Learning Objectives for Today

- One more **Behavioural** Patterns
  - **Visitor**
- **Structural** Design Patterns:
  - **Decorator**

The Visitor Pattern

In many cases we want to apply operations to elements contained within a structure.

For example, recall the **iterator pattern.**

This allows to access **sequential** elements of a **collection;** it **polymorphically** executes actions on each element.

A drawback of this pattern is that **all elements of the structure need share a common data type**!

The Visitor Pattern

Allows operations to be performed on the **elements of a structure.**

Lets you define operations to be performed **without changing the classes of elements that are targets of the operations.**

Components of The Visitor Pattern

Visitor: An interface/abstract class that declares the **visit operations** for all types of visitable classes.

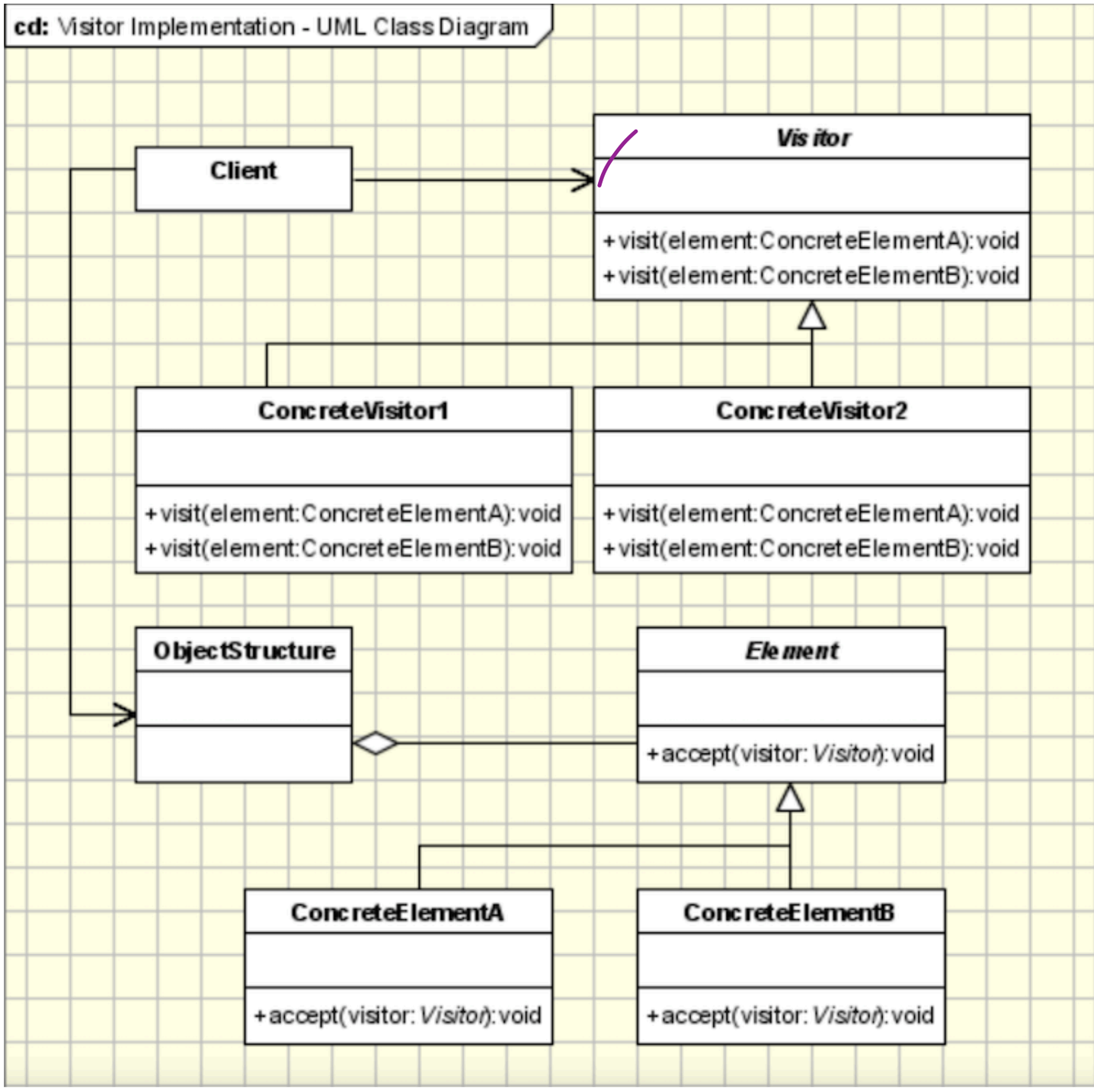
ConcreteVisitor: Implements the methods declared in the visitor interface/abstract class.

Visitable: An abstraction which declares an **accept** operation. It will be used to allow an object to **accept** a **visitor**.

ConcreteVisitable: Implements the Visitable abstraction and defines the **accept** operation.

ObjectStructure: The software structure containing all the objects that can be visited.

The Visitor Pattern



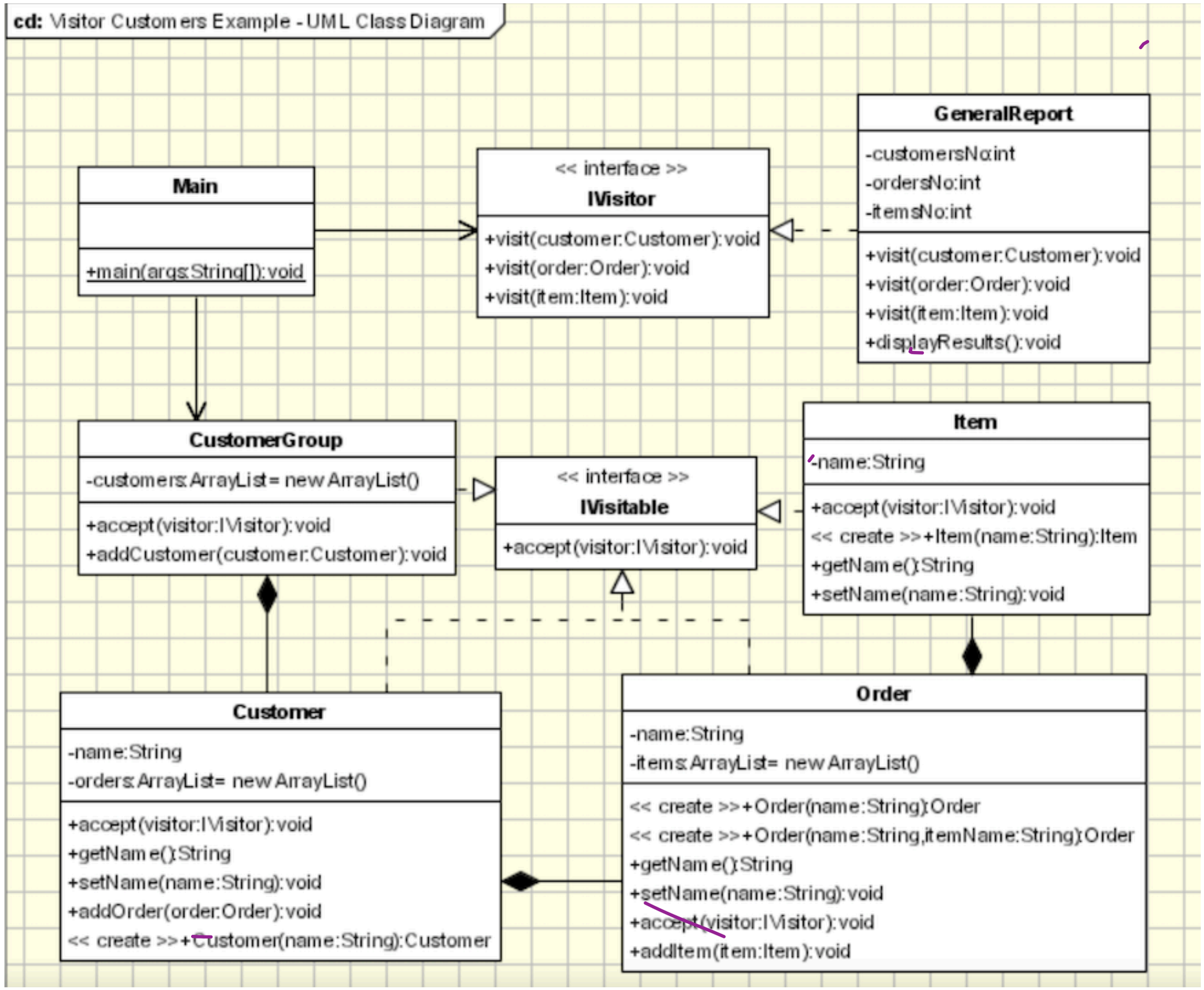
The Visitor Pattern: When to Use It

Similar operations have to be performed on objects of different types grouped within a **structure**, and **distinct and/or unrelated operations** must be performed on each object.

