SOFTWARE ENGINEERING

Chapter 3.4B: Design Patterns

MOTIVATION...

“Don’t worry if it doesn’t work right. If everything did, you’d be out of a job.””

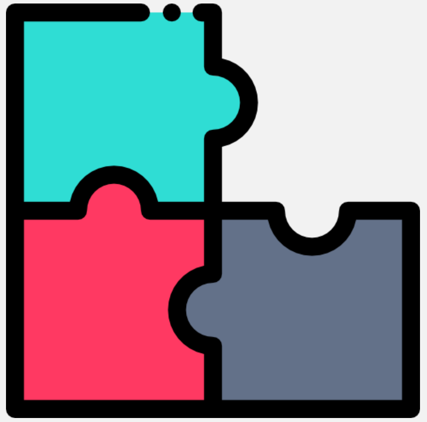
Mosher’s Law of Software Engineering

What is a design pattern?

Design Patterns

Are general repetible solutions to common software design problems.

Design Patterns Are..



Descriptions on how to solve a given problem

They can't be directly translated into code

They provide tested and proven development paradigms

Design Patterns

Creational

Structural

Behavioural

Creational Patterns

Deal with object creation problems controlling subtle problems for a given situation

Structural Patterns

Deal with relationships between entities

Behavioural Patterns

Deal with common communication and interaction between objects

Abstract Factory

Abstract Factory

Problem

- Platform dependencies are not always engineered in advance…
- Operators #ifdef may appear *a lot* detecting multiple platforms…
- The operator new is considered harmful

Abstract Factory

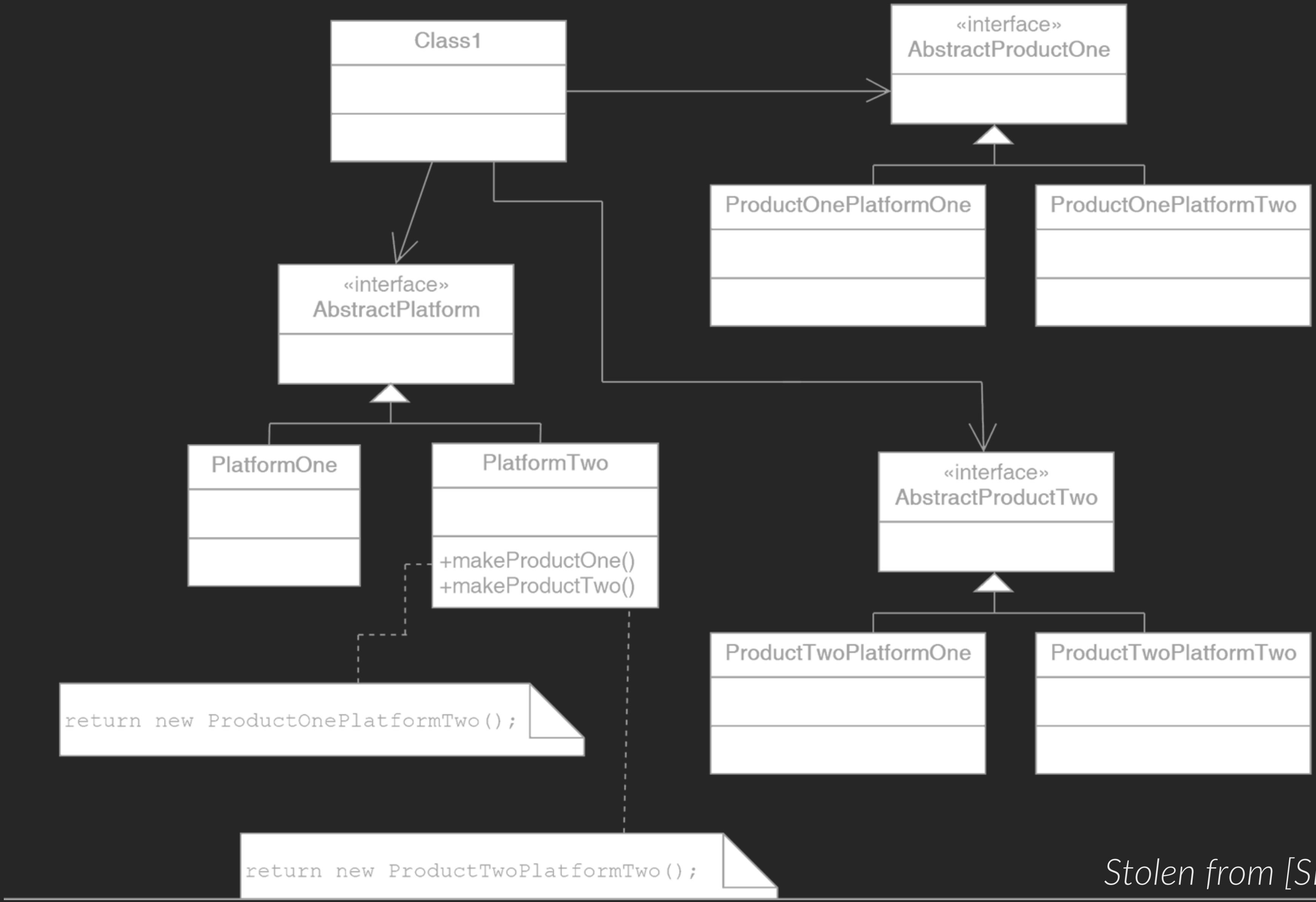
Solution!

- Create a factory object that creates all the services for the entire platform family

```
#ifdef __IOS__
//Code!

#ifdef __ANDROID___
//Code!
```

