempio: class Computer

```
public class Computer {
// Componenti obbligatori
private final String cpu;
private final int ramGB;


// Componenti opzionali
private final String gpu;
private final String storageType; // SSD o HDD
private final int storageCapacityGB;
private final boolean hasOS; // Sistema Operativo
```

```
/**
* Costruttore accessibile solo dal ComputerBuilder
* (presumendo che si trovino nello stesso package).
*/
Computer(String cpu, int ramGB, String gpu, String storageType, int storageCapacityGB, boolean hasOS) {
this.cpu = cpu;
this.ramGB = ramGB;
this.gpu = gpu;
this.storageType = storageType;
this.storageCapacityGB = storageCapacityGB;
this.hasOS = hasOS;
}


}
```

Example: class ComputerBuilder

```
public class ComputerBuilder {
   // Parametri obbligatori (final, inizializzati nel costruttore del Builder)
   private final String cpu;
   private final int ramGB;

   // Parametri opzionali (con valori di default)
   private String gpu = null;
   private String storageType = "Sconosciuto";
   private int storageCapacityGB = 0;
   private boolean hasOS = false;

   // Costruttore per i parametri OBBLIGATORI del Computer.
public ComputerBuilder(String cpu, int ramGB) {
      this.cpu = cpu;
      this.ramGB = ramGB;
   }

   // Metodi setter opzionali: restituiscono il Builder stesso
   public ComputerBuilder withDedicatedGPU(String gpu) {
      this.gpu = gpu;
      return this;
   }

   public ComputerBuilder withSSD(int capacity) {
      this.storageType = "SSD";
      this.storageCapacityGB = capacity;
      return this;
   }

...
      // Metodo BUILD finale: crea e restituisce l'istanza finale del Computer.

public Computer build() {
      // Logica di validazione
      if (ramGB < 4) {
         throw new IllegalStateException("RAM insufficiente. Minimo richiesto: 4GB.");
      }
       // Chiama il costruttore package-private del Prodotto
      return new Computer(cpu, ramGB, gpu, storageType, storageCapacityGB, hasOS);
   }
}
```

Example : Main

```
public class Main {
public static void main(String[] args) {
// 1. Costruzione di un PC "Gaming" completo
System.out.println("--- Ordine 1: PC Gaming ---");
// le chiamate . sono innestate perché i vari metodi with di ComputerBuilder restituiscono lo stesso ComputerBuilder
Computer gamingPC = new ComputerBuilder("AMD Ryzen 7", 32).withDedicatedGPU("AMD Radeon RX 7800").withSSD(2000).includeOS().build();
System.out.println(gamingPC);


System.out.println("\n--- Ordine 2: PC Ufficio Base ---");
// Nessuna GPU dedicata, no OS
Computer officePC = new ComputerBuilder("Intel Core i3", 8).withHDD(500).build();
System.out.println(officePC);

System.out.println("\n--- Ordine 3: PC obsoleto ---");
Computer obsoletePC = new ComputerBuilder("Intel Core i3", 2)
.withHDD(250)
.build();
System.out.println(obsoletePC);

}
}
```

Example : Output

--- Ordine 1: PC Gaming ---

Configurazione Computer:

- CPU: AMD Ryzen 7
- RAM: 32GB
- GPU: AMD Radeon RX 7800
- Storage: 2000GB SSD
- OS Installato: true

--- Ordine 2: PC Ufficio Base ---

Configurazione Computer:

- CPU: Intel Core i3
- RAM: 8GB
- GPU: Integrata
- Storage: 500GB HDD
- OS Installato: false

--- Ordine 3: PC obsoleto ---

Exception in thread "main"
java.lang.IllegalStateException: RAM insufficiente. Minimo richiesto: 4GB.

at ComputerBuilder.build(ComputerBuilder.java:52)

at Main.main(Main.java:24)

The third PC has not been built: the logic related to obsolete computers is not in the Computer class but in the Builder class

Builder in Java libraries

- The Builder pattern is widely used in standard Java library classes:
  - java.lang.StringBuilder
    - Used to build strings
  - java.util.Locale.Builder
    - Used to build complex locale objects by defining language, country, scripts, and variants
  - java.time.format.DateTimeFormatterBuilder
    - Builder to create date-time formatters

Builder in the REST API we developed

```
private String createJWT(String username, long ttlMillis) {

    String token = JWT.create()
            .withIssuer(ISSUER)
            .withClaim("username", username)
            .withIssuedAt(new Date())
            .withExpiresAt(new Date(System.currentTimeMillis() + ttlMillis))
            .withJWTId(UUID.randomUUID().toString())
            .sign(algorithm);

    return token;
}
```

```
return Response
        .status(Response.Status.OK)
        .entity(token)
        .build();
```

Adapter

(Structural)

Design Pattern : Adapter

Name: Adapter

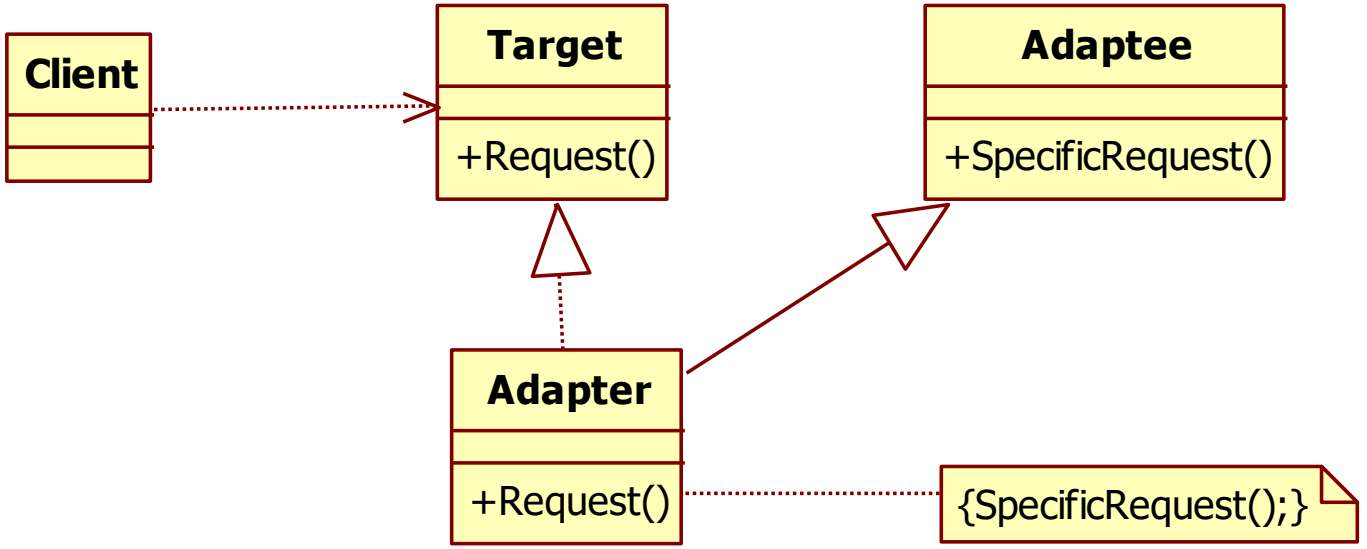
Synonym : Wrapper

Purpose : Converts the interface of one class into another interface, expected by a client (thus allowing the cooperation of classes that would otherwise have incompatible interfaces)

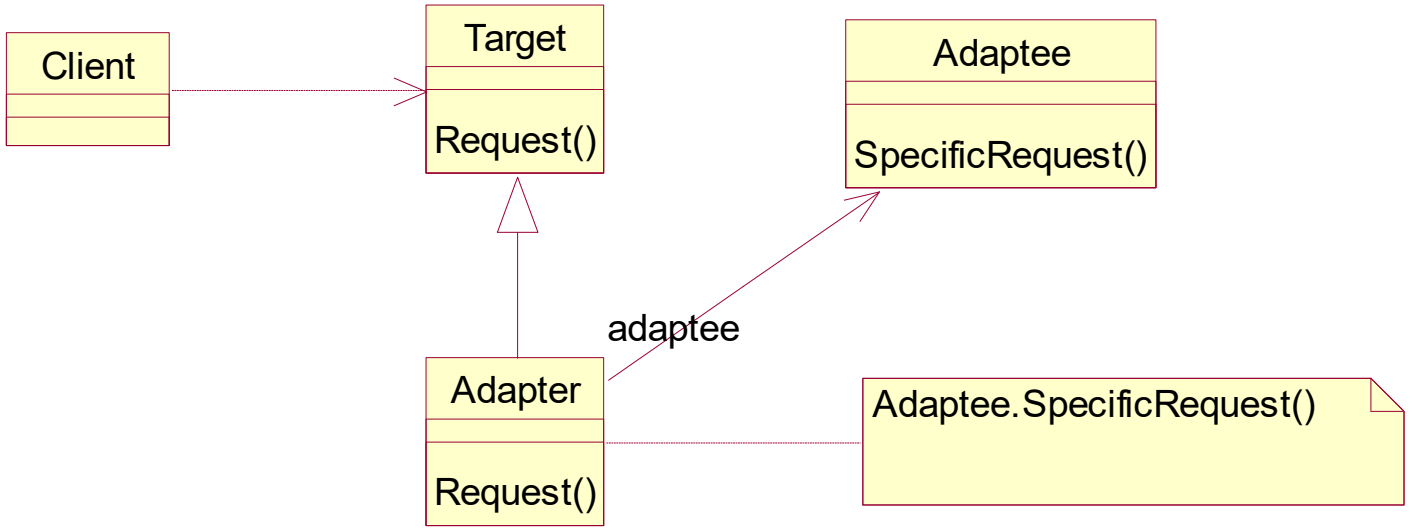
Motivation : When you want to use an already existing and tested class but it does not match the interface you need

Adapter

- **Participants :**
  - *Client* : the class that wants to use the service to be adapted
  - *Target* : the interface we would like from the service to use
  - *Adaptee* : the existing class that provides the service
  - *Adapter* : The class that adapts the interface provided by adaptee to the Target interface



Adapter : Alternative solution



Adapter

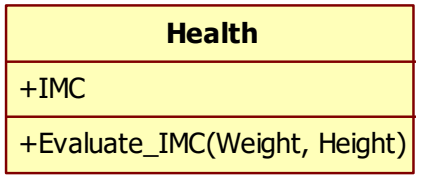
Uses:

Adapter can be used :

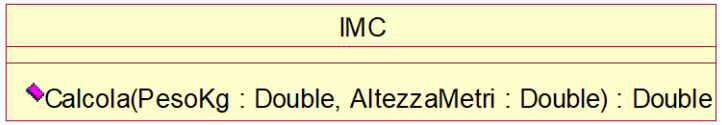
1. If you want to use an existing class (Adaptee) whose interface is incompatible
2. If you want to create a (reusable) class that will have to collaborate with classes that are not predictable at the time of its creation
3. If you want to reuse a set of existing subclasses, but it is inconvenient to adapt the interface by creating a subclass for each: better to create an adapter for the parent class

Example

In the creation of an object-oriented software for the US market, we want to create a Health class (Client class) that provides, among other things, the body mass index (IMC) of a person, according to the interface shown below:



We do not know the algorithm that allows to calculate this index, thus we decided to reuse an IMC class (Adaptee class), taken from a software made for the Italian market that has the following interface:



We know that 1 pound is equal to 0.453 kilograms and 1 inch is equal to 0.0254 meters

We want to Design the class diagram at the level of detail that is able to solve the proposed problem by freeing the programmer of the Health class from the knowledge of the interface and implementation of the IMC class, using the Adapter design pattern.