We can handle this by creating **separate visitors** for each operation and by separating **operation implementation** from **object structure**.

New operations can be added without needing to change the object structure. So, the design is open to extension!

The Visitor Pattern: Example



The Visitor Pattern: Another Example

Remember **Abstract Syntax Trees (ASTs)** from CSC148?

This is a tree whose nodes can represent **Expressions** or **Statements** that show up in code, like **Integers** or **Strings** or **Binary operations (like addition).**

We may want to traverse an AST to print it, evaluate it (i.e. perform semantic analysis), parse or compile it.

Structural Design Patterns

Structural Design Patterns realize relationships between objects or entities. Some examples of these patterns include:

- The **Adapter Pattern,** which matches interfaces of different classes
- The **Bridge Pattern,** which separates an object's interface from its implementation
- The **Composite Pattern,** which creates a tree structure of simple and composite objects
- The **Decorator Pattern,** which adds responsibilities to objects dynamically
- The **Facade Pattern,** which creates a single class to represents an entire subsystem
- The **Proxy Pattern,** which allows objects to represent other objects

We will cover the **Adapter**, **Composite** and **Decorator** Pattern in more detail today.

You can look up some of the other patterns online, and via the course Quercus instance!!

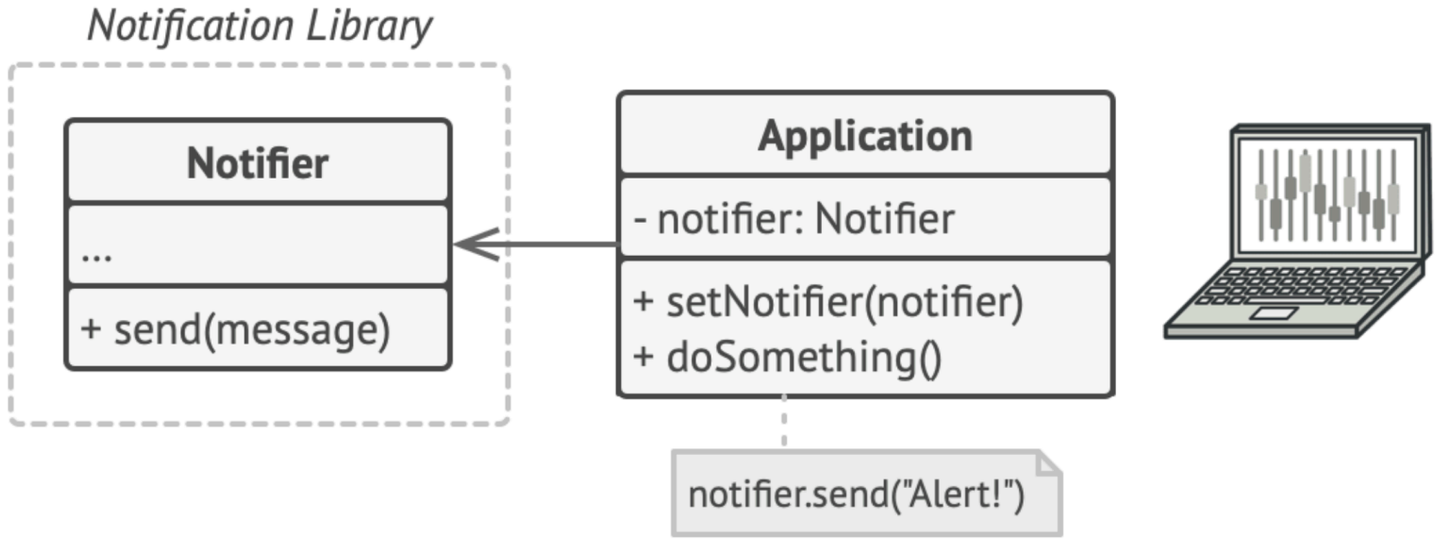
The Decorator Pattern

Decorator lets you attach **behaviours** to objects by placing these objects inside special **wrapper objects** that contain the behaviours.

When have we seen wrapper classes of objects before?

The Decorator Pattern: Why?

Imagine that you are working on a notification library which lets other programs notify their users about important events.

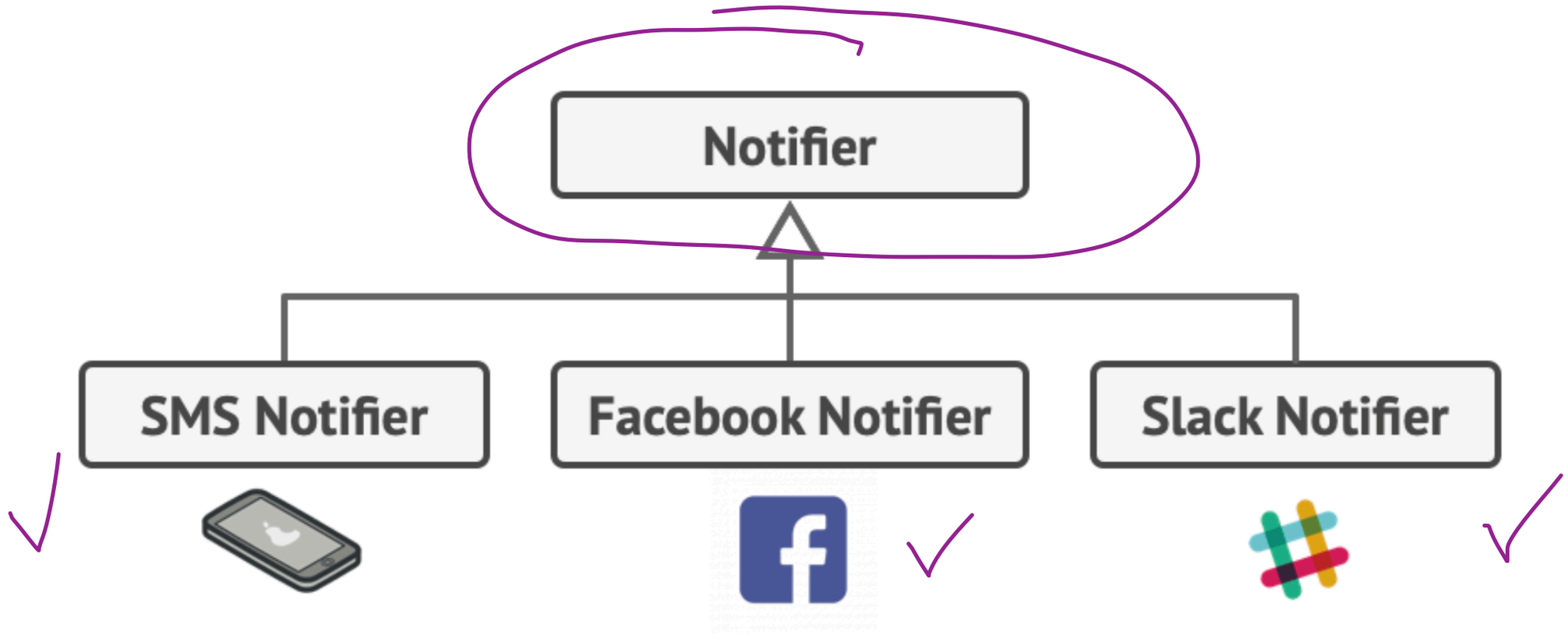


A program could use the notifier class to send notifications about important events to a predefined set of emails.

What pattern does this remind you of?

The Decorator Pattern: Why?