Generic Pattern



```
return new ProductOnePlatformTwo();
```

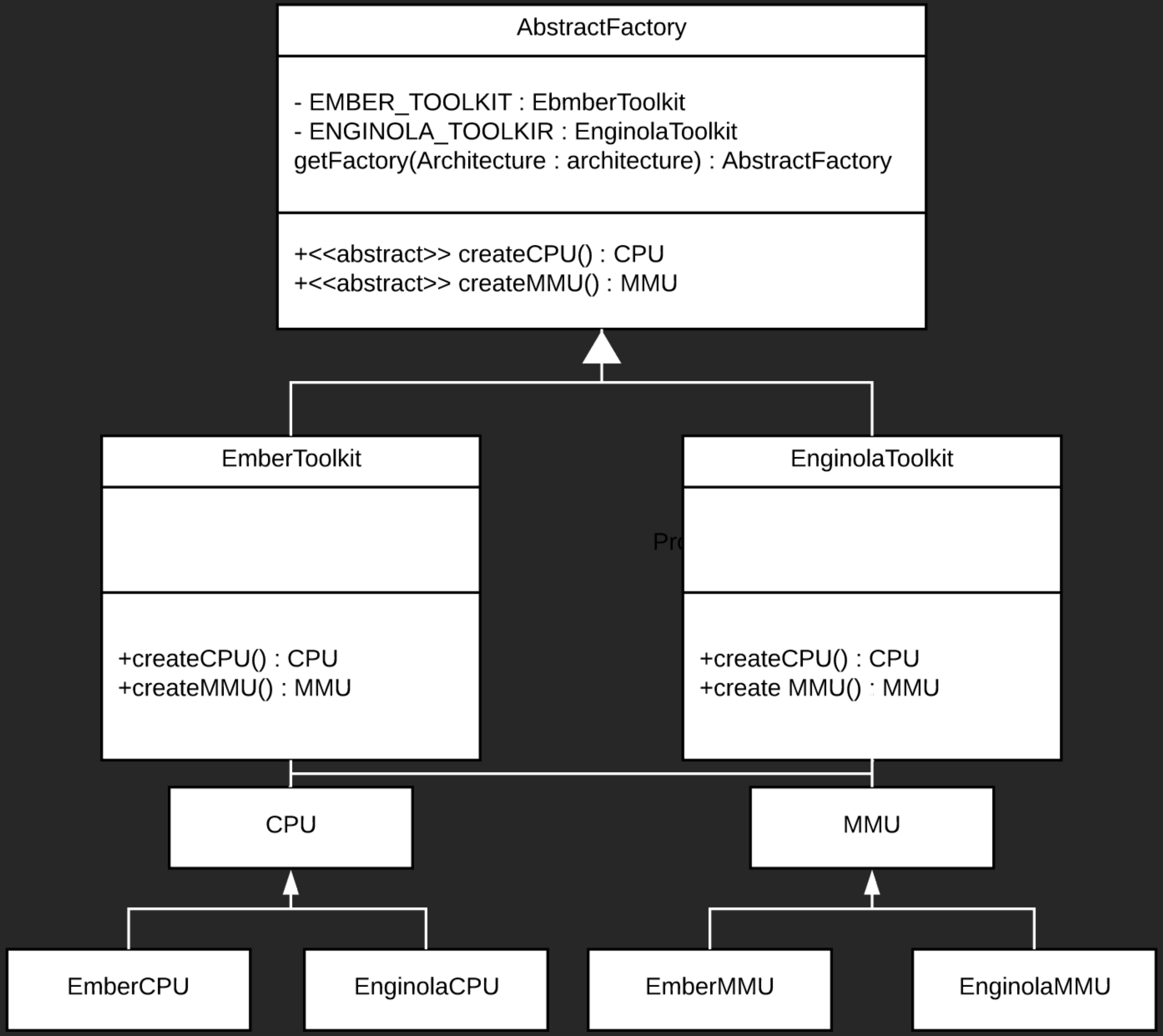
```
return new ProductTwoPlatformTwo();
```

Stolen from [SHVETS]

Can’t it be simplier?....

Let’s see an example!!!

Example!

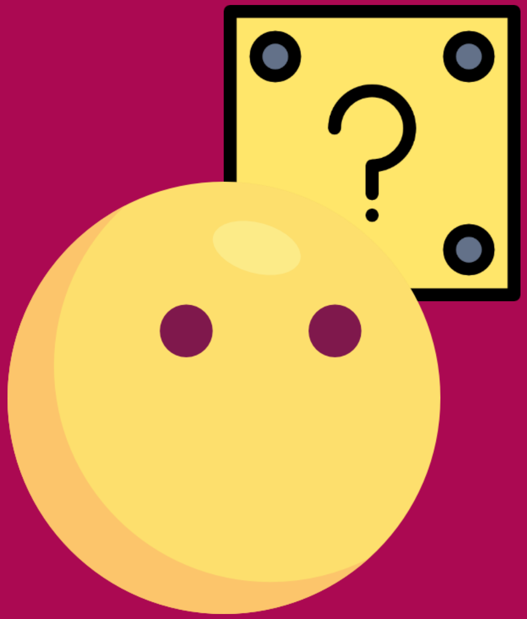

AbstractFactory
- EMBER_TOOLKIT : EbmberToolkit
- ENGINOLA_TOOLKIR : EnginolaToolkit
getFactory(Architecture : architecture) : AbstractFactory
+<<abstract>> createCPU() : CPU
+<<abstract>> createMMU() : MMU
EmberToolkit
+createCPU() : CPU
+createMMU() : MMU
EnginolaToolkit
+createCPU() : CPU
+create MMU() : MMU
Pr
CPU
MMU
EmberCPU
EnginolaCPU
EmberMMU
EnginolaMMU

Factory Object

It only appears once in the implementation. Thus, it can be instantiated as a Singleton

Take a look at the board!!

Singleton?



Singleton

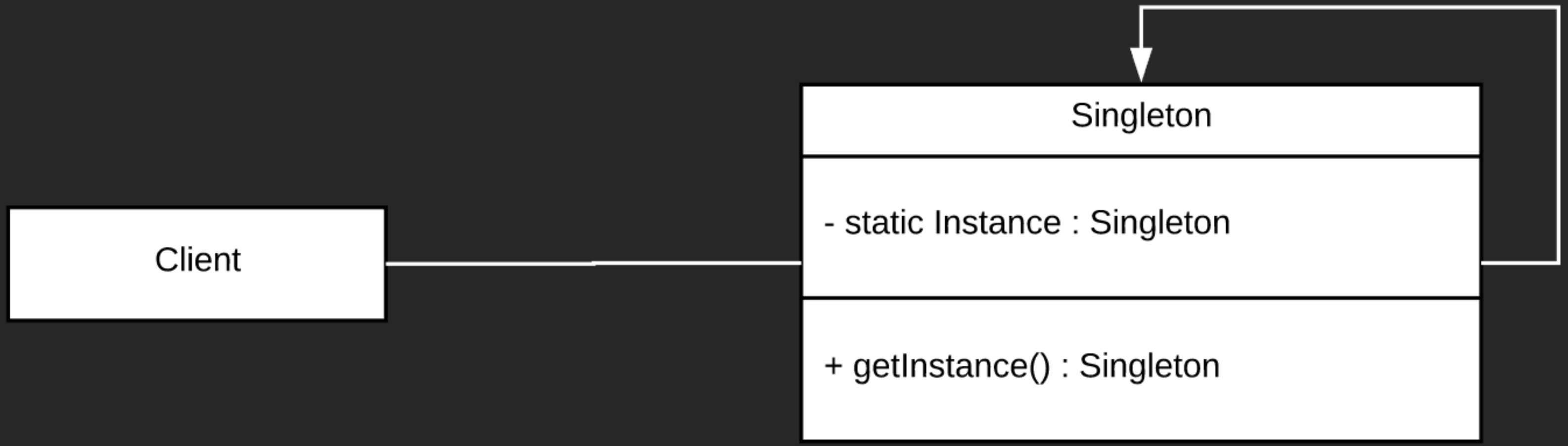
Problem

- Application needs one and only one instance of an object!
- Don’t use it to replace global variables!!!

Singleton

Solution!

- Create the class with only one private static attribute and one access method!



