Design Patterns

Object Oriented Programming week 10

Ruben Holubek

Who am I?

- Ruben Holubek
- 2nd year Master Student Software Science
- Research related to teaching programming
- Right now: how to effectively teach OO programming, especially design patterns

Online Quiz

- Please join the online quiz for exercises:
- Please go to www.socrative.com
- Click on login (upper right corner)
- Click on Student login
- Insert room name HOLUBEK60

Question 1

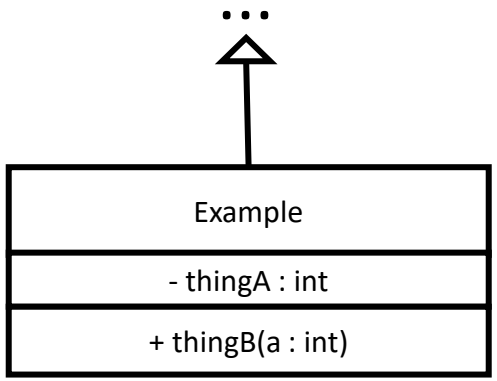
Which code fragment is represented by this UML?

A.

```
public class Example {
    int thingA;

    void thingB (int a) {
        // Some code
    }
}
```

B.

```
public class Example {
    int thingB;

    void thingA (int a) {
        // Some code
    }
}
```



C.

```
public class Example {

    void thingB (int a) {
        // Some code
    }
}
```

D.

```
public class Example {
    int thingA;

    void thingB (boolean a) {
        // Some code
    }
}
```

Question 1

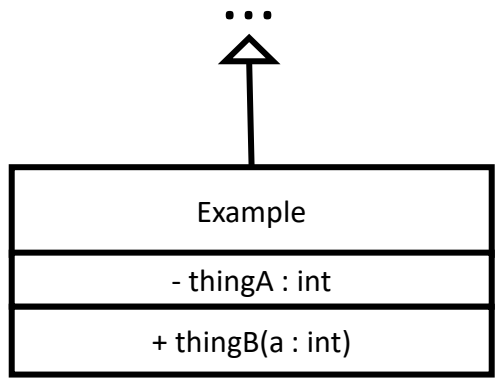
A is correct; B switched around the names for the attribute and function, the attribute thingA is missing in C and the type for thingB is incorrect in D

A.

```
public class Example {
    int thingA;

    void thingB (int a) {
        // Some code
    }
}
```

B.

```
public class Example {
    int thingB;

    void thingA (int a) {
        // Some code
    }
}
```



C.

```
public class Example {

    void thingB (int a) {
        // Some code
    }
}
```

D.

```
public class Example {
    int thingA;

    void thingB (boolean a) {
        // Some code
    }
}
```

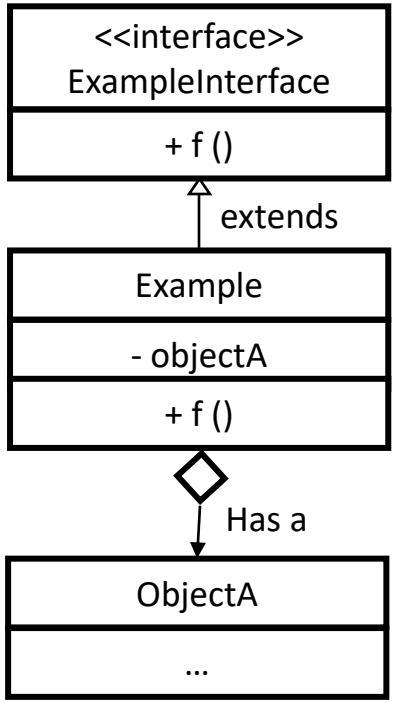
Question 2

Which UML represents this code fragment?

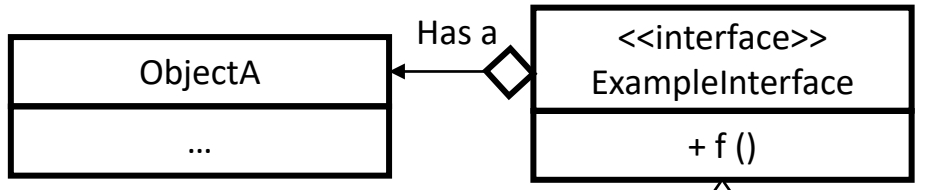
```java
public interface ExampleInterface {
    public void f();
}

public class Example implements ExampleInterface {
    ObjectA x;

    @Override
    public void f () {
        // Some code
    }
}
```

```java
public class ObjectA {
    // different attributes

    // different functions
}
```

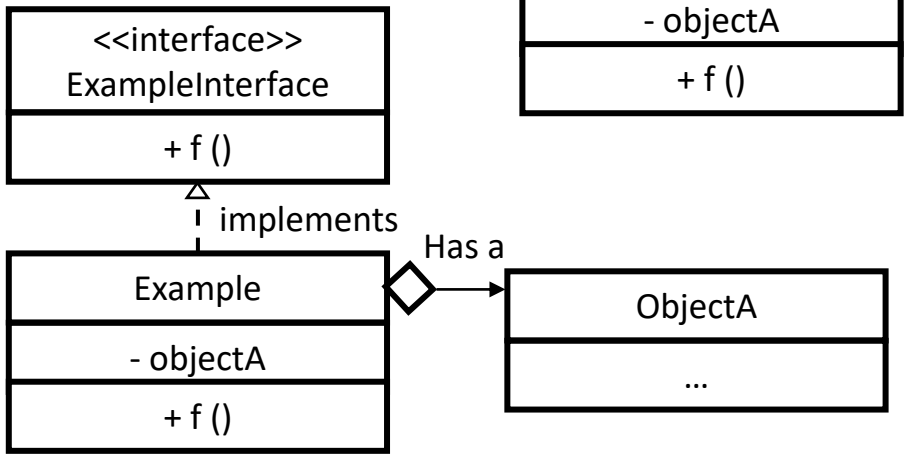
A.



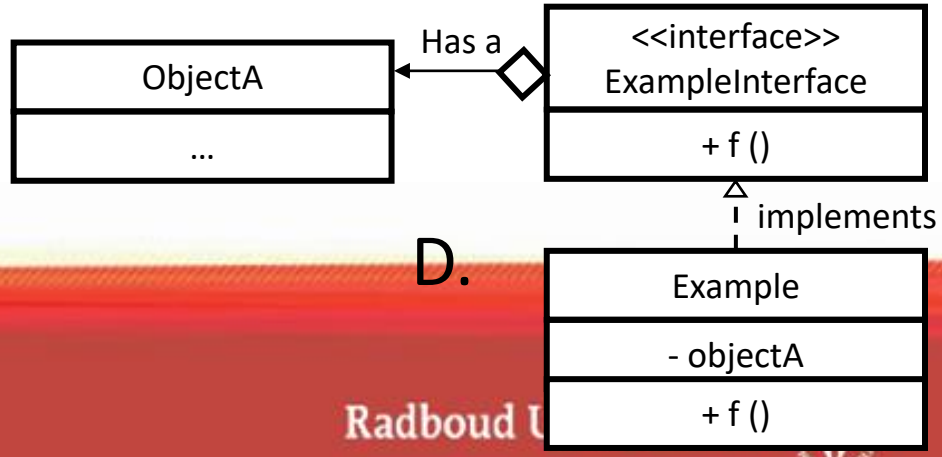
B.



C.



D.



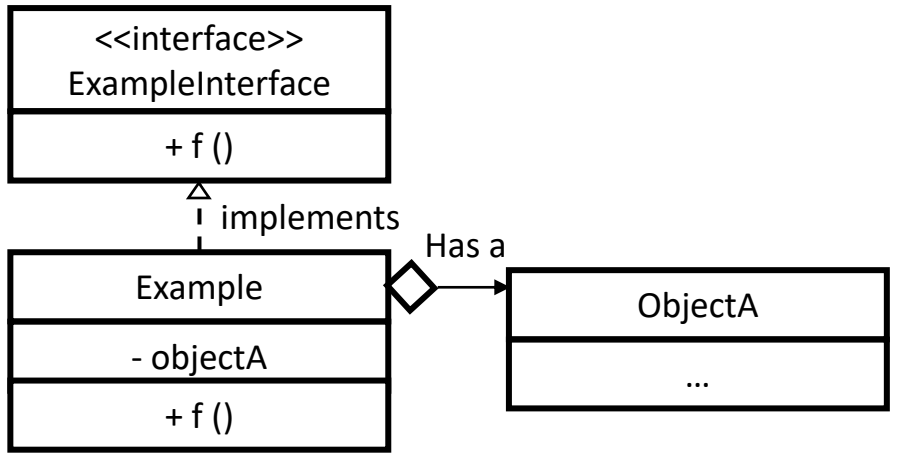
Question 2

C is correct, as Example implements ExampleInterface (it doesn't extend it) and Example uses ObjectA (ExampleInterface does not)

```java
public interface ExampleInterface {
    public void f();
}

public class Example implements ExampleInterface {
    ObjectA x;

    @Override
    public void f () {
        // Some code
    }
}
```

```java
public class ObjectA {
    // different attributes

    // different functions
}
```

C.



Question 3

Which of the following programs have a similar structure? (e.g. containing a loop, using similar interfaces etc)

1. Getting the last names of a list of students
2. Evaluating the value of a given boolean formula (e.g. T /\ F -> T)
3. For the numbers between 1 and 100, calculate the square root

A. 1 and 2 have a similar structure
B. 1 and 3 have a similar structure
C. 2 and 3 have a similar structure
D. None of these programs have a similar structure

Question 3

1. Getting all the last names of a list of students
2. Evaluating the value of a given boolean formula (e.g. T /\ F -> T)
3. For the numbers between 1 and 100, calculate the square root

A. 1 and 2 have a similar structure

B. 1 and 3 have a similar structure

C. 2 and 3 have a similar structure

D. None of these programs have a similar structure

Both of these formulas iterate over something (list of students or numbers) and perform an action on each element (return the name or calculate square root). Program 2 doesn't perform such an iterating operation.