Solution

```
public class Main {
   public static void main(String[] args) {
      Health health = new Health();
      health.evaluateIMC(165,70);
      System.out.println(health.IMC);
   }
}

//Classe che dobbiamo realizzare (inglese)

public class Health {
   public double IMC;
   public double evaluateIMC(double weight, double height) {
      Target adapter = new Adapter();
      IMC = adapter.evaluate(weight, height);
      return IMC;
   }
}

//interfaccia che richiediamo

public interface Target {
   public double evaluate(double weight, double height);
}
```

```
//Classe che va ad adattare la soluzione esistente

public class Adapter implements Target {
   public Adaptee adaptee = new Adaptee();
   public double evaluate(double weight, double height) {
      return adaptee.calcolaIMC(weight/0.453,height/0.0254);
   }
}


public class Adaptee {
   public double calcolaIMC(double pesoKg, double altezzaM){
      IMC imc = new IMC();
      return imc.calcolaIMC(pesoKg,altezzaM);
   }
}


public class IMC {
   public double calcolaIMC(double pesoKg, double altezzaM){
      return pesoKg/(altezzaM*altezzaM);
   }
}
```

Typical Uses of Adapter

- The Adapter class intervenes continuously when dealing with services or microservices:
  - The design of our software makes us design an interface
  - An existing microservice provides the service we need, but with its own interface
  - We implement an Adapter class that interacts with the class that provides the service, converts the parameters provided by the service (Adaptee) into the ones we are interested in, possibly reorganizing and processing them, and provides them to the Target class
    - The target class could also be an Interface, since it only requests a service
  - Why don't we change the service (Adaptee) directly?
    - Because we didn't do it and we only have the interface or because it is already used by others

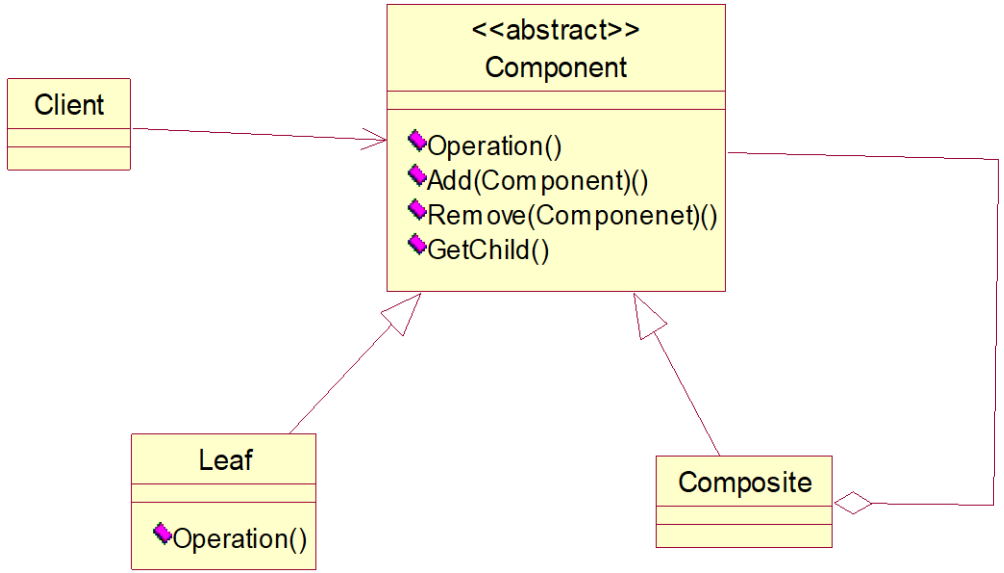
Composite

(Structural)

		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

Design Pattern Composite

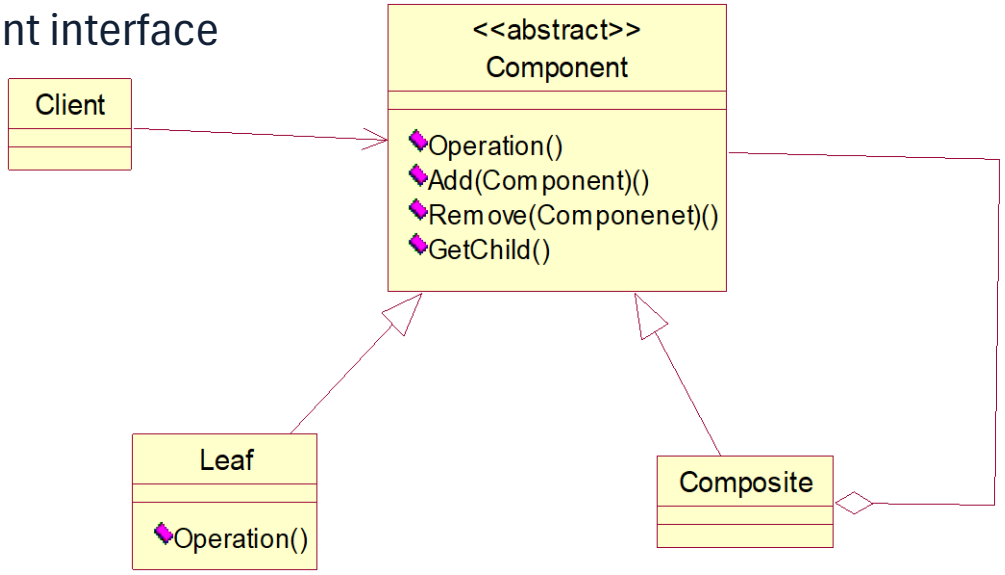
- Name : Composite
- Purpose: Compose objects into tree structures that can represent all-part hierarchies. You want clients to treat both composite objects and individual objects the same way



Design Pattern Composite

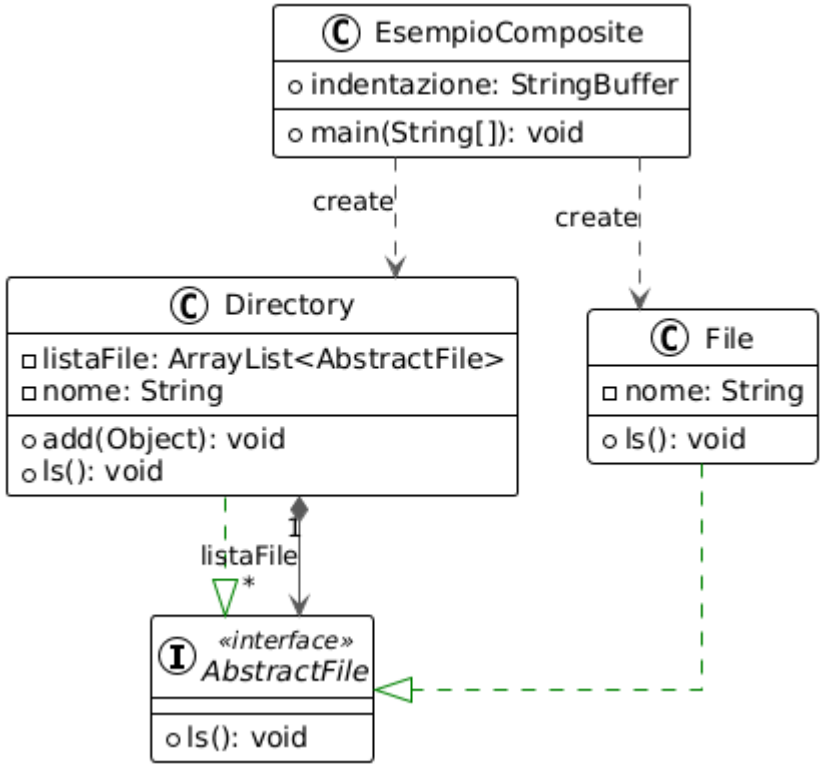
Partecipants

- **Component** declares the object interface and default behavior, declares an interface for accessing and managing its child components
- **Leaf** implements childless component objects and their behavior
- **Composite** defines the behavior of components with children, stores child components, implements child-related operations defined by the Component interface
- **Client** uses composition objects using the Component interface



Example : File System

- We want to implement a directory and a file tree
  - With the **ls** function
  - Taking into account the *indentation*
- A **Directory** can be composed of **Files** and other directories, possibly empty
- **AbstractFile** represents items that can be included in Directories



Composite: Esempio (1/3)

```
interface AbstractFile
{
  public void ls();
}

class File implements AbstractFile
{
    public File(String n)
    {
        nome = n;
    }
    public void ls()
    {
        System.out.println(EsempioComposite.indentazione + nome);
    }
    private String nome;
}
```

Composite: Esempio (2/3)

```
class Directory implements AbstractFile {
    private String nome;
    private ArrayList listaFile = new ArrayList();

    public Directory(String name)
        { nome = name; }

    public void add(Object obj)
        { listaFile.add(obj); }

    public void ls()    {
        System.out.println(EsempioComposite.indentazione + nome);
        EsempioComposite.indentazione.append("   ");
        for (int i = 0; i < listaFile.size(); ++i)   {
            AbstractFile obj = (AbstractFile)listaFile.get(i); //Polimorfiscmo
            obj.ls();
        }
        EsempioComposite.indentazione.setLength(EsempioComposite.indentazione.length() - 3);
    }
}
```

Composite: Esempio (3/3)

```
public class EsempioComposite
{
    public static StringBuffer indentazione = new StringBuffer();

    public static void main(String[] args)
    {
        Directory one = new Directory("dir111"),
        two = new Directory("dir222"),
        thr = new Directory("dir333");
        File a = new File("a"), b = new File("b"),
        c = new File("c"), d = new File("d"), e = new File("e");
        one.add(a);
        one.add(two);
        one.add(b);
        two.add(c);
        two.add(d);
        two.add(thr);
        thr.add(e);
        one.ls();
    }
}
```

Output

```
dir111
  a
  dir222
    c
    d
    dir333
      e
  b
```

Consequences

- Composite :
  - Allows to define class hierarchies consisting of primitive and composite objects.
    - Primitive objects can be composed to form more complex objects, which can be composed recursively.
    - At all points where the client expects to use a primitive object, a composite object can be used indifferently.
  - Simplifies the client.
    - Clients can use composite structures and individual objects uniformly.
    - Clients usually don't know (and shouldn't even care) whether they're working with a leaf or a composite component.
  - Is flexible
    - Makes it easier to add new types of components (new Leaf or Composite subclasses can be automatically used in existing structures and work with client code)
  - Is often used to instantiate the hierarchy of widgets of a GUI

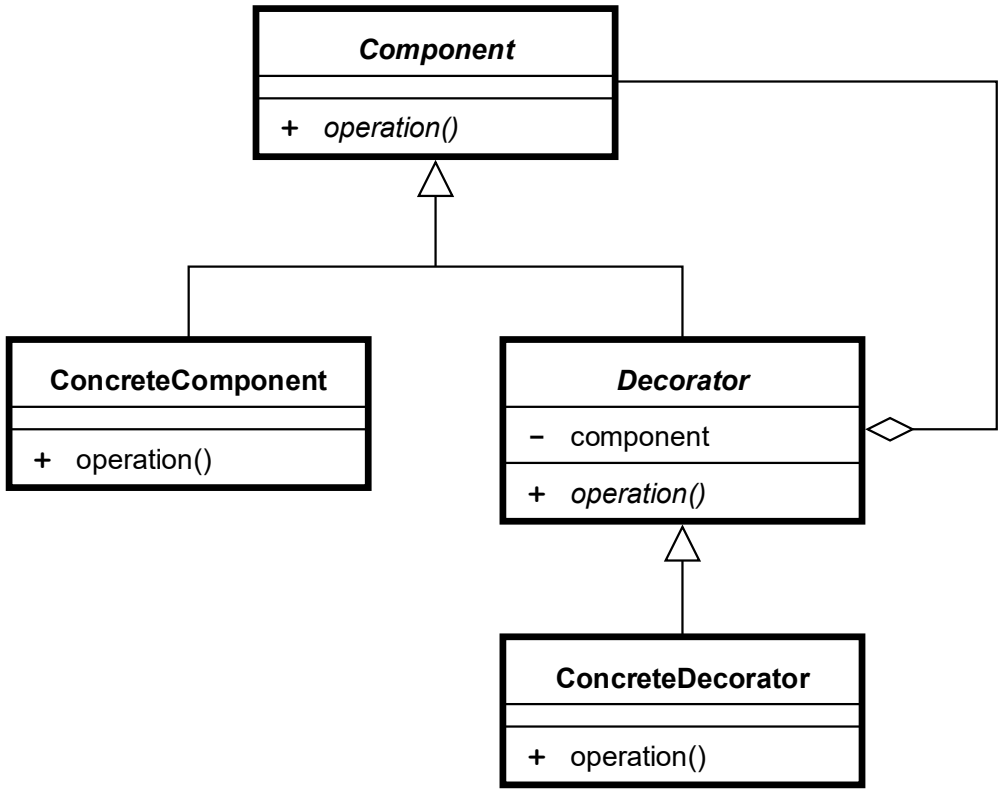
Decorator

(Structural)

Pattern Decorator

- The pattern decorator allows to create complex objects by adding functionality to simpler objects.
- The complex object includes the simple object and "decorates" it by adding functionality
  - The original object is passed to the decorator's constructor as a parameter and the "decorated" object is returned
  - Multiple decorators can also be chained to each other, thus incrementally adding functionality to the complex class (which is represented by the last link in the chain).
    - The concatenation of decorators can take place according to an arbitrary composition: the number of possible behaviors of the composite object therefore varies with combinatorial law with respect to the number of decorators available.
- This pattern is an alternative to the use of single or multiple inheritance. With inheritance, in fact, the addition of features occurs statically according to the links defined in the class hierarchy and it is not possible to obtain at run-time an arbitrary combination of features, nor their addition/removal.