From code:

```
public class Singleton {
    private Singleton() {}

    private static class SingletonHolder {
        private static final Singleton INSTANCE = new Singleton();
    }

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

Builder!

Builder

Problem!

- An application needs to create the elements of a complex aggregate.

Builder

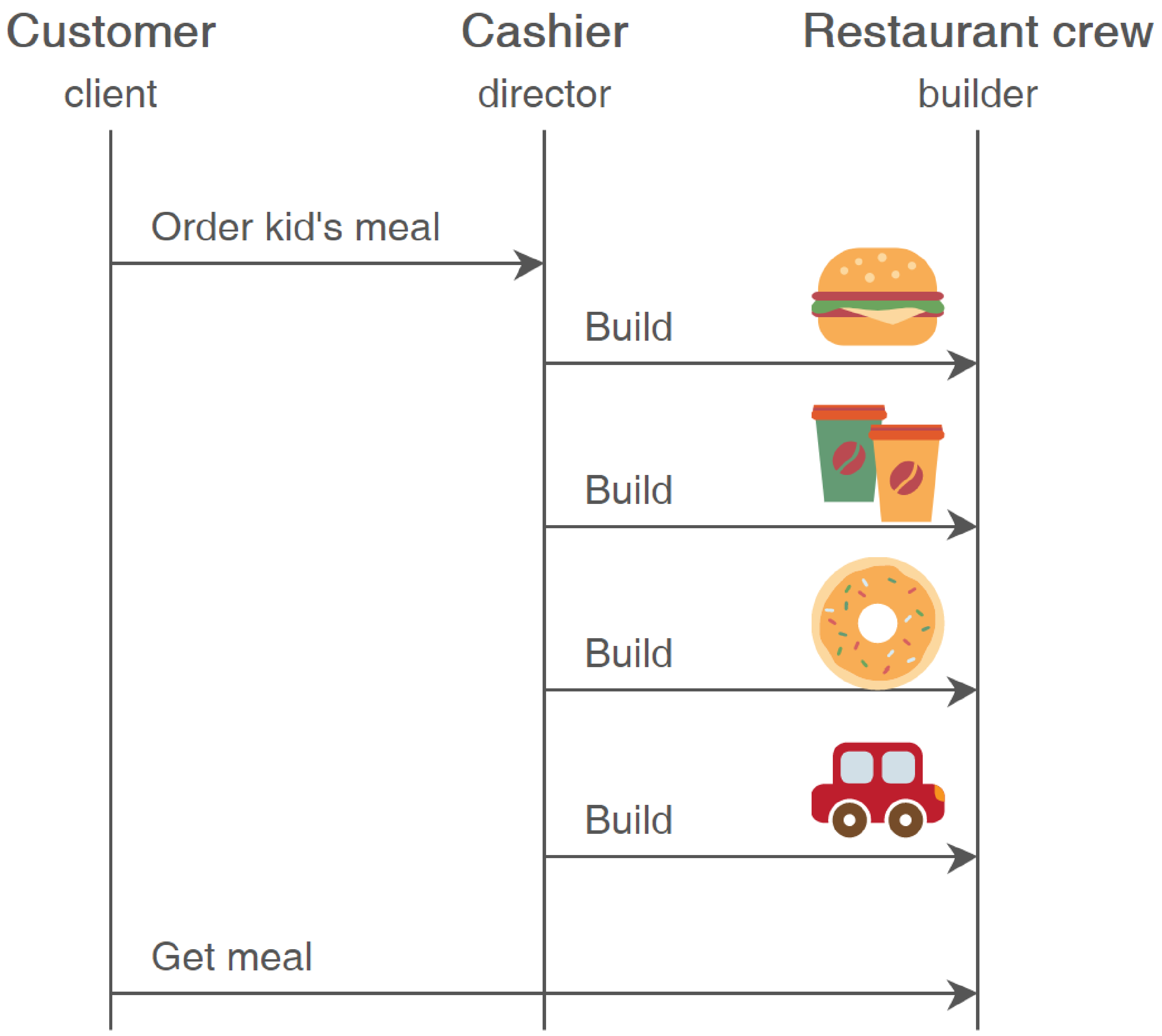
Solution

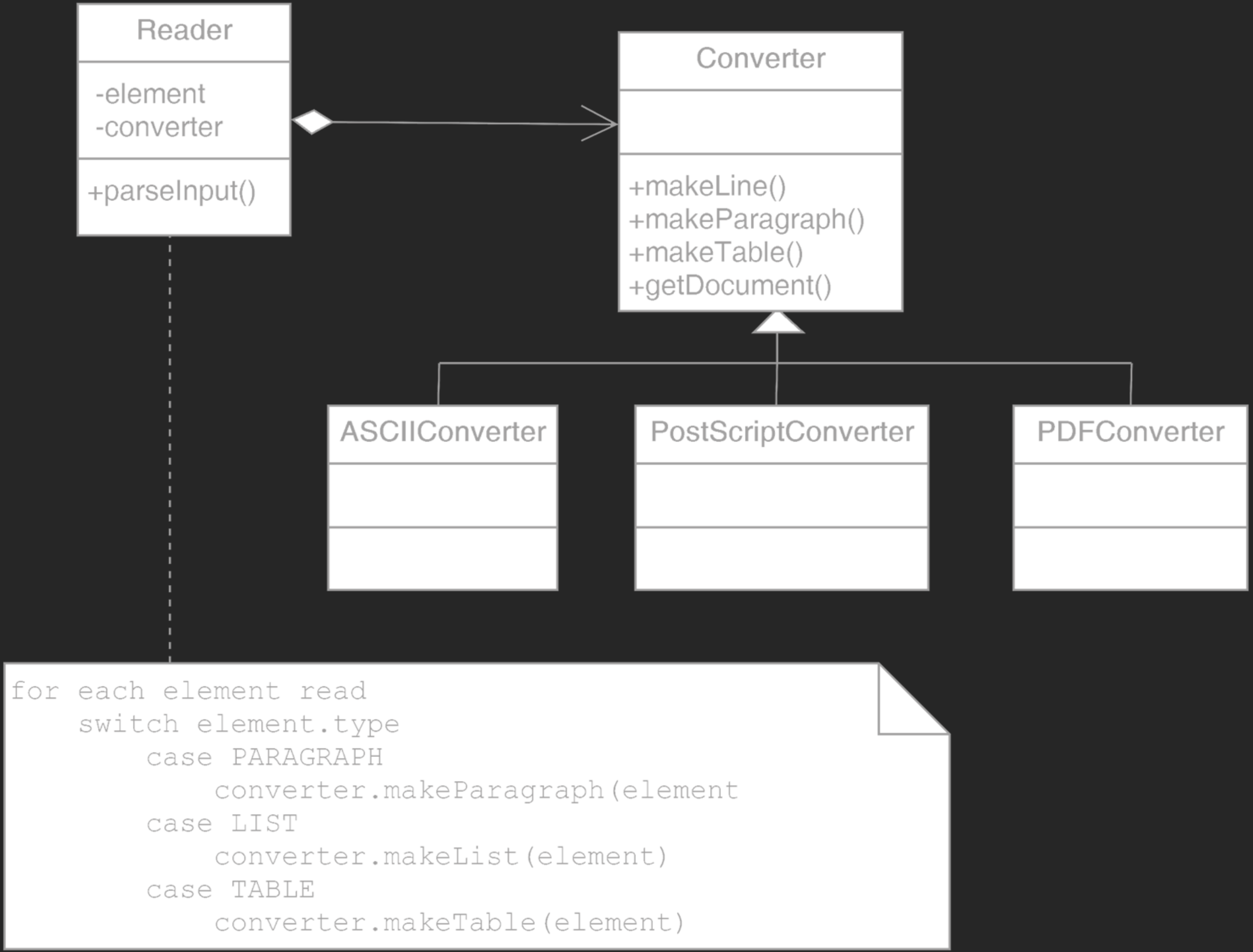
- Separate the construction of a simple object from its representation.

- The construction can create different representations.

Example

This real life situation has a builder pattern!