Question 4

Which of the following objects have a similar structure?

1. Many different bikes with an option to add a specific, special saddle

2. A pizza with many options for different toppings, e.g. tomato, cheese, chicken etc

3. A server where users can have a combination of different rights, e.g. reading, writing, editing, adding new users etc

1. 1 and 2 have a similar structure

A. 1 and 3 have a similar structure

B. 2 and 3 have a similar structure

C. None of these programs have a similar structure

Question 4

1. Many different bikes with an option to add a specific, special saddle
2. A pizza with many options for different toppings, e.g. tomato, cheese, chicken etc
3. A server where users can have a combination of different rights, e.g. reading, writing, editing, adding new users etc

A. 1 and 2 have a similar structure

B. 1 and 3 have a similar structure

C. 2 and 3 have a similar structure

D. None of these programs have a similar structure

Both these programs have one base object (pizza or user), which can have multiple "extras" (toppings or rights). Program 1 has different base objects (bikes), but with only one "extra" (special saddle), so this is a simple attribute

Design patterns

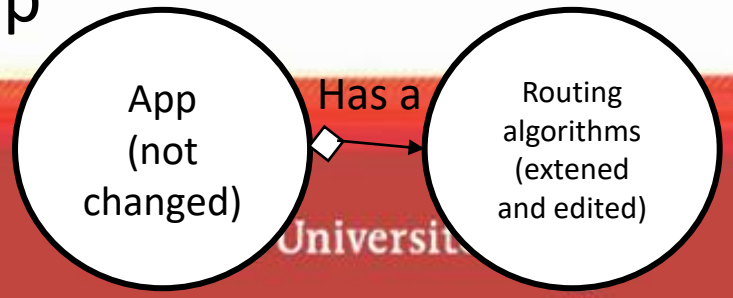
Design Patterns

- Design patterns are specific structures that reoccur in different programs
  - Iterating over a loop and performing a specific action
  - Having a base object with many different possibilities of extras
  - And many more…
- You probably used a lot of these without even thinking about it
- Now we simply give them a name
- This lecture: we discuss 3 different design patterns
  - Strategy pattern
  - Decorator pattern
  - Visitor pattern

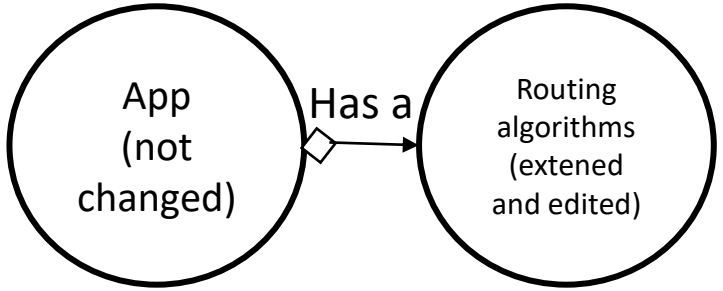
Strategy Pattern

Problem

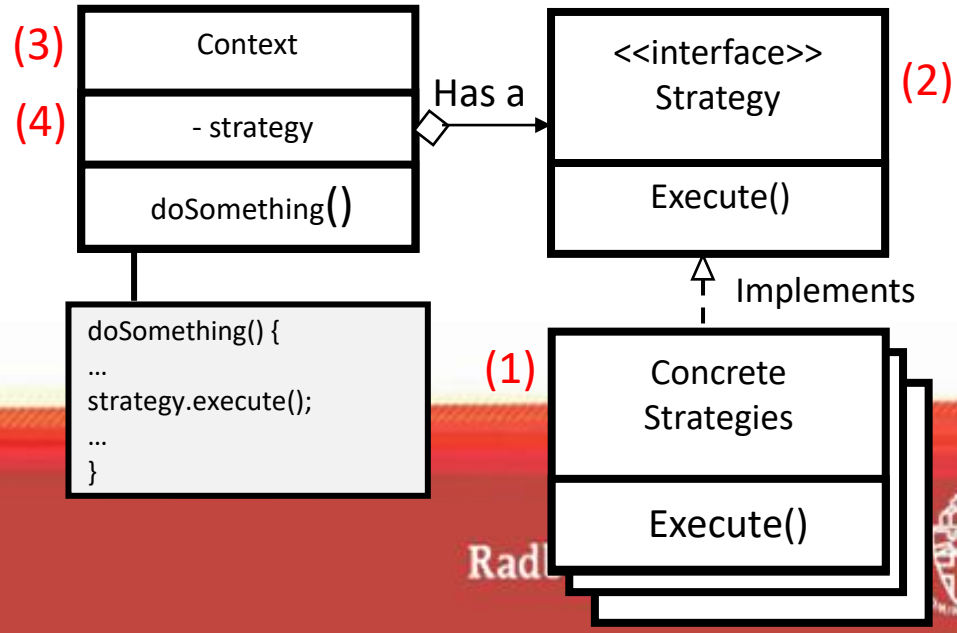
- Navigation app that provides the fastest route for 2 given points
- First version: provides routes traveling by car
- Later versions: algorithms for other routes
  - Biking, walking, busses, along highlights, shortest route etc.
- Maintaining this without a proper structure is impossible
  - Code gets messy
  - Altering the code for the whole navigation app,
  while only changing the routing algorithms
- Idea: Split the routing algorithms from the whole app
- Solution: Strategy Pattern



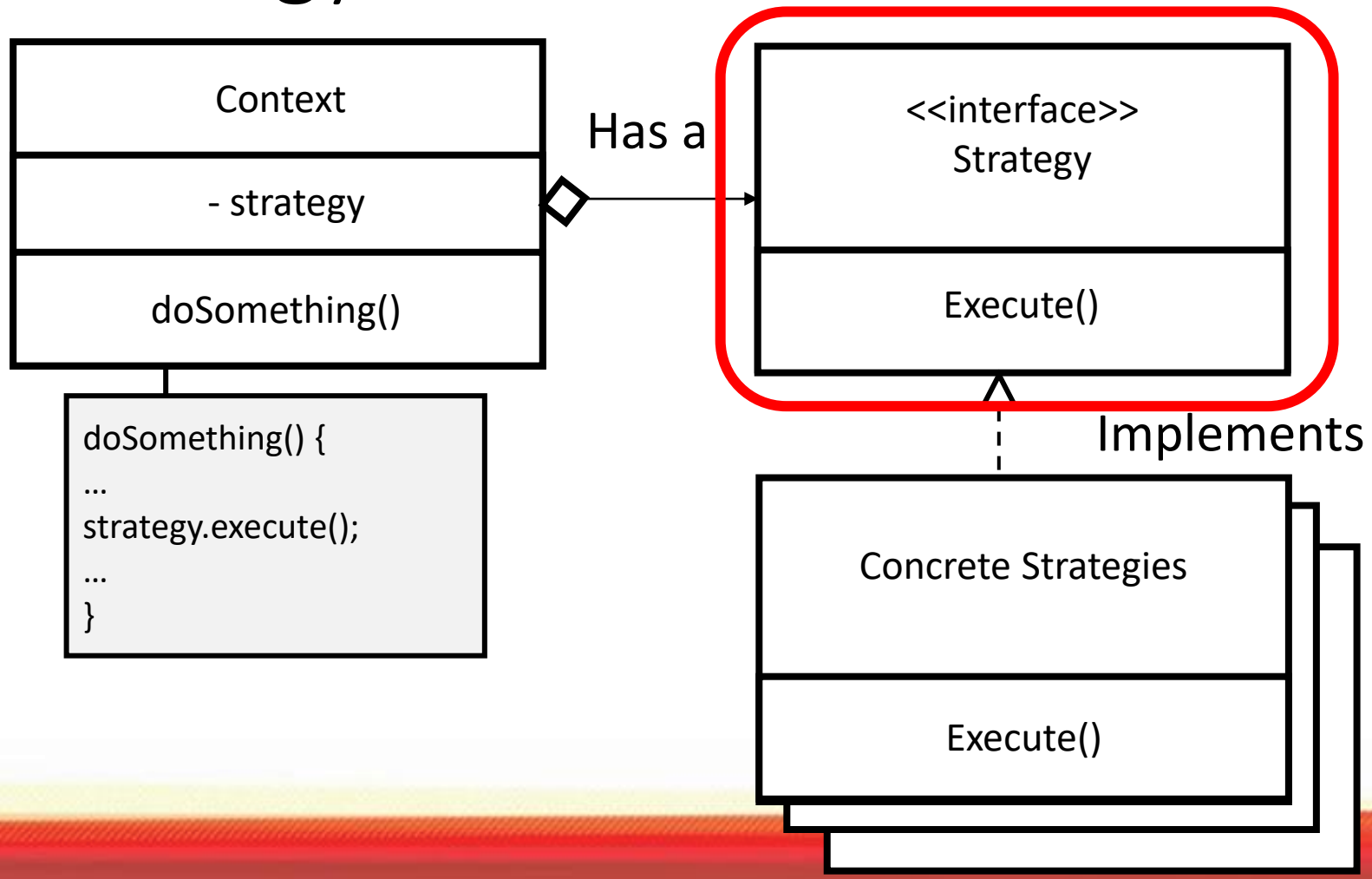
Strategy Pattern



- Previously: Split the routing algorithms from the whole app
- Strategy Pattern: Split the strategies from the context
- Applied with “a class that does something specific in a lot of ways”
- The concrete strategies (1) are all “stored” in the interface Strategy (2)
- The context (3) has a Strategy attribute (4)
- The context uses this reference as a “generic interface”
- The actual strategies are completely split from the context!