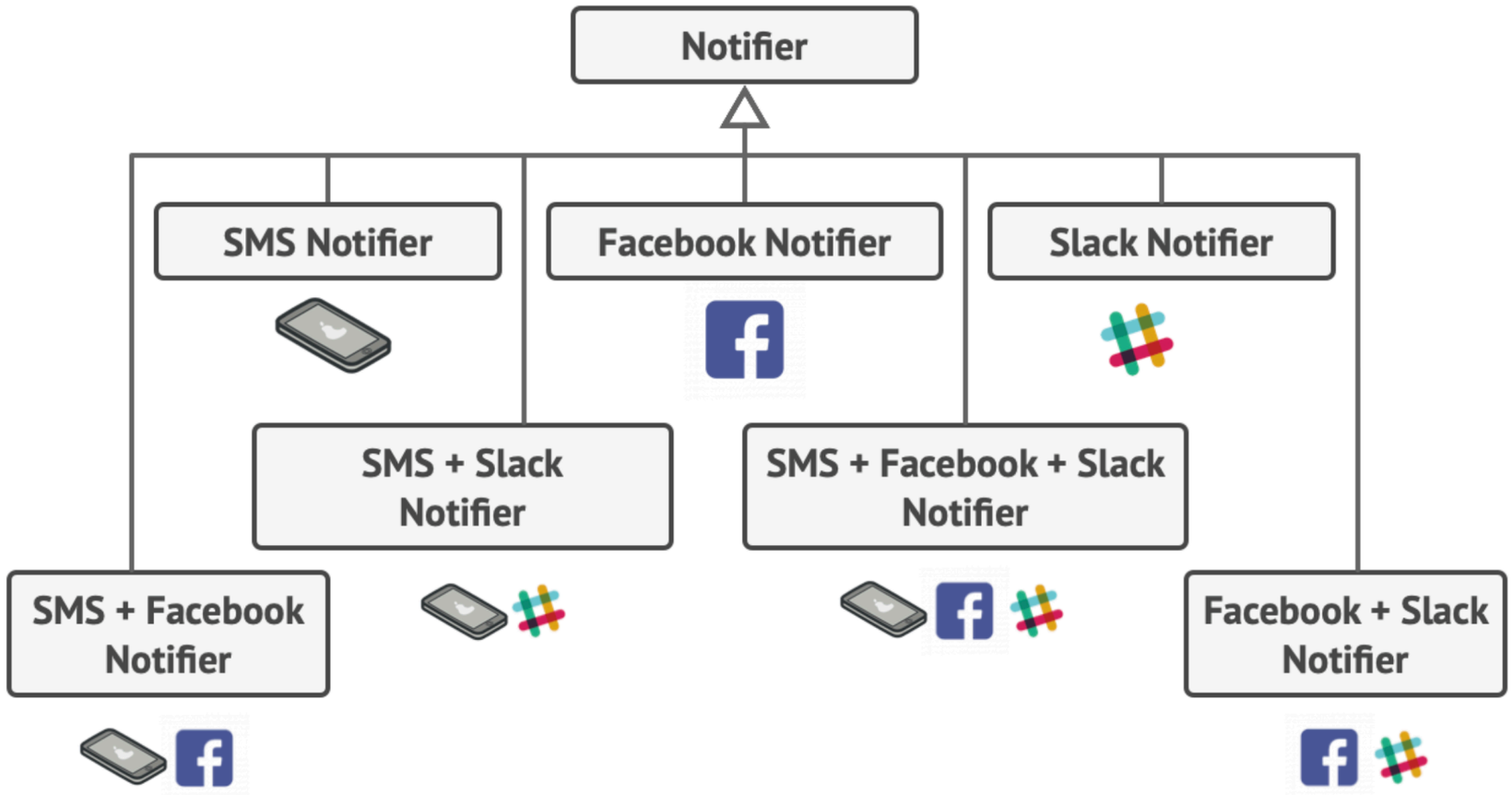
Many users get SMS, Facebook, Slack notifications.



Each notification type is implemented as a notifier's subclass.

The Decorator Pattern: Why?

What happens when we try to accommodate people who need many combinations of notifications?



What's wrong with this design???

The Decorator Pattern: How?

Extending a class may be the first thing that comes to mind when you need to alter an object's behaviour.

However, inheritance has several serious caveats:

- **Inheritance is static.** You can't alter the behaviour of an existing object at runtime.
- **Subclasses have just one parent class.**

This can be fixed by using **Aggregation** or **Composition**.

Both of the alternatives work almost the same way:

- one object has a reference to other objects and delegates work to them
- with inheritance, the object itself does the work

The Decorator Pattern: How?

Inheritance vs. Aggregation

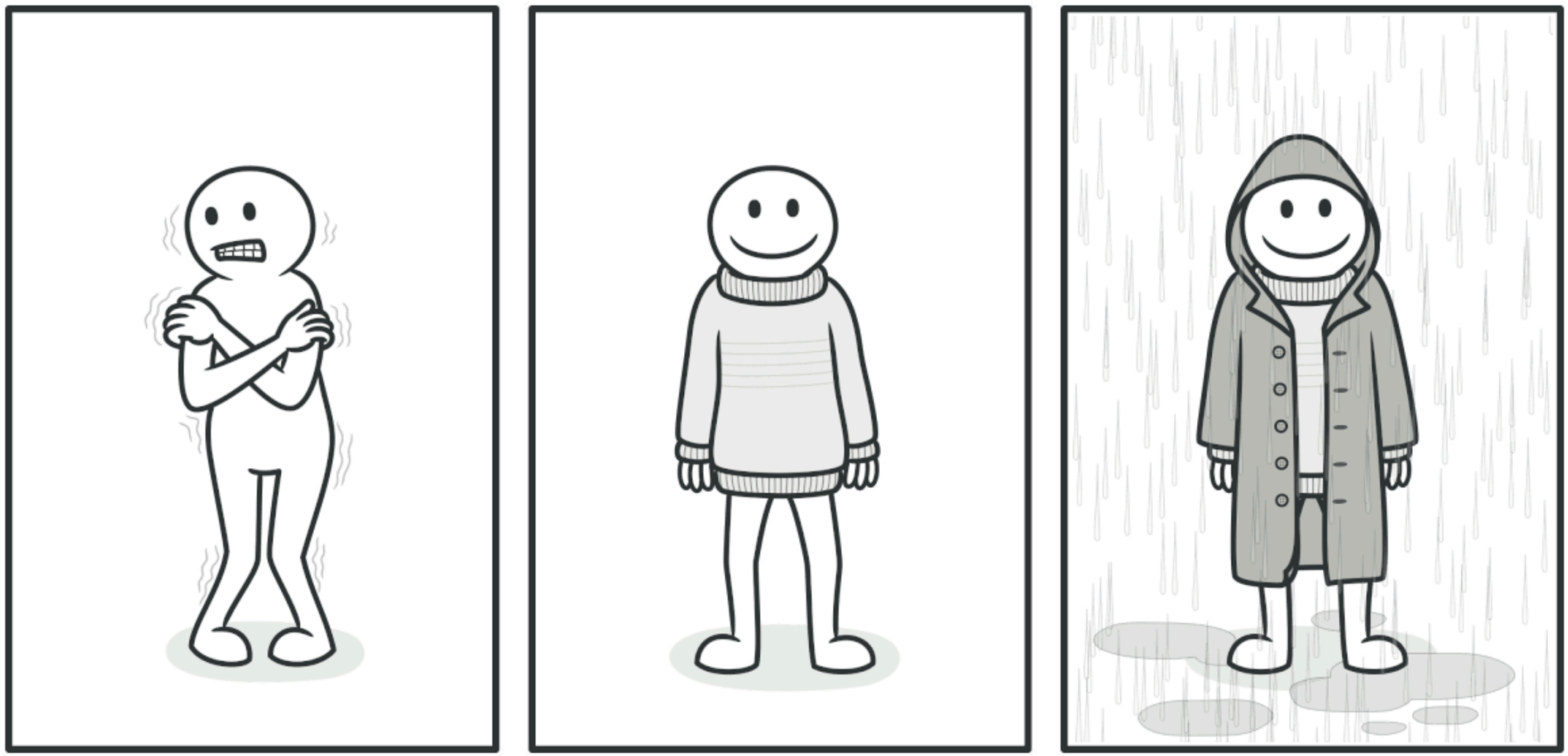
The Decorator Pattern: How?

A *wrapper* is an object that can be linked with a target object.

The wrapper contains the same methods as the target and delegates manages requests to the target for method results.

The wrapper may alter the result by doing something either before or after it passes the request to the target.

