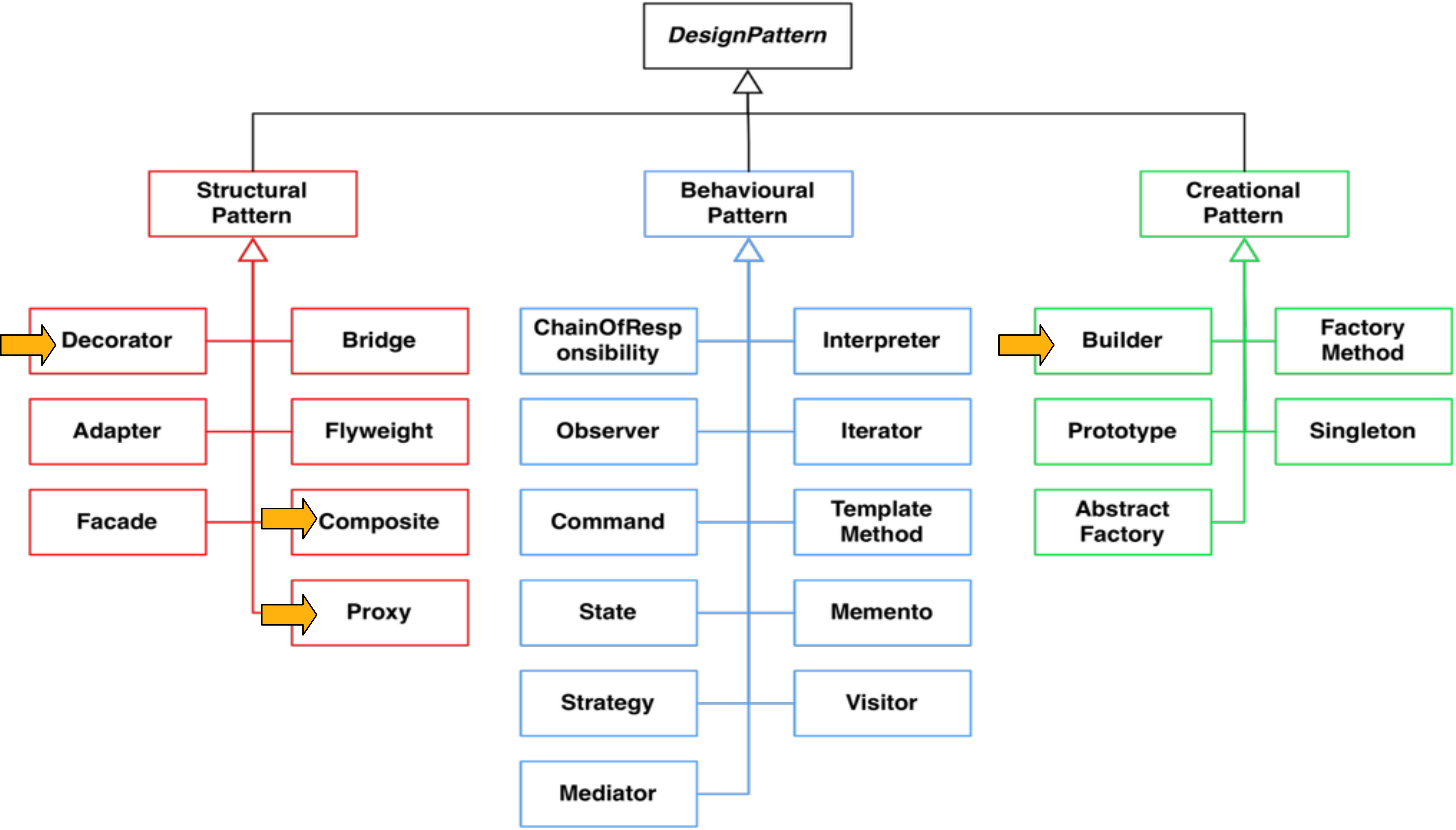
COIS3040 Lecture 7

Taxonomy of Design Patterns



DesignPattern

Structural Pattern

Behavioural Pattern

Creational Pattern

Decorator

Bridge

Adapter

Flyweight

Facade

Composite

Proxy

ChainOfResp onsibility

Interpreter

Observer

Iterator

Command

Template Method

State

Memento

Strategy

Visitor

Mediator

Builder

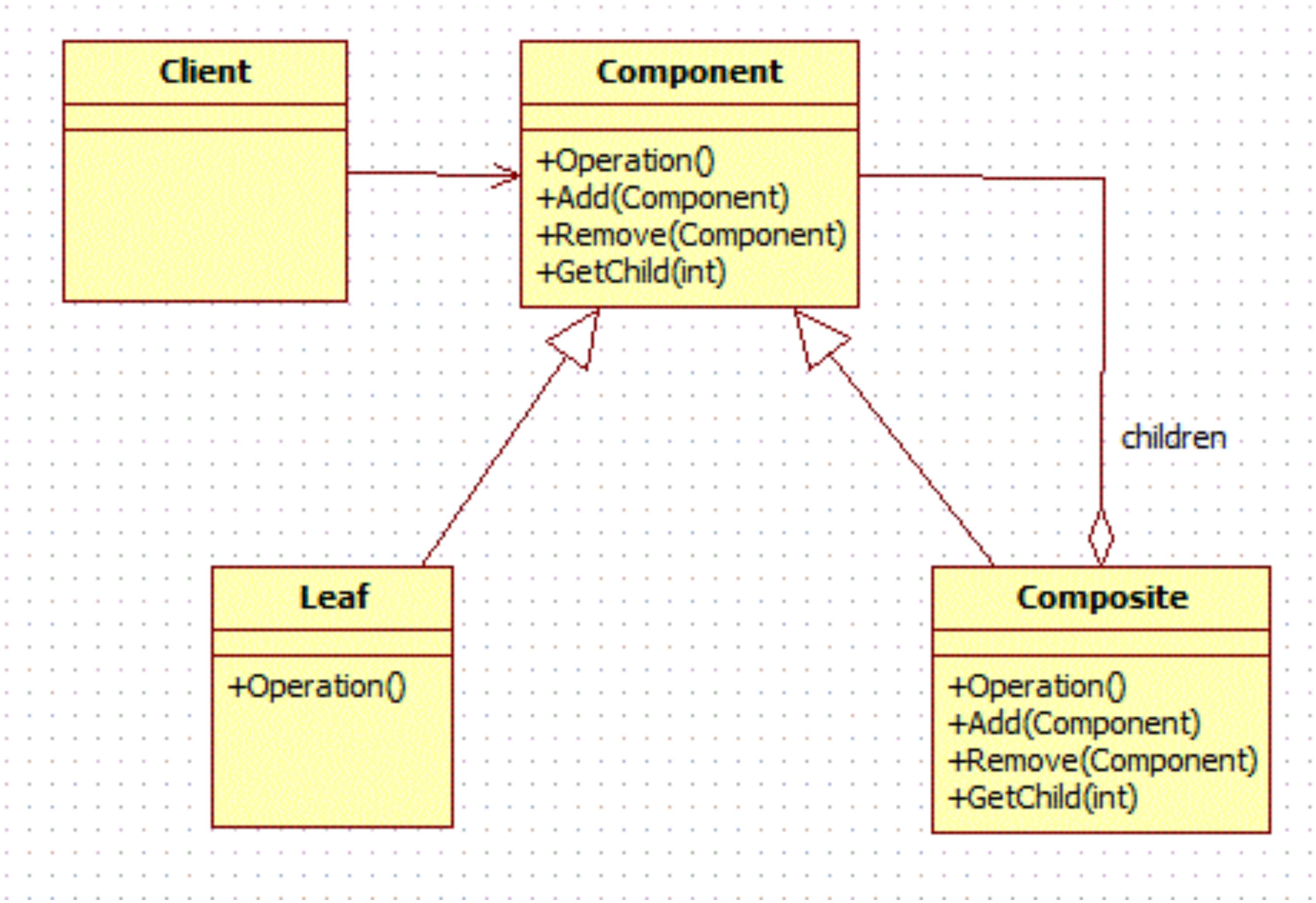
Factory Method

Prototype

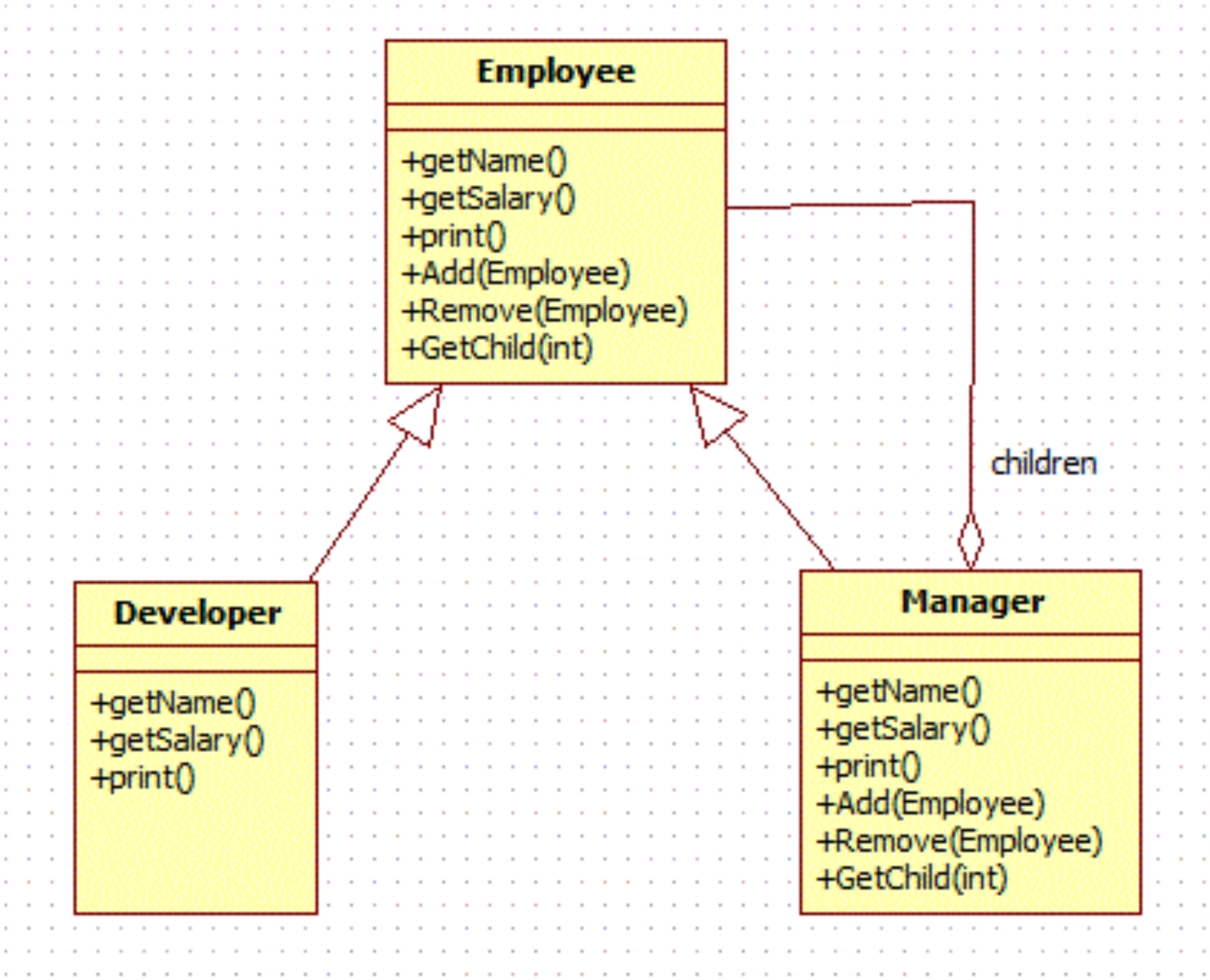
Singleton

Abstract Factory

Composite


Client
Component
+Operation()
+Add(Component)
+Remove(Component)
+GetChild(int)
children
Leaf
+Operation()
Composite
+Operation()
+Add(Component)