Schema concettuale



- Component is abstract/interface
- ConcreteComponent is the basic implementation of component
- Decorator is an abstract class that includes a Component and inherit operation
- ConcreteDecorator is a class that improves Component because it can take on all its characteristics (it has a component attribute) and can modify its methods in override

Pizza Example: Interface, Decorator, and Base Class

```
//component
public interface Pizza {String getDescription();
double getCost();
}

//Decorator
public abstract class PizzaDecorator implements Pizza {
protected Pizza pizza;

public PizzaDecorator(Pizza pizza) {
this.pizza = pizza;}

@Override
public String getDescription() {
return pizza.getDescription(); }
@Override
public double getCost() {
return pizza.getCost(); }
}

//ConcreteComponent
public class OlioEPomodoro implements Pizza {

@Override
public String getDescription() {
return "Pizza Olio e Pomodoro";}

@Override
public double getCost() {
return 3.00;}
}
```

Example : decorators

```
// Concrete Decorators
public class Mozzarella extends PizzaDecorator {
public Mozzarella(Pizza pizza) {
super(pizza); } //Mozzarella è una Pizza

@Override
public String getDescription() {
return pizza.getDescription() + ", Mozzarella";}

@Override
public double getCost() {
return pizza.getCost() + 1.50;}
}
```

```
public class Prosciutto extends PizzaDecorator {
public Prosciutto(Pizza pizza) {
super(pizza); } // Prosciutto è una Pizza

@Override
public String getDescription() {
return pizza.getDescription() + ", Prosciutto"; }

@Override
public double getCost() {
return pizza.getCost() + 1.00; }
}
```

Pizza example : Main

```
public class Main {
public static void main(String[] args) {
// Ordiniamo una pizza base
Pizza myPizza = new OlioEPomodoro();
System.out.println("Pizza ordinata: " + myPizza.getDescription());
System.out.println("Costo: €" + myPizza.getCost());
System.out.println("--------------------");

// Aggiungiamo formaggio alla pizza
myPizza = new Mozzarella(myPizza);
System.out.println("Pizza ordinata: " + myPizza.getDescription());
System.out.println("Costo: €" + myPizza.getCost());
System.out.println("--------------------");

// Pizza doppia Mozzarella
Pizza mia = new OlioEPomodoro();
mia = new Mozzarella(mia);
mia = new Mozzarella(mia);
System.out.println("\n*** La mia pizza ***");
System.out.println("Descrizione: " + mia.getDescription());
System.out.println("Costo totale: €" + mia.getCost());

//prosciutto e funghi
Pizza tua= new OlioEPomodoro();
tua = new Prosciutto(tua);
tua = new Fungo(tua);
System.out.println("\n*** La tua pizza ***");
System.out.println("Descrizione: " + tua.getDescription());
System.out.println("Costo totale: €" + tua.getCost());
}
}
```

Decorators in Java

- The Decorator pattern is often used within the Java libraries
- Examples:
  - Reader()
  - BufferedReader(Reader in)
  - InputStream()
  - InputStreamReader(InputStream in)
  - BufferedInputStream(InputStream in)

```
BufferedReader in = new BufferedReader(new
  InputStreamReader(System.in));
```

Decorator : Advantages and disadvantages

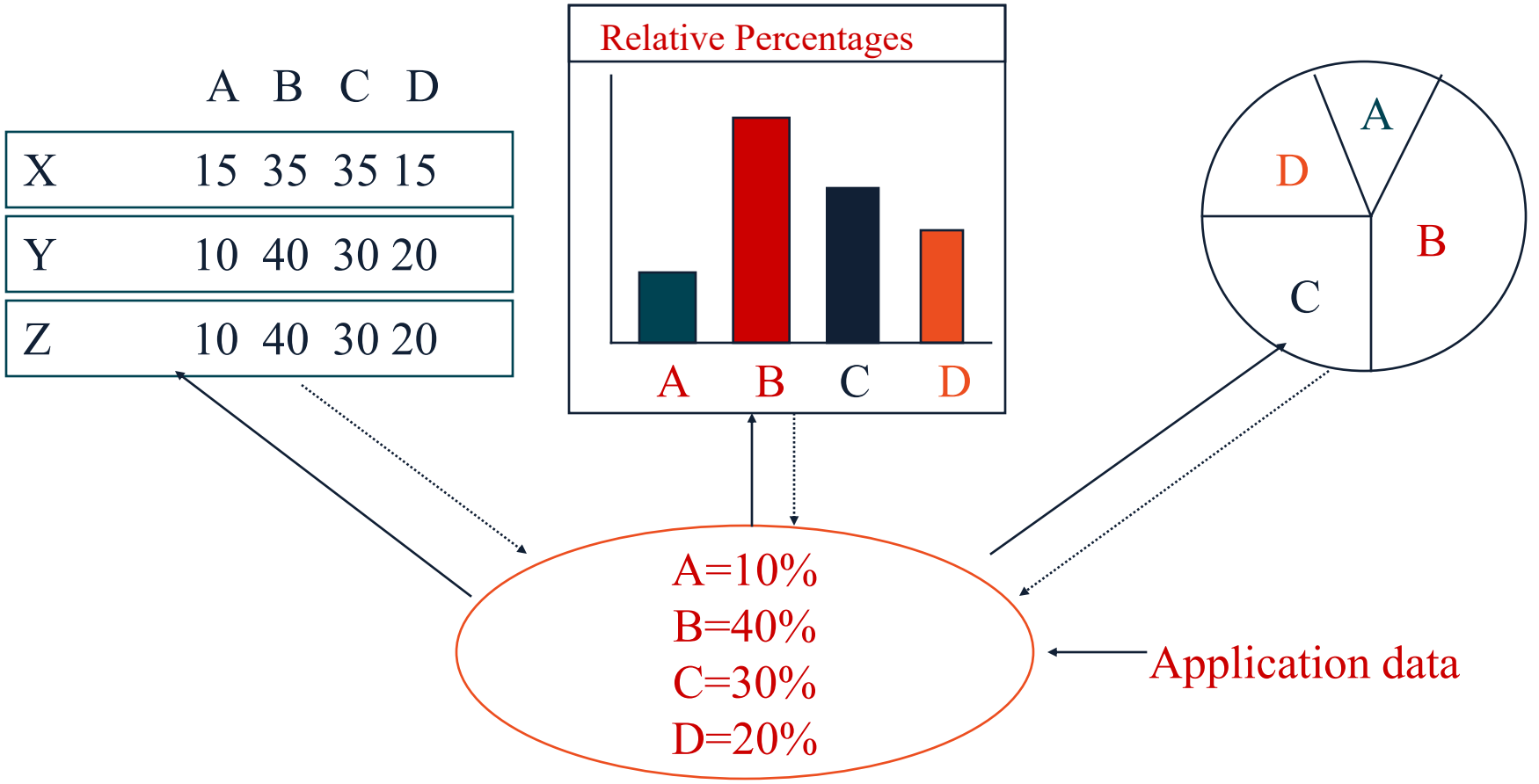
- Advantages :
  - By combining decorators we can get an unlimited number of combinations of builder calls
    - We don't need to declare classes «prosciutto e funghi», «margherita», «margherita prosciutto e funghi», «margherita doppia mozzarella»
- Disadvantage :
  - The construction of an object implies the need to instantiate more than one object and to call more than one constructor: the pattern is inefficient from a computational point of view (and, consequently, also from an energy point of view)

Observer

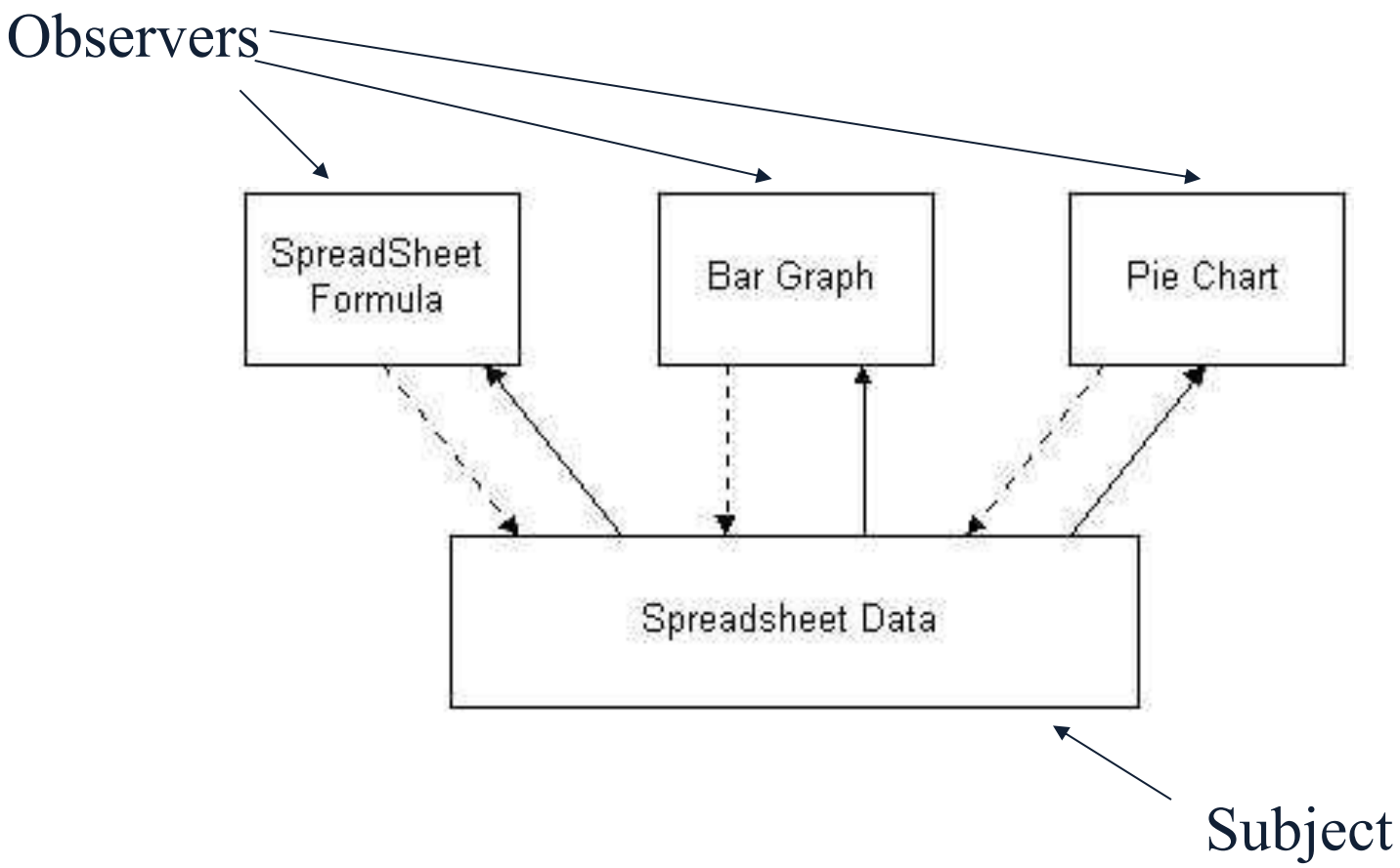
(Behavioral)