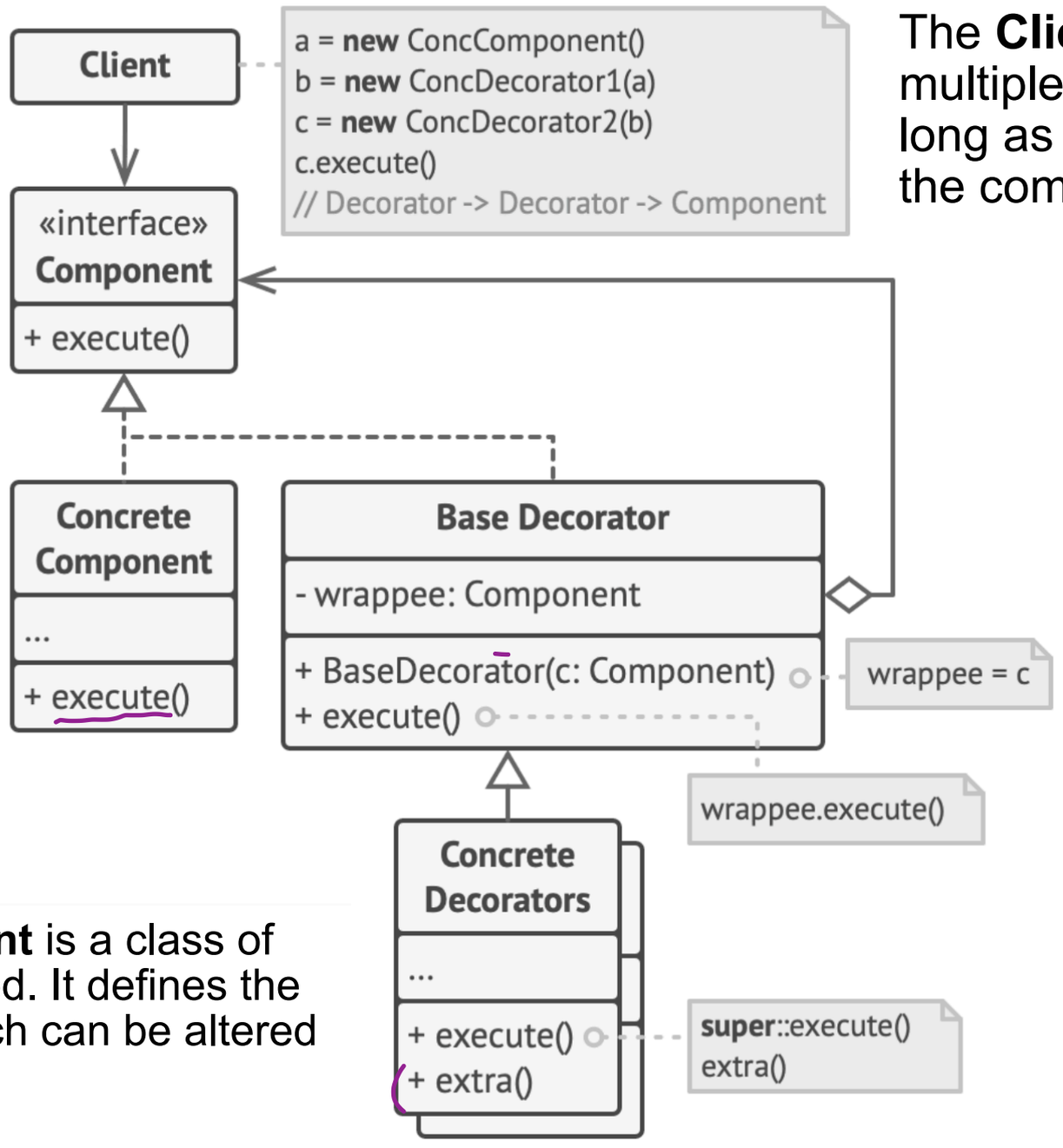
The Decorator Pattern: Example



You get a combined effect from wearing multiple pieces of clothing.

The Decorator Pattern: UML

The **Component** declares the common interface for both wrappers and wrapped objects.



The **Client** can wrap components in multiple layers of decorators, as long as it works with all objects via the component interface.

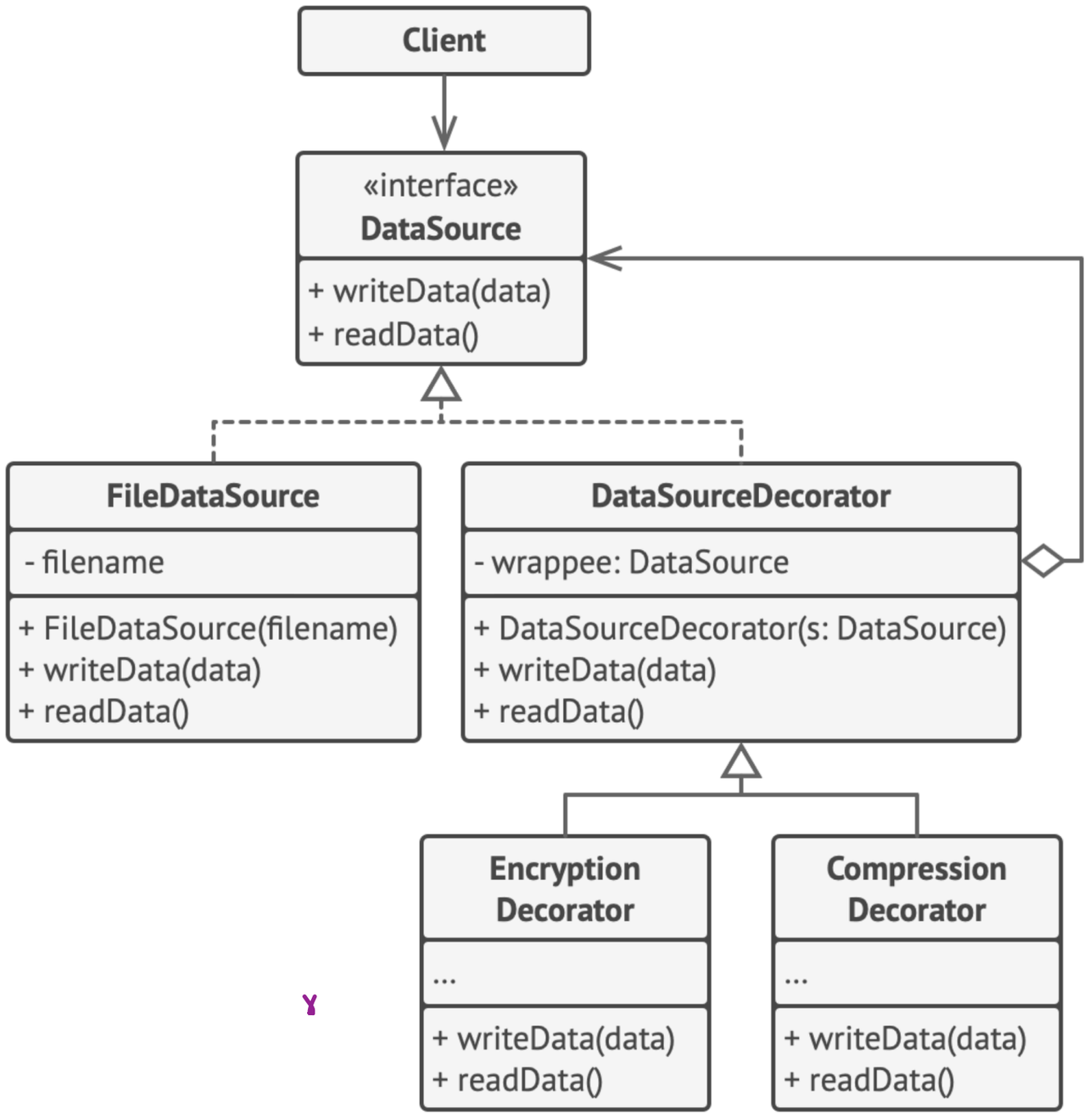
Concrete Decorators define extra behaviours that can be added to components dynamically. Concrete decorators override methods of the base decorator and execute their behaviour either before or after calling the parent method.

The **Base Decorator** class has a field for referencing a wrapped object. The field's type should be declared as the component interface so it can contain both concrete components and decorators. The base decorator delegates all operations to the wrapped object.

Concrete Component is a class of objects being wrapped. It defines the basic behaviour, which can be altered by decorators.

FROM https://refactoring.guru/design-patterns/decorator

The Decorator Pattern: UML



The Decorator Pattern: Example

This example shows how you can adjust the behaviour of an object without changing its code.

Initially, the business logic class could only read and write data in plain text.

Then we created several small wrapper classes that add new behaviour after executing standard operations in the wrapped object.

The first wrapper encrypts and decrypts data, and the se**cond one** compresses and extracts data.

You can **combine** wrappers by wrapping a decorator with another.

The Decorator Pattern: When?

Use the Decorator pattern:

- when you need to assign extra behaviours to objects
- when you have code that depends on objects you want to enhance but don’t want to break existing code
- when it’s awkward or not possible to extend an object's behaviour using inheritance.

Decorator lets you structure your logic into layers. Create a decorator for each layer and compose objects with combinations at runtime.