+Remove(Component)
+GetChild(int)

Composite


Employee
+getName()
+getSalary()
+print()
+Add(Employee)
+Remove(Employee)
+GetChild(int)
children
Developer
+getName()
+getSalary()
+print()
Manager
+getName()
+getSalary()
+print()
+Add(Employee)
+Remove(Employee)
+GetChild(int)

Composite

- Use it when you want to represent part-whole hierarchies of objects.
- You want client to be able to ignore difference between compositions of objects and individual objects.Clients will treat all objects in the composite structure uniformly.

Composite

```
public interface Employee {
   public void add(Employee employee);
   public void remove(Employee employee);
   public Employee getChild(int i);
   public String getName();
   public double getSalary();
   public void print();
}
```

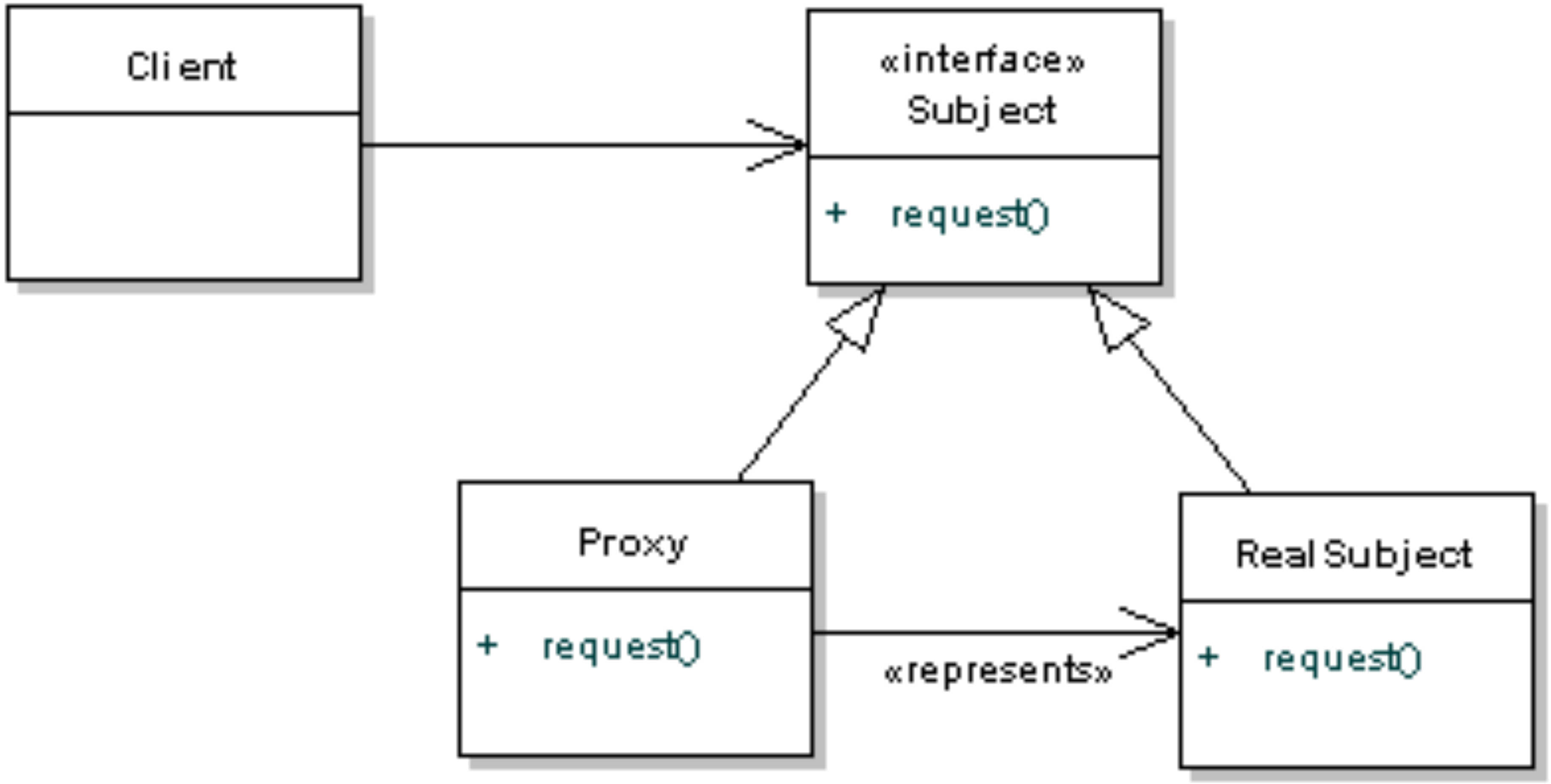
Composite

```
public class Manager implements Employee{

public Manager(String name,double salary){
  this.name = name;
  this.salary = salary;
 }
 List<Employee> employees = new ArrayList<Employee>();
 public void add(Employee employee) {
    employees.add(employee);
 }
 public Employee getChild(int i) {
  return employees.get(i);
 }
 public String getName() {
  return name;
 }
 public double getSalary() {
  return salary;
 }
 public void remove(Employee employee) {
  employees.remove(employee);
 }
}
```