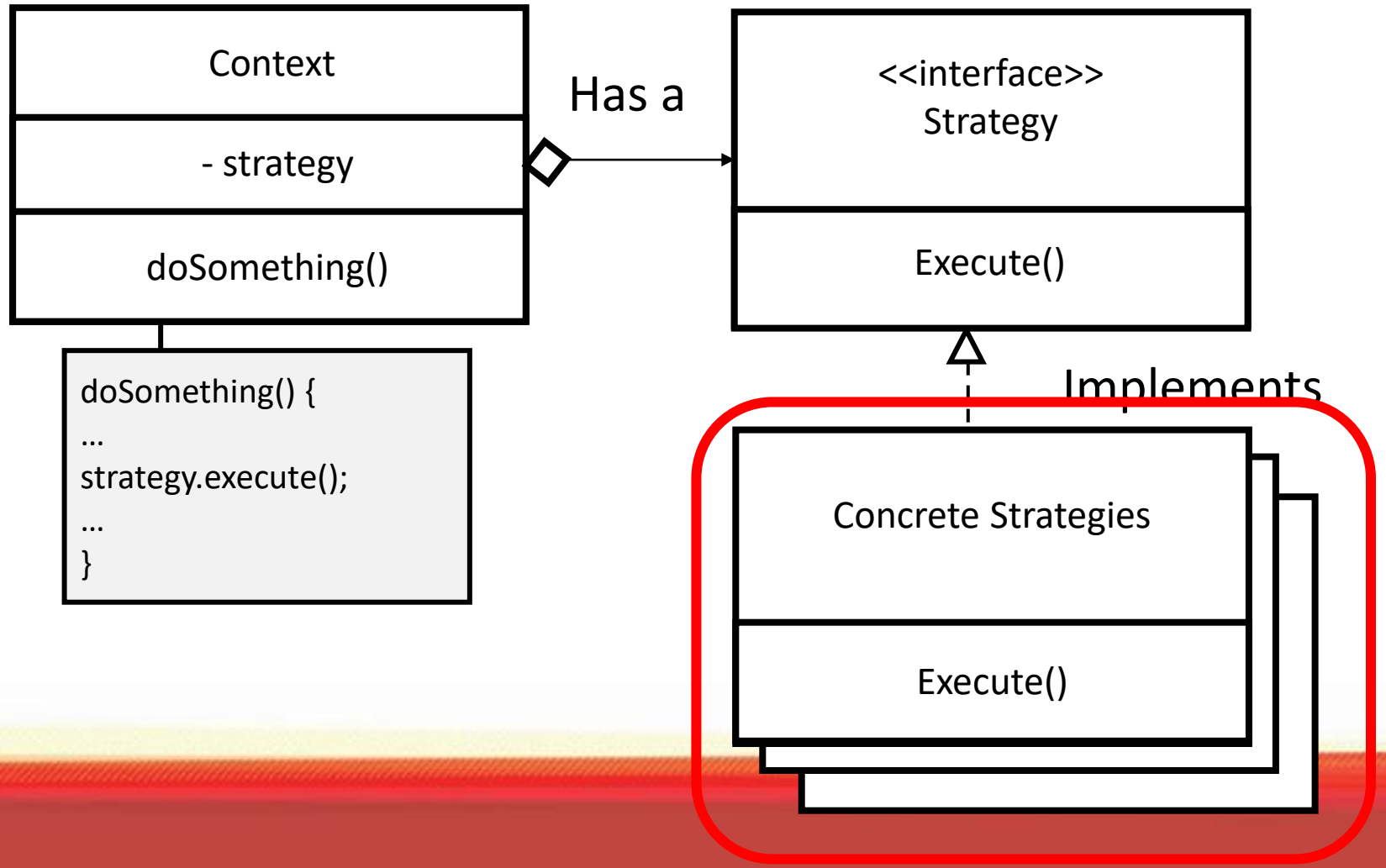
Strategy Pattern UML



Interface Strategy

- The interface for the implemented strategies
- It has an abstract function execute() which every concrete strategy should implement

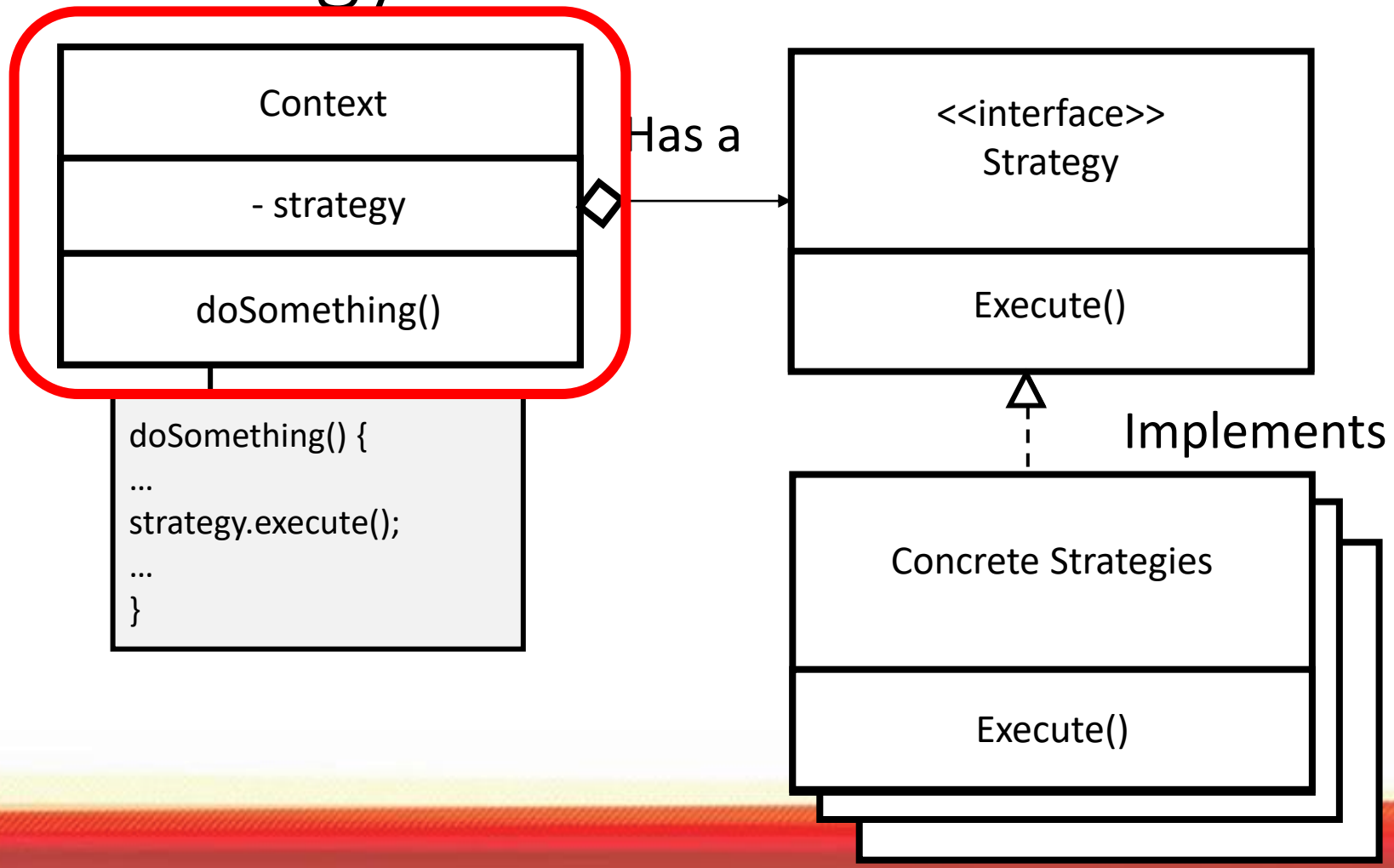
Strategy Pattern UML



The concrete strategies

- These implement the Strategy interface
- So every strategy has an implemented Execute function