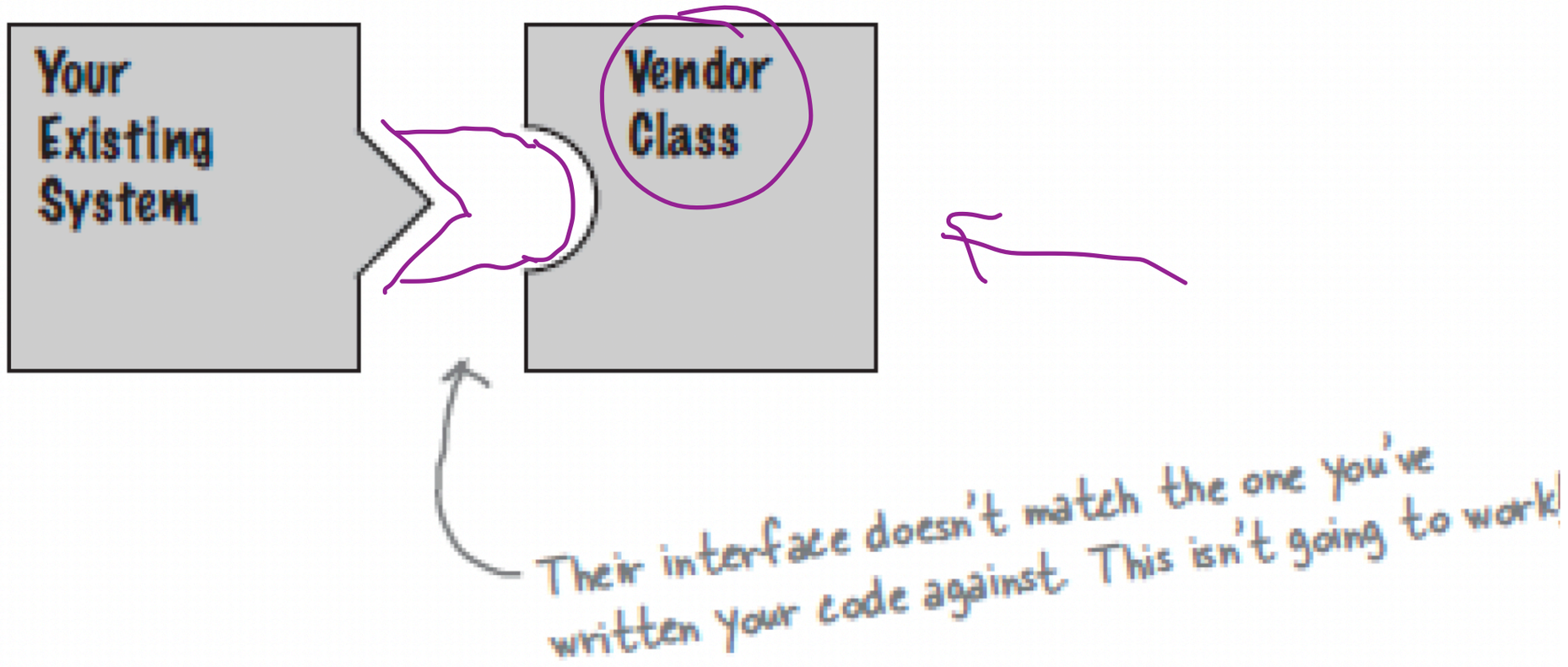
Client code can treat all decorated objects in the same way, since they all follow a common interface.

Learning Objectives for Today

- **Structural** Design Patterns:
  - **Adapter**
  - **Composite**
- **Creational** Design Patterns:
  - **Factory**

The Adapter Pattern: Why?

If we have an existing system which need to work with a vendor class library, with interfaces designed differently from last vendor's, we would write a class that adapts the new vendor interface into the one we are expecting:



The Adapter Pattern: Why?

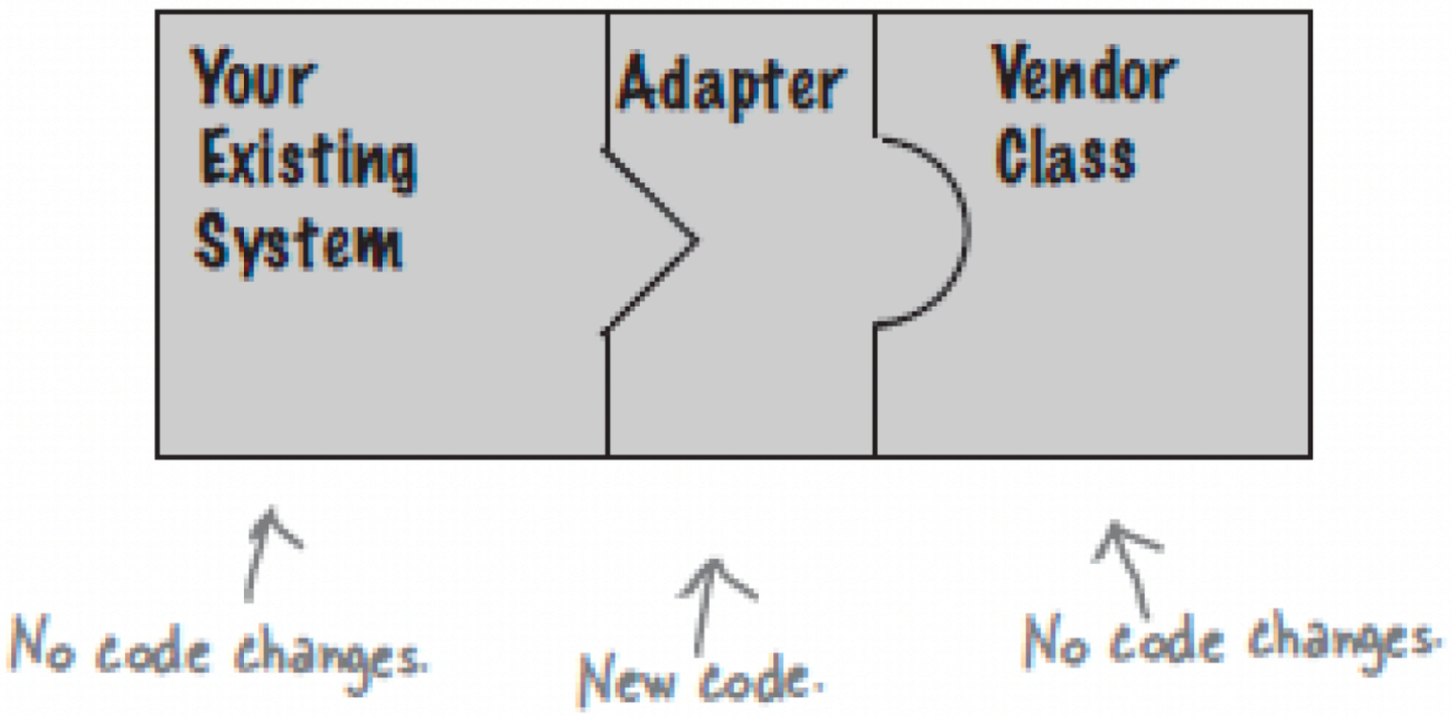
Adapter Pattern

- Adapter pattern allows allows us to use a client with an incompatible interface by creating an Adapter that does the conversation.
- Adapter decouples the client from the implemented interface, and if we expect the interface to change over time, the adapter encapsulates that change so that the client doesn't have to be modified each time it needs to operate against a different interface.

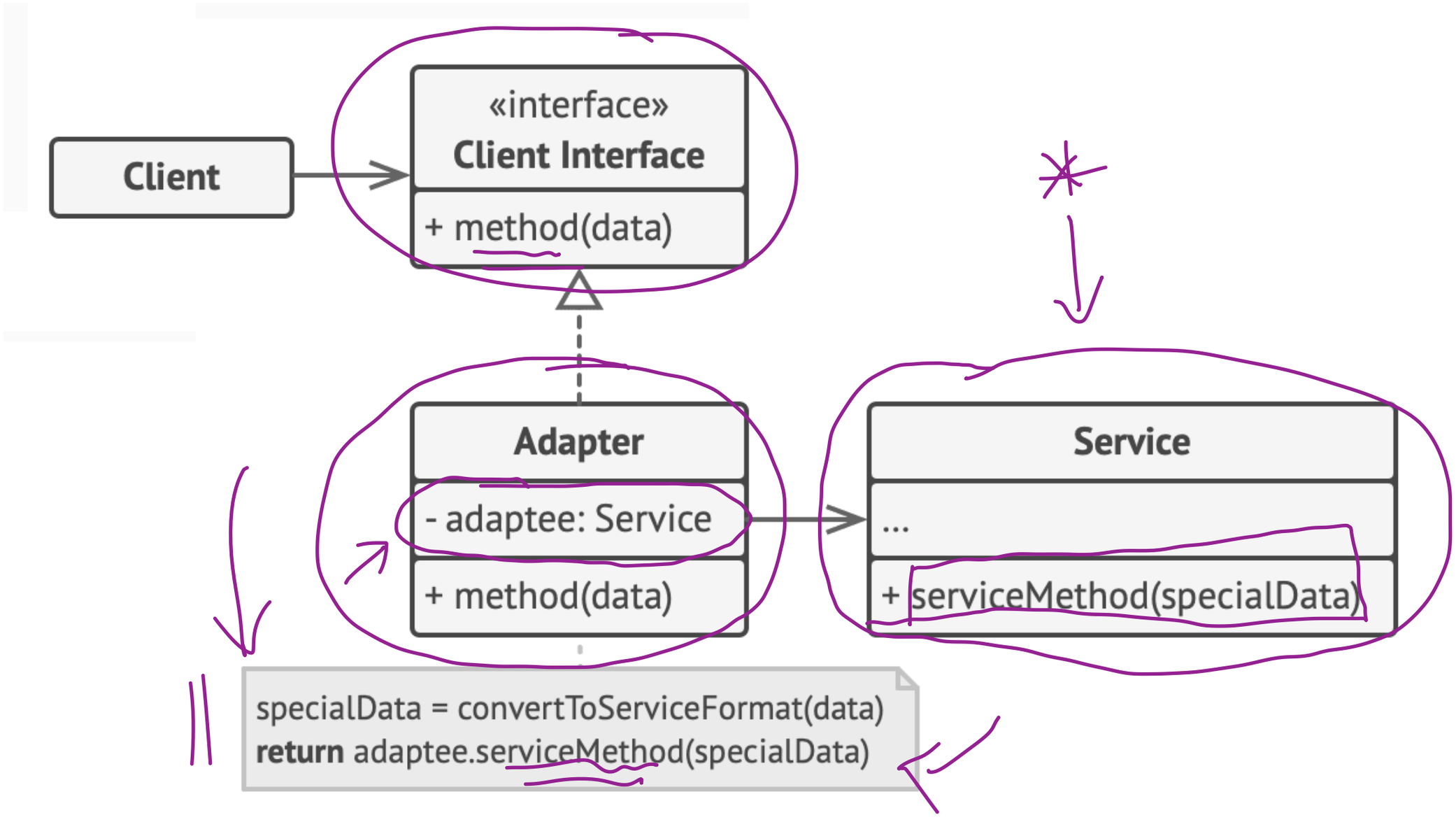
The Adapter Pattern: What?