		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

Patterns by Example:
Multiple displays enabled by Observer



Schematic Observer Example

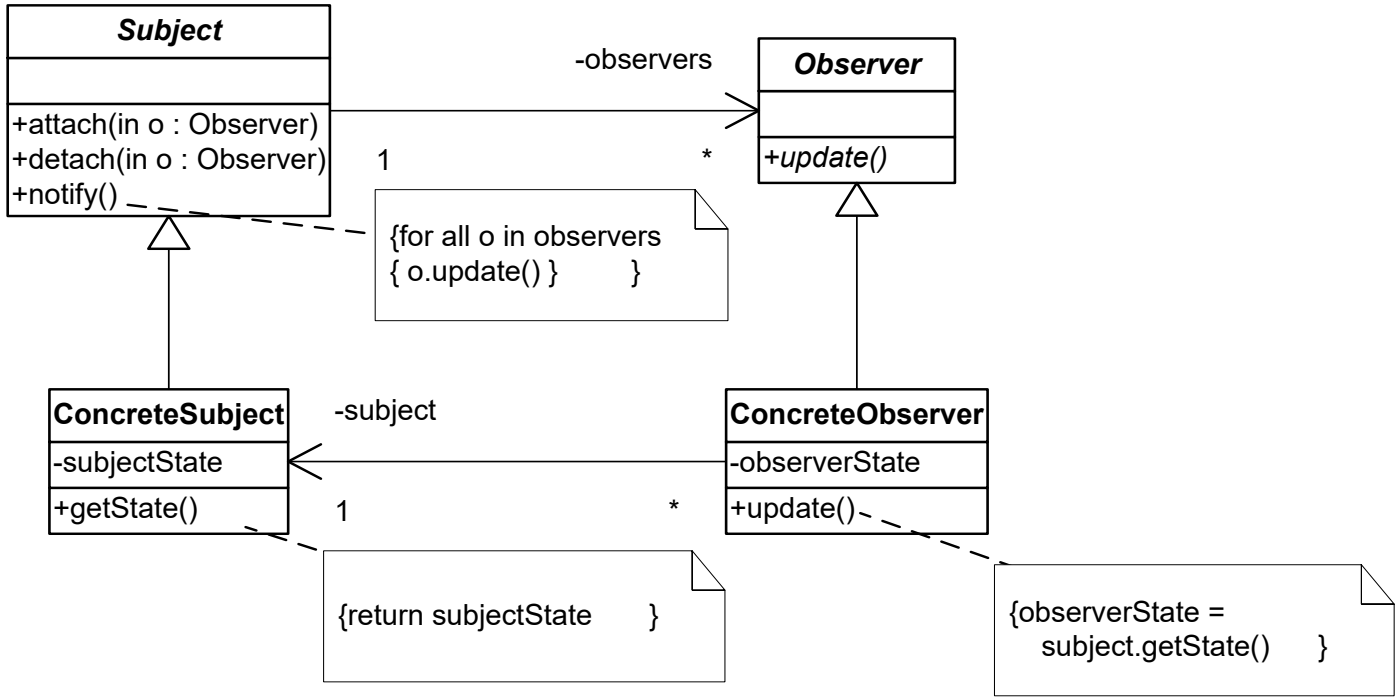


Observer (Behavioral)

- 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 – Example 1/2

```
//osservatore (interfaccia) – invariante –
potrebbe essere una interfaccia di libreria
public interface Osservatore {
  public void aggiorna();
}



//osservatore (concreto)

//e' legato ad un soggetto del quale vuole
conoscere le modifiche
public class OsservatoreConcreto implements
Osservatore {
  private int stato; //stato dell'osservatore
  private Soggetto osservato; //sogg. osservato

  public OsservatoreConcreto(Soggetto soggetto) {
    this.osservato=soggetto; //sogg. osservato
  }


 @Override
 public void aggiorna() {
   stato = osservato.getStato();
   System.out.println("Nuovo Stato:");
   System.out.println(stato);
 }
}
```

```
//soggetto (astratto) – invariante – potrebbe
essere una
classe di libreria

public abstract class Soggetto {
  protected int stato; //lo stato del soggetto
originale
  protected ArrayList<Osservatore> osservatori;

  public void registraOsservatore (Osservatore o)
{
    osservatori.add(o);
  }


  // informa osservatori del cambiamento di stato

  protected void update() {
     for (int i=0; i<osservatori.size(); i++) {
        Osservatore Osvtr = osservatori.get(i);
        Osvtr.aggiorna();
     }
  }
  public abstract void setStato(int s);
  public abstract int getStato();
}
```

```
//soggetto (concreto)

public class SoggettoConcreto extends Soggetto {
    public SoggettoConcreto() {
       osservatori=new ArrayList<Osservatore>();
       stato=0;
    }
    @Override
    public void setStato(int s) {
       stato = s;
       update(); //rispetto ad una normale classe
c'è questa chiamata extra
    }
    @Override
    public int getStato() {
       return stato;
    }
}
```

Esempio 2/2

```
public class Main {

    public static void main(String[] args) {
       Soggetto soggetto = new SoggettoConcreto();
       Osservatore o1 = new OsservatoreConcreto (soggetto);
       soggetto.registraOsservatore(o1);

       Osservatore o2 = new OsservatoreConcreto (soggetto);
       soggetto.registraOsservatore(o2);

       soggetto.setStato(20);
    }
}
```

Strategy

(Behavioural)

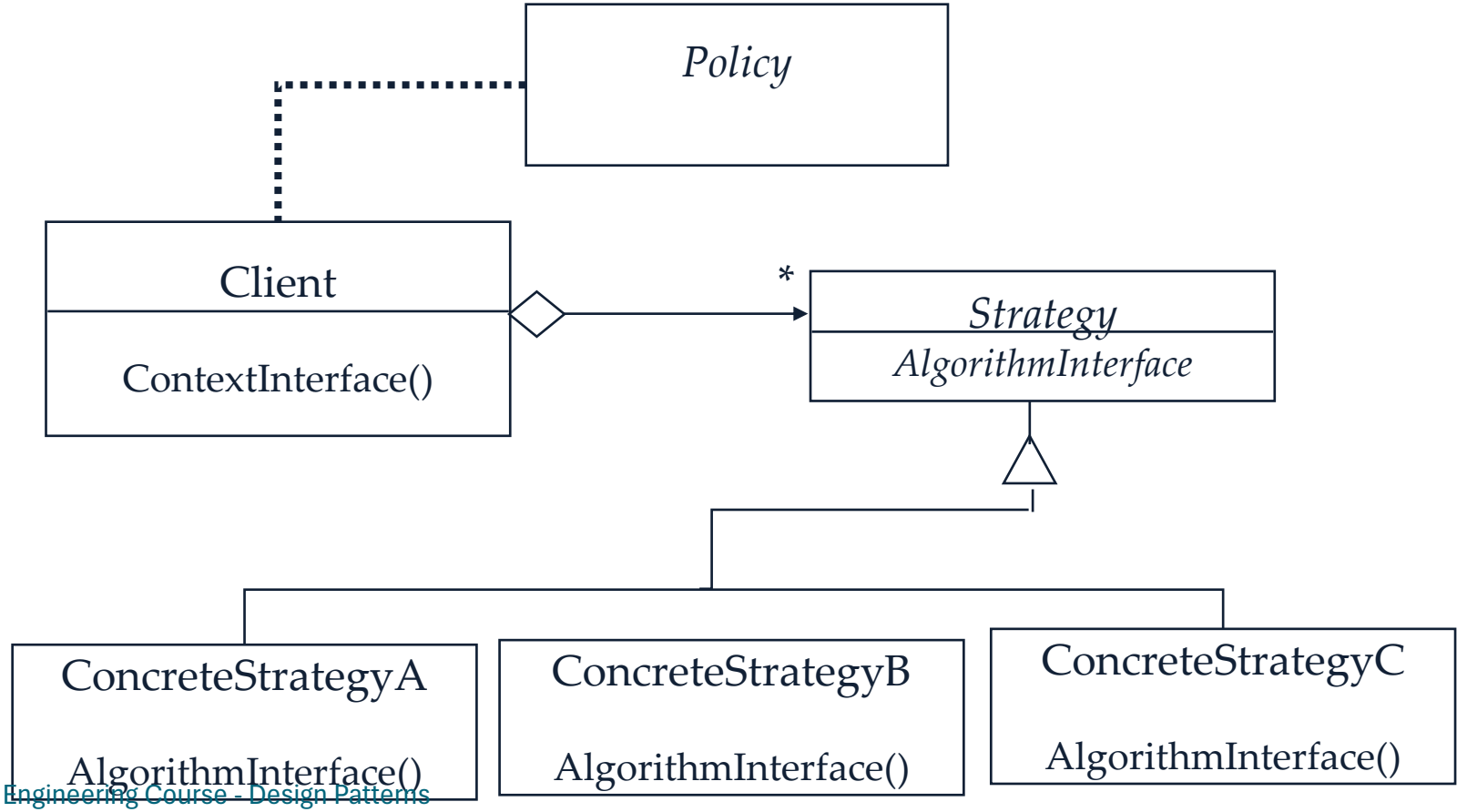
		Purpose		
		Creational	**Structural**	**Behavioral**
Scope	**Class**	Factory Method	Adapter (class)	Interpreter Template Method
	Object	Abstract Factory Builder Prototype Singleton	Adapter (object) Bridge Composite Decorator Flyweight Facade Proxy	Chain of Responsibility Command Iterator Mediator Memento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

Strategy Pattern

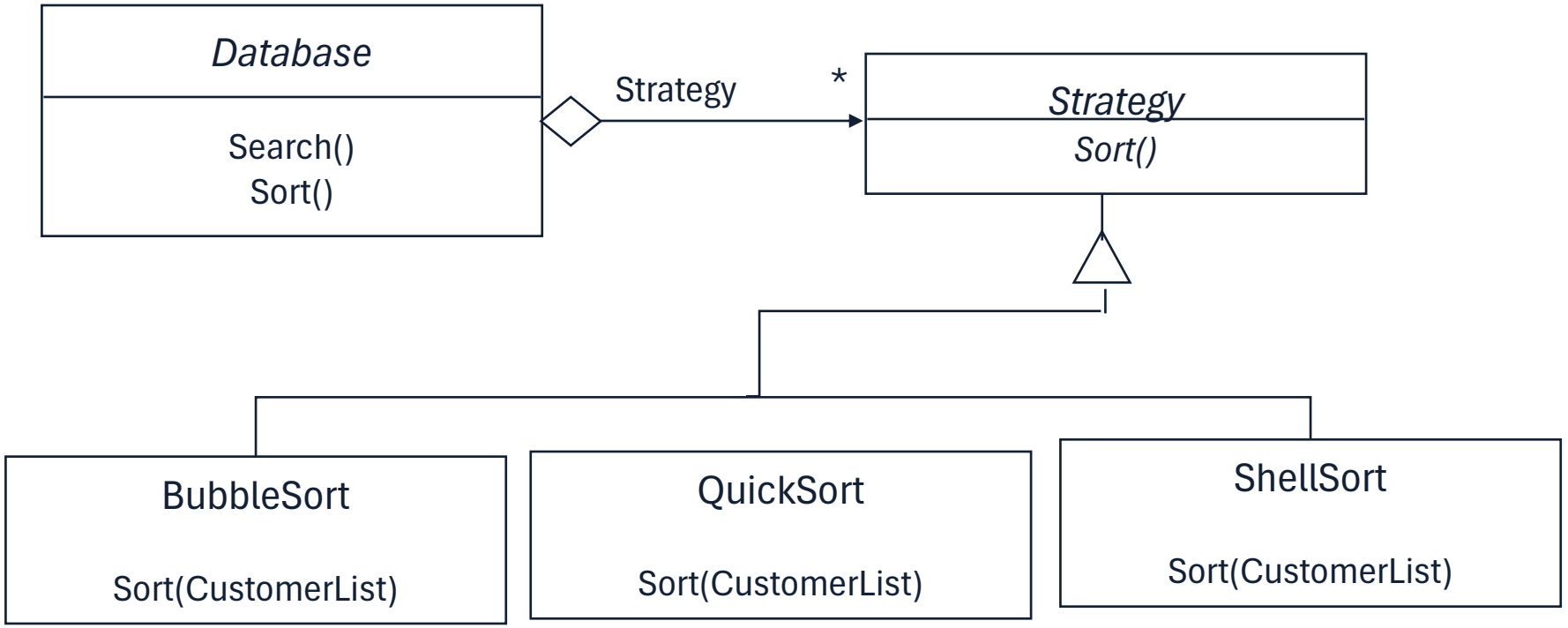
- Many different algorithms exists for the same task
- Examples:
  - Breaking a stream of text into lines
  - Parsing a set of tokens into an abstract syntax tree
  - Sorting a list of customers
- The different algorithms will be appropriate at different times
- We don’t want to support all the algorithms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

