Software Design Patterns

Agenda

1. GRASP
    - Low Coupling
    - High Cohesion
2. GoF Design Patterns
    - Factory
        - Simple Factory
        - Factory Method
        - Abstract Factory

Software Design Patterns

GRASP

- Information Expert
- Creator
- Low Coupling
- High Cohesion
- Controller
- Polymorphism
- Pure Fabrication
- Indirection
- Protected Variations

GoF Design Patterns

- Creational Patterns
  - Abstract Factory
  - Builder
  - Factory Method
  - Prototype
  - Singleton
- Structural Patterns
  - Adapter
  - Bridge
  - Composite
  - Decorator
  - Facade
  - Flyweight
  - Proxy
- Behavioural Patterns
  - Chain of Responsibility
  - Command
  - Interpreter
  - Iterator
  - Mediator
  - Memento
  - Observer
  - State
  - Strategy
  - Template Method
  - Visitor

GRASP: Cohesion & Coupling

Coupling – concerns relationships between modules (classes)

Cohesion – concerns relationships within a module (class)

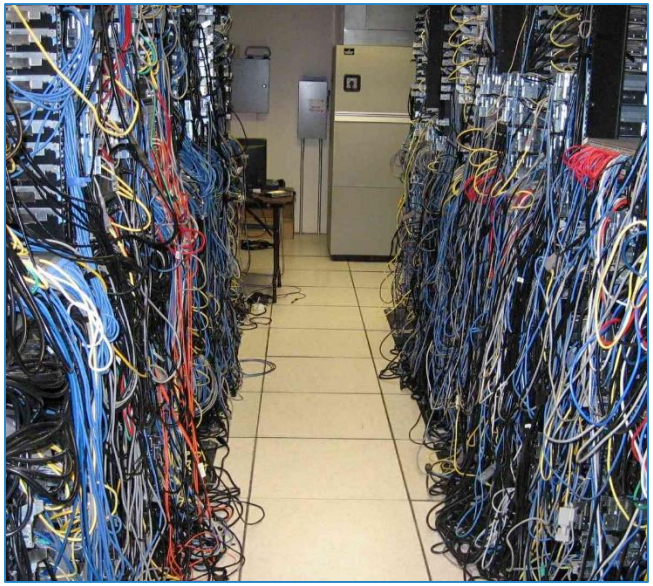
*Goal: We want **loosely coupled** modules (classes)*
*with **high** internal **cohesion***

Coupling

Coupling refers to how much a class knows about other classes.

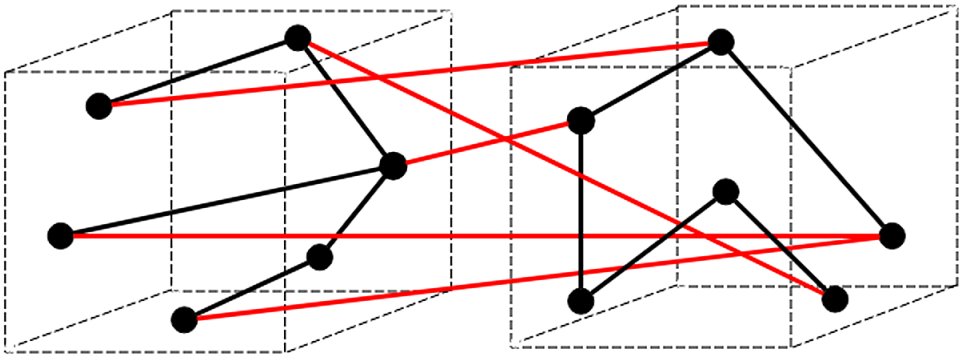
There are two types of coupling:

- **Tight (High) Coupling** (a bad programming design)
- **Loose (Low) Coupling** (a good programming design)



Tight (High) Coupling

If a class A has some **public data members** and another class B is accessing these data members **directly using the dot operator**, the two classes are said to be **tightly coupled**.



Tight (High) Coupling

Example 1

Checking a valid access to "name"

```
class A {
  public String name;
  public String getName() {
    if (name != null) return name;
    else  return "not initiaized";
  }

  public void setName(String name) {
    if (name == null) System.out.println("You can't initialized name to a null");
    else  this.name = name;
  }
}
```

Directly setting the value of data member of **class A**

Direct access of data member of **class A**

```
class B {
  public static void main(String... arg) {
    A obj = new A();
    obj.name = null;
    System.out.println("Name is " + obj.name);
  }
}
```

Output:

Name is null

https://www.decodejava.com/coupling-cohesion-java.htm

Tight (High) Coupling

Example 2

skypeId → skypeName

```
class Author {
  public String name;
  public String skypeId;

  public String getSkypeId() {
   return skypeId;
  }
}
```

changing in instance variable name

```
class Author {
  public String name;
  public String skypeName;

  public String getSkypeId() {
   return skypeName;
  }
}
```

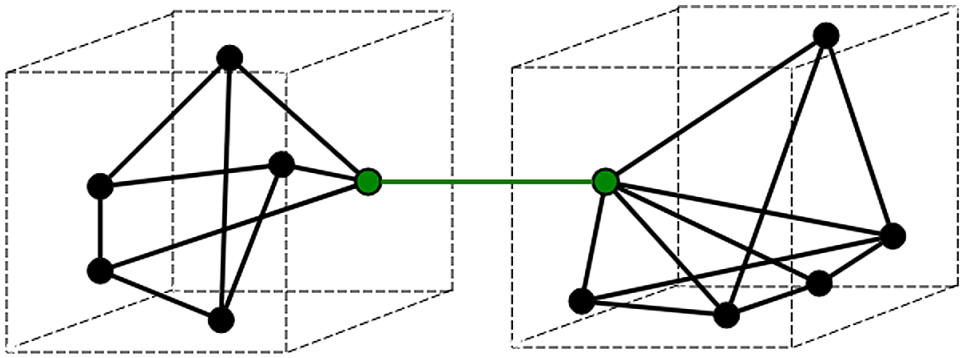
```
class Editor {
  public void clearEditingDoubts(Author author){
    setUpCall(author.skypeId);
    converse(author);
  }

  void setUpCall(String skypeId){ /* ... */ }
  void converse(Author author){ /* ... */ }
}
```

```
class Editor {
  public void clearEditingDoubts(Author author){
    setUpCall(author.skypeId);
    converse(author);
  }

  void setUpCall(String skypeId){ /* ... */ }
  void converse(Author author){ /* ... */ }
}
```

Loose (Low) Coupling

A good application designing is creating an application with loosely coupled classes by following proper **encapsulation** by

- ✓ declaring data members of a class with the **private** access modifier, forcing them to call **public getter**, **setter methods** to access these private data members.



class A class B

Loose (Low) Coupling

Example 1

```
class A {
  private String name;
  public String getName() {
    if (name != null) return name;
    else  return "not initiaized";
  }

  public void setName(String name) {
    if (name == null) System.out.println("You can't initialized name to a null");
    else  this.name = name;
  }
}
```

Checking a valid access to "name"

```
class B {
  public static void main(String... arg) {
    A obj = new A();
    obj.setName(null);
    System.out.println("Name is " + obj.getName());
  }
}
```

Calling setter method, as the direct access of "name" is not possible

Calling getter method, as the direct access of "name" is not possible

Output:

You can't initialized name to a null
Name is not initiaized

https://www.decodejava.com/coupling-cohesion-java.htm

Loose (Low) Coupling

Example 2

skypeId → skypeName

```
class Author {
  private String name;
  private String skypeId;

  public String getSkypeId() {
   return skypeId;
  }
}
```

changing in instance variable name

```
class Author {
  private String name;
  private String skypeName;

  public String getSkypeId() {
   return skypeName;
  }
}
```

```
class Editor {
  public void clearEditingDoubts(Author author){
    setUpCall(author.getSkypeId());
    converse(author);
  }

  void setUpCall(String skypeId){ /* ... */ }
  void converse(Author author){ /* ... */ }
}
```

```
class Editor {
  public void clearEditingDoubts(Author author){
    setUpCall(author.getSkypeId());
    converse(author);
  }

  void setUpCall(String skypeId){ /* ... */ }
  void converse(Author author){ /* ... */ }
}
```

Loose (Low) Coupling

Example 2

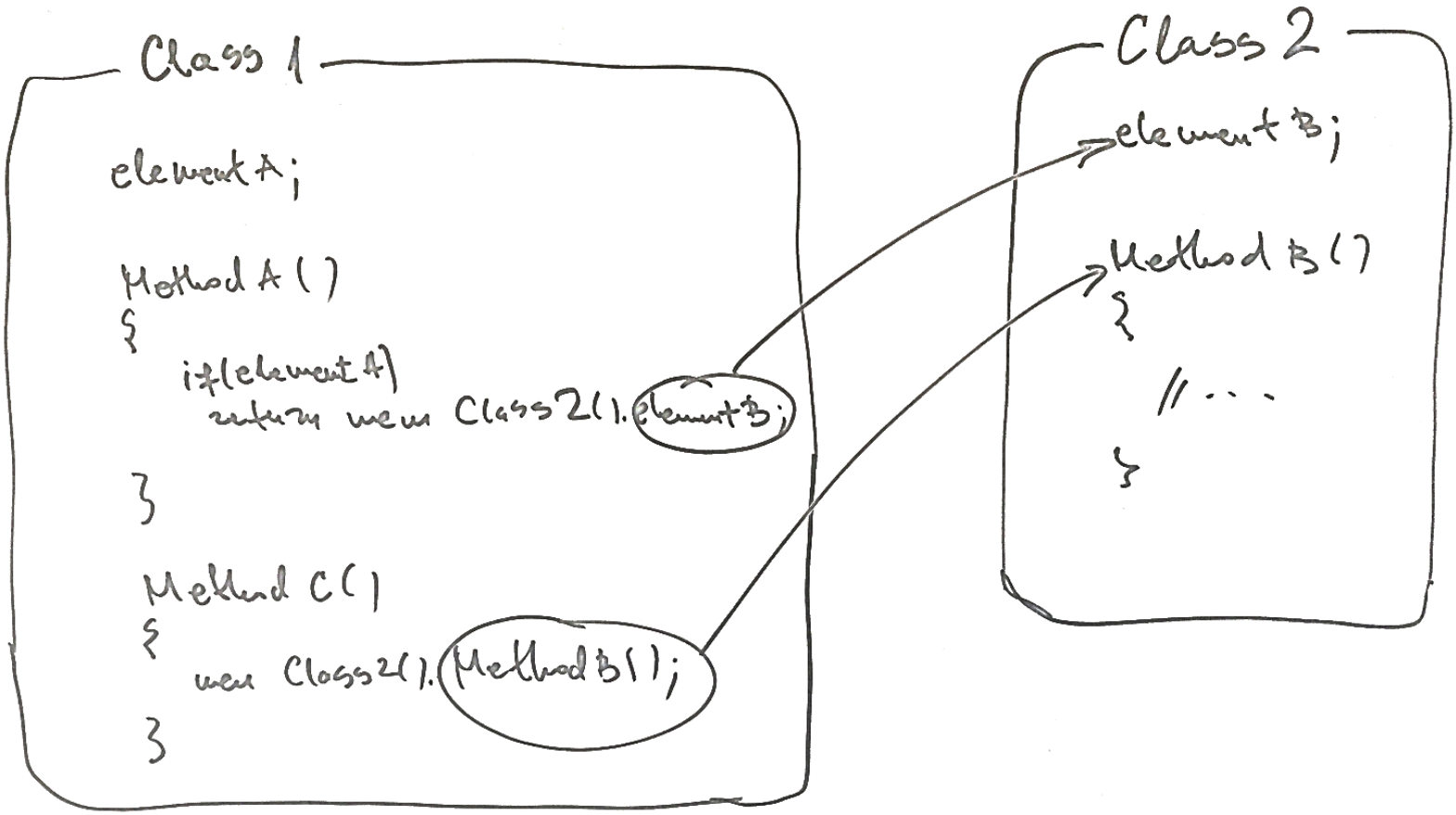
```
public interface Author {
  String getSkypeId();
}
```

```
class TechnicalAuthor implements Author {
  public String name;
  public String skypeName;

  public String getSkypeId() {
   return skypeName;
  }
}
```

```
class Editor {
  public void clearEditingDoubts(Author author){
    setUpCall(author.getSkypeId());
    converse(author);
  }

  void setUpCall(String skypeId){ /* ... */ }
  void converse(Author author){ /* ... */ }
}
```