Definition

The **Adapter Pattern** converts the interface of a class into another interface the clients expects. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.



The Adapter Pattern: UML



The Adapter Pattern: Example

An audio player device can play **mp3** files only.

We want to use an audio player capable of playing **vlc** and **mp4** files.

The Adapter Pattern: Example

Components:

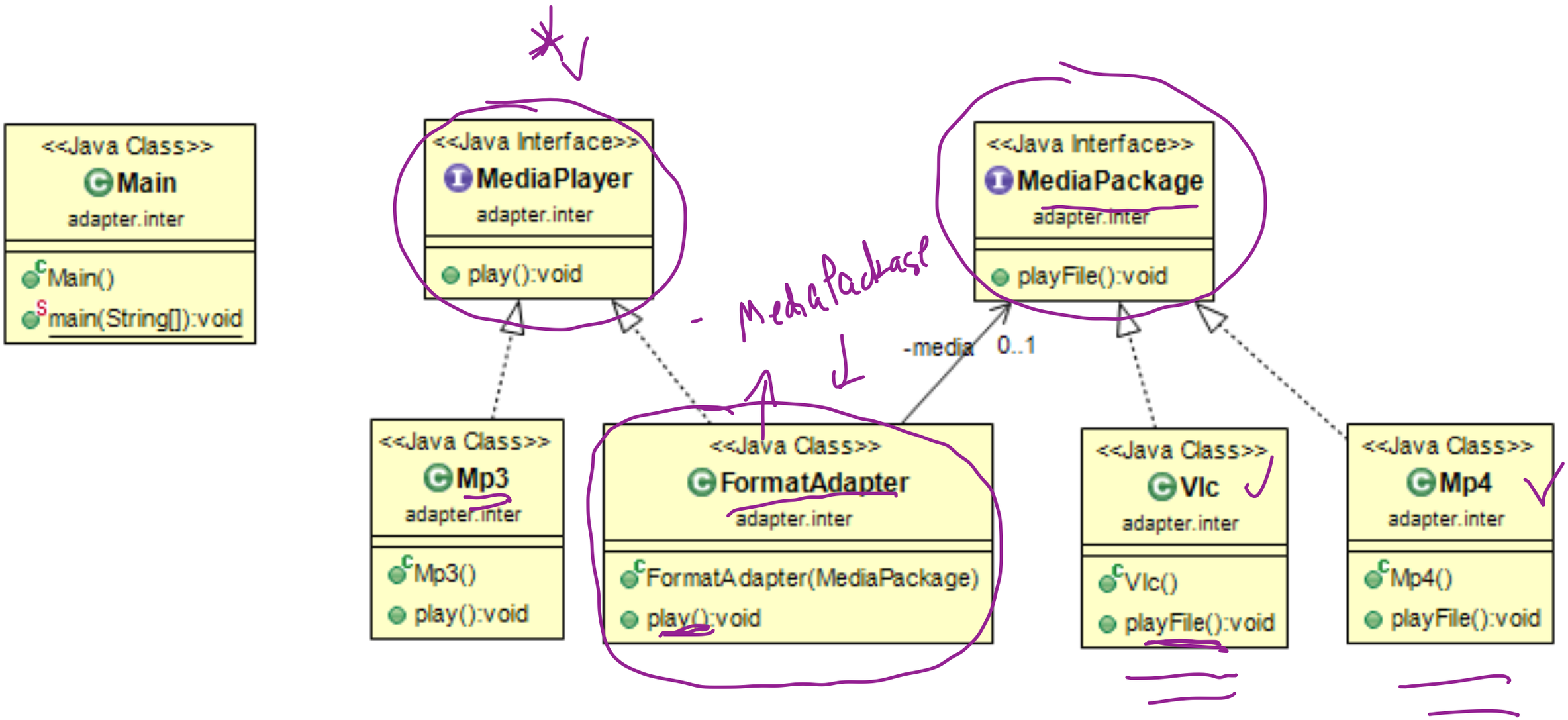
A **MediaPlayer interface** and a concrete class **MP3** that implements the MediaPlayer interface. These are able to play **mp3** format files.

A **MediaPackage interface** and **concrete classes** that implement the MediaPackage interface. These are able to play **vlc** and **mp4** format files.

To adapt the MP3 class for MediaPackage, we create an adapter class **FormatAdapter.** This implements the **MediaPlayer** interface and uses **MediaPackage** objects to play the required format.

MP3 class will use the adapter class **FormatAdapter.** FormatAdapters will accept **MediaPackage** method calls and convert to **MP3** compatible calls as appropriate.

The Adapter Pattern: UML



https://ssaurel.medium.com/implement-the-adapter-design-pattern-in-java-f9adb6a8828f

The Adapter Pattern: Example

Create interfaces for **MediaPlayer** and **MediaPackage.**

```
public interface MediaPlayer {
   public void play(String fileName);
}

public interface MediaPackage {
   public void playFile(String fileName);
}
```

The Adapter Pattern: Example

Create **concrete classes** implementing the **MediaPlayer** interface

public class **MP3** implements **MediaPlayer** {

```
    @Override
    public void play(String filename) {
      System.out.println("Playing MP3 File " + filename);
    }
}
```

The Adapter Pattern: Example

Create concrete classes implementing the **MediaPackage** interface.

public class **VlcPlayer** implements **MediaPackage** {

```
@Override
public void playFile(String fileName) {
   System.out.println("Playing vlc file. Name: "+ fileName);
   }

}
```

```
public class Mp4Player implements MediaPackage {
   @Override
   public void playFile(String fileName) {
      System.out.println("Playing mp4 file. Name: "+ fileName);
   }
}
```

The Adapter Pattern: Example

Create **adapter** class.

```
public class FormatAdapter implements MediaPlayer {
    private MediaPackage media;
     public FormatAdapter(MediaPackage m) {
        media = m;
     }

     @Override
     public void play(String filename) {
        System.out.print("Using Adapter --> ");
        media.playFile(filename);
     }
}
```

The Adapter Pattern: Example

Use the **MediaPlayer** to play different types of audio formats.

```
public class Main {
    public static void main(String[] args) {
      MediaPlayer player = new MP3();
      player.play("file.mp3");
      player = new FormatAdapter(new MP4());
      player.play("file.mp4");
      player = new FormatAdapter(new VLC());
      player.play("file.avi");
      }
}
```

The Composite Pattern: What is it?