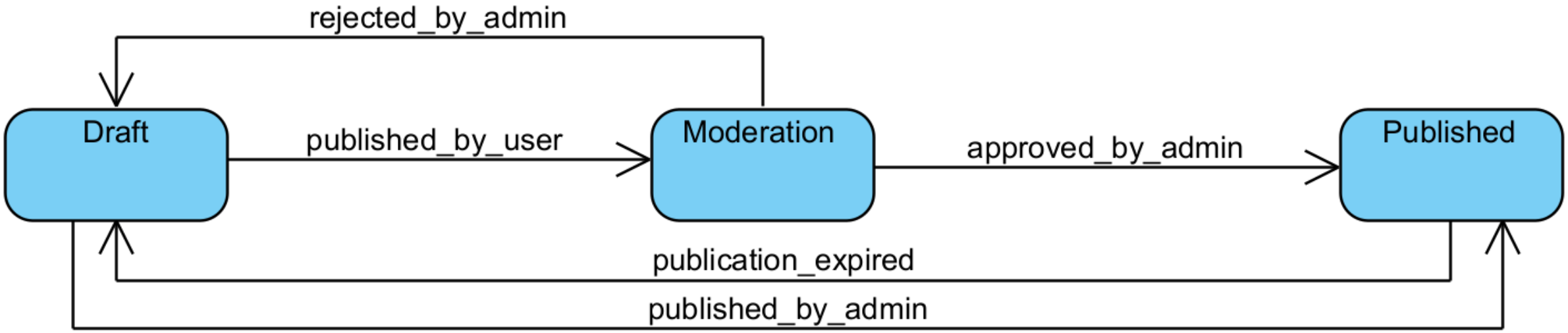
State

(Behavioural)

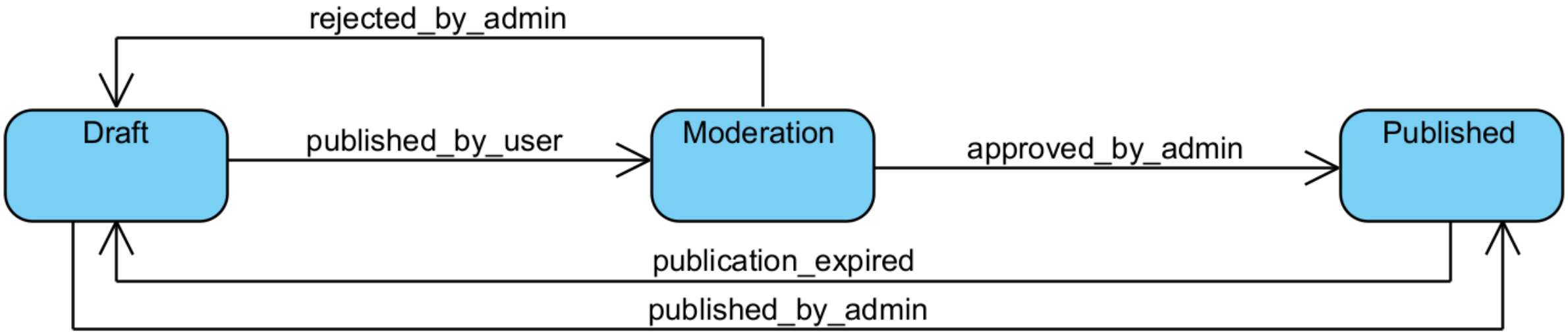
		Purpose		
		Creational	**Structural**	**Behavioral**
Scope	**Class**	Factory Method	Adapter (class)	Interpreter Template Method
	Object	Abstract Factory Builder Prototype Singleton	Adapter (object) Bridge Composite Decorator Flyweight Facade Proxy	Chain of Responsibility Command Iterator Mediator Memento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

MOTIVATION

- Classes/Subsystems can be seen as Reactive systems (see Lect. 9)
  - They **react** to external stimuli (i.e.: events) based on their current **state**
- Suppose you have a **Document** class, and the behaviour of a Document is described by the following Statechart



MOTIVATION



- How would you implement the **Document** class?
  - Probably, you'd find yourself using lots of conditional statements

MOTIVATION

- Code may look like this
- Might seem manageable...
- As we add more states and more transitions, most methods will end up including monstrous conditionals
- Changes to transition logic will likely require changing state conditionals in every method!

```
public class Document {
  private String state;
  // ...
  public void publish() {
    switch (state) {
      case "draft":
        state = "moderation"; break;
      case "moderation":
        if ("admin".equals(user.getRole())) {
           state = "published";
        }
        break;
      case "published": // Do nothing.
        break;
     // ...
```

THE STATE PATTERN

- Create new classes for all possible states of an object
- Delegate state-specific behaviour to these classes

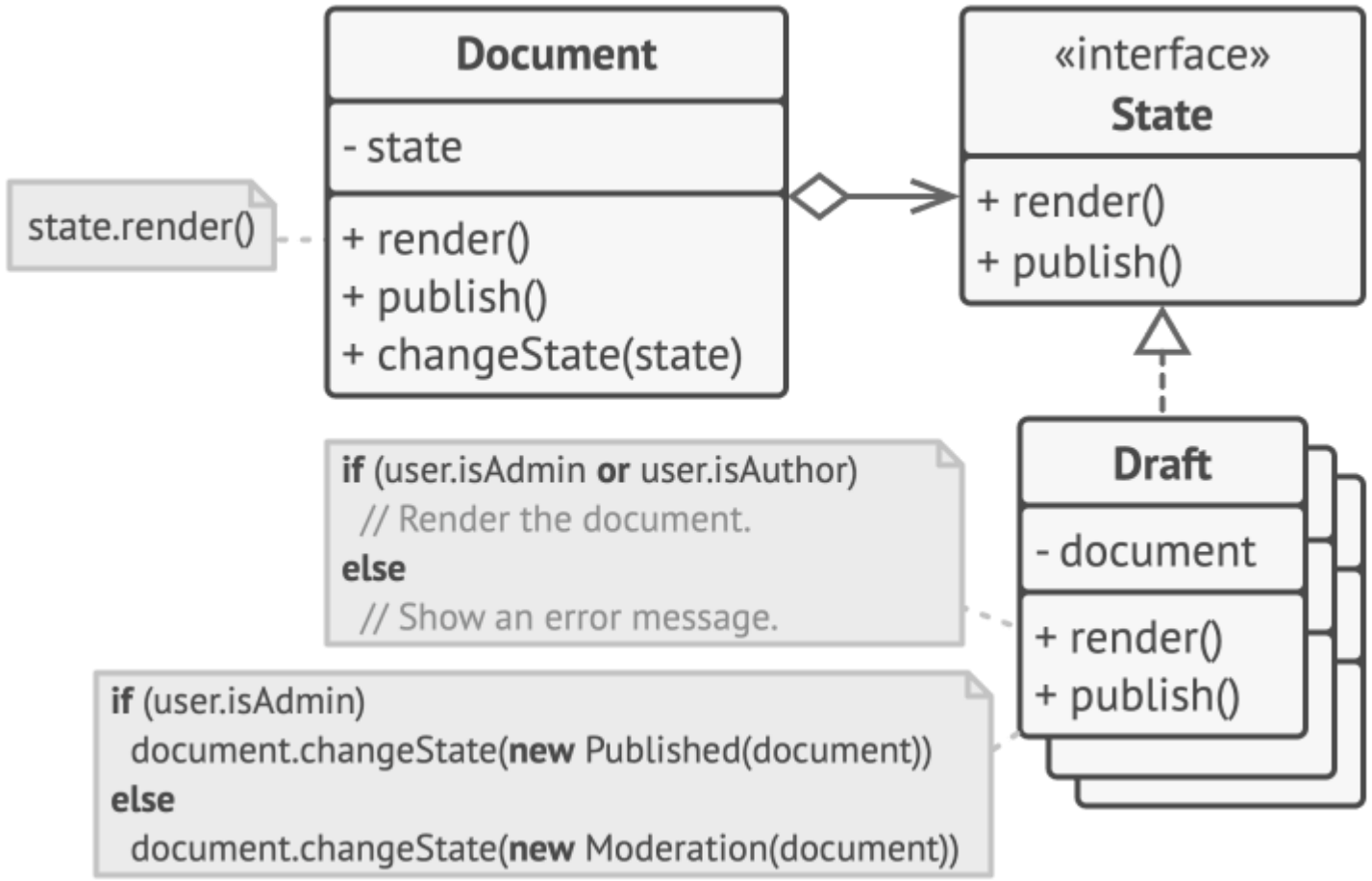


Image from https://refactoring.guru/design-patterns/state

STATE VS STRATEGY

- State is somewhat similar to Strategy
  - The implementation of certain operations is delegated to «pluggable» objects
- There is one key difference:
  - In the State pattern, the particular states may be aware of each other and initiate transitions from one state to another
    - E.g.: One state may decide what the next state will be
  - In the Strategy pattern, strategies almost never know about each other (they provide separate, independent implementations of an operation)

Dependency Injection

Dependency Injection

- It occurs when the resources on which a component depends are passed (injected) from outside rather than instantiated by the component itself during its execution
- The component have to declare (for example through its constructor) the type (possibly generic) and the quantity of all the resources on which it depends
- References to the required resources will then be passed (injected) e.g. through the constructor

Simple example

```
public class MyClass {
    private Logger logger;

    public MyClass(Logger logger) {
        this.logger = logger;
        // write an info log message
        logger.info("This is a log message.")
    }
}
```

- MyClass can write into a log
- The log object on which it writes is given by the object of type Logger passed as a parameter
- https://www.vogella.com/tutorials/DependencyInjection/article.html

Pattern consequences

- It makes the class more generic, as it separates its behavior from the specific object being passed
  - It introduces a dependency between an object and a class
- It may not be very optimal from a coupling point of view
- It is also called **inversion-of-control** because it requires to create first the dependent object and then the one that will use it, going in the opposite direction to, for example, the BCE model

Dependency Injection and Testing

- Dependency Injection is fundamental for isolation and integration testing
  - It allows to perform a test on an object having entered as input a fictitious object on which it acts, replacing the concrete object that may not be available or not yet tested

- Example:
  - you want to test a class that calls a service, but the service is not yet available.
  - If the *dependency injection* pattern is followed, then the class under test will receive via the constructor a reference to the service object
  - When we are testing the class, we can create a fictitious service object and pass it to the class
  - In this way, the class can be executed even without the original service
- Counterexample:
  - If the class does not follow this pattern, the reference to the service is hardcoded in the class under test or is instantiated in it and cannot be modified at testing time

DESIGN PATTERNS: CAUTION ADVISED

Beware of tw

- Patterns sy ed solutions to a class of pro
  - Do not tre oted to your specific context! C
- Beware of u
  - «When th ooks like a nail»
  - Often nov n when simpler solutions would wo
  - Using the t solution!