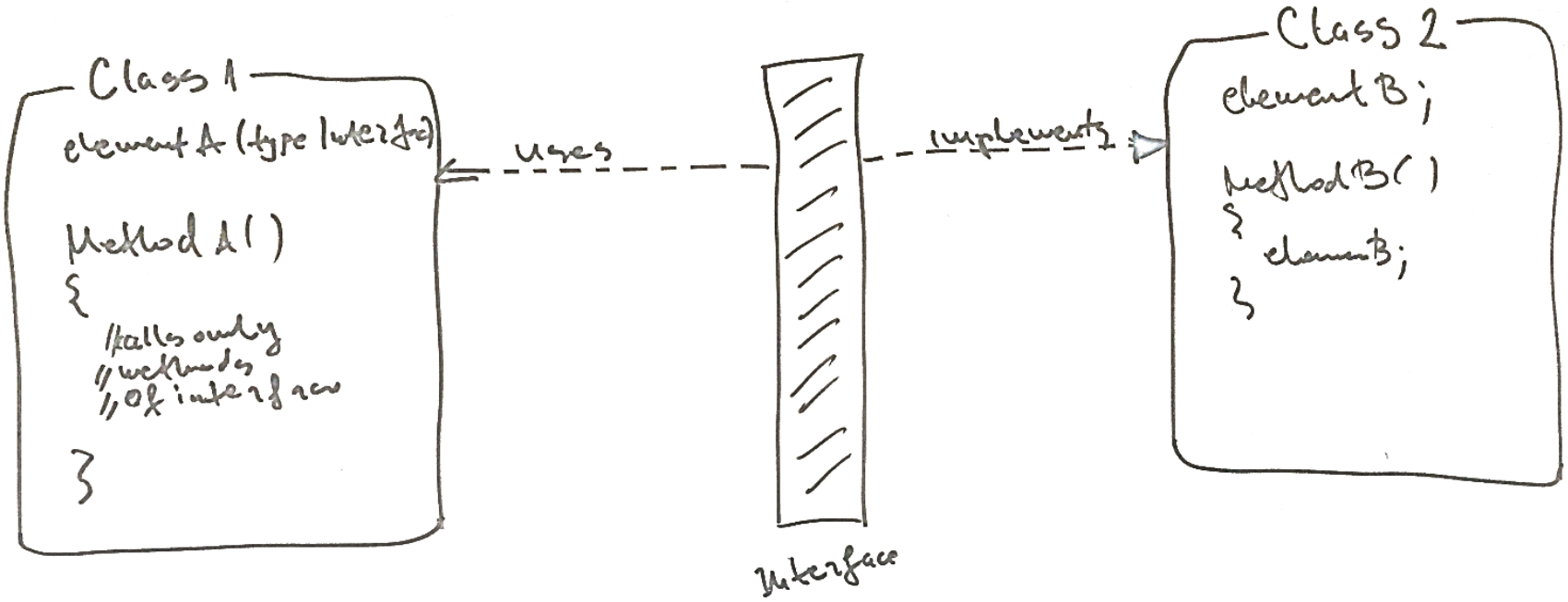
✓ Dependency Inversion Principle

Tight (High) Coupling



https://thebojan.ninja/2015/04/08/high-cohesion-loose-coupling/

Loose (Low) Coupling



https://thebojan.ninja/2015/04/08/high-cohesion-loose-coupling/

Cohesion

Cohesion refers to the extent to which a class is defined to do a specific specialized task.

There are two types of cohesion:

- **Low cohesion** (a bad programming design)
- **High Cohesion** (a good programming design)

A class created with high cohesion is targeted towards a single specific purpose, rather than performing many different specific purposes.

Cohesion

Example 1

- **Low cohesion** refers to modules that have different unrelated responsibilities.
- **High cohesion** refers to modules that have functions that are similar in many aspects.

Low cohesion

```
class Editor {
  public void editBooks() { /* ... */ }

  public void manageBookPrinting() { /* ... */ }

  public void reachOutToNewAuthors() { /* ... */ }
}
```

Editor is performing diverse set of unrelated tasks

High cohesion

```
class Editor {
  public void useEditTools() { /* ... */ }

  public void editFirstDraft() { /* ... */ }

  public void clearEditingDoubts() { /* ... */ }
}
```

Editor is performing multiple but related tasks

Cohesion

Example 2

Low cohesion

```
class PlayerDatabase
{
  public void connectDatabase(){};

  public void printAllPlayersInfo(){};

  public void printSinglePlayerInfo(){};

  public void printRankings(){};

  public void closeDatabase(){};
}
```

✓ Open-Closed Principle

High cohesion

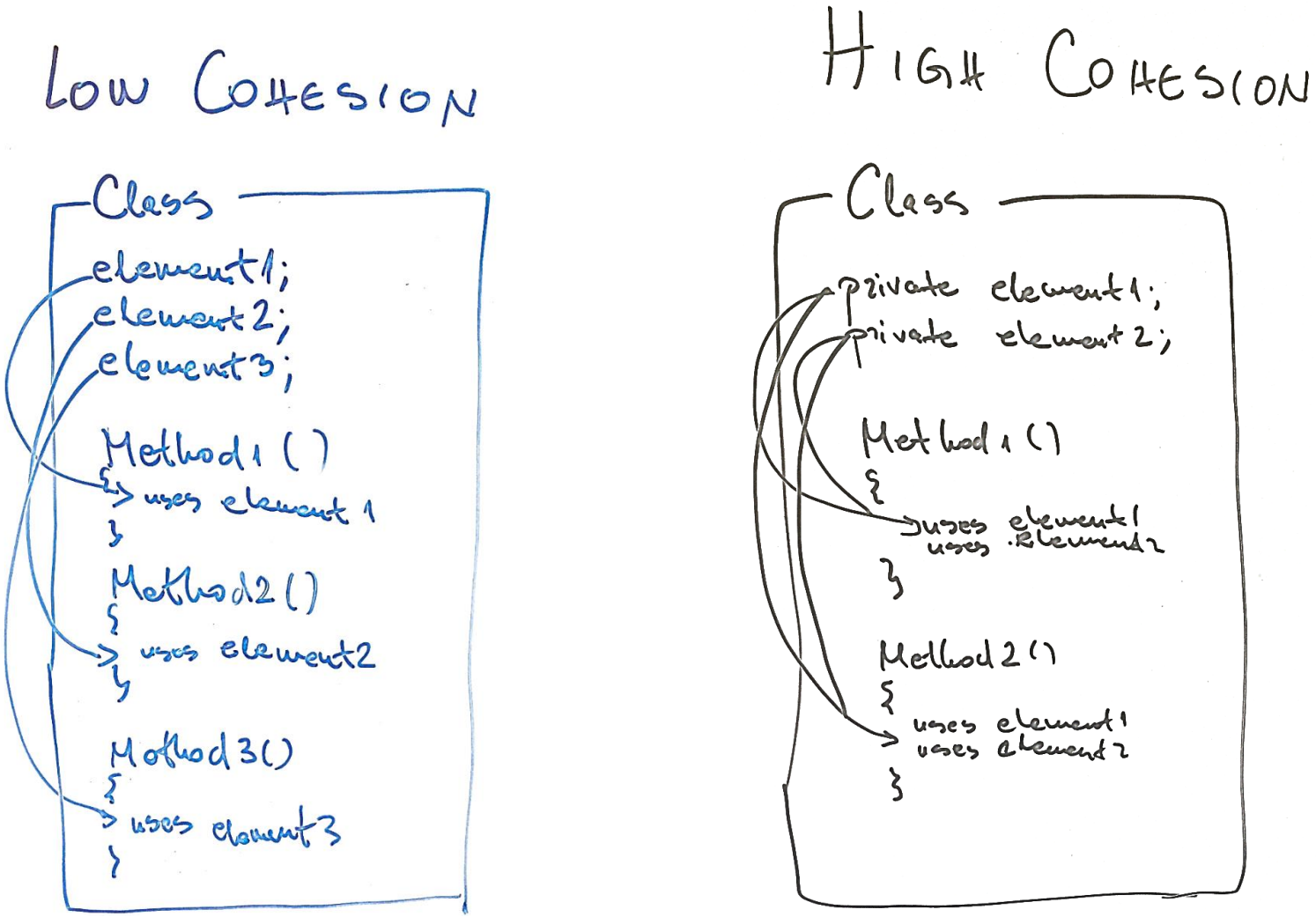
```
class PlayerDatabase {
  DatabaseManager databaseManager =
                                  new DatabaseManager();
  PrintManager printManager = new PrintManager();
}

class DatabaseManager {
  public void ConnectDatabase() {
    /* connecting to database */ }

  public void CloseDatabase() {
    /* closing the database connection */ }
}

class PrintManager{
  public void printRankings() {
    /* printing the players current rankings */ }

  public void printAllPlayersInfo() {
    /* printing all the players information */ }

  public void printSinglePlayerInfo() {
    /* printing a single player information */ }
}
```

Cohesion

- Coincidental cohesion (worst)
- Logical cohesion
- Temporal cohesion
- Procedural cohesion
- Communicational cohesion
- Sequential cohesion
- Functional cohesion (best)

https://thebojan.ninja/2015/04/08/high-cohesion-loose-coupling/

Cohesion



- **Coincidental cohesion** (worst)
- **Functional cohesion** (best)

https://thebojan.ninja/2015/04/08/high-cohesion-loose-coupling/

Gang of Four (GoF)

Ralph Johnson, Richard Helm, Erich Gamma, and John Vlissides (left to right)

GoF Design Patterns

- Creational Patterns

Frequency of use:

- Abstract Factory 
- Builder 
- Factory Method 
- Prototype 
- Singleton 

- Structural Patterns

- Adapter 
- Bridge 
- Composite 
- Decorator 
- Facade 
- Flyweight 
- Proxy 

- Behavioural Patterns

Frequency of use:

- Chain of Responsibility 
- Command 
- Interpreter 
- Iterator 
- Mediator 
- Memento 
- Observer 
- State 
- Strategy 
- Template Method 
- Visitor 

https://www.dofactory.com/net/design-patterns

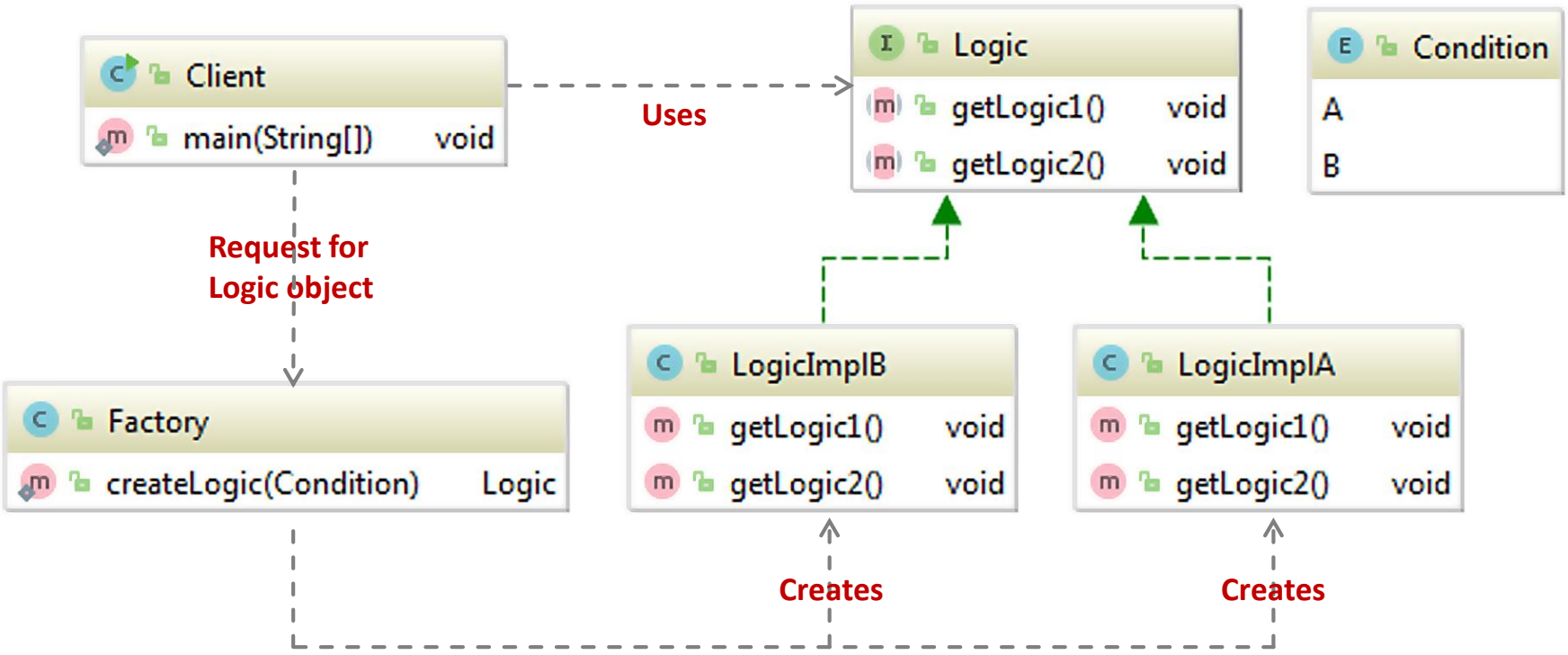
Creational Patterns: **Factory**

- ✓ The Factory Pattern deals with the creation of similar types of objects and producing them in a centralized manner, depending on the condition or type of object requested.

- ❖ Simple Factory
- ❖ Factory Method
- ❖ Abstract Factory

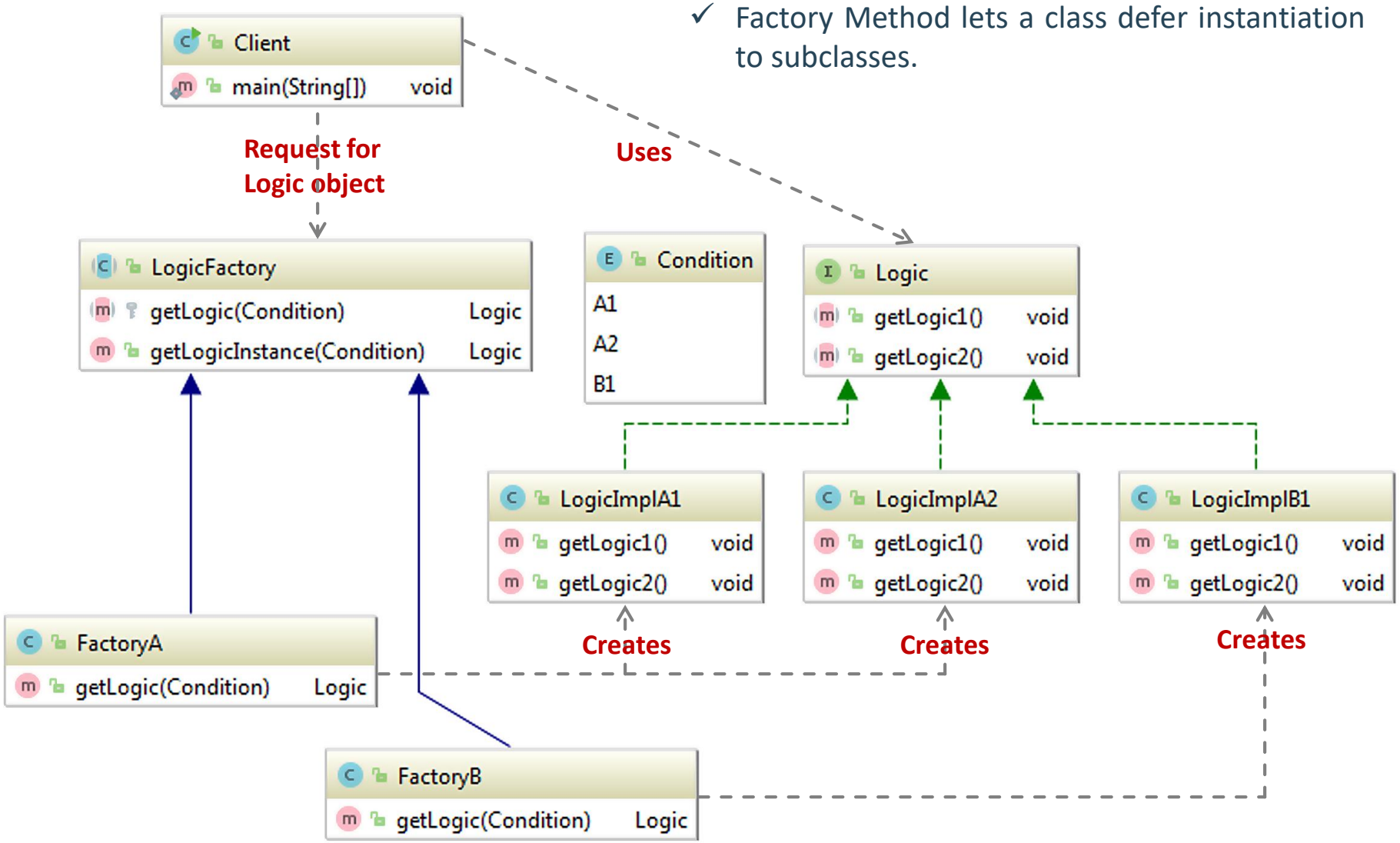
Creational Patterns: **Simple Factory**

- ✓ Is used to create a specific type of product (object) depending on a condition.



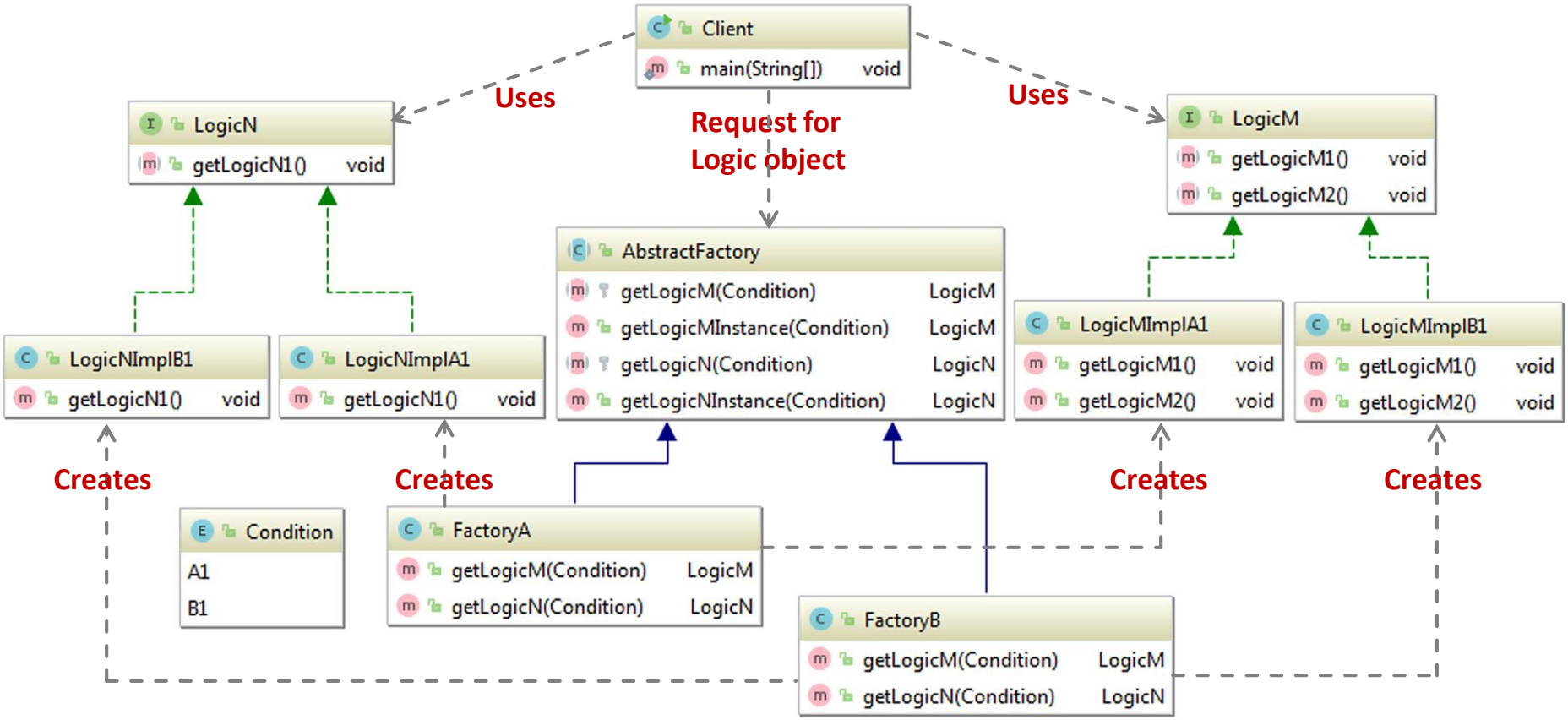
Creational Patterns: **Factory Method**

- ✓ Define an interface for creating an object, but let subclasses decide which class to instantiate.
- ✓ Factory Method lets a class defer instantiation to subclasses.



Creational Patterns: **Abstract Factory**

- ✓ The Abstract Factory pattern is used to create a family of related products (Factory Method pattern creates one type of object)



Creational Patterns: **Benefits of Factory**

- ✓ Prefer method invocation over direct constructor calls
- ✓ Prevent tight coupling between a class implementation and your application
- ✓ Promote creation of cohesive classes
- ✓ Promote programming to an interface
- ✓ Promote flexibility. Object instantiation logic can be changed without affecting the clients that use objects. They also allow addition of new concrete classes.

Creational Patterns: **Java API (Factories)**

Class	Method
java.util.Calendar	getInstance()
java.util.Arrays	asList()
java.util.ResourceBundle	getBundle()
java.sql.DriverManager	getConnection()
java.sql.DriverManager	getDriver()
java.sql.Connection	createStatement()
java.util.concurrent.Executors	newFixedThreadPool() newCachedThreadPool() newSingleThreadExecutor()
java.text.NumberFormat	getInstance() getNumberFormat()

Agenda

GoF Design Patterns

- Singleton
  - Eager
  - Classic
  - Thread-Safe
  - Double-Checked Locking
  - Enum
- State

GoF Design Patterns

- Creational Patterns

Frequency of use:

- Abstract Factory 

- Builder 

- Factory Method 

- Prototype 

- Singleton 

- Structural Patterns
  - Adapter 

  - Bridge 

  - Composite 

  - Decorator 

  - Facade 

  - Flyweight 

  - Proxy 

- Behavioural Patterns

Frequency of use:

  - Chain of Responsibility 

  - Command 

  - Interpreter 

  - Iterator 

  - Mediator 

  - Memento 

  - Observer 

  - State 

  - Strategy 

  - Template Method 

  - Visitor 

https://www.dofactory.com/net/design-patterns

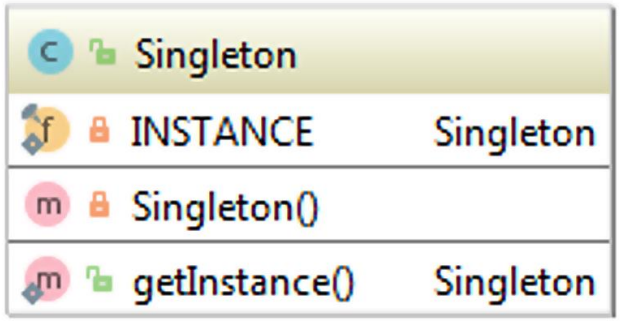
Creational Patterns: **Singleton**

- ✓ Ensure a class has only one instance and provide a global point of access to it.