```
for each element read
    switch element.type
        case PARAGRAPH
            converter.makeParagraph(element
        case LIST
            converter.makeList(element)
        case TABLE
            converter.makeTable(element)
```

Adapter

Adapter

Problem!

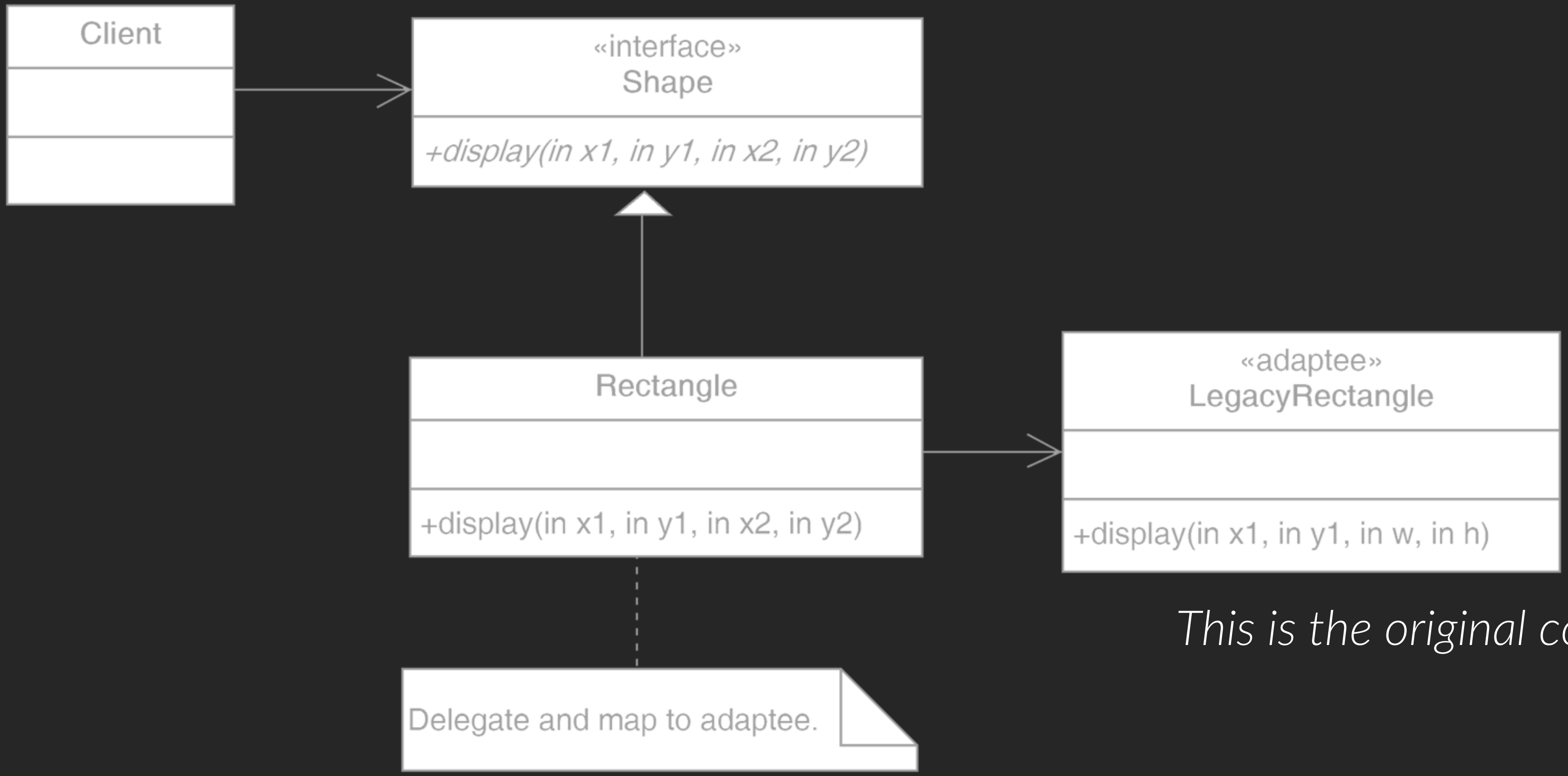
- The usage of components is normally mandatory, but their ‘knowledge of the world’ is not compatible with the system currently being developed.

Adapter

Solution

- Adapt or transform inputs in order to reuse components!

Example


Client
«interface»
Shape
+display(in x1, in y1, in x2, in y2)
Rectangle
+display(in x1, in y1, in x2, in y2)
«adaptee»
LegacyRectangle