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; handling state-dependant behaviour
- 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

Conclusions

- We could go on and on and present different patterns.
  - More patterns at reported the end of the slide, after the references
- 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

- 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
- AntiPatterns, Brown, et al.; Wiley, 1998; ISBN 0-471-19713-0
- https://refactoring.guru/design-patterns/

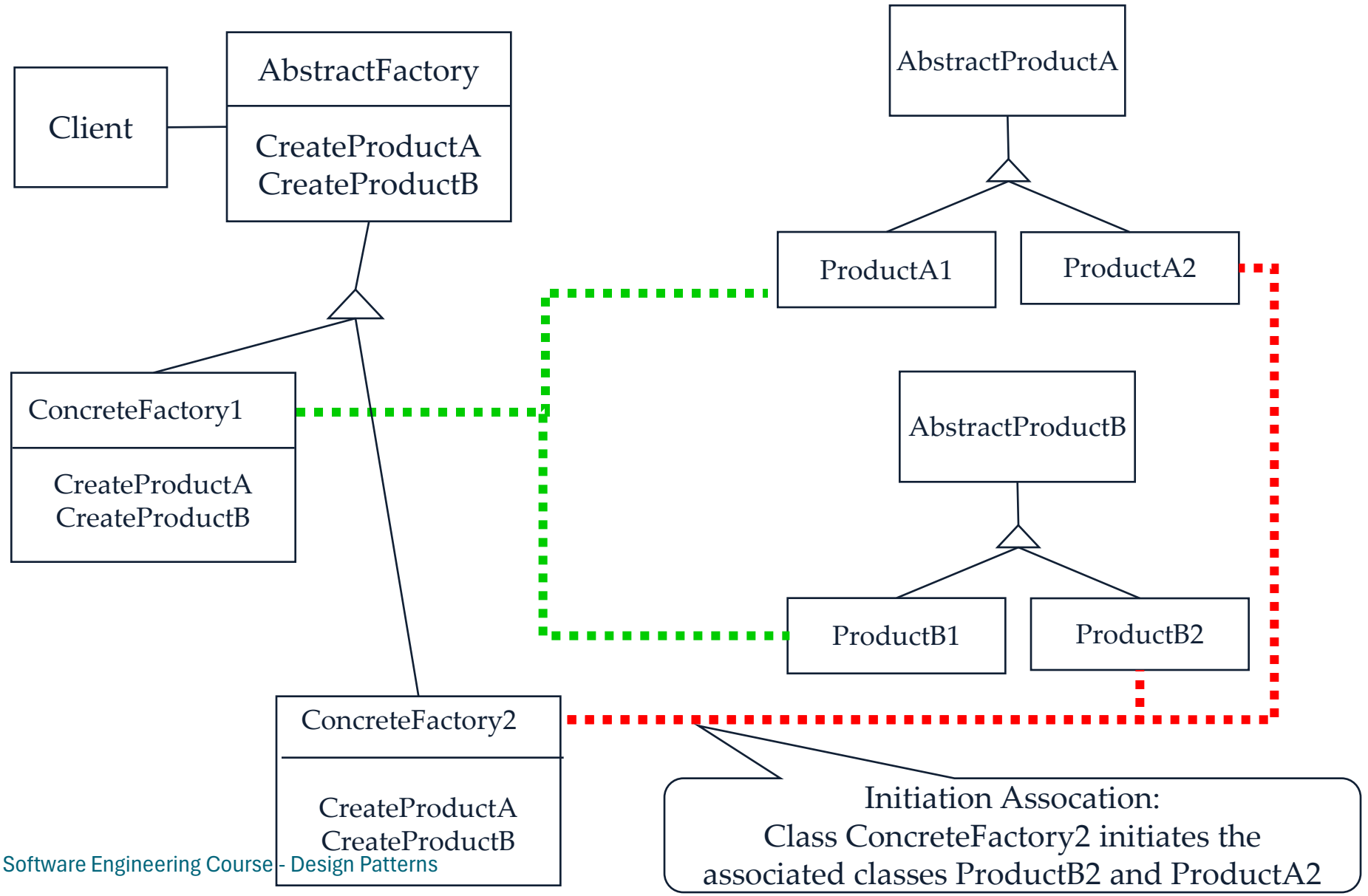
Abstract Factory

		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 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?

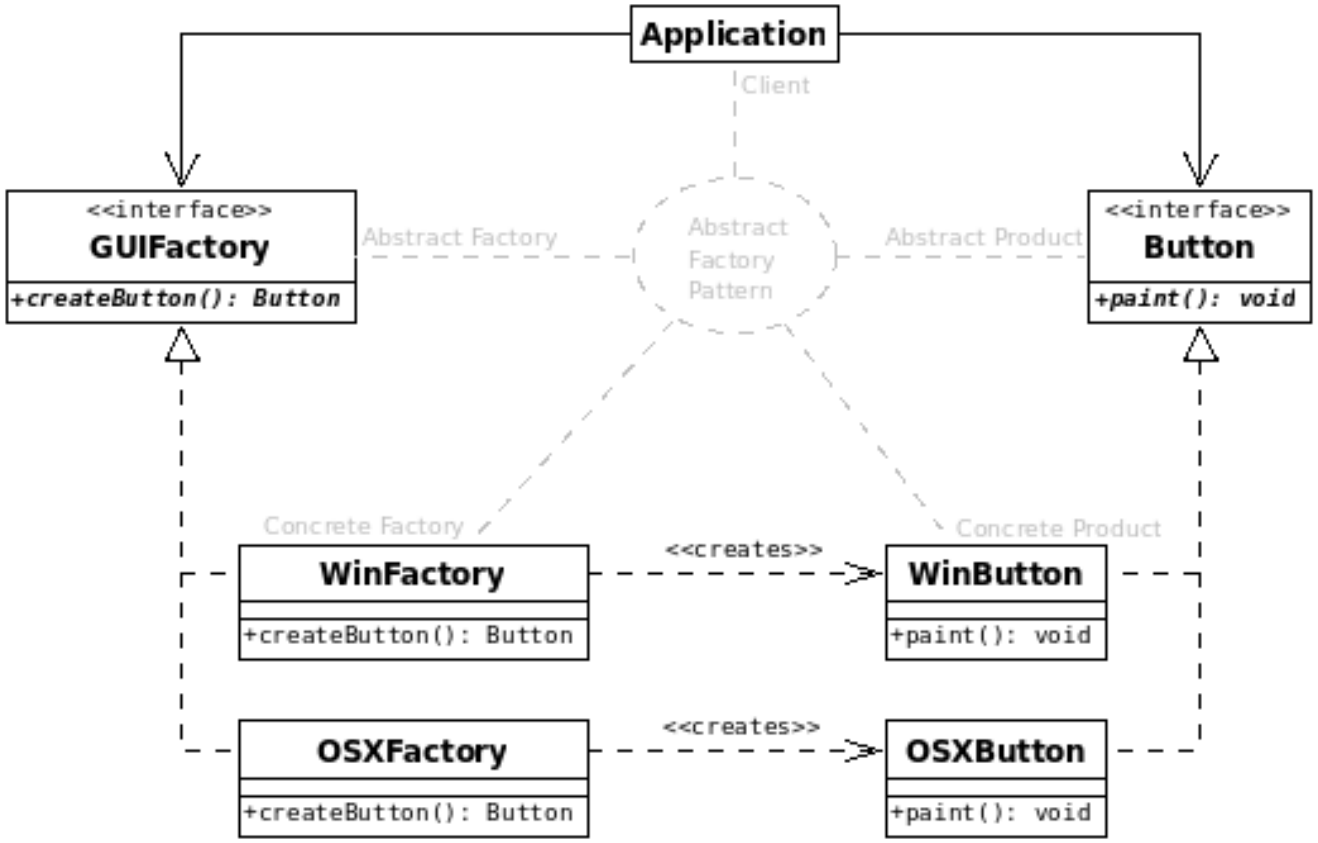
Abstract Factory



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();
}
```

Bridge

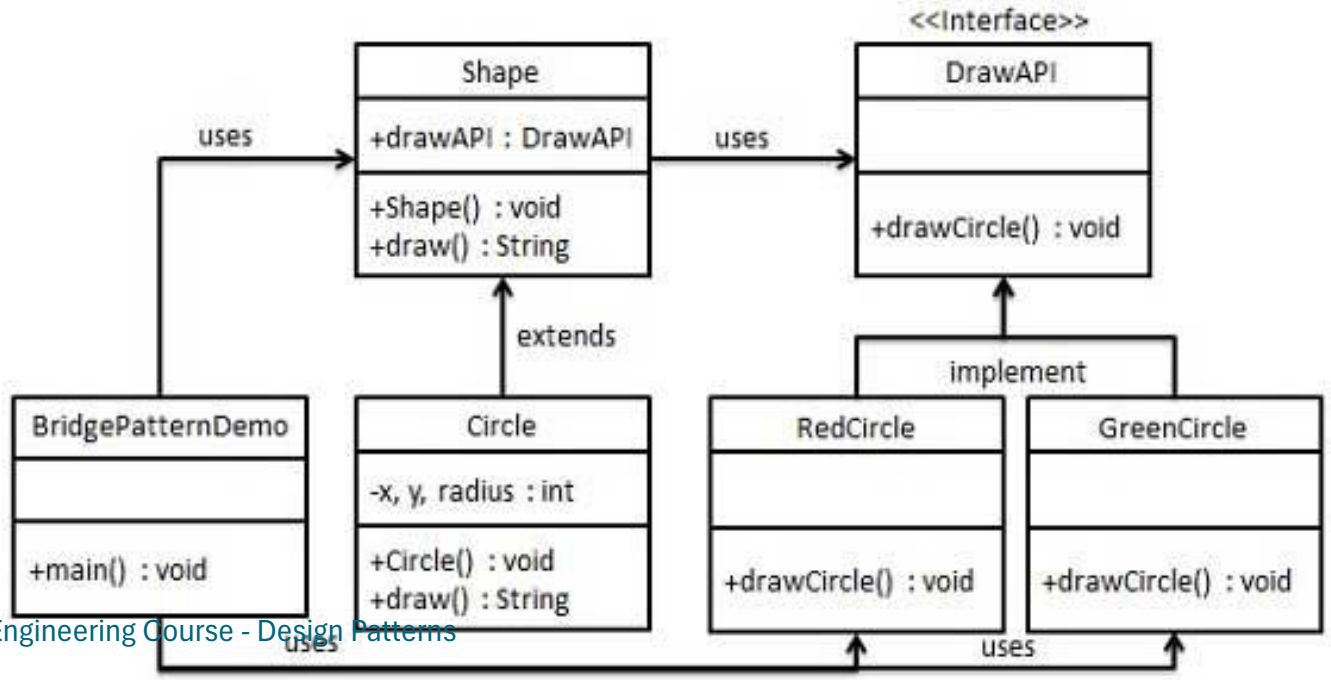
		Purpose		
		Creational	**Structural**	**Behavioral**
Scope	**Class**	Factory Method	Adapter (class)	Interpreter Template Method
	Object	Abstract Factory Builder Prototype Singleton	Adapter (object) Bridge Composite Decorator Flyweight Facade Proxy	Chain of Responsibility Command Iterator Mediator Memento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

Bridge 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.

Adapter (Class)

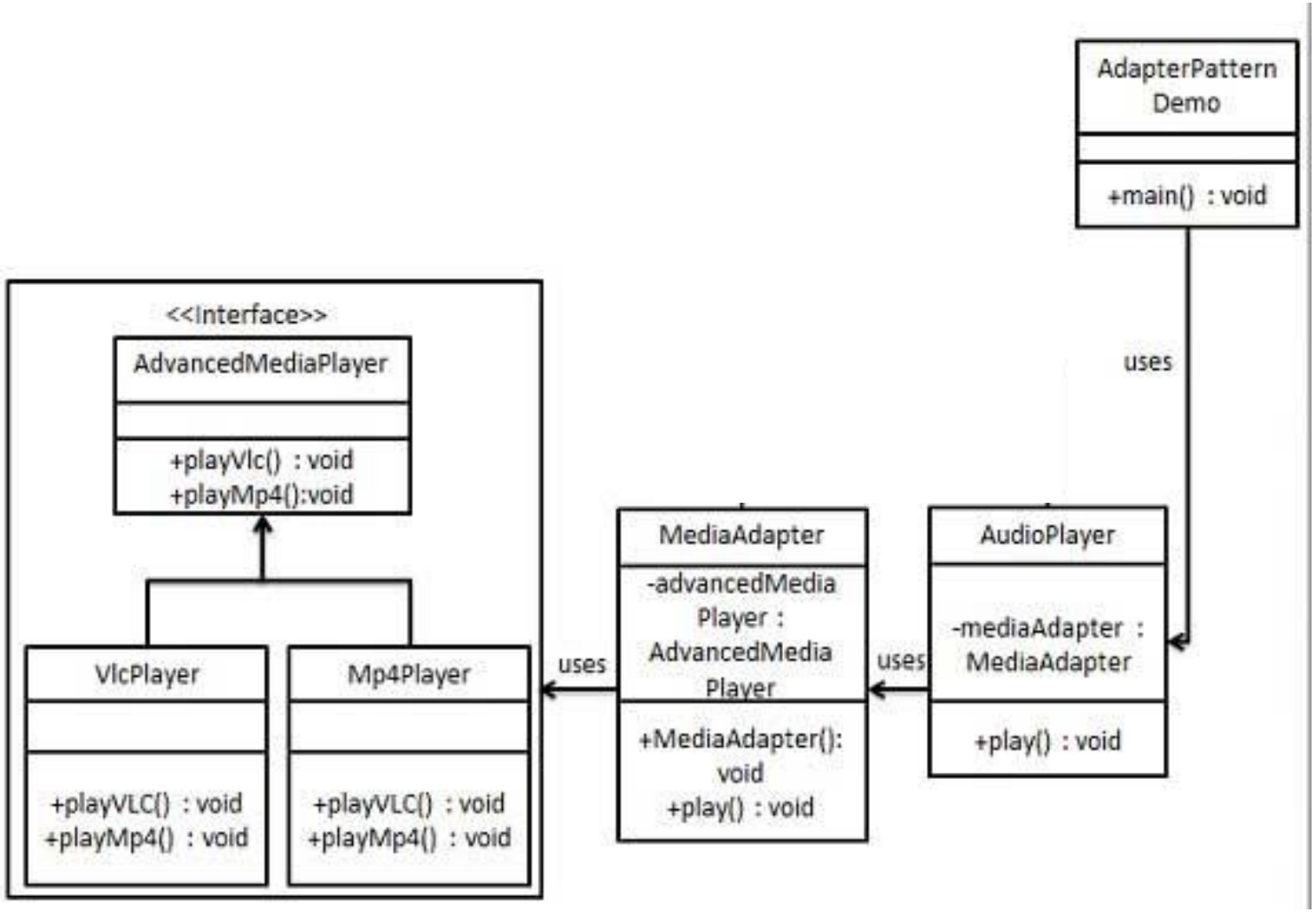
Structural

		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 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);
      }
   }
}
```