Composite

```
public class Developer implements Employee{
 private String name;
  private double salary;
  public Developer(String name,double salary){
    this.name = name;
    this.salary = salary;
  }
  public void add(Employee employee) {
    //this is leaf node so this method is not applicable to this class.
  }
  public Employee getChild(int i) {
    //this is leaf node so this method is not applicable to this class.
    return null;
  }
  public String getName() {
    return name;
  }
  public double getSalary() {
    return salary;
  }
  public void remove(Employee employee) {
    //this is leaf node so this method is not applicable to this class.
  }

}
```

Proxy


Client
«interface»
Subject
+ request()
Proxy
+ request()
«represents»
Real Subject
+ request()

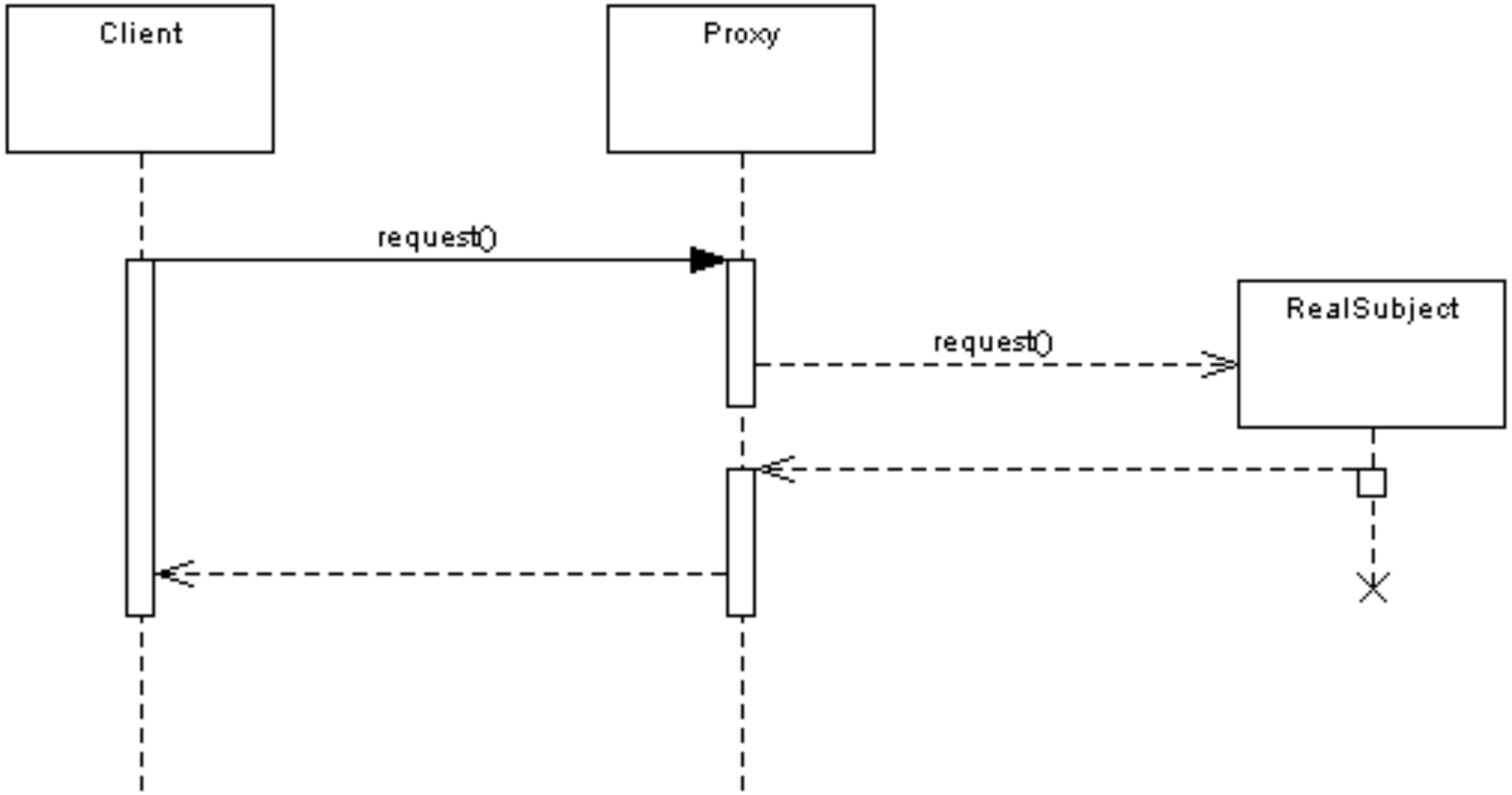
Proxy

- A Proxy can also be defined as a surrogate.
- In the real world a cheque or credit card is a proxy for what is in our bank account.