+display(in x1, in y1, in w, in h)
Delegate and map to adaptee.

This is the original component!!

Private Class Data

Private Class Data

Problem!

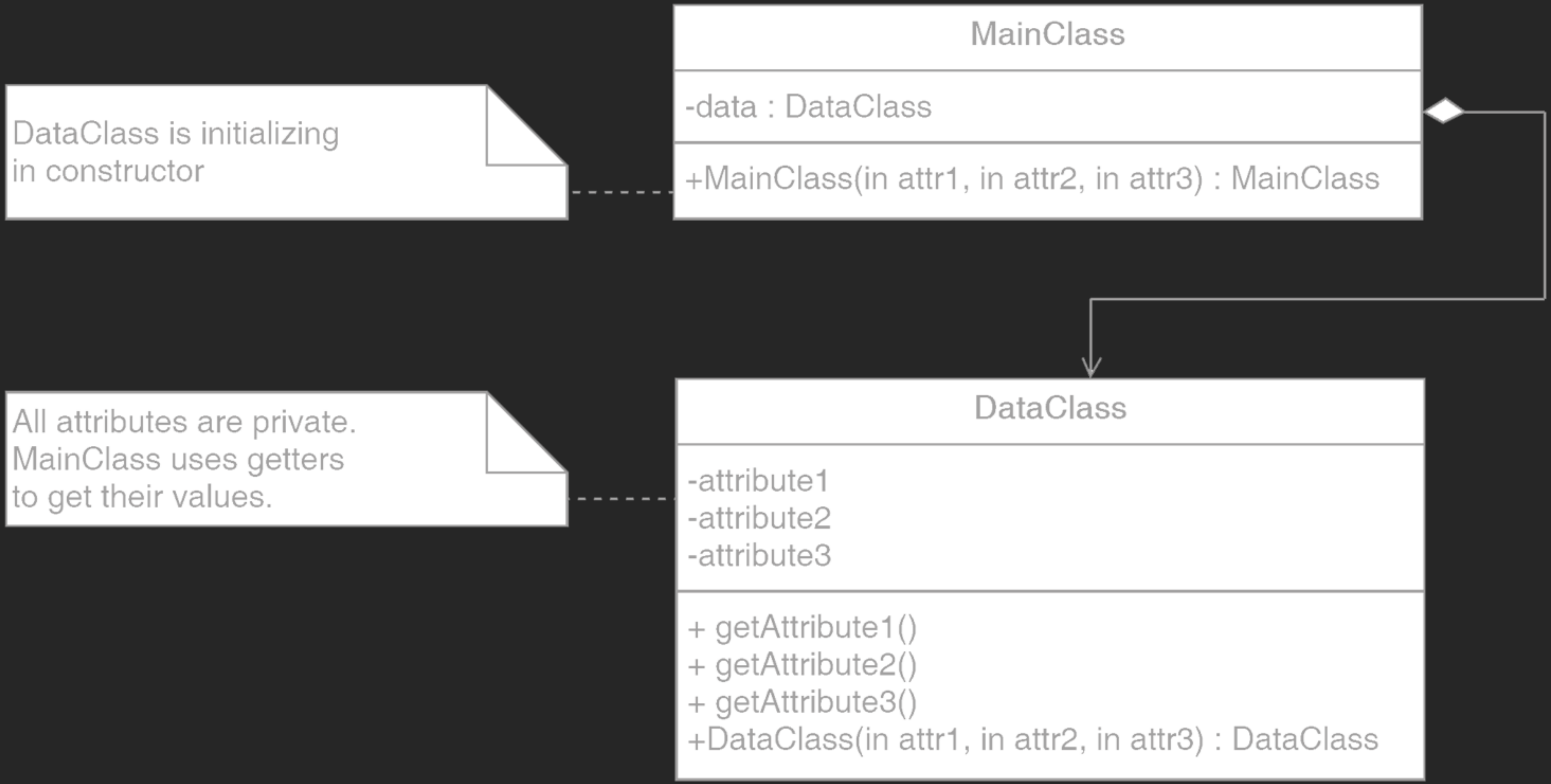
- A class may expose its attributes after construction and sometimes it’s not desirable to enable that manipulation.

Private Class Data

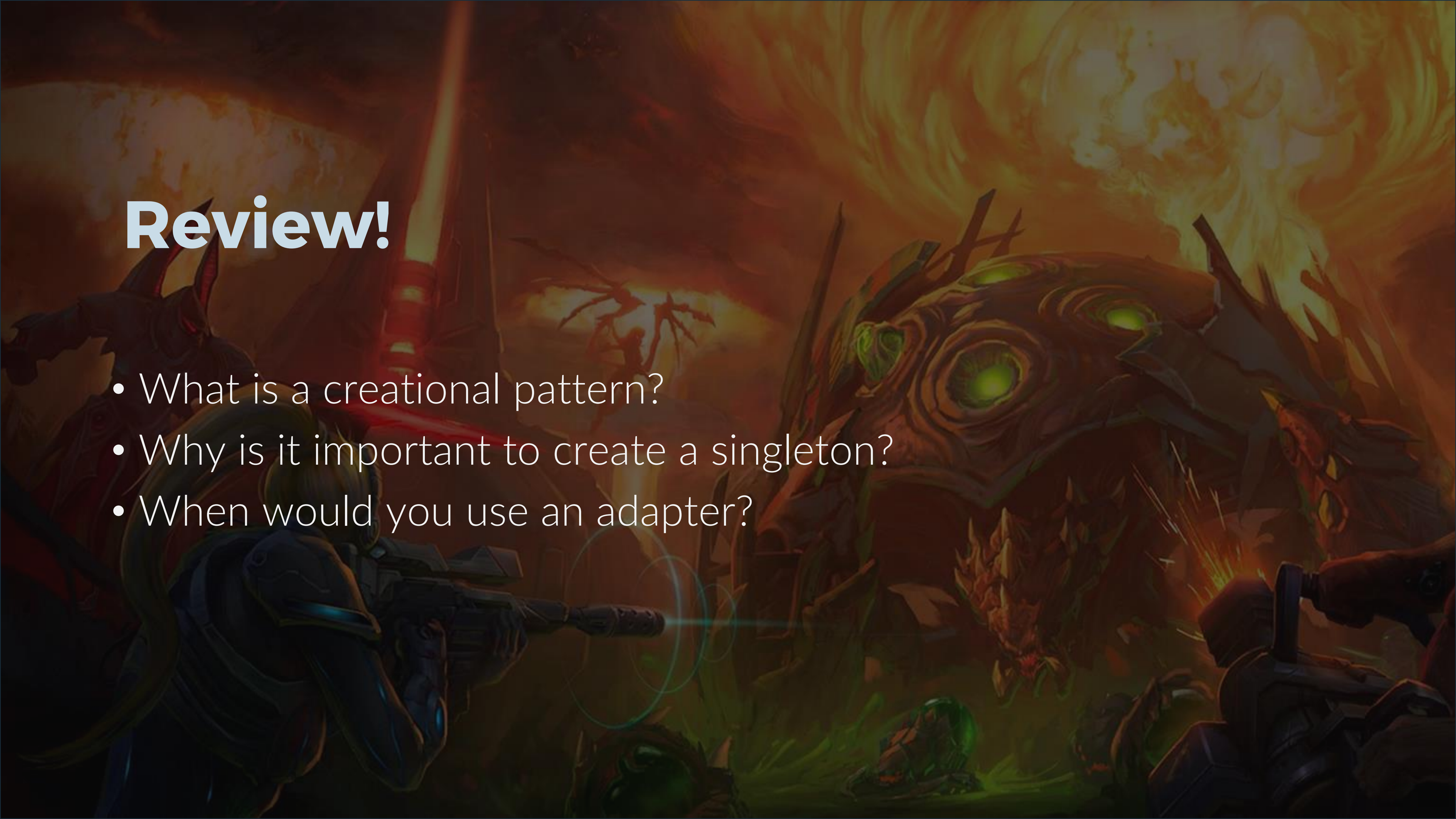
Solution!

- Encapsulate the data on a main class

Example


DataClass is initializing in constructor
MainClass
-data : DataClass
+MainClass(in attr1, in attr2, in attr3) : MainClass
All attributes are private. MainClass uses getters to get their values.
DataClass
-attribute1
-attribute2
-attribute3
+ getAttribute1()