Adapter (Method)

Structural

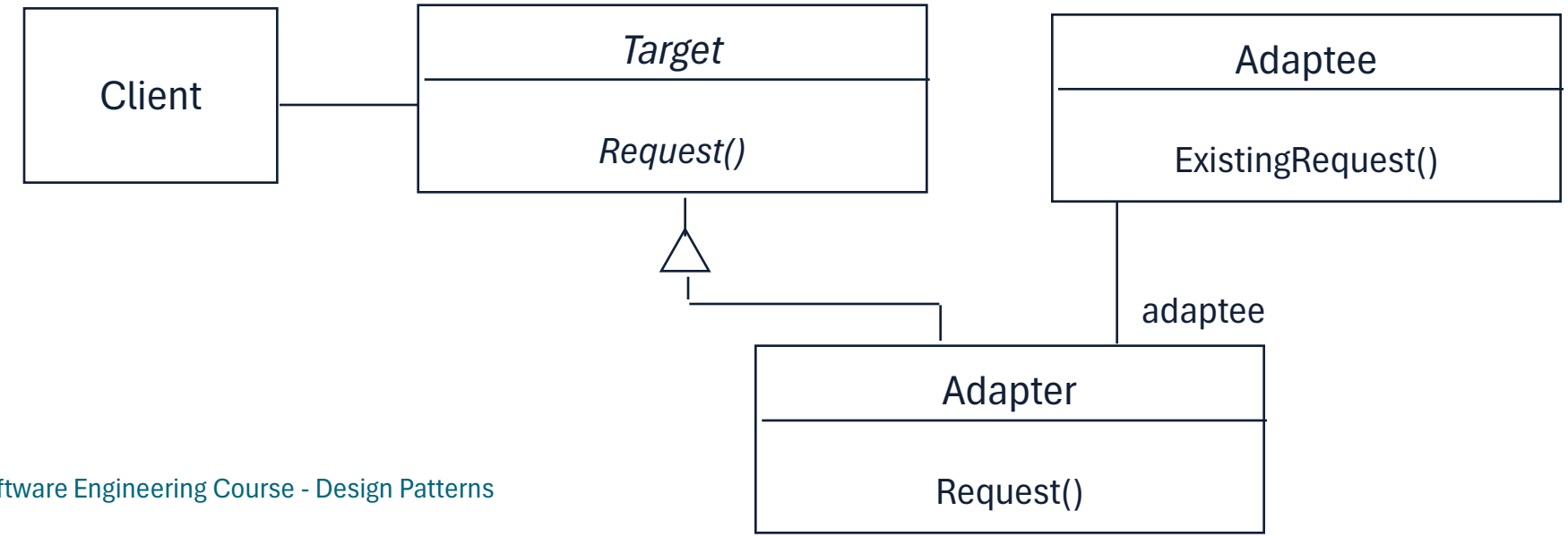
		Purpose		
		Creational	**Structural**	**Behavioral**
Scope	**Class**	Factory Method	Adapter (class)	Interpreter Template Method
	Object	Abstract Factory Builder Prototype Singleton	Adapter (object) Bridge Composite Decorator Flyweight Facade Proxy	Chain of Responsibility Command Iterator Mediator Memento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

Adapter Method 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.