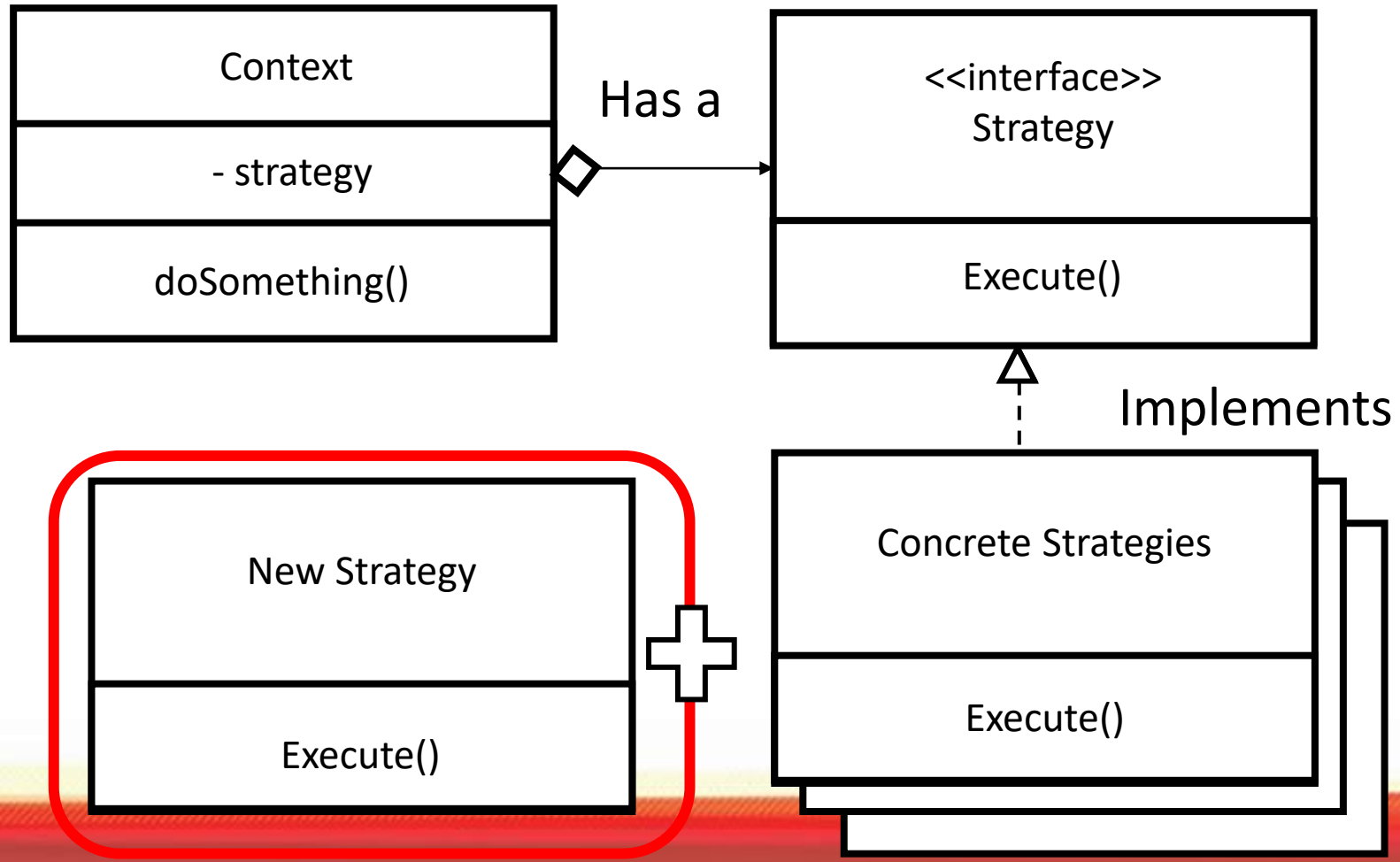
Strategy Pattern UML



Context

- The context has a reference to the used strategy
- In the functionality of Context, there is a function doSomething() that uses the stored strategy
- E.g. strategy.execute()
- Context is implemented with an abstract strategy!

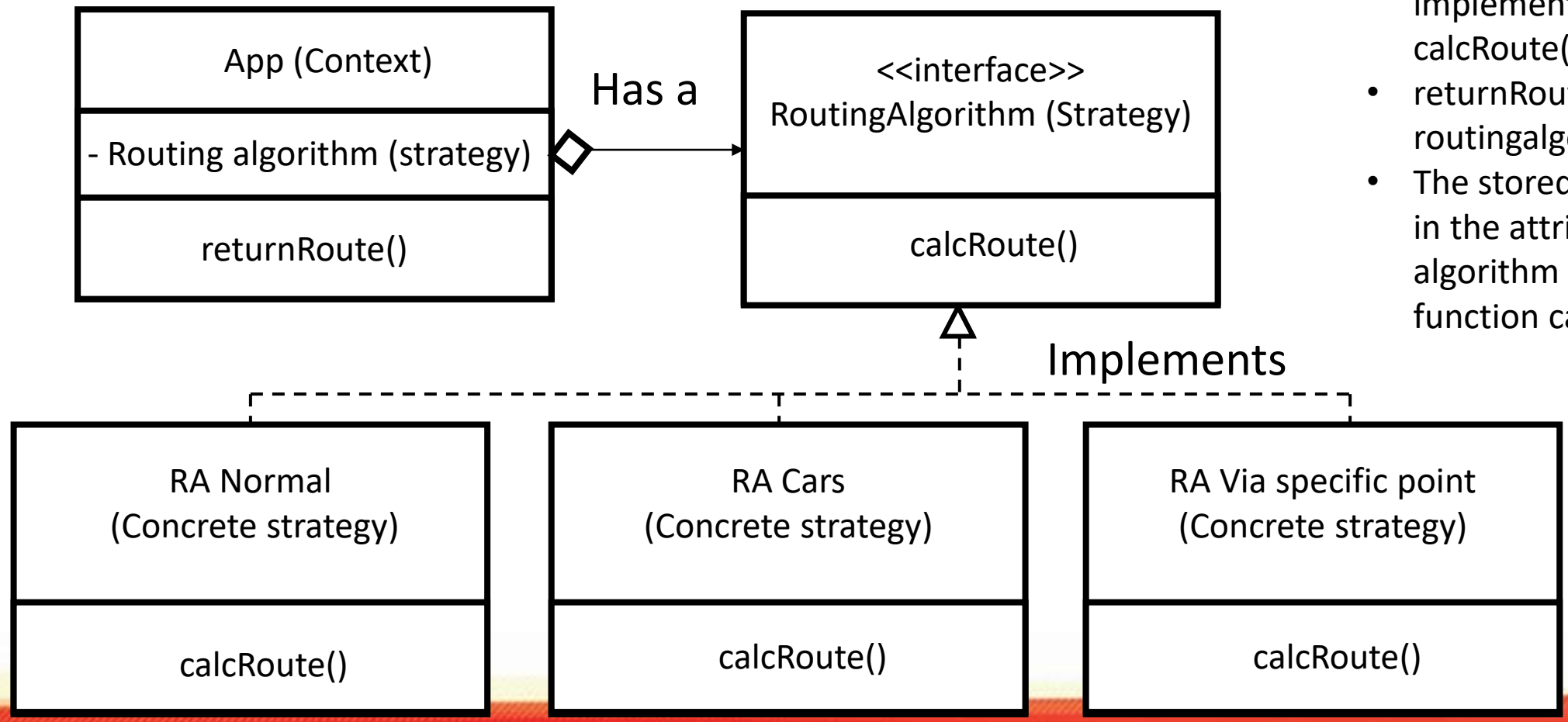
Strategy Pattern UML



Adding a new strategy

- Implement the Execute function for this strategy
- Done!
- Context is unchanged!
- Context works with an abstract strategy, so as long as a strategy has an execute function, it can use it (which it automatically has as it implements the interface strategy!)

Application on previous problem



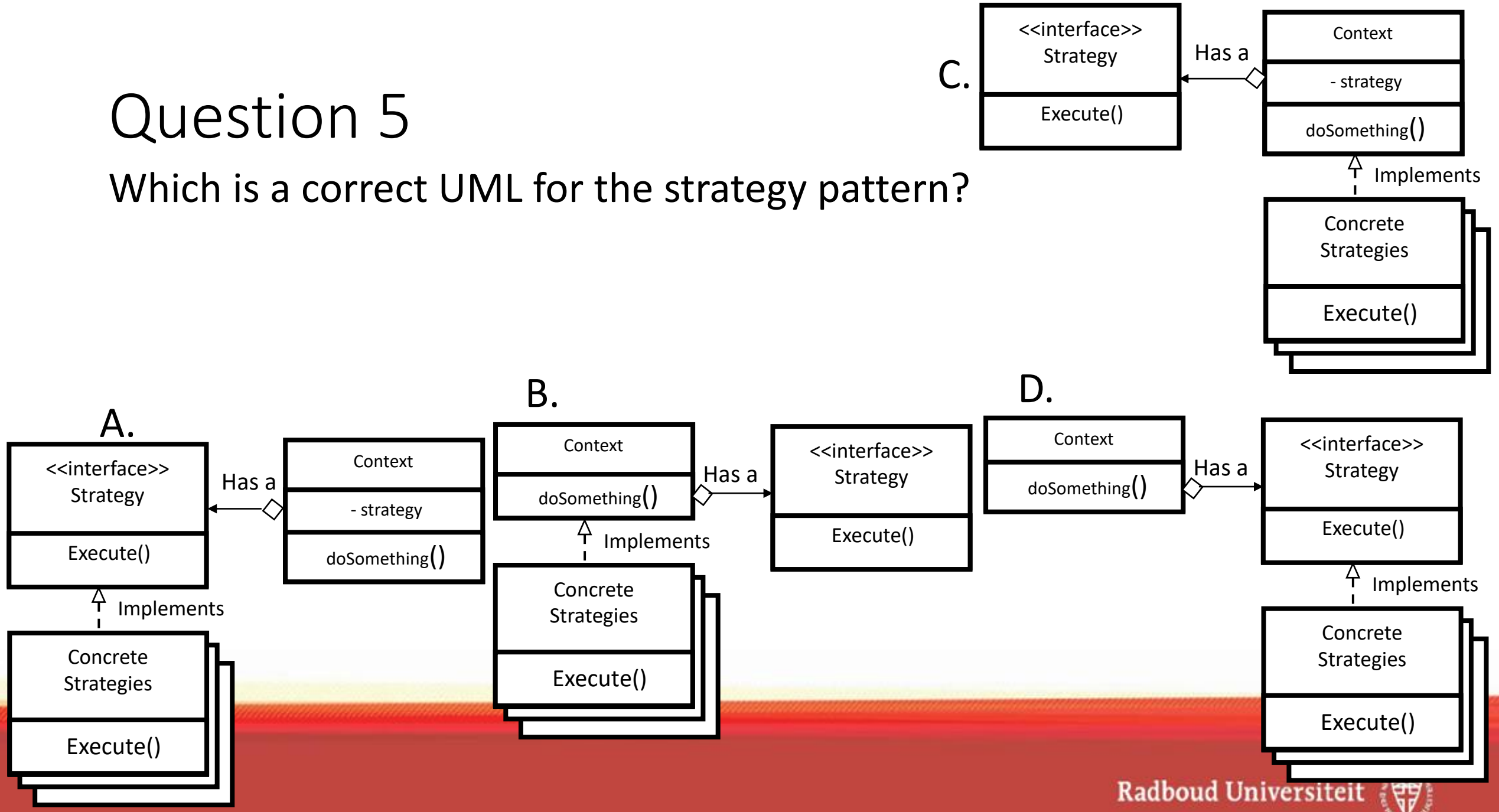
- All concrete routing algorithms implement the interface and have an implementation of calcRoute()
- returnRoute() uses routingalgorithm.calcRoute()
- The stored RoutingAlgorithm in the attribute decides which algorithm is used with this function call!