+ getAttribute2()
+ getAttribute3()
+DataClass(in attr1, in attr2, in attr3) : DataClass

One Shot Review



Review!

- What is a creational pattern?
- Why is it important to create a singleton?
- When would you use an adapter?

Observer

Observer

Problem!

- Static designs tend not to scale properly when dependent components get dynamic.

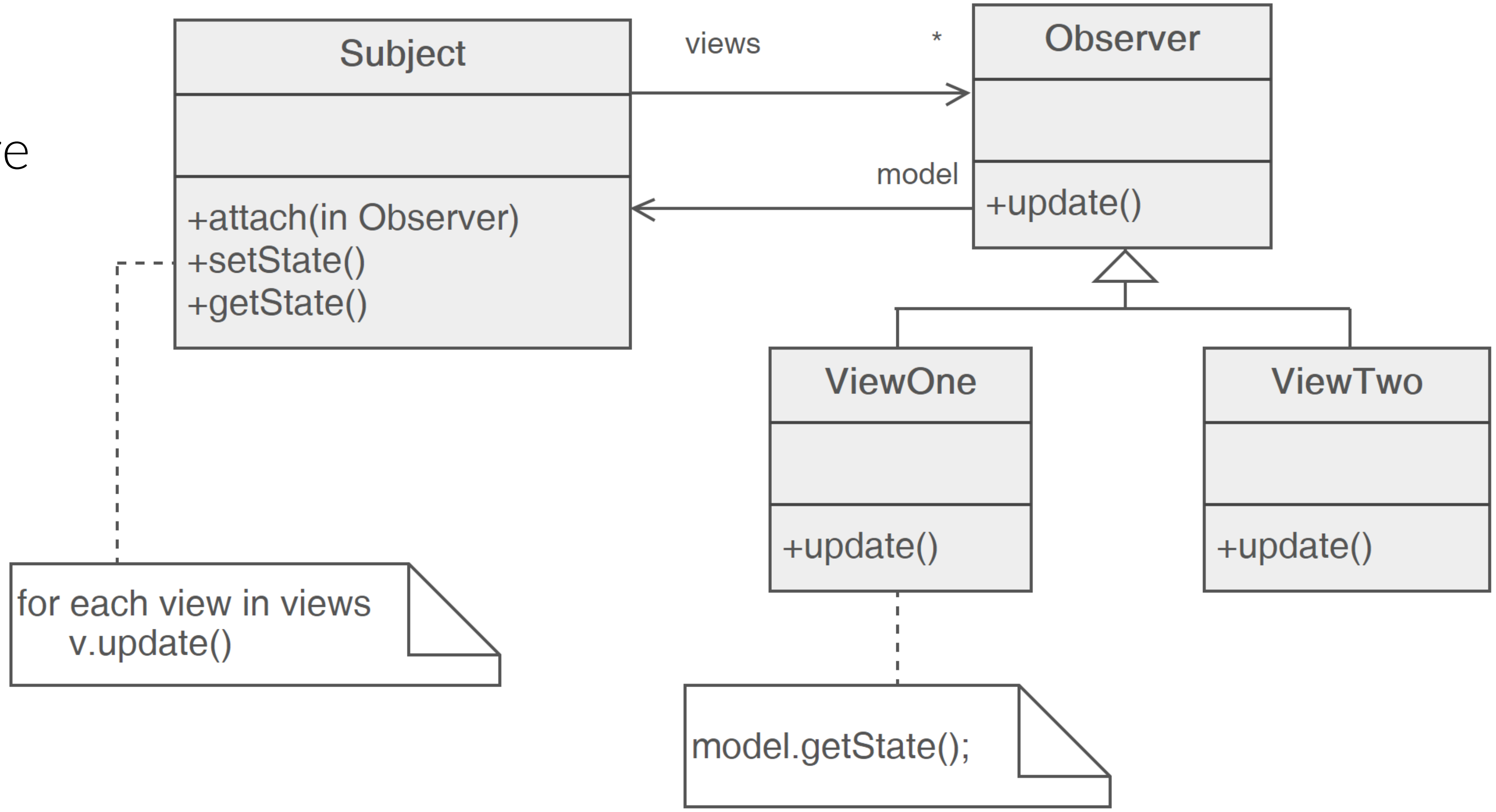
Observer

Solution!

- Define a one to many dependency that changes all views dynamically.
- Encapsulate core in a subject abstraction and variable components in an hierarchy

Example!

See how views are now more dynamic?



Decorator

Decorator

Problem!