- That's exactly what the Proxy pattern does - controls and manage access to the object they are "protecting".

When to use proxy?

- The object being represented is external to the system.
- Objects need to be created on demand.
- Access control for the original object is required
- Added functionality is required when an object is accessed.

Proxy



Proxy

```
public interface Image{ public void displayImage(); }

public class RealImage implements Image{
public RealImage(URL url)   {
//load up the image
loadImage(url);
}
public void displayImage()   {
//display the image
}
//a method that only the real image has
private void loadImage(URL url)  {
//do resource intensive operation to load image
}
}
```

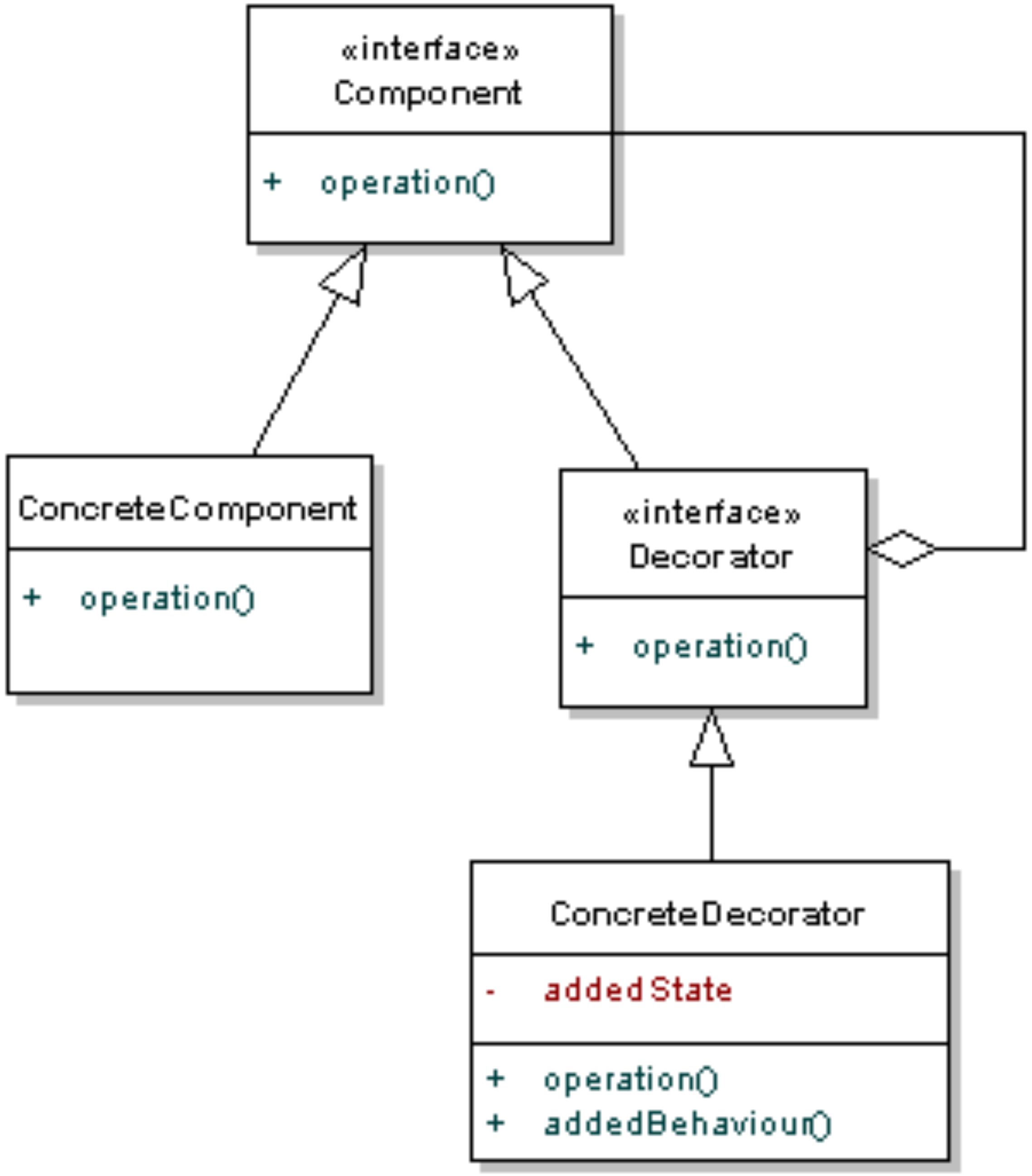
Proxy

```
public class ProxyImage implements Image{
private URL url;
public ProxyImage(URL url)
{       this.url = url;
}
//this method delegates to the real image
public void displayImage()
{
RealImage real = new RealImage(url);
real.displayImage();
}
}
```