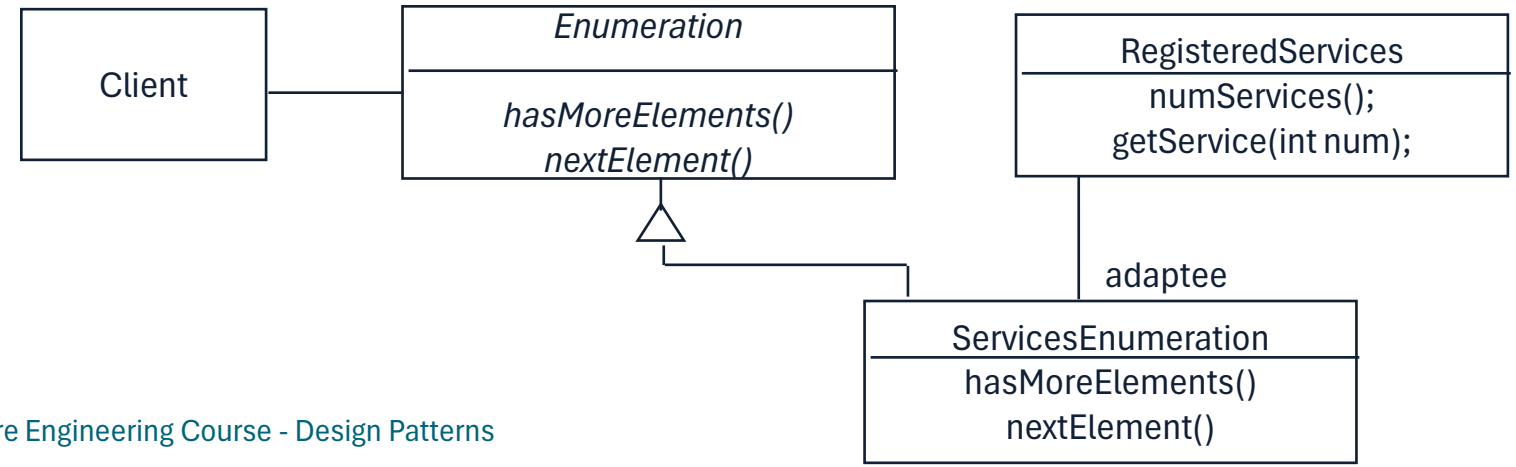
Adapter Method Pattern

- 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++);
  }
```



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

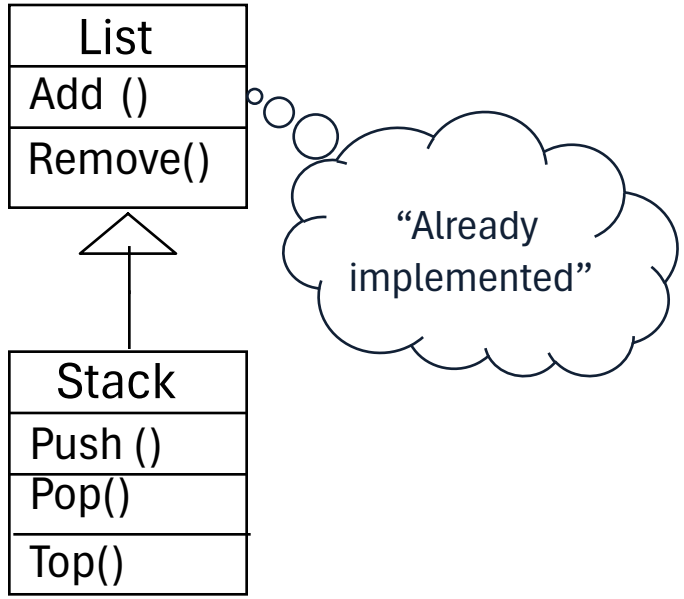
Implementation Inheritance vs Interface Inheritance

- 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()?

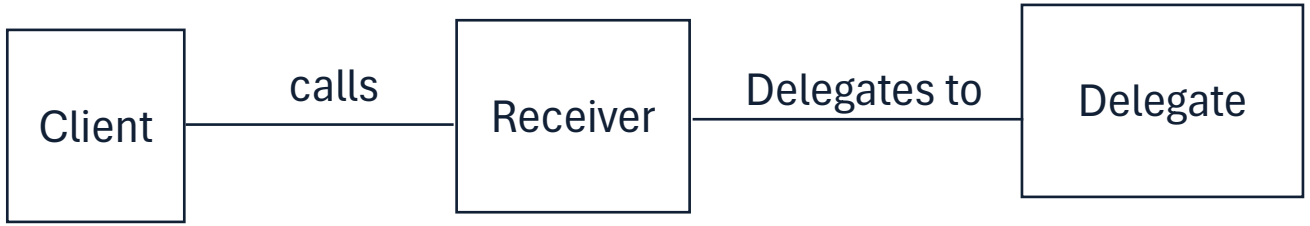


Problem with implementation inheritance

- 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()?

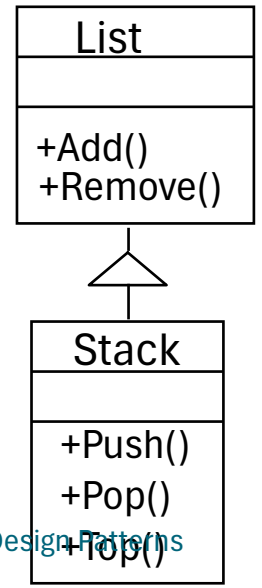
Delegation

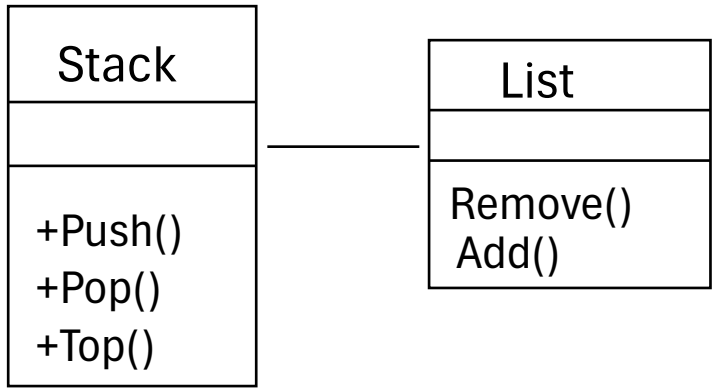
- Delegation is a way of making composition (for example aggregation) as powerful for reuse as inheritance
- In Delegation two objects are involved in handling a request
  - A receiving object delegates operations to its delegate.
  - The developer can make sure that the receiving object does not allow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

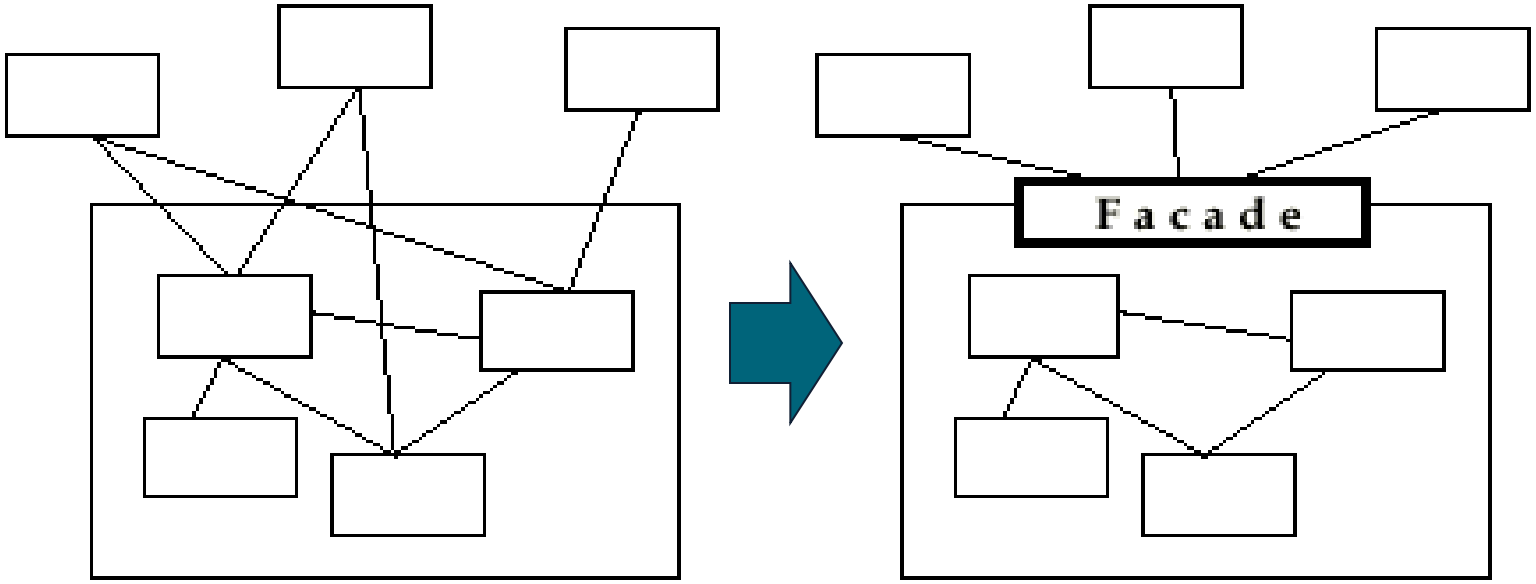
- Avoid to use implementation inheritance, always use interface inheritance
- A subclass should never hide operations implemented in a superclass
- If you are tempted to use implementation inheritance, use delegation instead

Facade

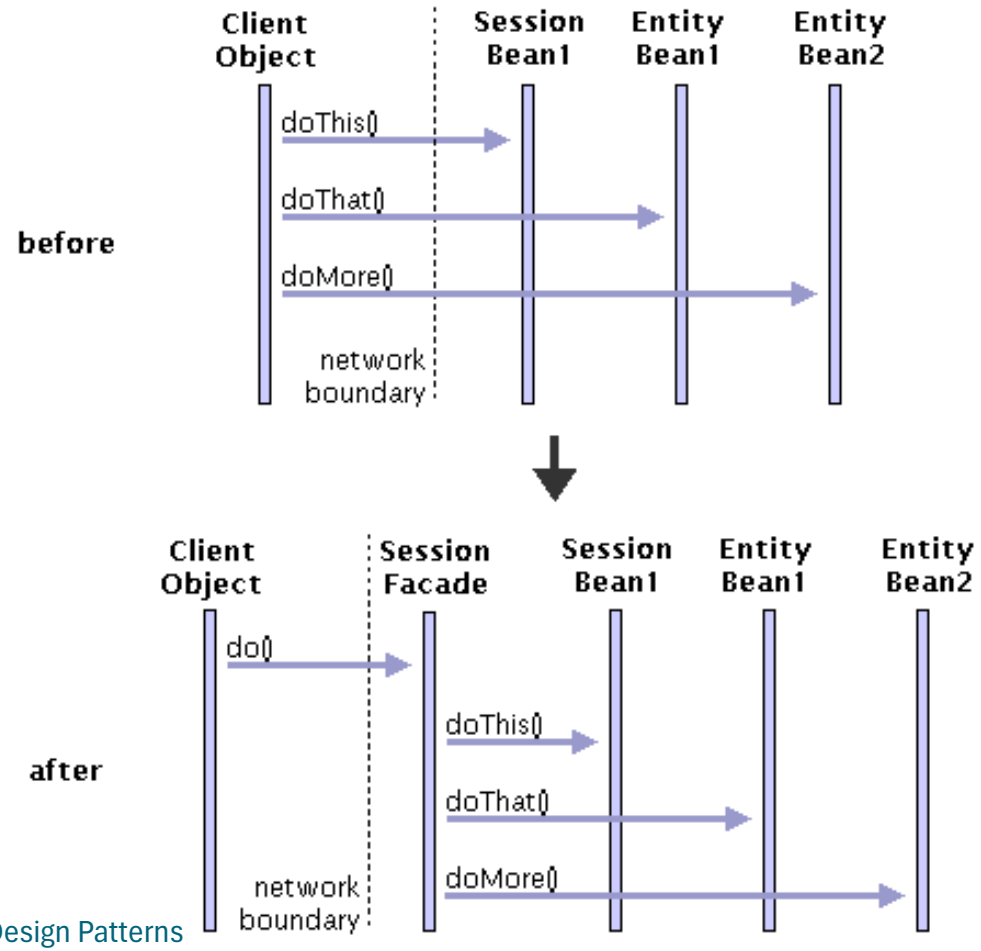
		Purpose		
		Creational	**Structural**	**Behavioral**
Scope	**Class**	Factory Method	Adapter (class)	Interpreter Template Method
	Object	Abstract Factory Builder Prototype Singleton	Adapter (object) Bridge Composite Decorator Flyweight Facade Proxy	Chain of Responsibility Command Iterator Mediator Memento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

Facade Pattern

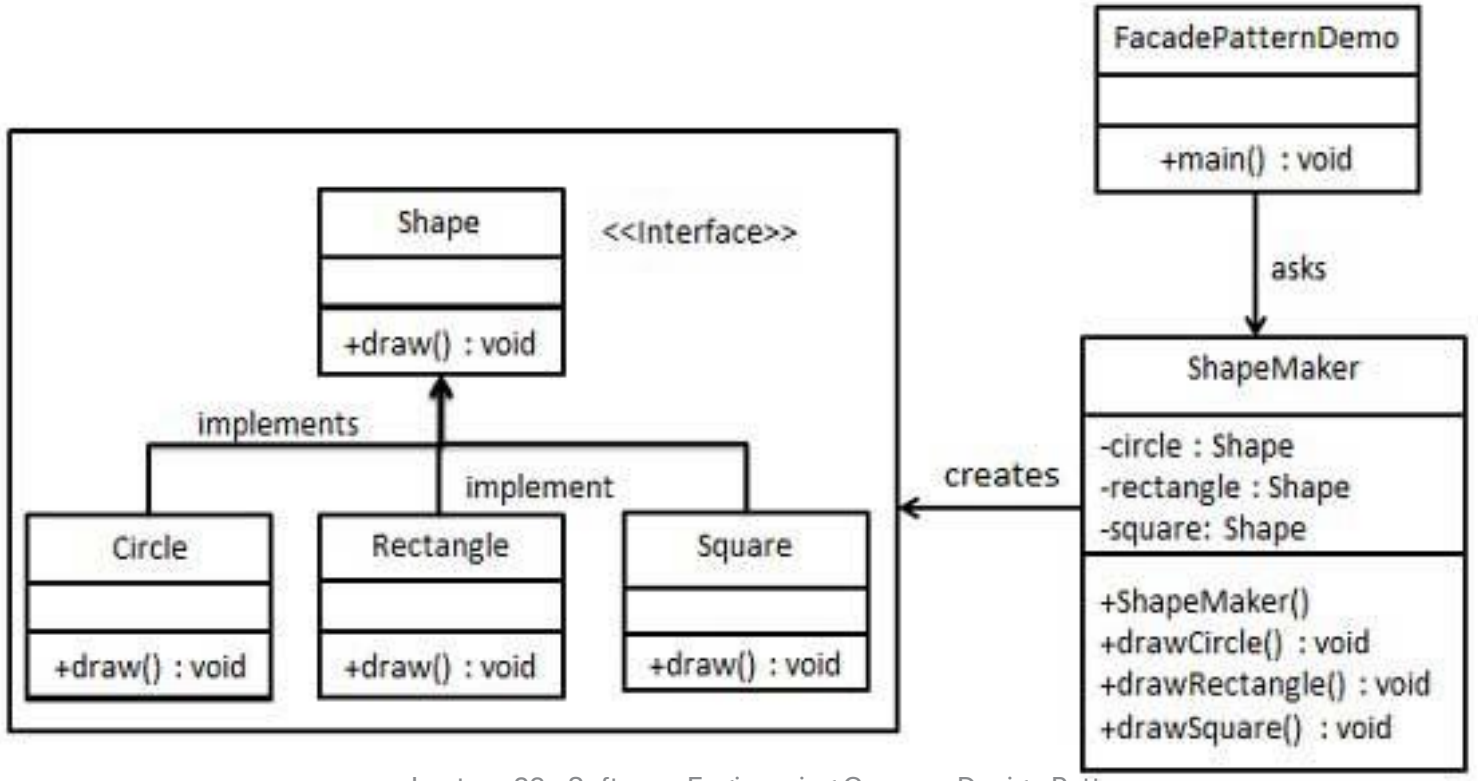
- Provides a unified interface to a set of objects in a subsystem.
- A facade defines a higher-level interface that makes the subsystem easier to use (i.e. it abstracts out the gory details)
- Facades allow us to provide a closed architecture



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!

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();
        }

}
```

Factory Method

(Creational)

		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

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 (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*

Iterator

(Behavioral)

		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

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.

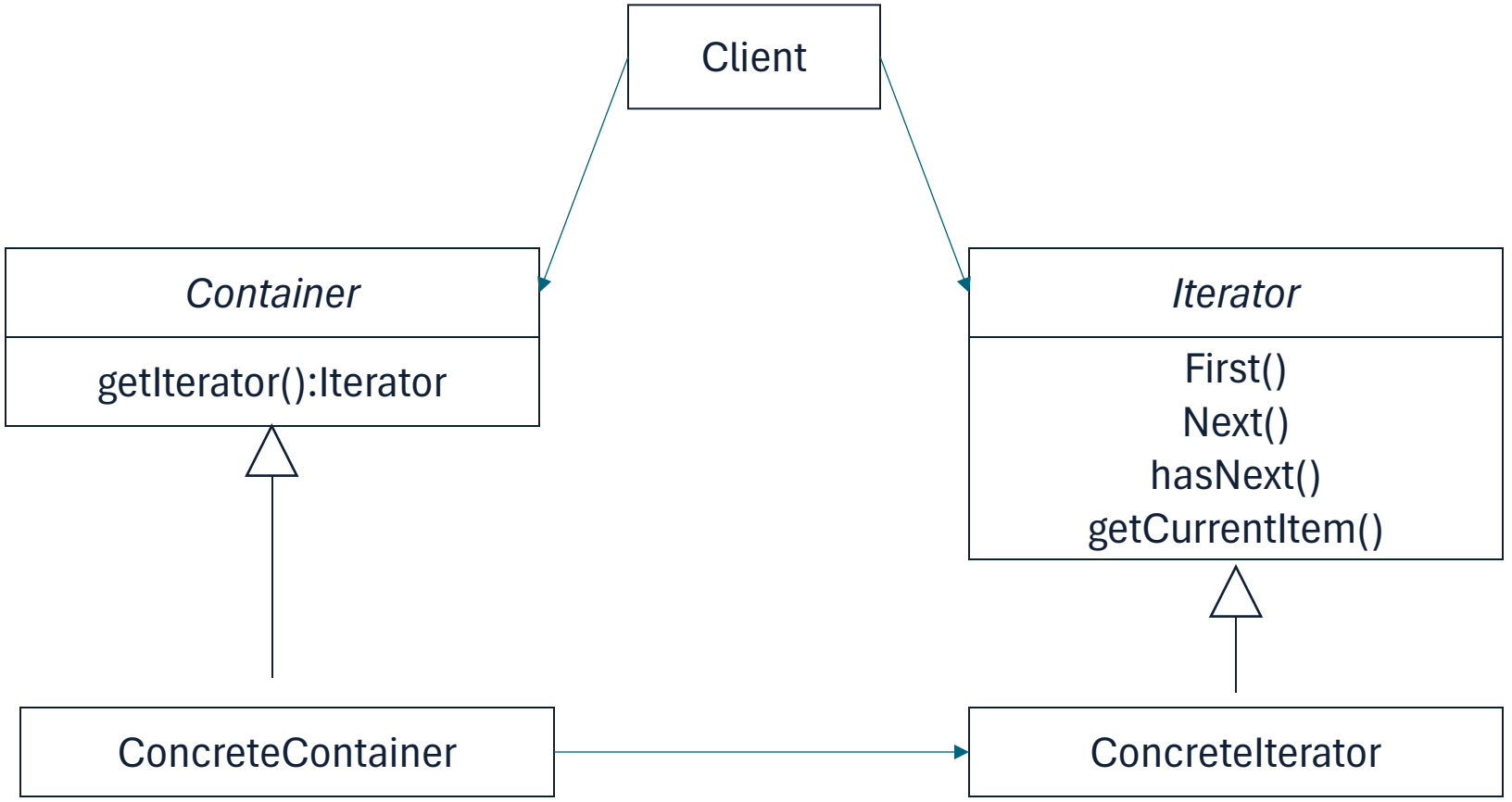
How can we do this?

- *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