- ❖ Eager Singleton
- ❖ Classic Singleton
- ❖ Thread-Safe Singleton
- ❖ Double-Checked Locking
- ❖ Enum Singleton

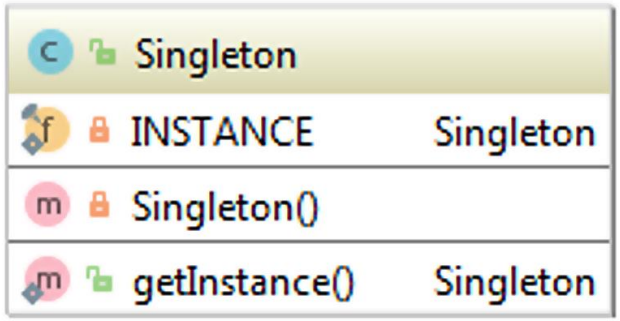
Creational Patterns: **Singleton**

- Eager Singleton




```
public class Singleton {

   private static final Singleton INSTANCE = new Singleton();

   private Singleton() {  }

   public static Singleton getInstance() {
      return INSTANCE;
   }
}
```

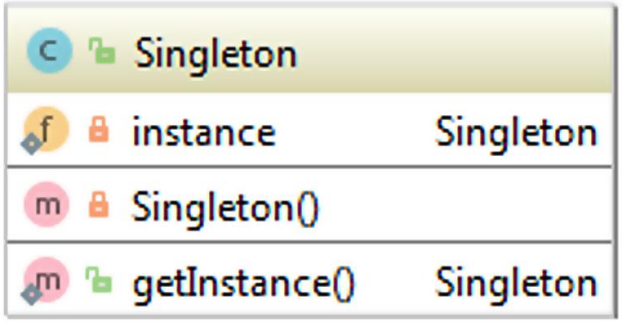
- Eager Singleton with static block




```
public class Singleton {
   private static final Singleton INSTANCE;

   static {
      try {
         INSTANCE = new Singleton();
      } catch (Exception ex) { throw ex; }
   }

   private Singleton() {  }

   public static Singleton getInstance() { return INSTANCE; }
}
```

Creational Patterns: **Singleton**

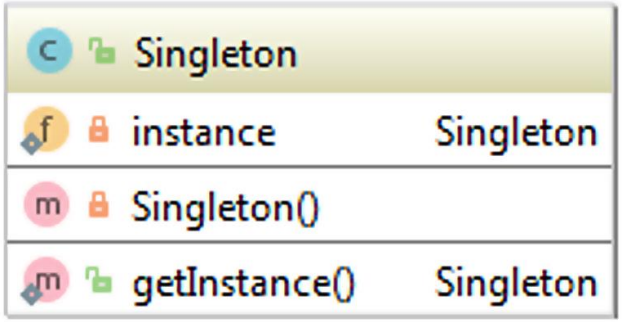
- Classic Singleton
  - Lazy initialization




```
public class Singleton {

   private static Singleton instance = null;

   private Singleton() {  }

   public static Singleton getInstance() {
      if (instance == null)  instance = new Singleton();
      return instance;
   }
}
```

Creational Patterns: **Singleton**

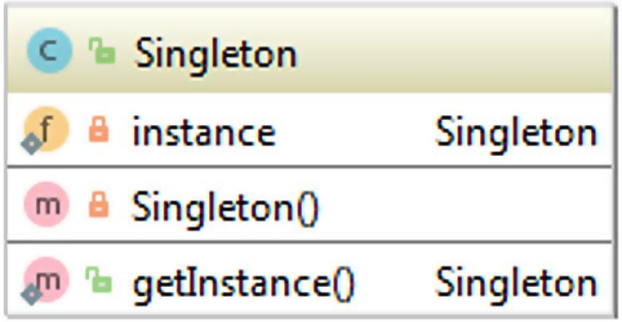
- Thread-Safe Singleton




```
public class Singleton {

   private static Singleton instance = null;

   private Singleton() {  }

   public synchronized static Singleton getInstance() {
      if (instance == null)  instance = new Singleton();
      return instance;
   }
}
```

Creational Patterns: **Singleton**

- Double-Checked Locking



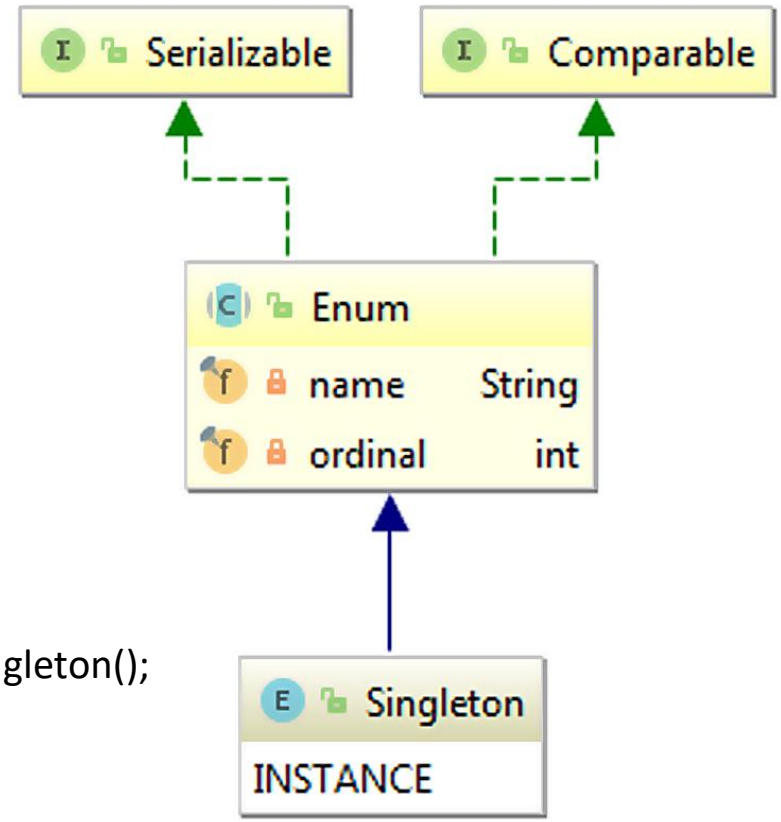

```
public class Singleton {
   private static volatile Singleton instance = null;

   private Singleton() {  }

   public static Singleton getInstance() {
      if (instance == null) {
         synchronized (Singleton.class) {
            if (instance == null) instance = new Singleton();
         }
      }
      return instance;
   }
}
```

❖ Enum Singleton

```
public enum Singleton {
   INSTANCE;
}
```

Which internally will be treated like

```
public class Singleton {

   public static final Singleton INSTANCE = new Singleton();

   private Singleton() {  }
}
```

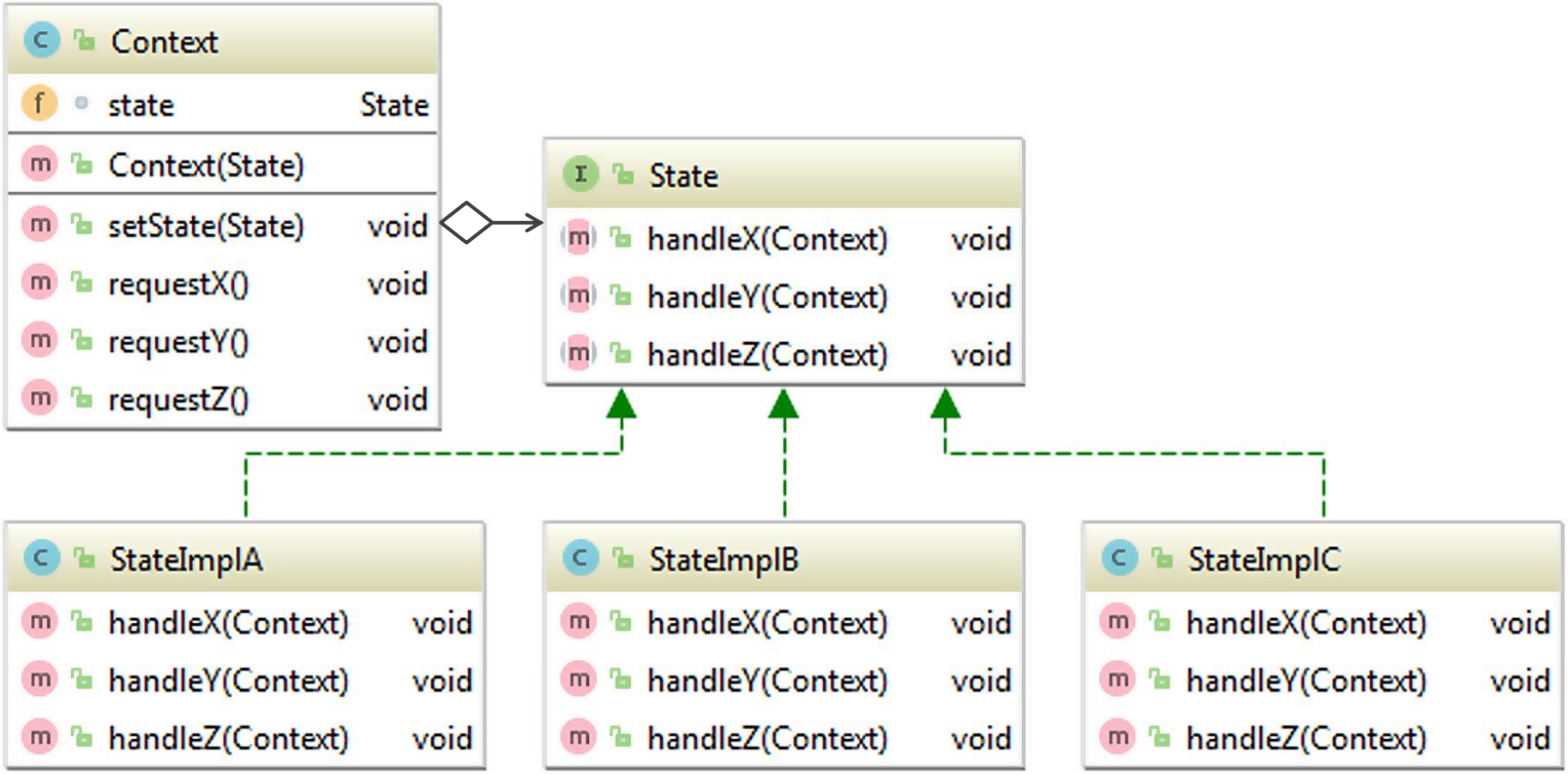


- An Enum is singleton by design (from Java 1.5).
- All the enum values are initialized only once at the time of class loading.