Proxy Vs. Adapter

- Adapter pattern is to change the interface of class/ library A to the expectations of client B. The typical implementation is a wrapper class or set of classes. The purpose is not to facilitate future interface changes, but current interface incompatibilities.
- The purpose of the proxy pattern is to create a stand-in for a real resource. Why?
    - The real resource resides on a remote computer —proxy facilities communication.
    - The real resource is expensive to create (the proxy ensures the cost is not incurred unless/ until really needed)

Decorator


«interface»
Component
+ operation()
ConcreteComponent
+ operation()
«interface»
Decorator
+ operation()
ConcreteDecorator
- addedState
+ operation()
+ addedBehaviour()

Decorator

- Attach additional responsibilities to an object dynamically.
- Decorators provide a flexible alternative to subclassing for extending functionality.
- The concept of a decorator is that it adds additional attributes to an object dynamically. A real world example of this would be a picture frame. The picture is our object, which has it's own characteristics. For display purposes we add a frame to the picture, in order to decorate it.
- Decorator acts as a wrapper object.

Decorator

```
public interface IEmail
{
  public String getContents();
}
//concrete component
public class Email implements IEmail{
private String content;
public Email(String content)   {
this.content = content;
 }
@Override
public String getContents() {
//general email stuff
return content;
}
}
```

Decorator

```
public abstract class EmailDecorator implements IEmail{
//wrapped component
IEmail originalEmail;
}
//concrete decorator
public class ExternalEmailDecorator extends EmailDecorator{
private String content;
public ExternalEmailDecorator(IEmail basicEmail)   {
  originalEmail = basicEmail;
 }
@Override   public String getContents()   {
    //  secure original
content = addDisclaimer(originalEmail.getContents());
  return content;
}
private String addDisclaimer(String message)   {
//append company disclaimer to message
 return  message + "\n Company Disclaimer";
 }
 }
```

Decorator

```
//concrete decorator
public class SecureEmailDecorator extends EmailDecorator{
private String content;
public SecureEmailDecorator(IEmail basicEmail)   {
originalEmail = basicEmail;
}
@Override
public String getContents()   {
 //  secure original
content = encrypt(originalEmail.getContents());
return content;
}
private String encrypt(String message)
{      //encrypt the string
  return  encryptedMessage;
}
}
```