- Add behavior or state in run-time as inheritance may not be feasible due to its static behavior.

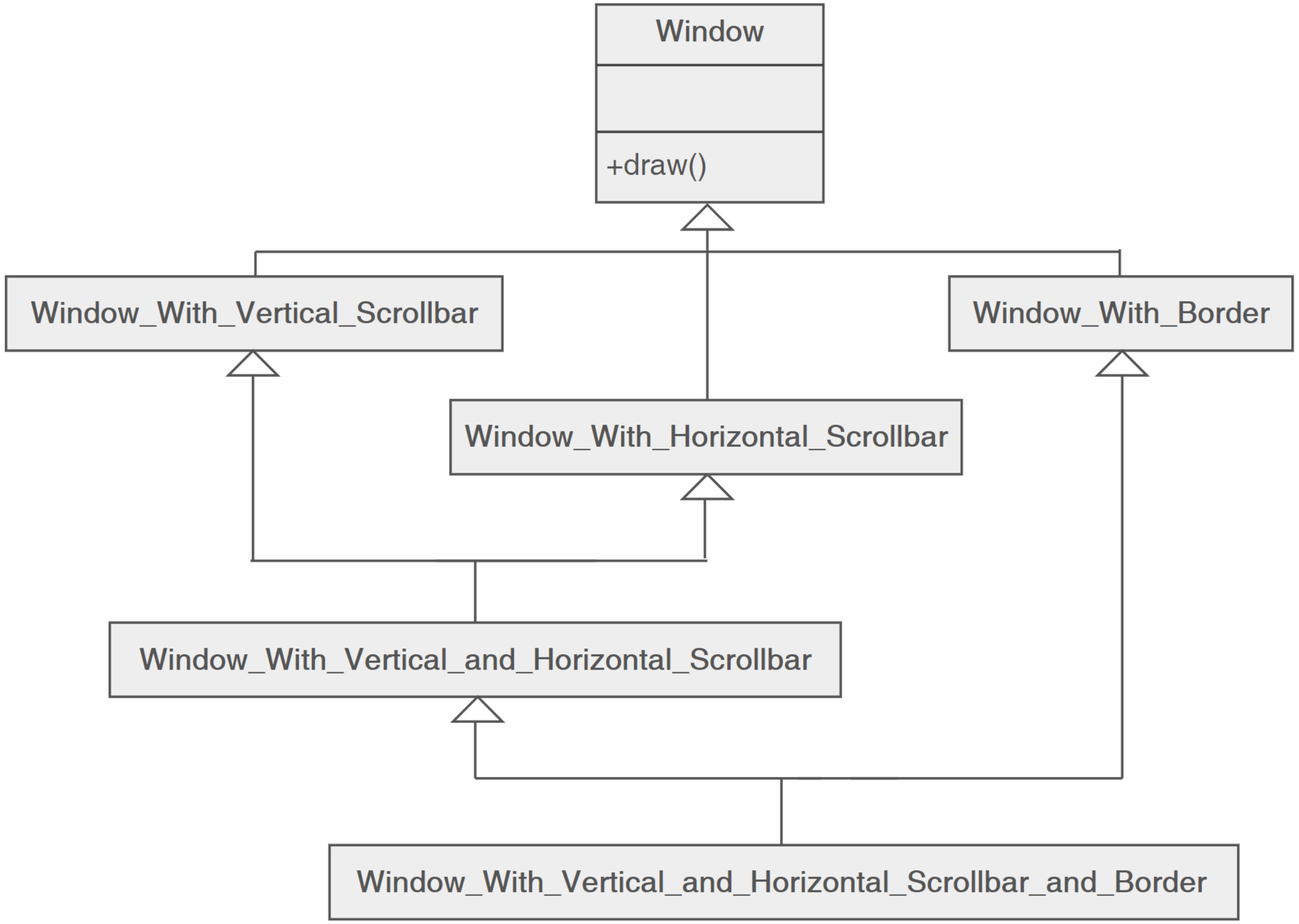
Decorator

Solution!

- Attach new responsibilities dynamically to a core object.
- Recursively wrapping of objects!

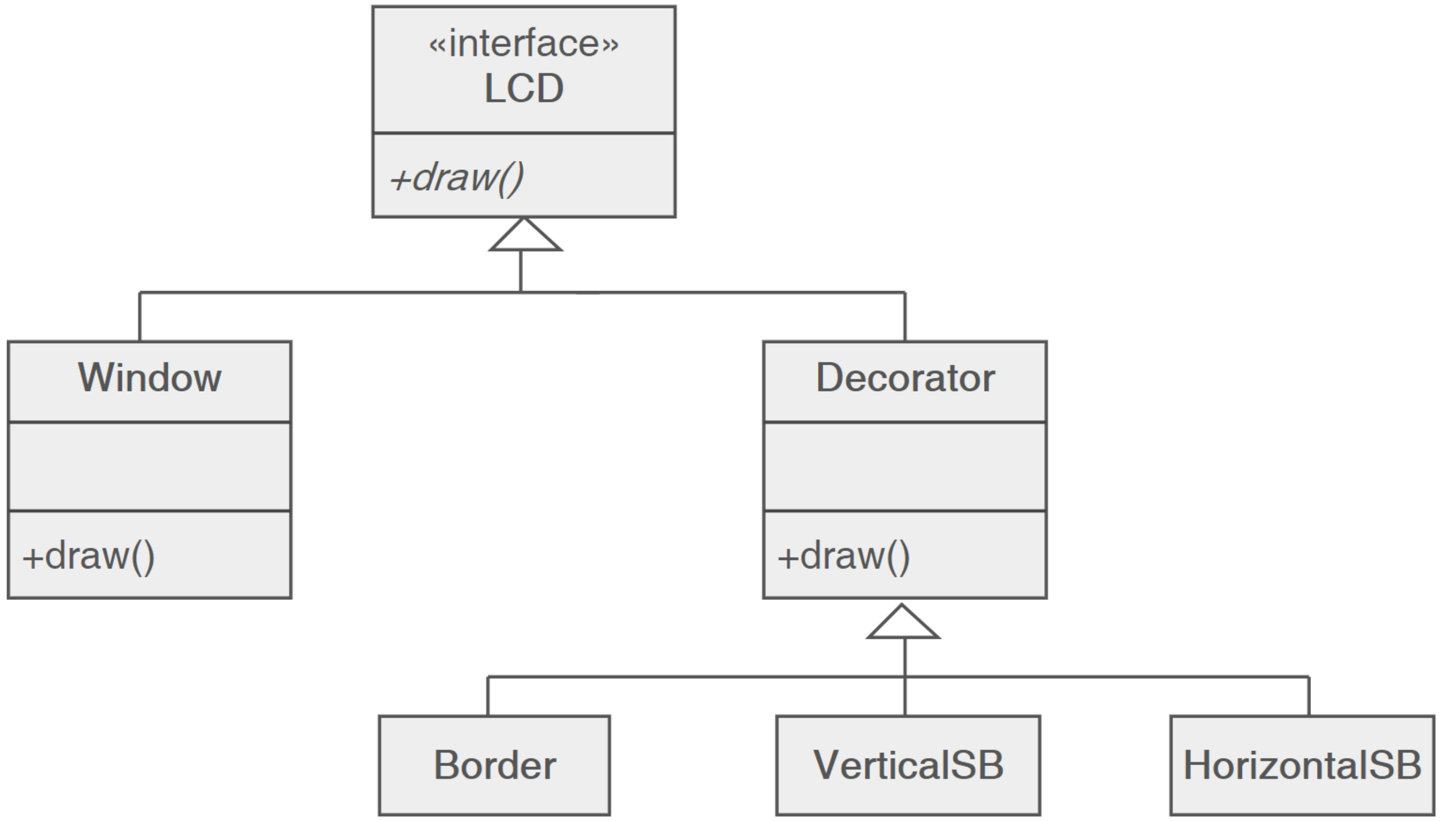
Example!

This design encapsulates all possible options before runtime...

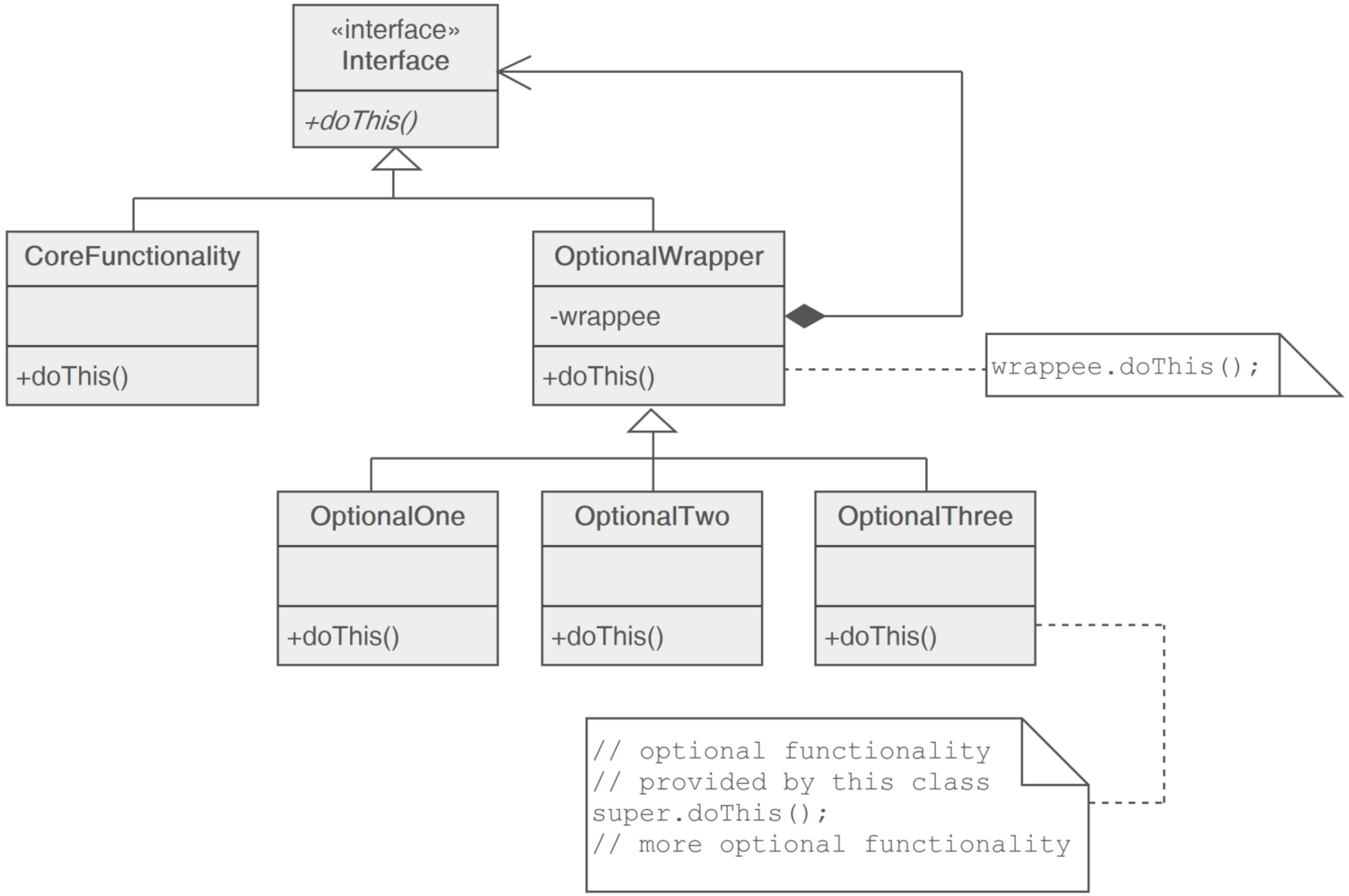

Window
+draw()
Window_With_Vertical_Scrollbar
Window_With_Border
Window_With_Horizontal_Scrollbar
Window_With_Vertical_and_Horizontal_Scrollbar
Window_With_Vertical_and_Horizontal_Scrollbar_and_Border

Example B

This one, on the other hand is more flexible with the client!



Generic View