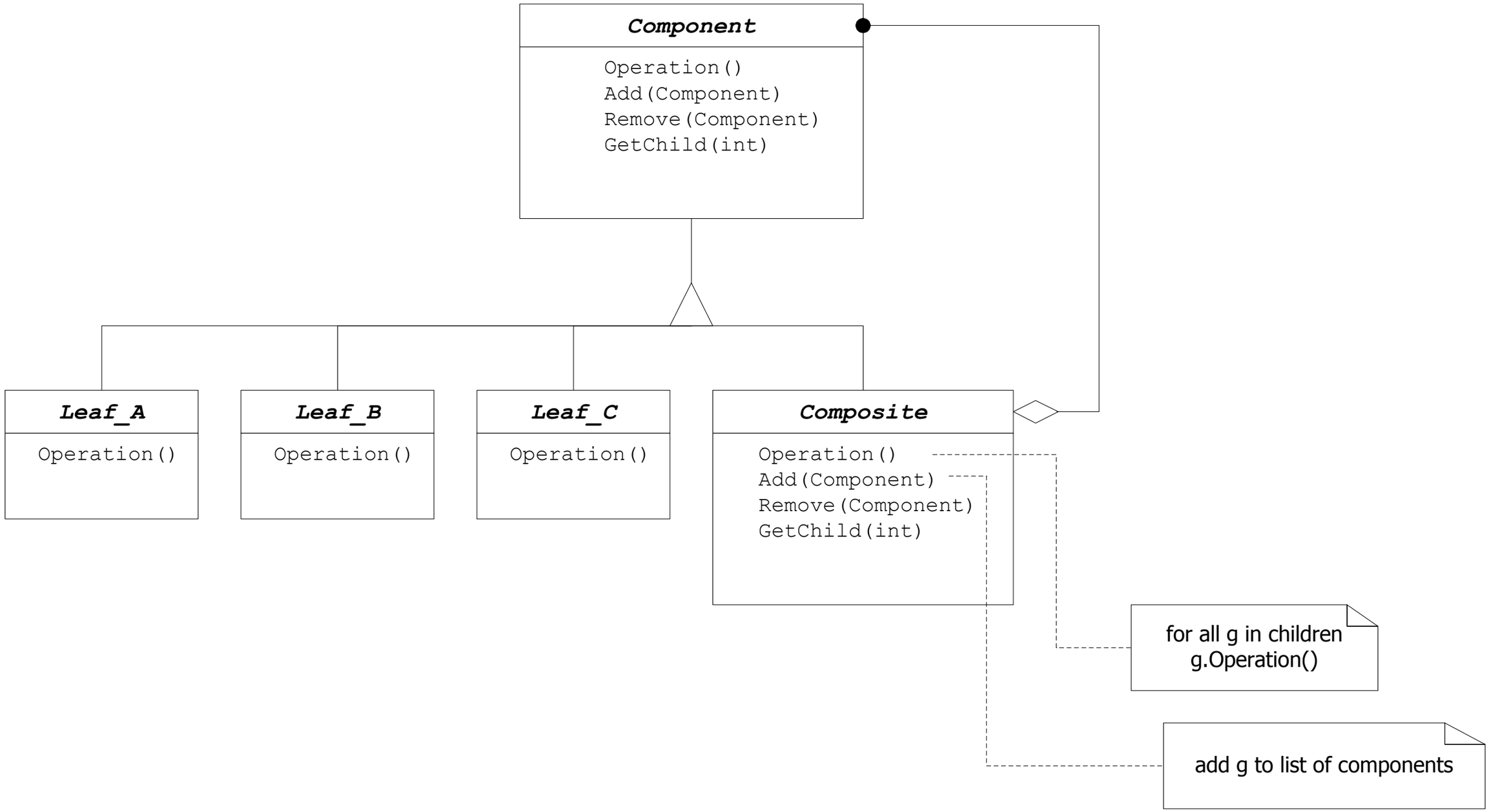
- The composite pattern collects objects into tree structures that represent part-whole hierarchies.
- Clients can then treat individual objects and compositions of objects in a uniform way.
- Scene graphs are examples of composites.
- You traversed a composite when you implemented the visitor pattern, too.

The Composite Pattern: When?

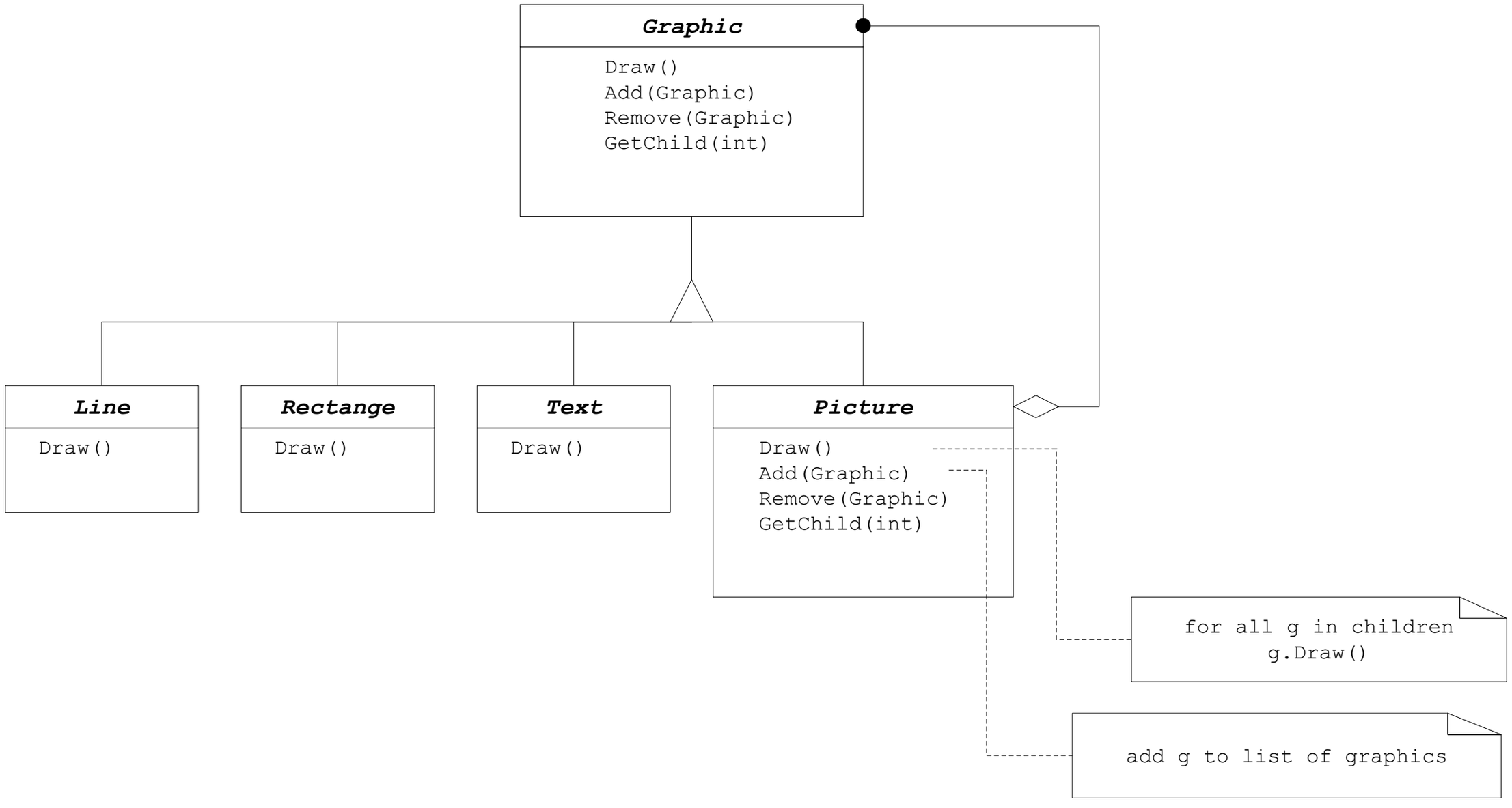
Use the Composite pattern:

- when you need to represent hierarchies of objects
- when you want clients to ignore the differences between individual objects and collections of objects
- When you want to create composites dynamically at run time, and/or create composites out of previously defined subsystems and parts