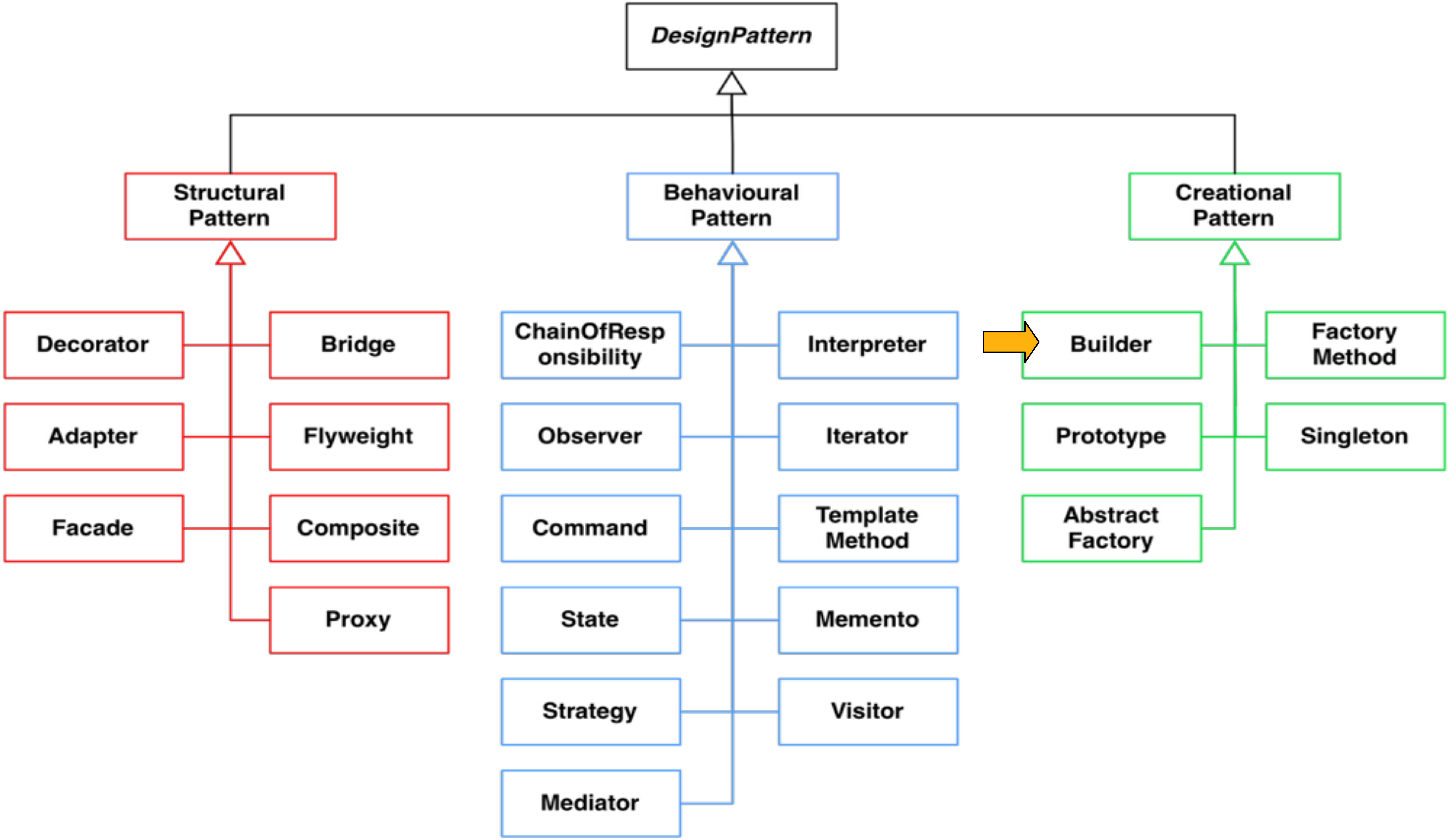
Decorator

```
public class EmailSender{
public void sendEmail(IEmail email)
{       //read the email to-address, to see if it's
going outside of the company
//if so decorate it
ExternalEmailDecorator external = new
ExternalEmailDecorator(email);
external.getContents();
//send
   }
}
```

Decorator Vs. Proxy

- Decorator Pattern focuses on dynamically adding functions to an object, while Proxy Pattern focuses on controlling access to an object.
- Relationship between a Proxy and the real subject is typically set at compile time, Proxy instantiates it in some way, whereas Decorator is assigned to the subject at runtime, knowing only subject's interface.