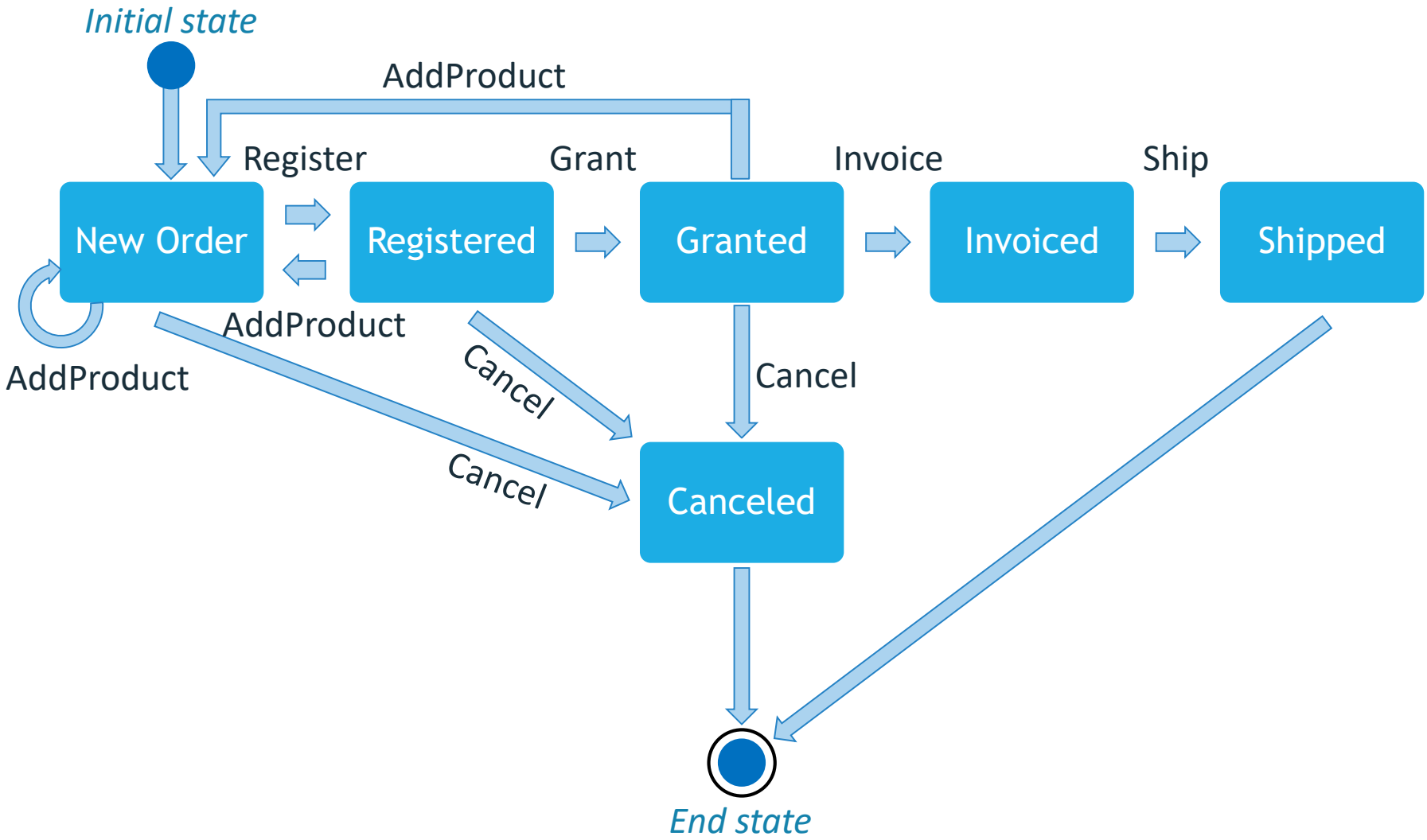
Behavioural Patterns: **State**

- ✓ Allow an object to alter its behavior when its internal state changes.

  The object will appear to change its class.

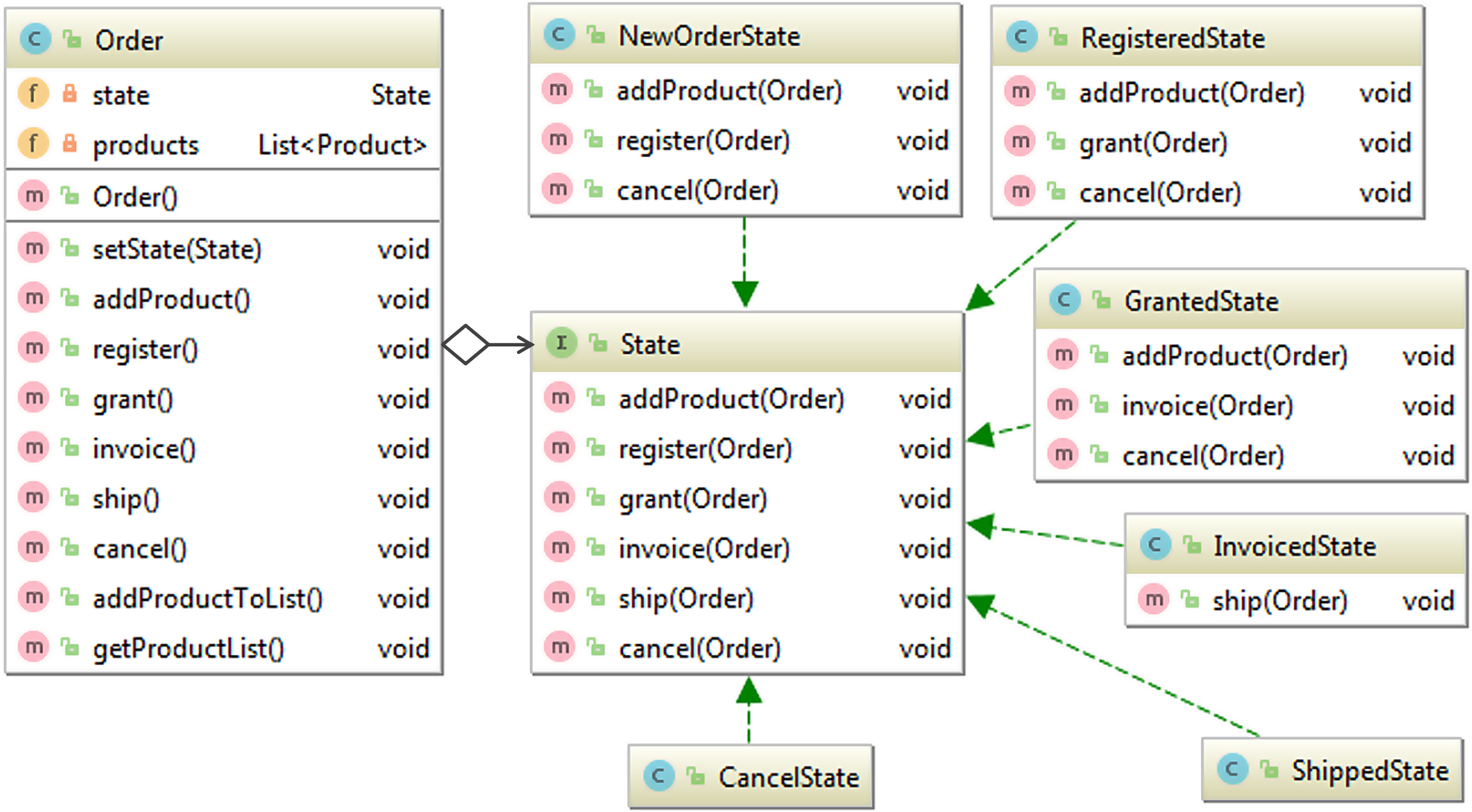


PROCESSING ORDERS



http://andriybuday.com/book

PROCESSING ORDERS



Agenda

GoF Design Patterns

- Adapter
- Decorator
- Facade
- Observer
- Strategy

GoF Design Patterns

- Creational Patterns

Frequency of use:

- Abstract Factory 

- Builder 

- Factory Method 

- Prototype 

- Singleton 

- Structural Patterns
  - Adapter 

  - Bridge 

  - Composite 

  - Decorator 

  - Facade 

  - Flyweight 

  - Proxy 

- Behavioural Patterns

Frequency of use:

  - Chain of Responsibility 

  - Command 

  - Interpreter 

  - Iterator 

  - Mediator 

  - Memento 

  - Observer 

  - State 

  - Strategy 

  - Template Method 

  - Visitor 

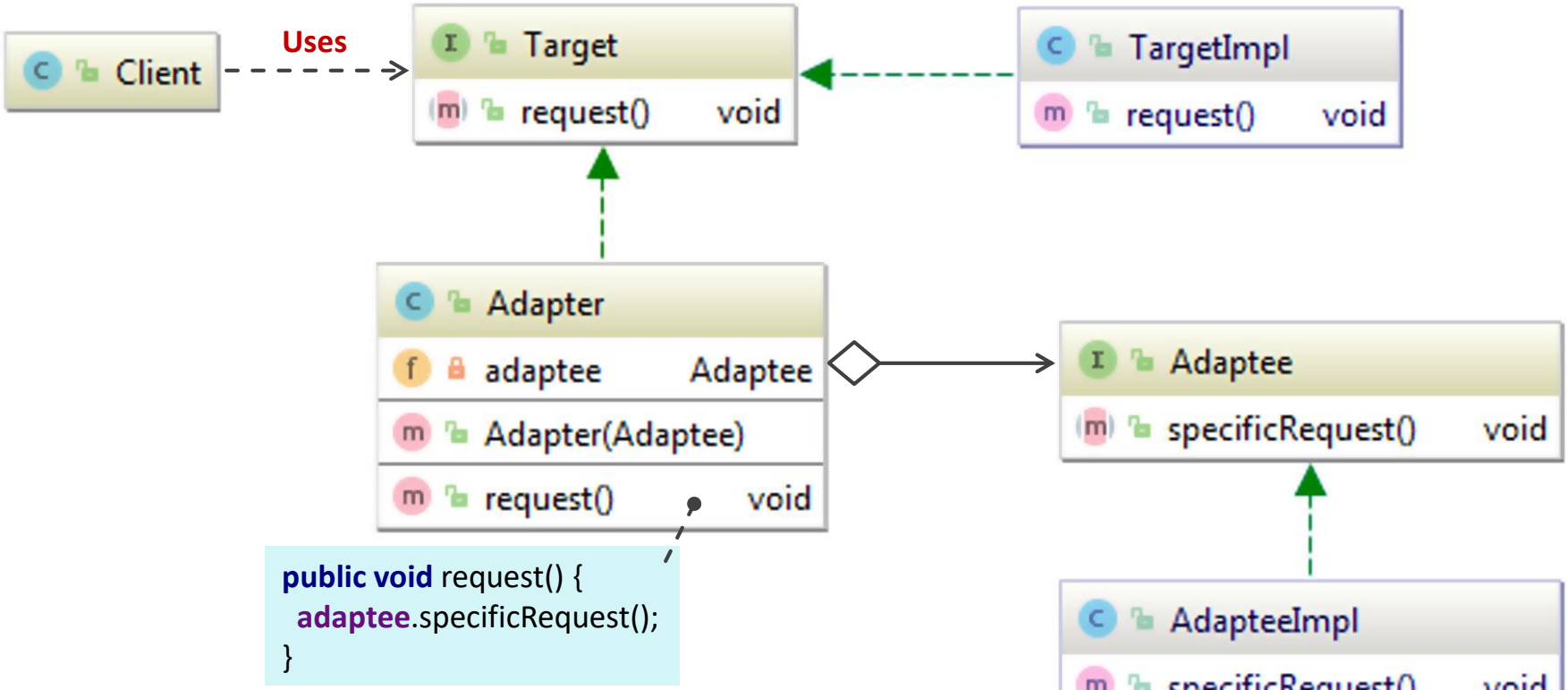
https://www.dofactory.com/net/design-patterns

Structural Patterns: **Adapter**

- ✓ Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.