Live Coding Session

- Lets take a look at the program from the previous example
- This can be found in StrategyPatternExample.zip
- Take a look at it in your own time as well and experiment!
- Hint: You can also use the debugger to investigate the flow of the program

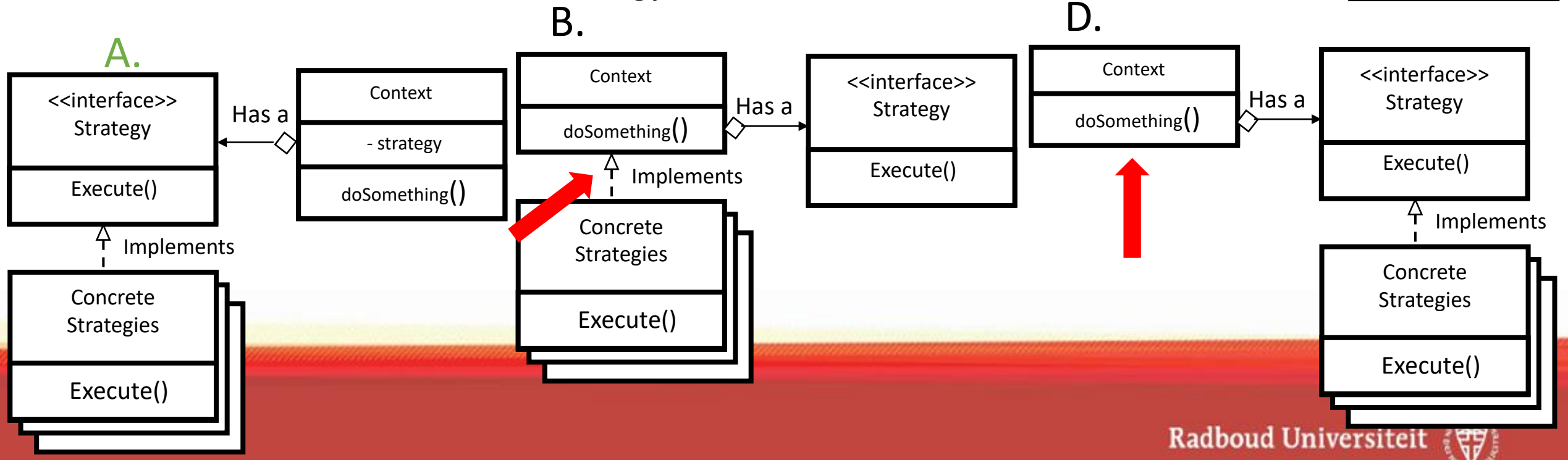
Question 5

Which is a correct UML for the strategy pattern?



Question 5

A is correct. The Concrete Strategies should Implement the interface Strategy (so A and D remain). Furthermore, the Context should have an attribute with the used strategy, so A is correct



Question 6

Suppose the following program:

We have a program that encrypts a given message with either RSA or Triple DES (2 encryption methods). The program probably won't be extended with other encryption methods.

Should we use the strategy pattern and how?

A. Yes, the encryption methods are the strategies

B. Yes, the different messages are the strategies

C. No, it is not necessary to apply the pattern

Question 6

We have a program that encrypts a given message with either RSA or Triple DES (2 encryption methods). The program probably won't be extended with other encryption methods.

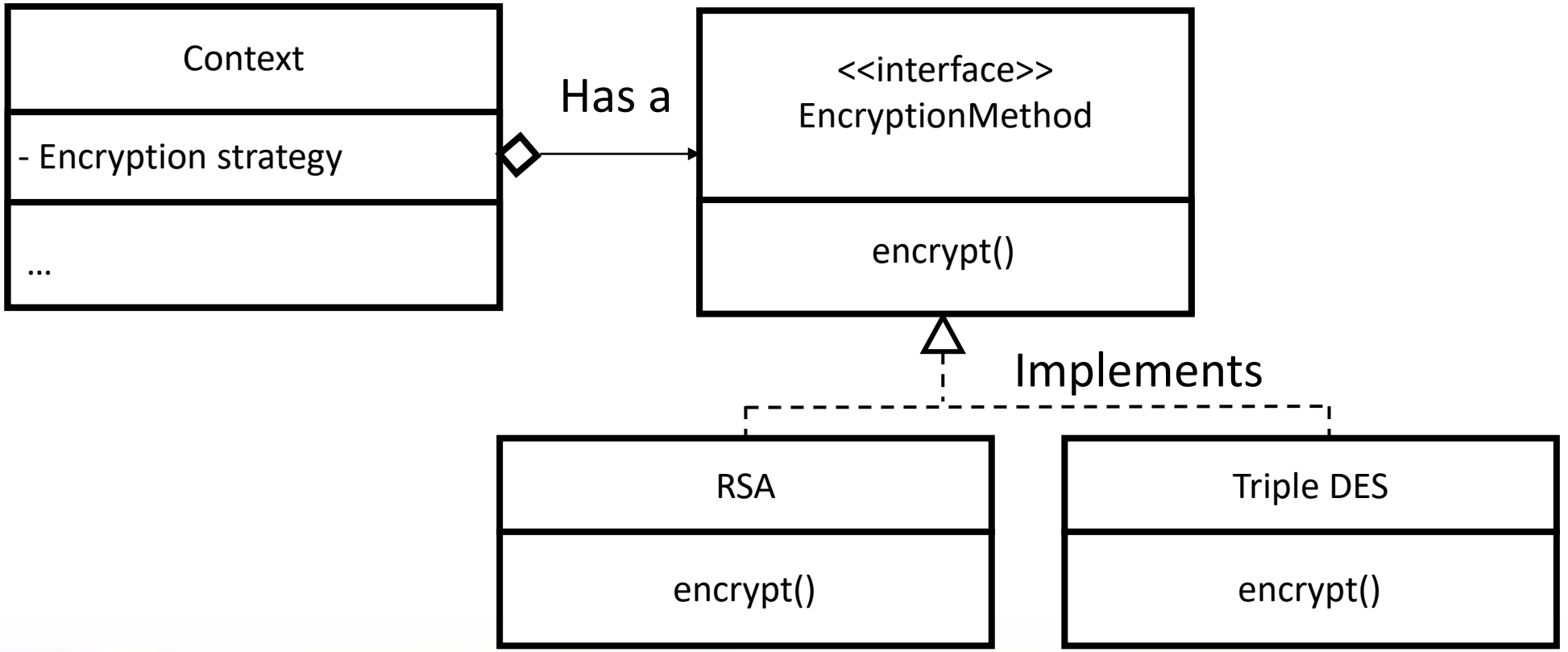
A. Yes, the encryption methods are the strategies

B. Yes, the different messages are the strategies

C. No, it is not necessary to apply the pattern

These encryption methods can be applied as strategies. However, it is debatable whether that is better, as there are only 2 methods used. Therefore, both answers are correct, but the preference is to still apply the strategy pattern for future adjustments

Question 6


Context
- Encryption strategy
…
Has a
<<interface>>
EncryptionMethod
encrypt()
Implements
RSA
encrypt()
Triple DES
encrypt()

Question 7

Suppose the following program:

We have a simulation of robots which move in a field. These robots have different behaviors, e.g. defensive, offensive or random. These behaviors can also be combined with each other or can have different variations.

Should we use the strategy pattern and how?

A. Yes, the robots are the strategies

B. Yes, the behaviors are the strategies

C. No, it is not necessary to apply the pattern

Question 7

We have a simulation of robots which move in a field. These robots have different behaviors, e.g. defensive, offensive or random. These behaviors can also be combined with each other or can have different variations.