The Composite Pattern: UML



The Composite Pattern: MVC Example



The Composite Pattern: Example

Individual species, particularly birds, migrate

So do flocks of birds

So do a mixture of flocks, even other species

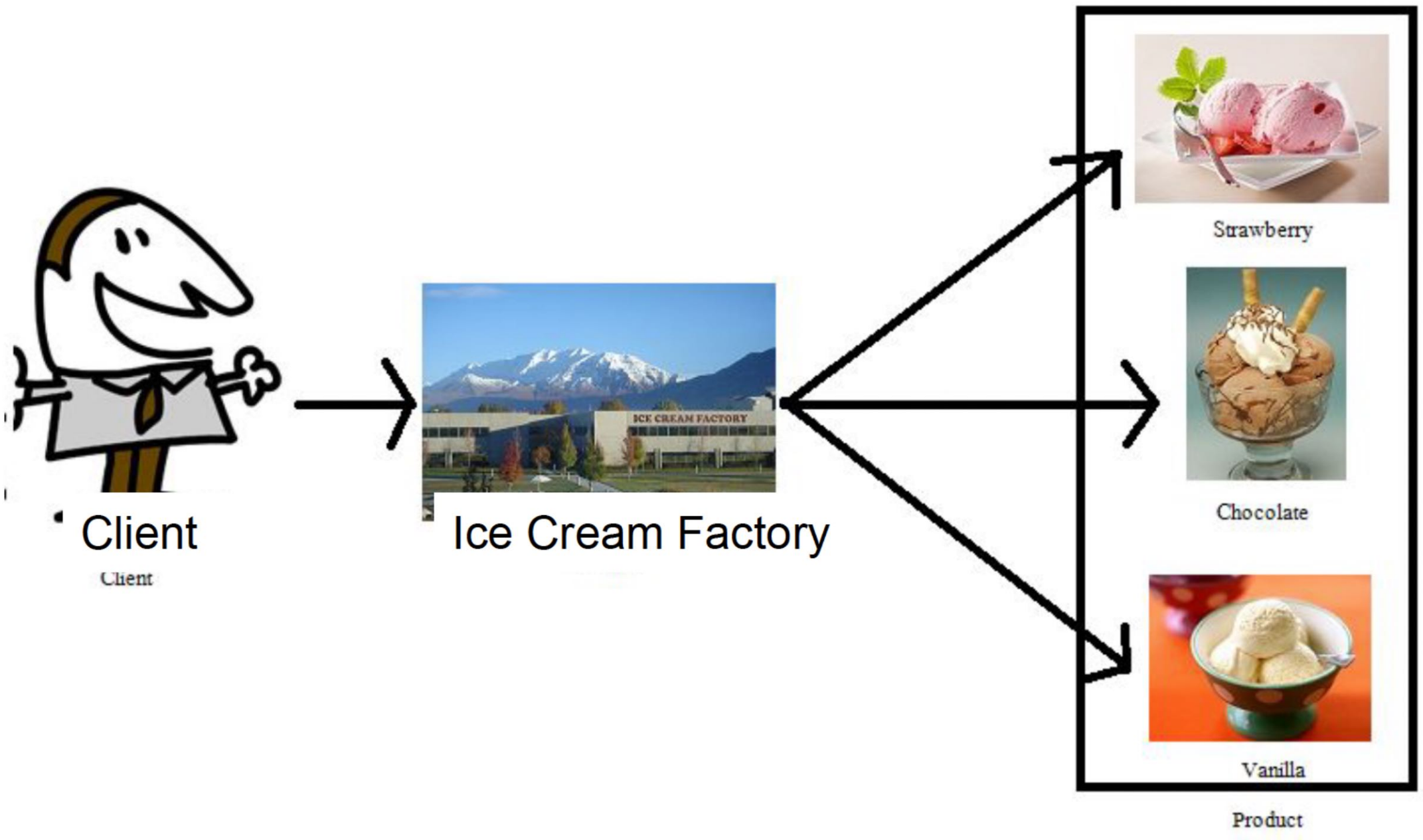
Creational Patterns

Creational design patterns deal with **object creation.**

Object creation can add complexity to software.

Creational design patterns attempt solve this problem by controlling object creation.

Factory Pattern: When Do We Need It?



Factory Pattern

This is one of the most commonly used design patterns.

It allows a client to create objects without exposing instantiation logic.

Food **food** = FoodFactory.createProduct(“Burger”);

Here, **food** is a **Burger** object.

The newly created object can then be referred to through a common interface:

food.getName();

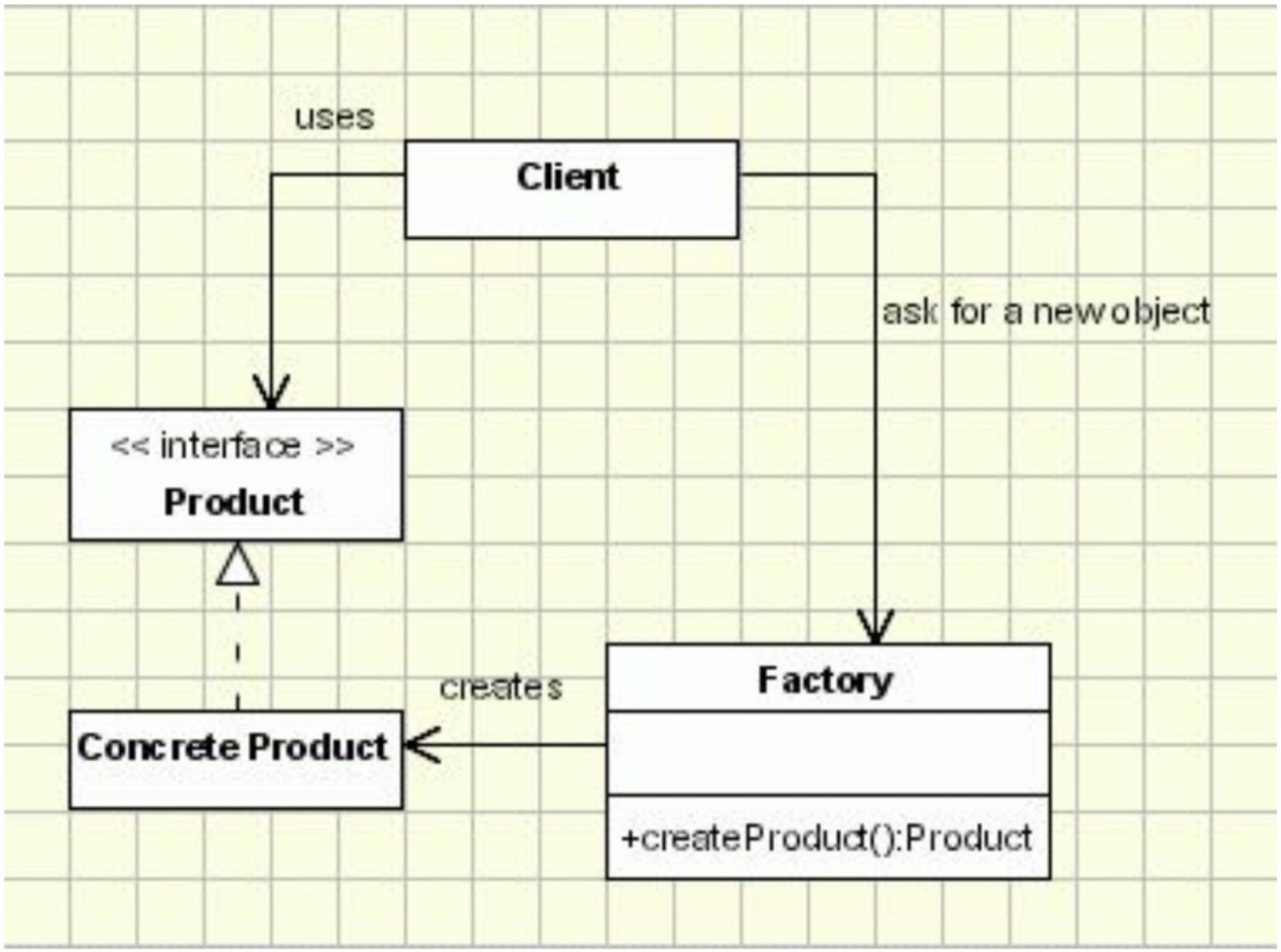
food.getCalories();

food.eat();

What do you think the UML for this arrangement might look like?

Factory Pattern: UML

UML of the **Factory** pattern



Factory Pattern: Implementation

Create a base class or interface for the product

e.g., **Food**

Implement concrete product classes by extending the base class

e.g., **Burger, Pizza, Salad, etc.**

Create the **Factory** class with a **createProduct()** method

ProductBase createProduct(String productID)

Return objects of different types according to **productID**.

Factory Example: Duck Factory

Step 1: Make some Ducks

```
public class Duck {

    private String name;

    public Duck(String name) {
        this.name = name;
    }

    public void quack() {
        System.out.println("Quack!");
    }
}
```

```
public class ToyDuck extends Duck{

    private int battery_life;

    public ToyDuck(String name, int battery_life) {
        super(name);
        this.battery_life = battery_life;
    }

    @Override
    public void quack() {
        if (this.battery_life > 0) {
            this.battery_life -= 1;
            super.quack();
        }
    }
}
```

UNIVERSITY OF TORONTO MISSISSAUGA

Factory Example: Duck Factory

Step 2: Make a Duck Factory

```
public class DuckFactory {

    public static Duck makeYellowDucks() {
        return new ToyDuck("Yellow", 1);
    }

    public static Duck makeRedDucks() {
        return new Duck("Red");
    }
}
```

Factory Example: Duck Client

Step 3: Use the Duck Factory

```
public class DuckClient {

    public static void main(String[] args) {
        int r;
        List<Duck> myDucks = new ArrayList<>(10);
        for (int i=0; i < 5; i++) {
            r = new Random().nextInt(2);
            System.out.println(r);
            if (r < 1)
                myDucks.add(DuckFactory.makeRedDucks());
            else
                myDucks.add(DuckFactory.makeYellowDucks());
        }

        // How many quacks?
        for (Duck d: myDucks) {
            for (int i=0; i < 3; i++)
                d.quack();
        }
    }

}
```