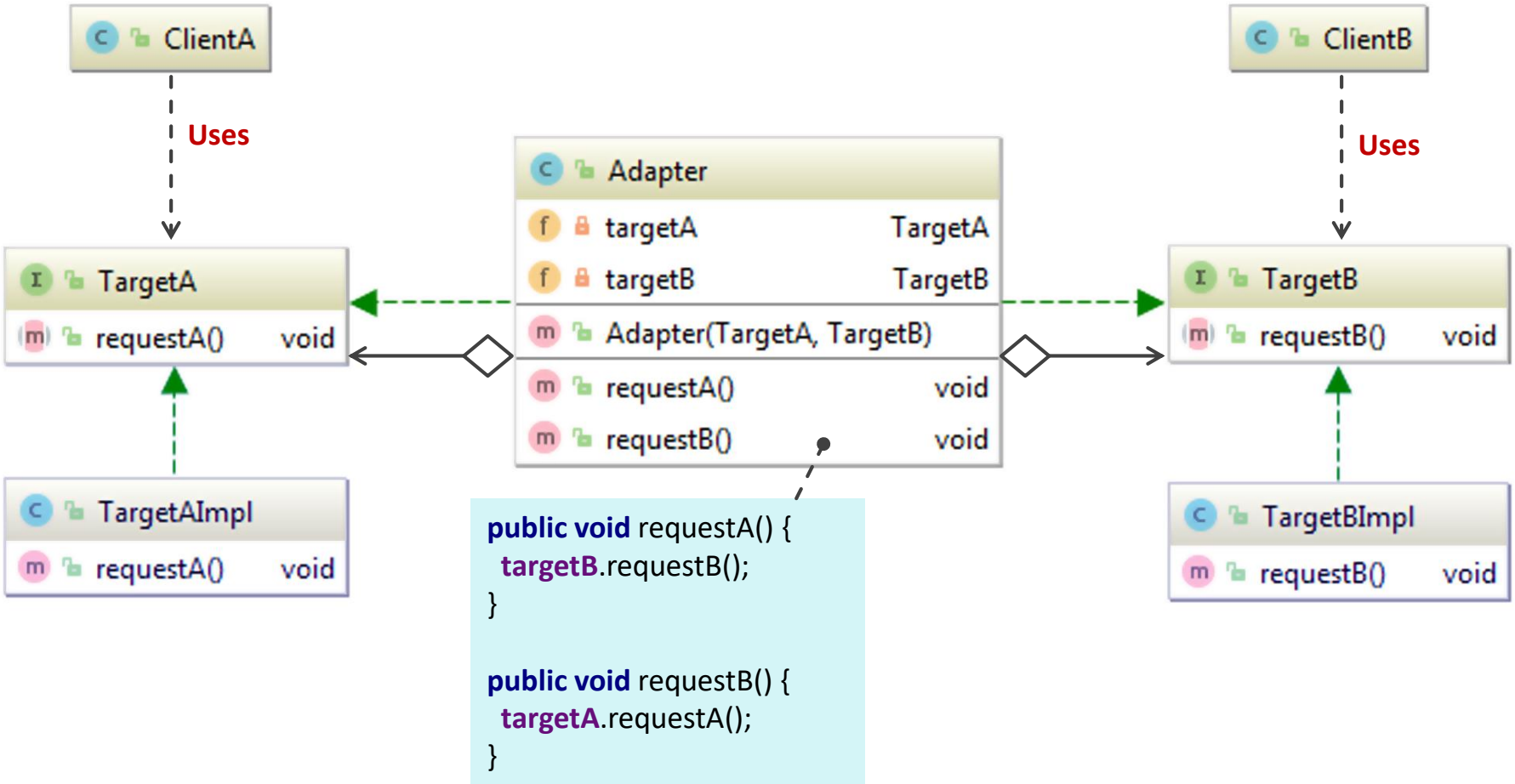
Structural Patterns: (object) **Adapter**

❖ Ordinary object Adapter



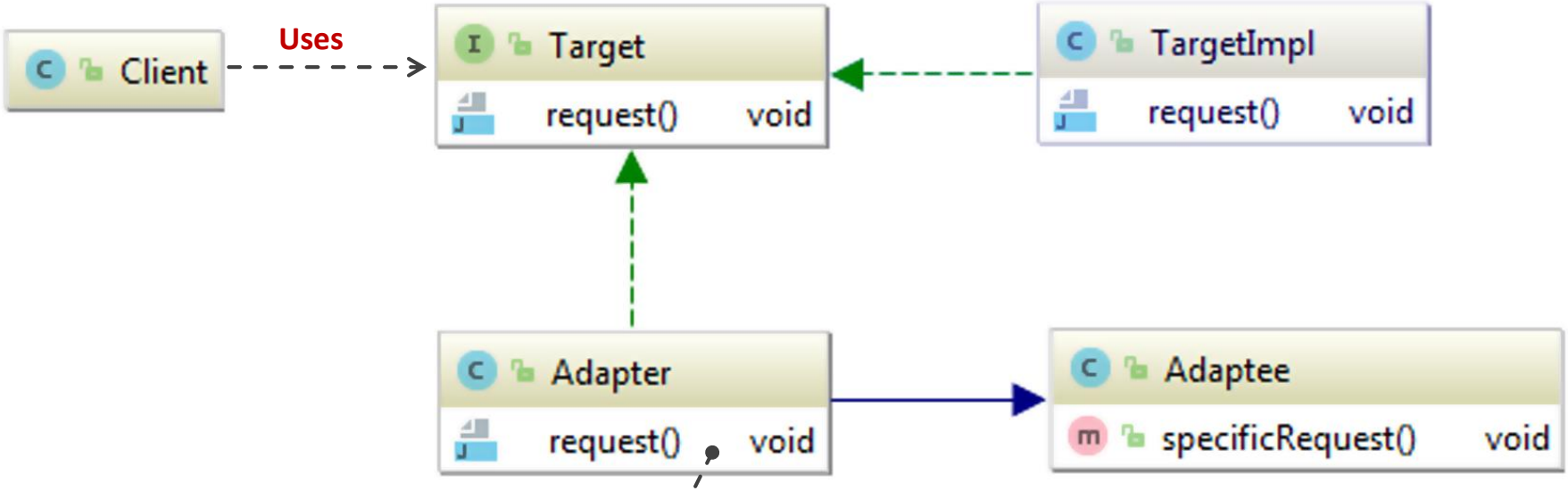
Structural Patterns: (object) **Adapter**

❖ Duplex object Adapter



Structural Patterns: (class) **Adapter**

❖ Ordinary class Adapter




```
public void request() {
  specificRequest();
}
```

Structural Patterns: **Java API (Adapter)**

java.util.Arrays**#asList**()

java.util.Collections**#list**()

java.util.Collections**#enumeration**()

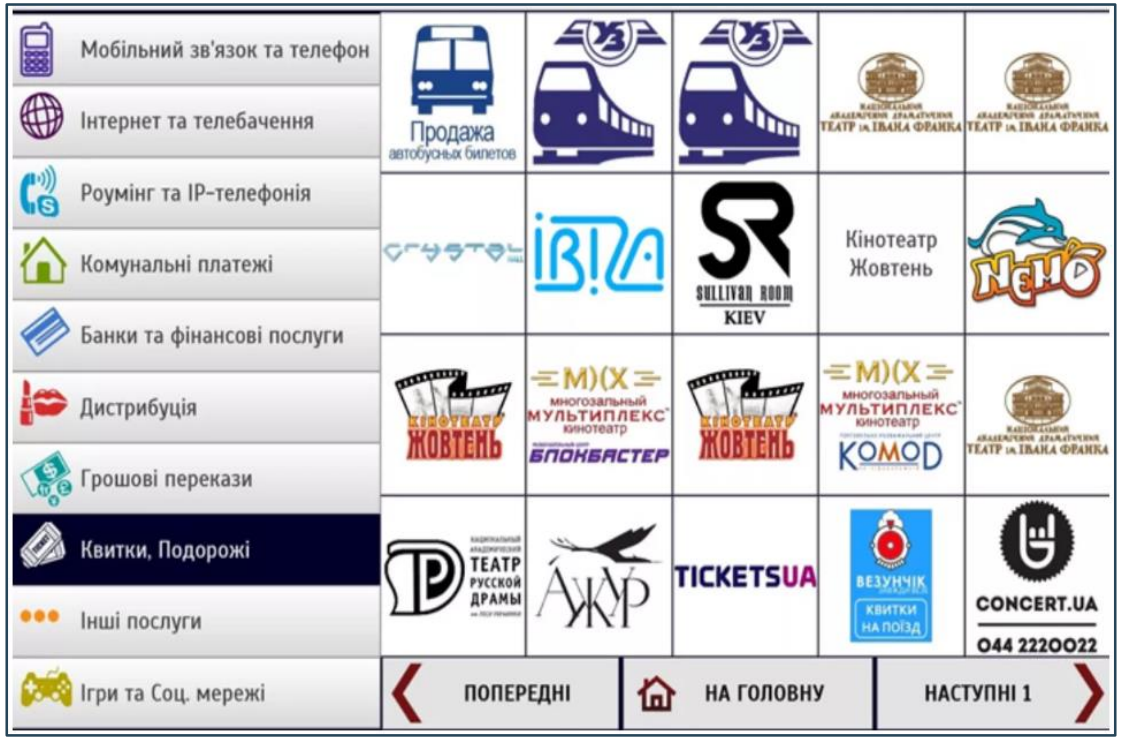
java.io.**InputStreamReader**(InputStream) (returns a Reader)

java.io.**OutputStreamWriter**(OutputStream) (returns a Writer)

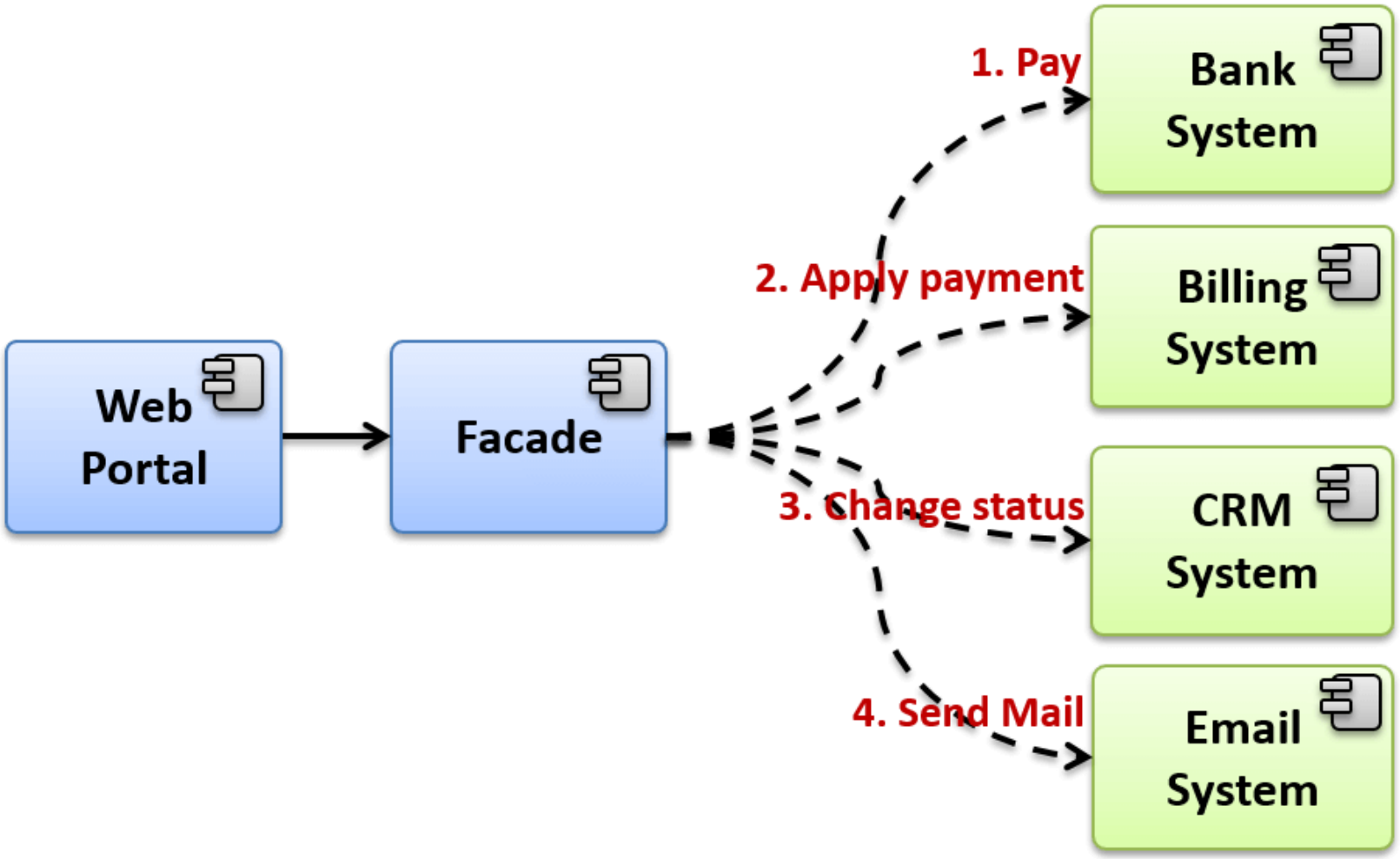
javax.xml.bind.annotation.adapters.XmlAdapter**#marshal**() and **#unmarshal**()