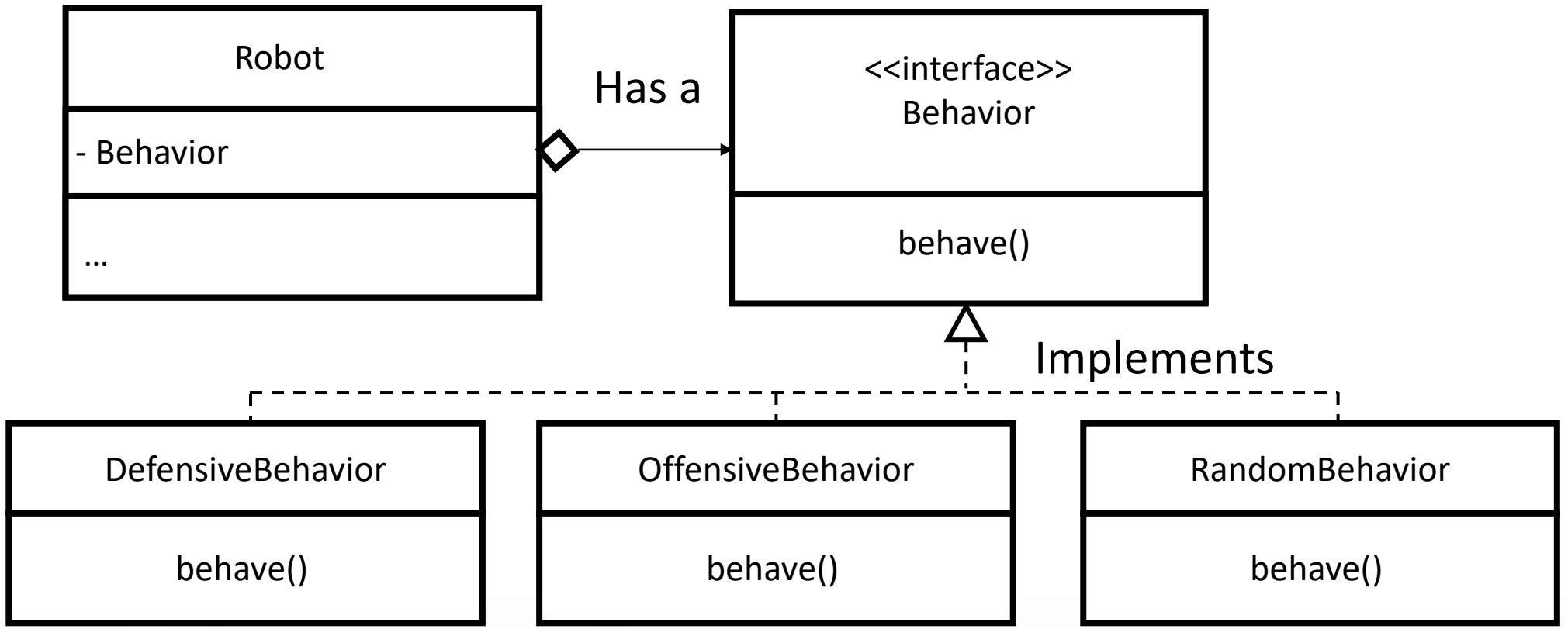
A. Yes, the robots are the strategies

B. Yes, the behaviors are the strategies

C. No, it is not necessary to apply the pattern

The behaviors are the strategies used by the robots; see the UML on the next slide

Question 7


Robot
- Behavior
...
Has a
<<interface>>
Behavior
behave()
Implements
DefensiveBehavior
behave()
OffensiveBehavior
behave()
RandomBehavior
behave()

Question 8

Suppose the following program:

We have a program that extracts plain text from different types of files, e.g. pdf, xml or html. This is then returned as a string.

Should we use the strategy pattern and how?

A. Yes, the different types of files are the strategies

B. Yes, the different extraction algorithms are the strategies

C. No, it is not necessary to apply the pattern

Question 8

We have a program that extracts plain text from different types of files, e.g. pdf, xml or html. This is then returned as a string.

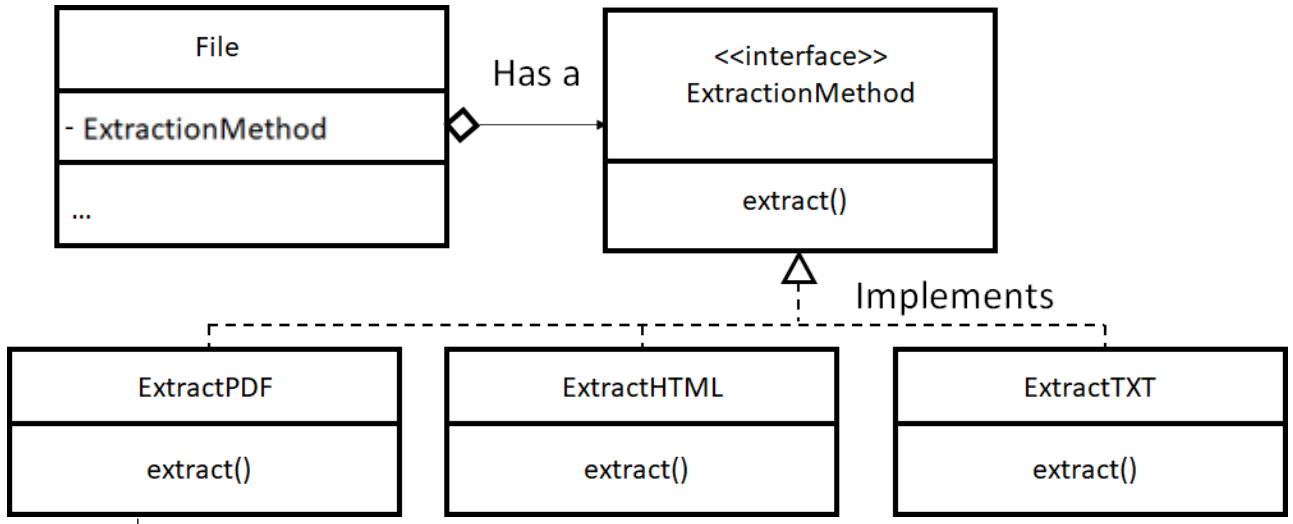
A. Yes, the different types of files are the strategies

B. Yes, the different extraction algorithms are the strategies

C. No, it is not necessary to apply the pattern

Using the Strategy pattern here will result in non-modular code, as the different methods for the specific text files can only be applied on the corresponding text file instead of universally on all files. It is better to have an abstract function extract() implemented for all the different files. See the next slide for the UMLs.

Question 8



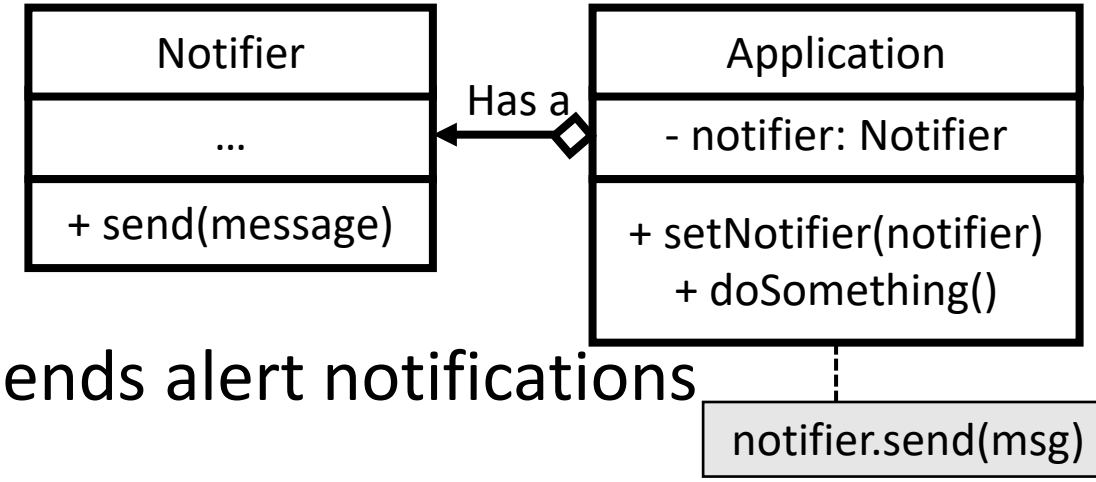

```
extract() {
  if (file.getClass() == pdf.getClass()) {
  // do something
  } else {
    ???
  }
}
```

Recap

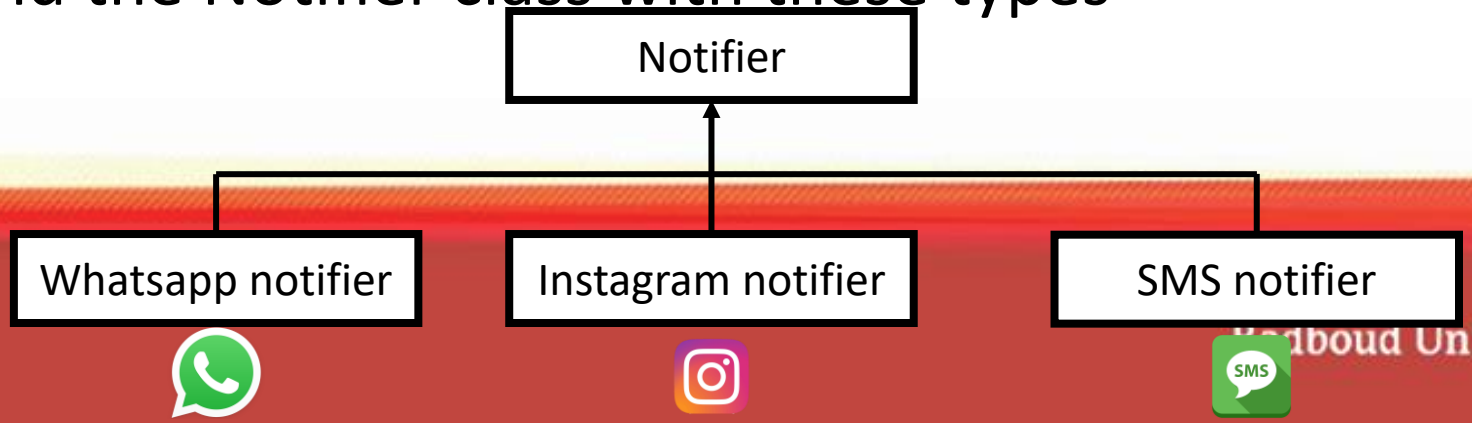
- The Strategy pattern can be used with classes that do something specific with different strategies
  - E.g. calculating a route, different encryption methods, different behaviors…
- It splits the different strategies (by using an interface) from the context
- The code in the context is unchanged if
  - A strategy contains a bug and is fixed
  - A new strategy is added