```
wrappee.doThis();
```

```
// optional functionality
// provided by this class
super.doThis();
// more optional functionality
```

From code:

```
Widget* aWidget = new BorderDecoratorWidget* aWidget = new
BorderDecorator(
  new HorizontalScrollBarDecorator(
    new VerticalScrollBarDecorator(
      new Window( 80, 24 ))));
aWidget->draw();
```

Null Object

Null Object

Problem!

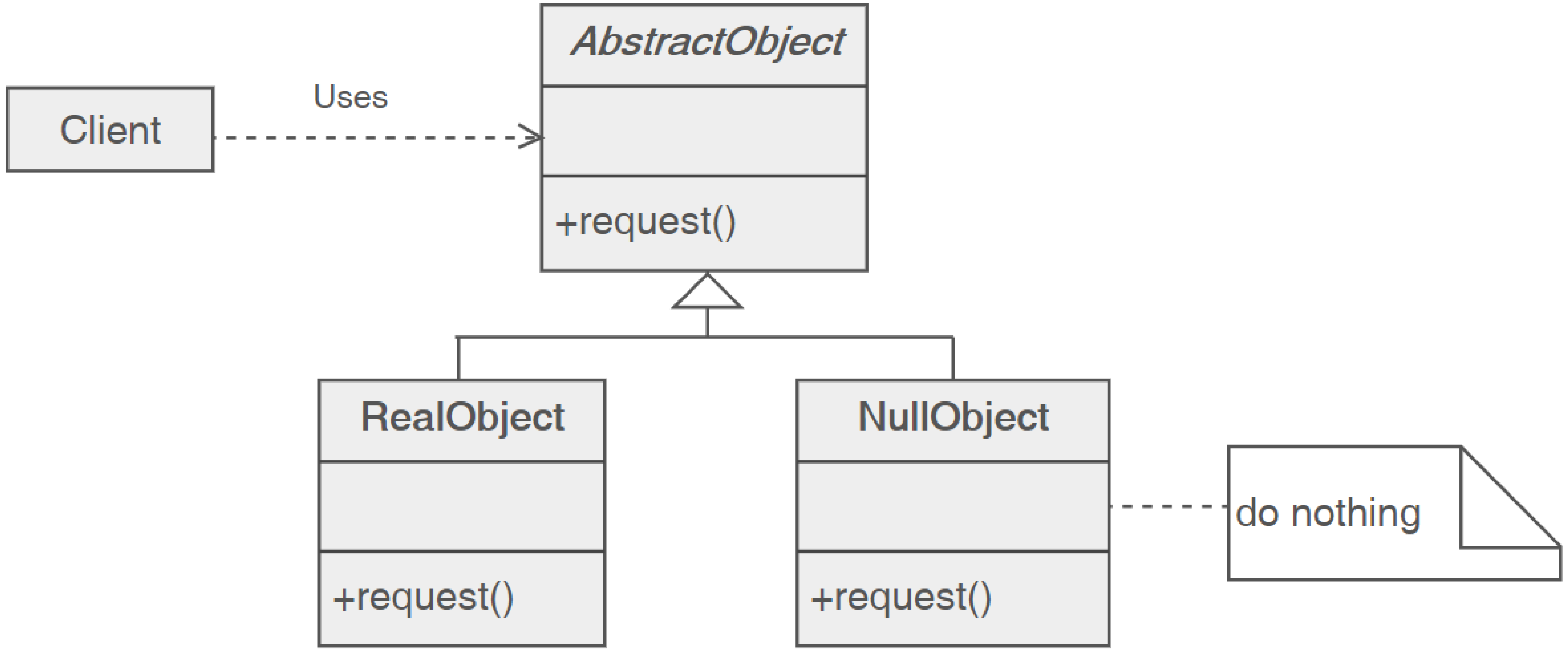
- Sometimes you want to encapsulate a null representation of an object instead of letting the client deal with those ‘null’s’

Null Object

Solution

- Provide a null representation then!

Generic View



References

- [LARMAN] Applying UML and Patterns – Craig Larman
- [SM] Design Patterns Explained Simply - SourceMaking

Class has died… for today!