Structural Patterns: **Facade**

- ✓ Provide a unified interface to a set of interfaces in a subsystem. Façade defines a higher-level interface that makes the subsystem easier to use.
- ✓ Provides a simplified interface to a library, a framework, or any other complex set of classes.

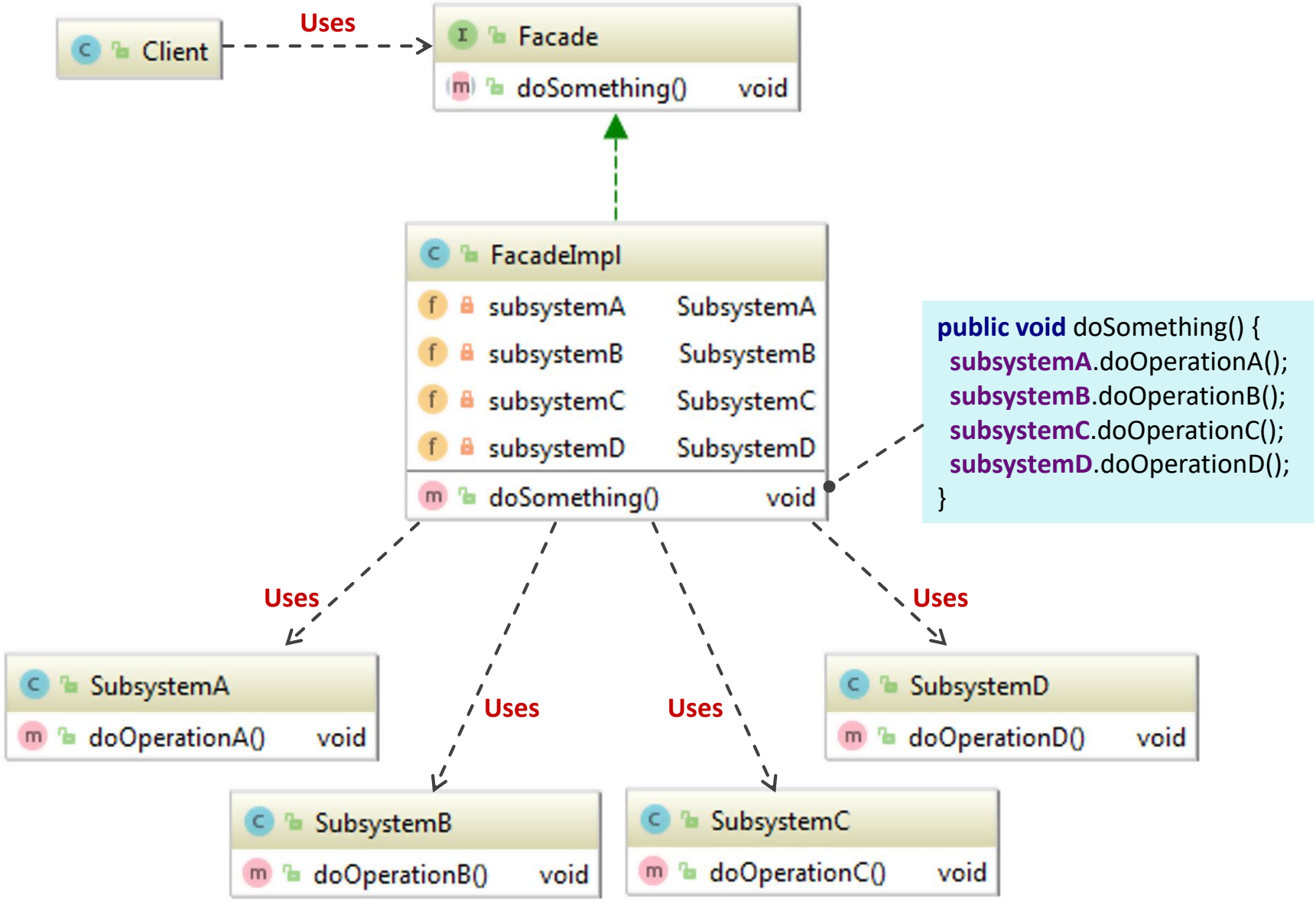


Structural Patterns: **Facade**



https://reactiveprogramming.io/books/design-patterns/es/catalog/facade

Structural Patterns: **Facade**



Structural Patterns: **Java API (Facade)**

javax.faces.context.FacesContext

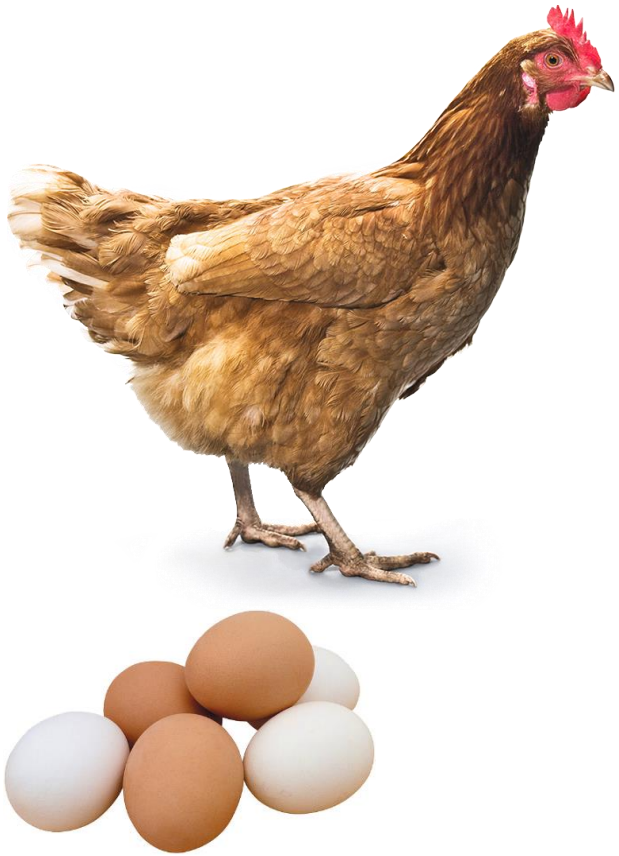
it internally uses among others the abstract/interface types **LifeCycle**, **ViewHandler**, **NavigationHandler** and many more without that the enduser has to worry about it (which are however overrideable by injection).

javax.faces.context.ExternalContext

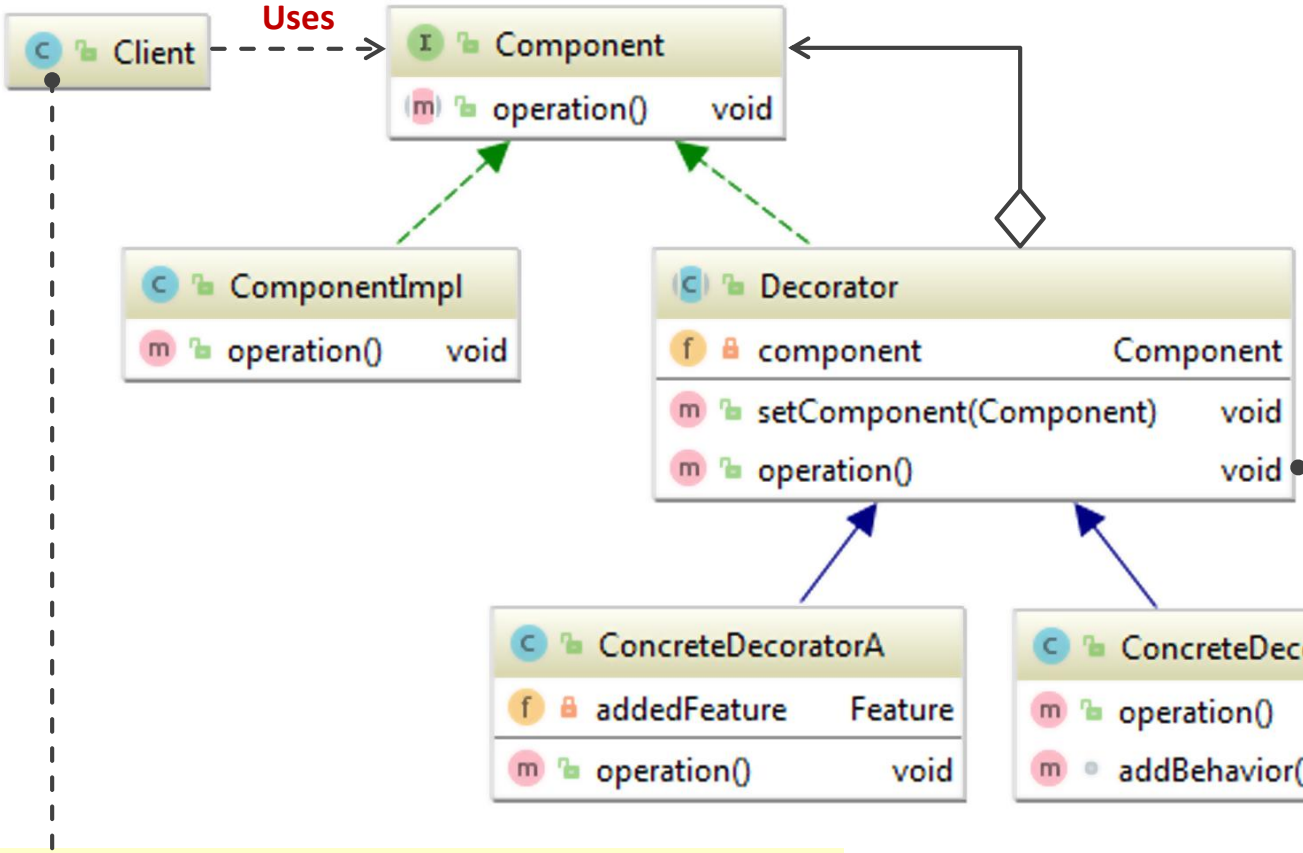
which internally uses **ServletContext**, **HttpSession**, **HttpServletRequest**, **HttpServletResponse**, etc.

Structural Patterns: **Decorator**

- ✓ Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.