Decorator Pattern

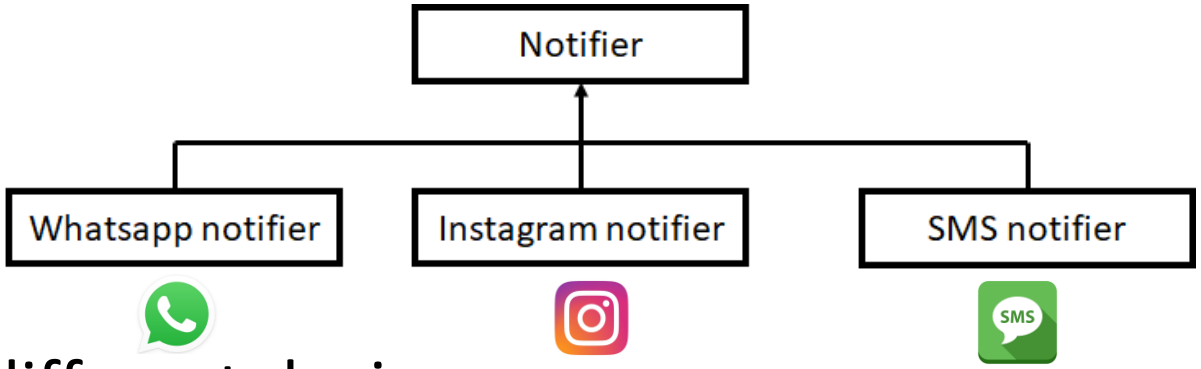
Problem I



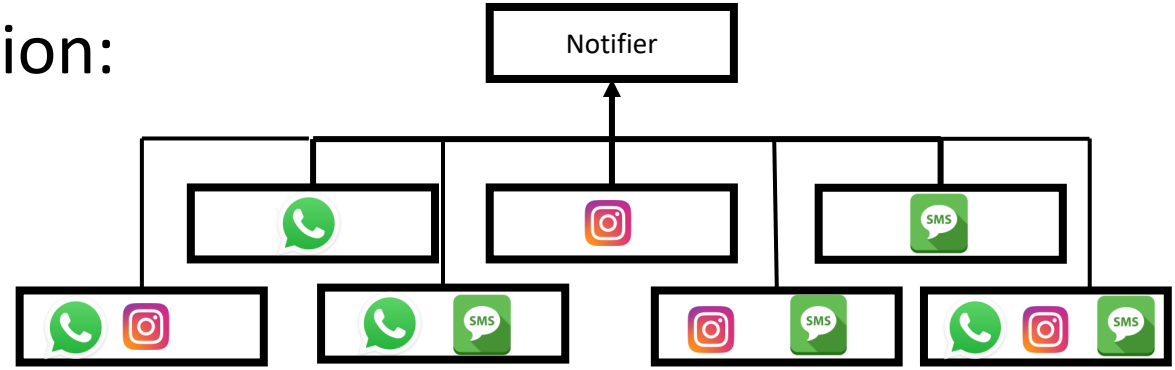
- Suppose we have an application which sends alert notifications
- First only email notifier
  - Solution: make a separate Notifier class
- Afterwards, demand for additional notifiers, next to the standard email notifier
  - E.g. Whatsapp, Instagram, SMS notification
- Solution: Extend the Notifier class with these types



Problem II



- Now: persons want notifiers on different devices
  - E.g. Whatsapp and SMS notifier
- “Solution”: Lets make subclasses for all these combinations:
- Unfortunately, this is not a great solution:
  - Duplicate code
  - A lot of unnecessary code
  - Not extendable
    - Imagine adding one extra Notifier

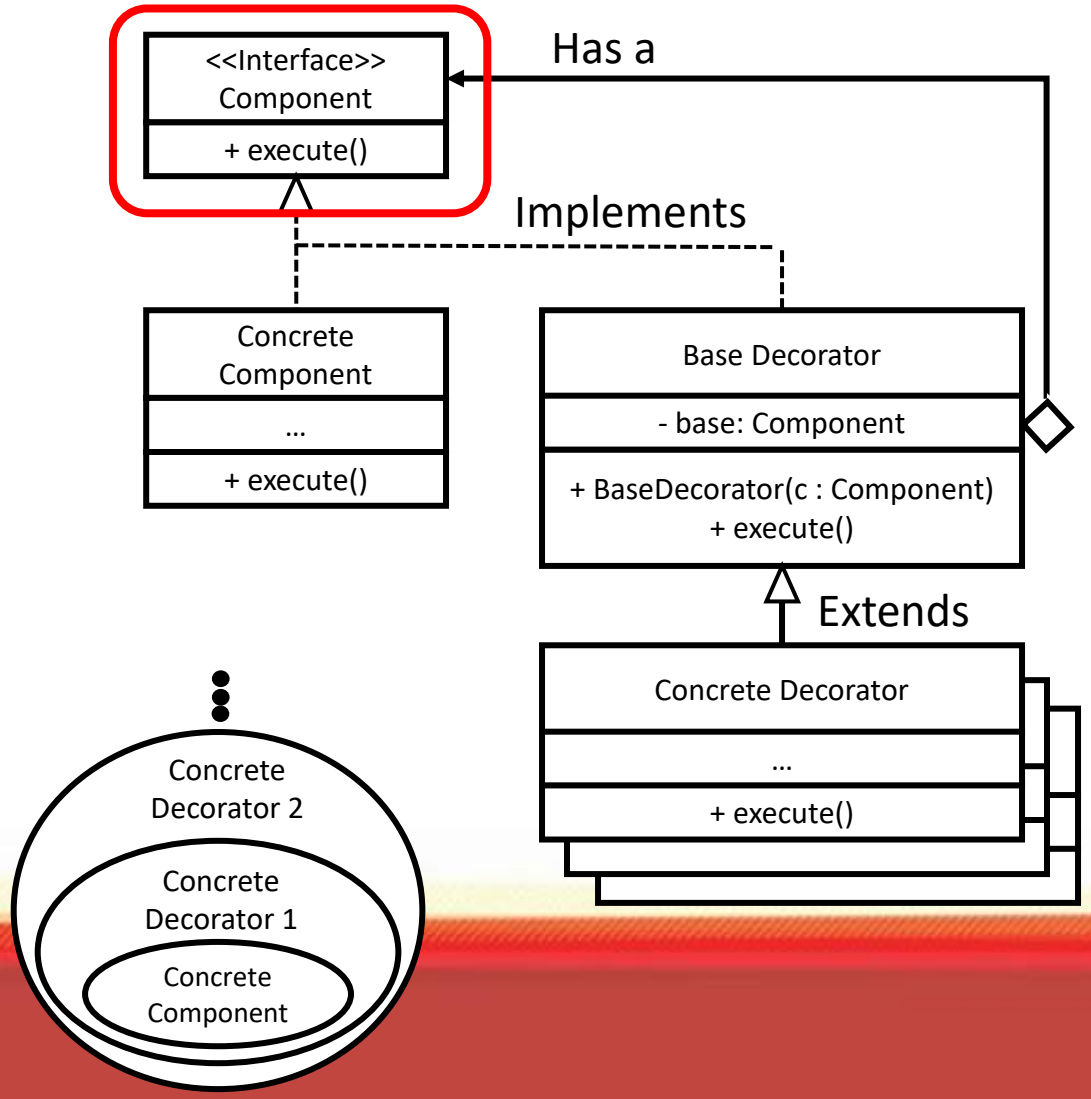


- Solution: **Decorator Pattern**

Decorator Pattern

- Previous problem: add multiple notifiers to the basic email notifier
- Decorator Pattern: add multiple decorators to the concrete component
  - E.g. multiple toppings on a pizza
- Instead of subclasses for all combinations
- Simply add the desired decorators to the concrete component
- Advantages:
  - No duplicate code
  - Easily extendable with new decorators
  - A lot of flexibility

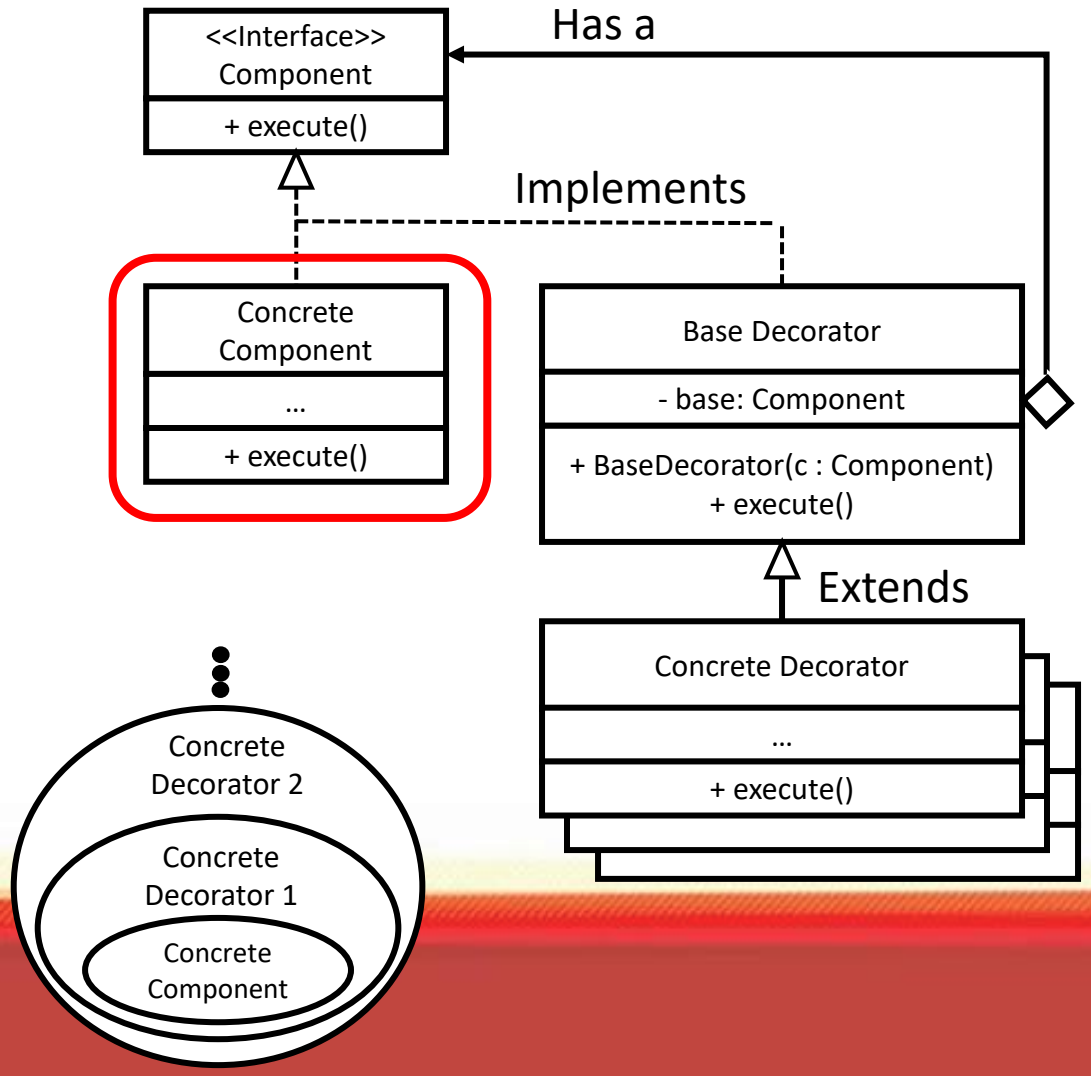
Decorator Pattern UML



Interface Component

- This is the interface for both the Concrete Component and the Decorators
- It contains a function which is implemented for all Concrete Components and Decorators
  - E.g. a cost() or toString() function for the pizzas with toppings