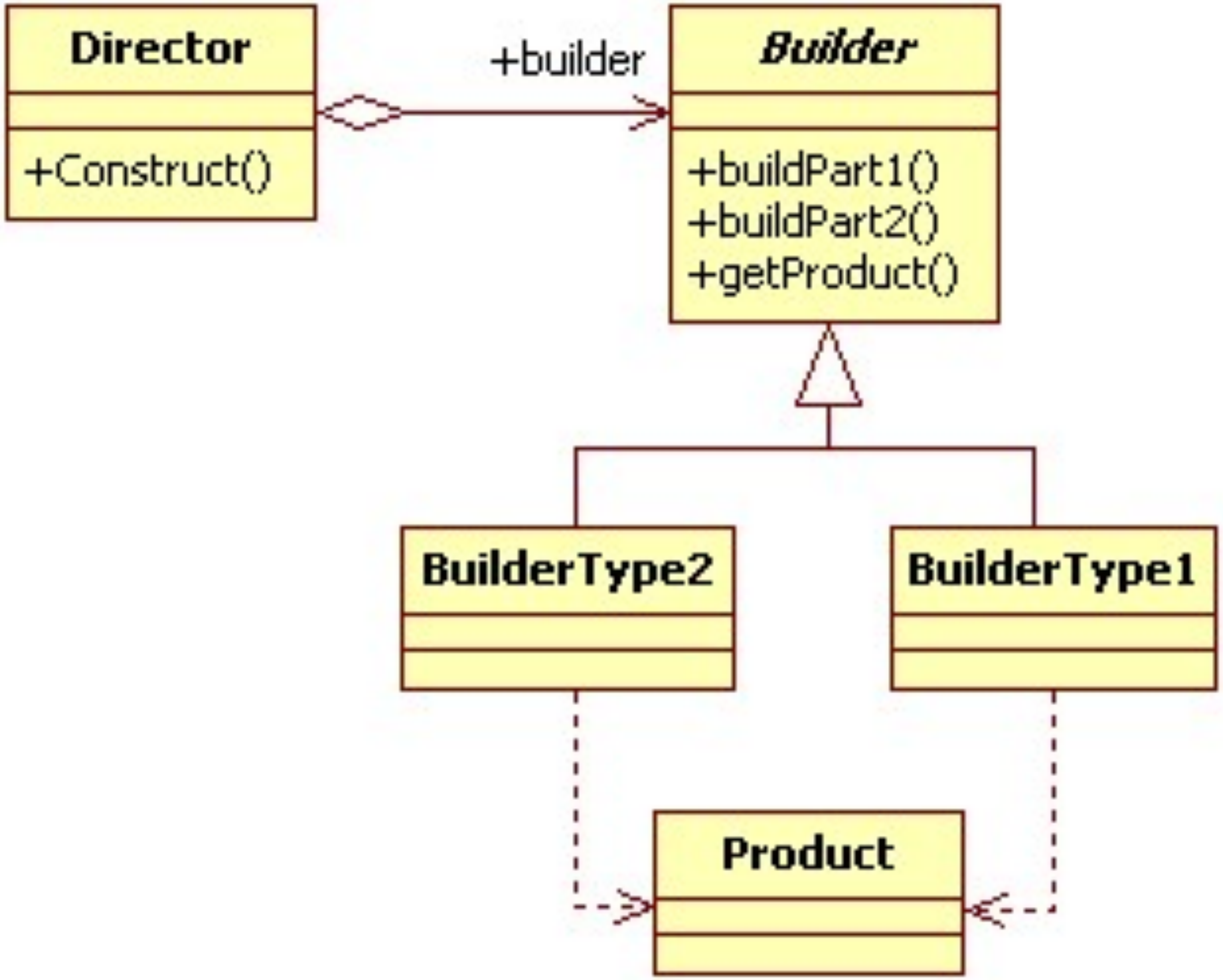
Taxonomy of Design Patterns



Builder

- Separates the construction of a complex object from its representation so that the same construction process can create different representations.
- The algorithm for creating a complex object should be independent of the parts that make up the object and how they're assembled.
- The construction process must allow different representations for the object that's constructed.

Builder Pattern



Builder

```
class Car {
    private int wheels;
    private String color;
    public Car() {
    }
    @Override
    public String toString() {
        return "Car [wheels = " + wheels + ", color = " + color + "]";
    }
    public int getWheels() {
        return wheels;
    }
    public void setWheels(final int wheels) {
        this.wheels = wheels;
    }
    public String getColor() {
        return color;
    }
    public void setColor(final String color) {
        this.color = color;
    }
}
```

Builder

```
interface CarBuilder {
    CarBuilder setWheels(final int wheels);

    CarBuilder setColor(final String color);

    Car build();
}
```

Builder

```
class CarBuilderImpl implements CarBuilder {
    private Car car;
    public CarBuilderImpl() {
        car = new Car();
    }
    @Override
    public CarBuilder setWheels(final int wheels) {
        car.setWheels(wheels);
        return this;
    }
    @Override
    public CarBuilder setColor(final String color) {
        car.setColor(color);
        return this;
    }
    @Override
    public Car build() {
        return car;
    }
}
```

Builder

```
public class CarBuildDirector {
    private CarBuilder builder;

    public CarBuildDirector(final CarBuilder builder) {
        this.builder = builder;
    }

    public Car construct() {
        return builder.setWheels(4).setColor("Red").build();
    }

    public static void main(final String[] arguments) {
        CarBuilder builder = new CarBuilderImpl();
        CarBuildDirector carBuildDirector = new CarBuildDirector(builder);
        System.out.println(carBuildDirector.construct());
    }
}
```

Builder with Static Inner Class

```
public class NutritionalFacts {
    private int sodium; private int fat; private int carbo;
    public static class Builder {
        private int sodium; private int fat; private int carbo;
        public Builder(int s) {
            this.sodium = s;
        }
        public Builder fat(int f) {
            this.fat = f;
            return this;
        }
        public Builder carbo(int c) {
            this.carbo = c;
            return this;
        }
        public NutritionalFacts build() {
            return new NutritionalFacts(this);
        }
    }
    private NutritionalFacts(Builder b) {
        this.sodium = b.sodium;
        this.fat = b.fat;
        this.carbo = b.carbo;
    }
}
```

Builder vs. Abstract Factory

- Abstract Factory: Emphasizes a family of product objects (either simple or complex)
- Builder: Focuses on constructing a complex object step by step
- Abstract Factory: Focus on *what* is made
- Builder: Focus on *how* it is made
- Abstract Factory: Focus on defining many different types of *factories* to build many *products*, and it is not a one builder for just one product
- Builder: Focus on building a one complex but one single *product*
- Abstract Factory: Defers the choice of what concrete type of object to make until run time
- Builder: Hide the logic/ operation of how to compile that complex object
- Abstract Factory: *Every* method call creates and returns different objects
- Builder: Only the *last* method call returns the object, while other calls partially build the object