Structural Patterns: **Decorator**




```
public void operation() {
  if (component != null) {
    component.operation();
  } }
```

```
Component component = new ComponentImpl();
ConcreteDecoratorA d1 = new ConcreteDecoratorA();
ConcreteDecoratorB d2 = new ConcreteDecoratorB();
d1.setComponent(component);
d2.setComponent(d1);
d2.operation();
```

Link decorators

```
public void operation() {
  super.operation();
  addBehavior();
}
```

Structural Patterns: **Decorator**

90 UAH

HAWAIIAN PIZZA

110 UAH

PEPPERONI PIZZA

180 UAH

CAULIFLOWER PIZZA

+

+5 UAH

+

+20 UAH

+

+30 UAH

+

-100 UAH

Structural Patterns: **Java API (Decorator)**

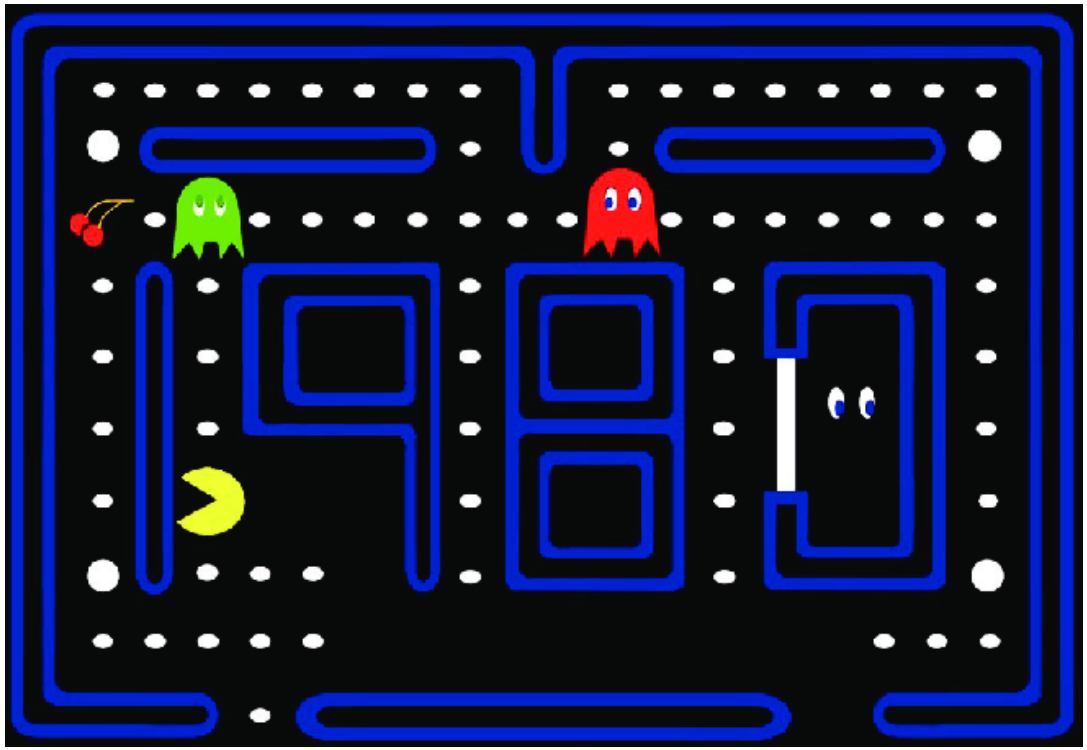
All subclasses of java.io.InputStream, OutputStream, Reader and Writer have constructors that accept objects of their own type.

java.util.Collections, methods checkedXXX(), synchronizedXXX() and unmodifiableXXX().

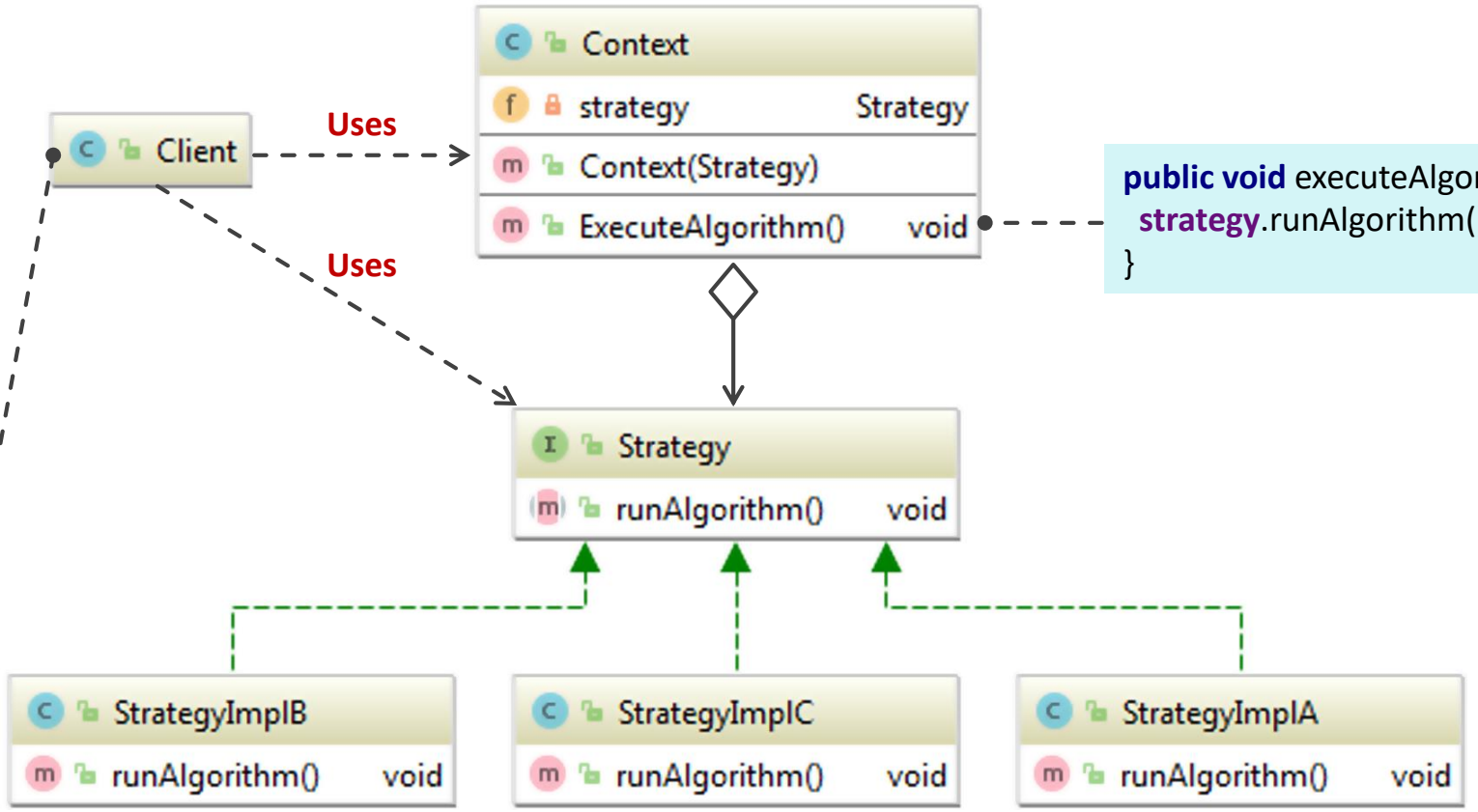
javax.servlet.http.HttpServletRequestWrapper and HttpServletResponseWrapper

Behavioural Patterns: **Strategy**

- ✓ Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

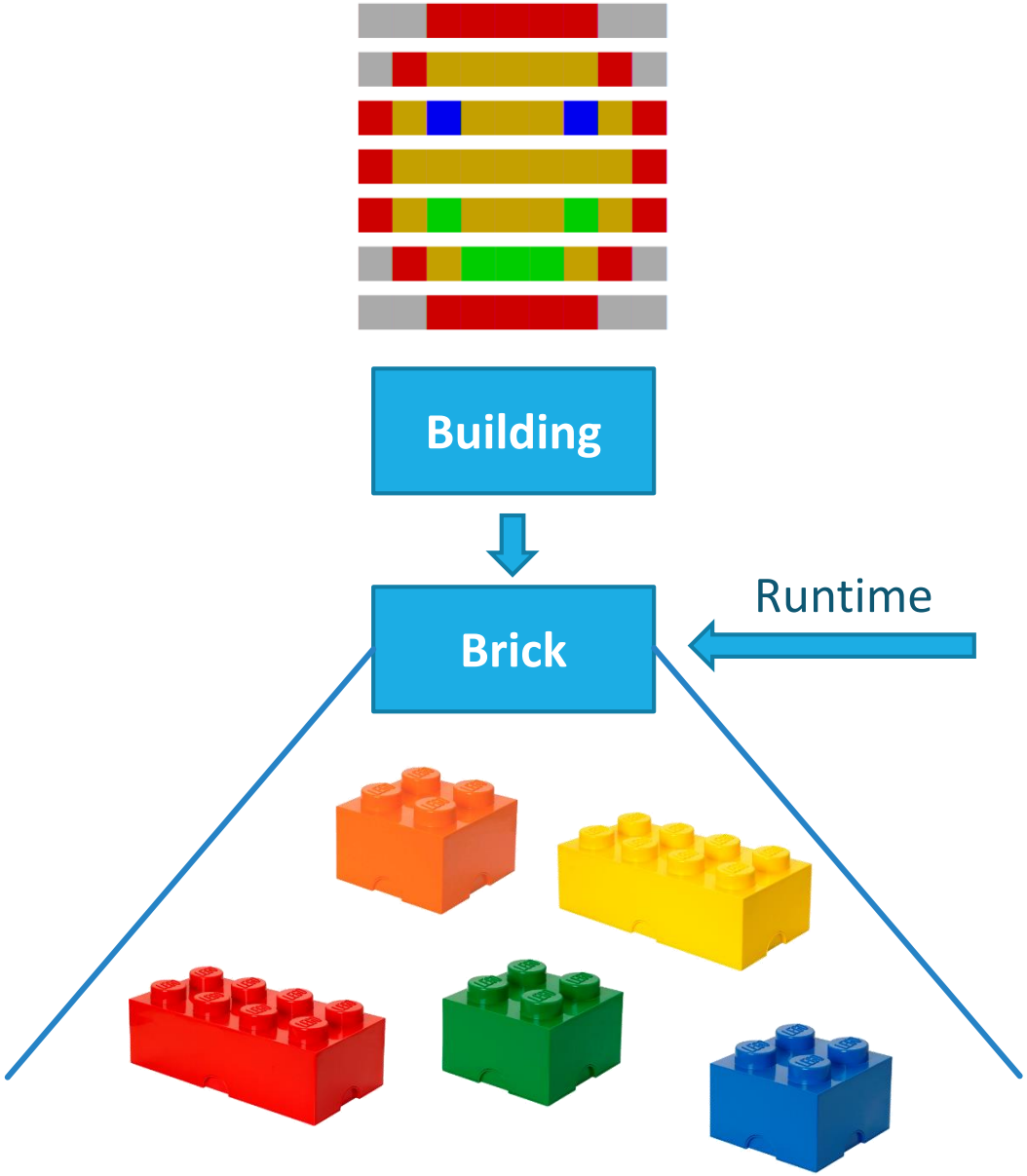


Behavioural Patterns: **Strategy**




```
Context context = new Context(new StrategyImplA());
context.executeAlgorithm();

context = new Context(new StrategyImplB());
context.executeAlgorithm();
```

Behavioural Patterns: **Strategy**



Behavioural Patterns: **Java API (Strategy)**

java.util.Comparator#compare() called from Collections#sort().
javax.servlet.http.HttpServlet: service() method, plus all of the doXXX() methods that accept HttpServletRequest and HttpServletResponse objects as arguments.
javax.servlet.Filter#doFilter()

Q&A