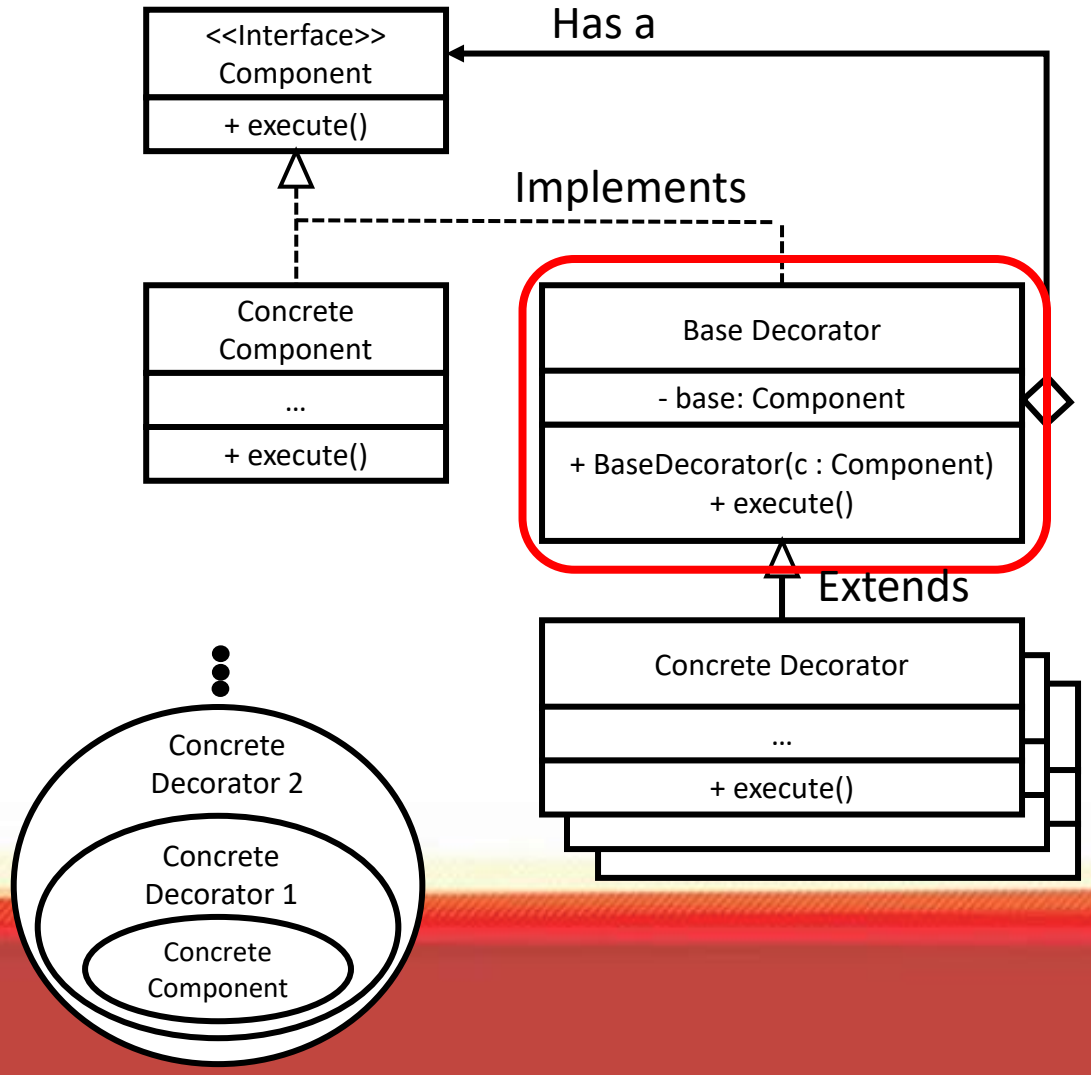
Decorator Pattern UML



Concrete Component

- This is the base on which the Decorators can be "wrapped"
  - E.g. the base of a pizza on which we can "wrap" the toppings
- Note that we have the same function execute() here as well
- Note that we **don't** have a Component object in a Concrete Component, as it is a base and it cannot be wrapped around something in contrast to a Decorator

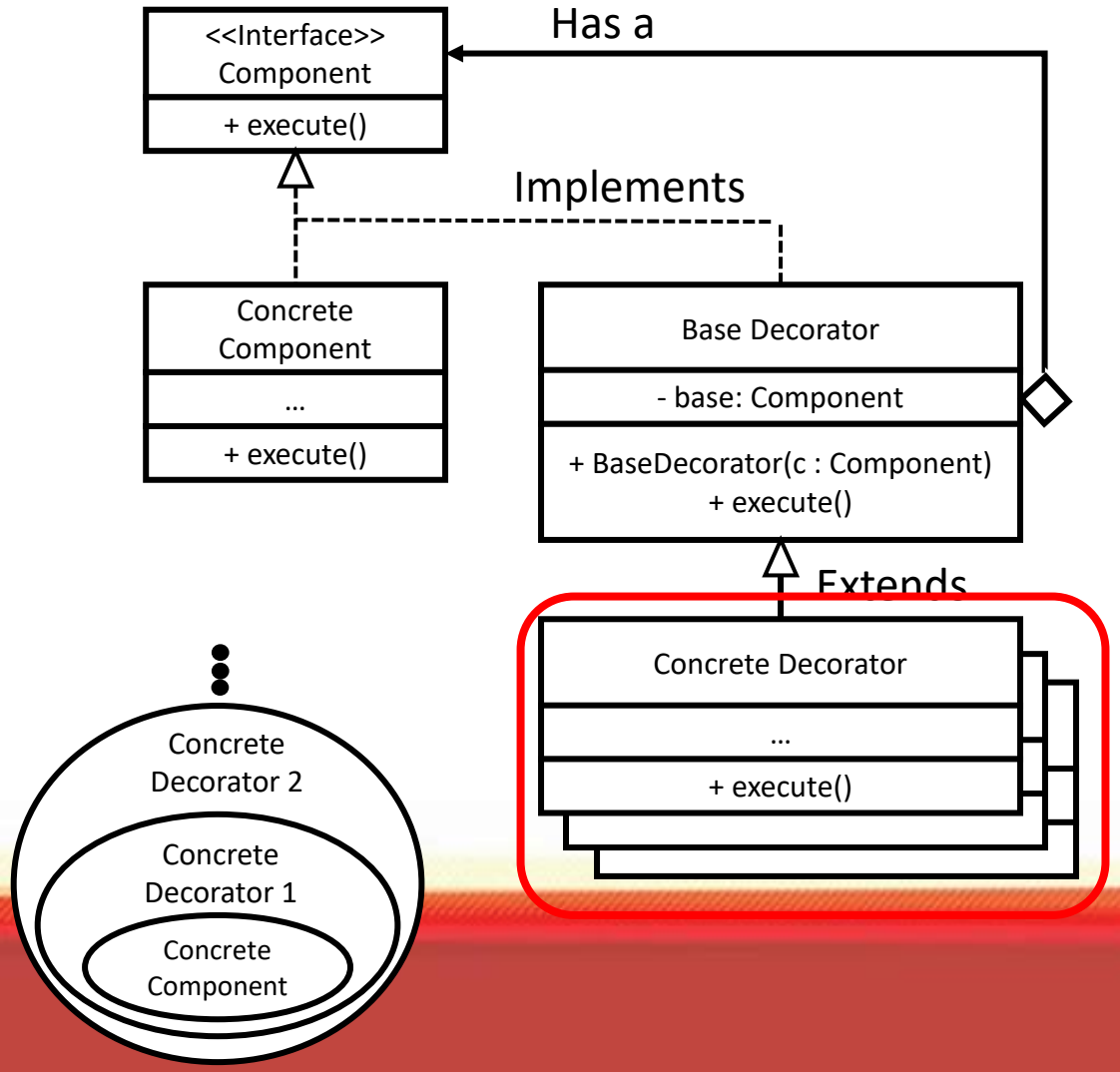
Decorator Pattern UML



Base Decorator

- This serves as the standard Decorator for all the Concrete decorators
- It contains a Component object, which is the object on which the current Decorator is wrapped
  - Note that this can either be a Concrete Component or a Concrete Component with several Decorators wrapped around it represented as a Component
- It contains a Constructor which simply sets the base
- It contains an execute() function, which is standard defined as base.execute() (simply calls the execute of the stored base)
- Note that the arrow with a diamond states that the Base Decorator uses a Component

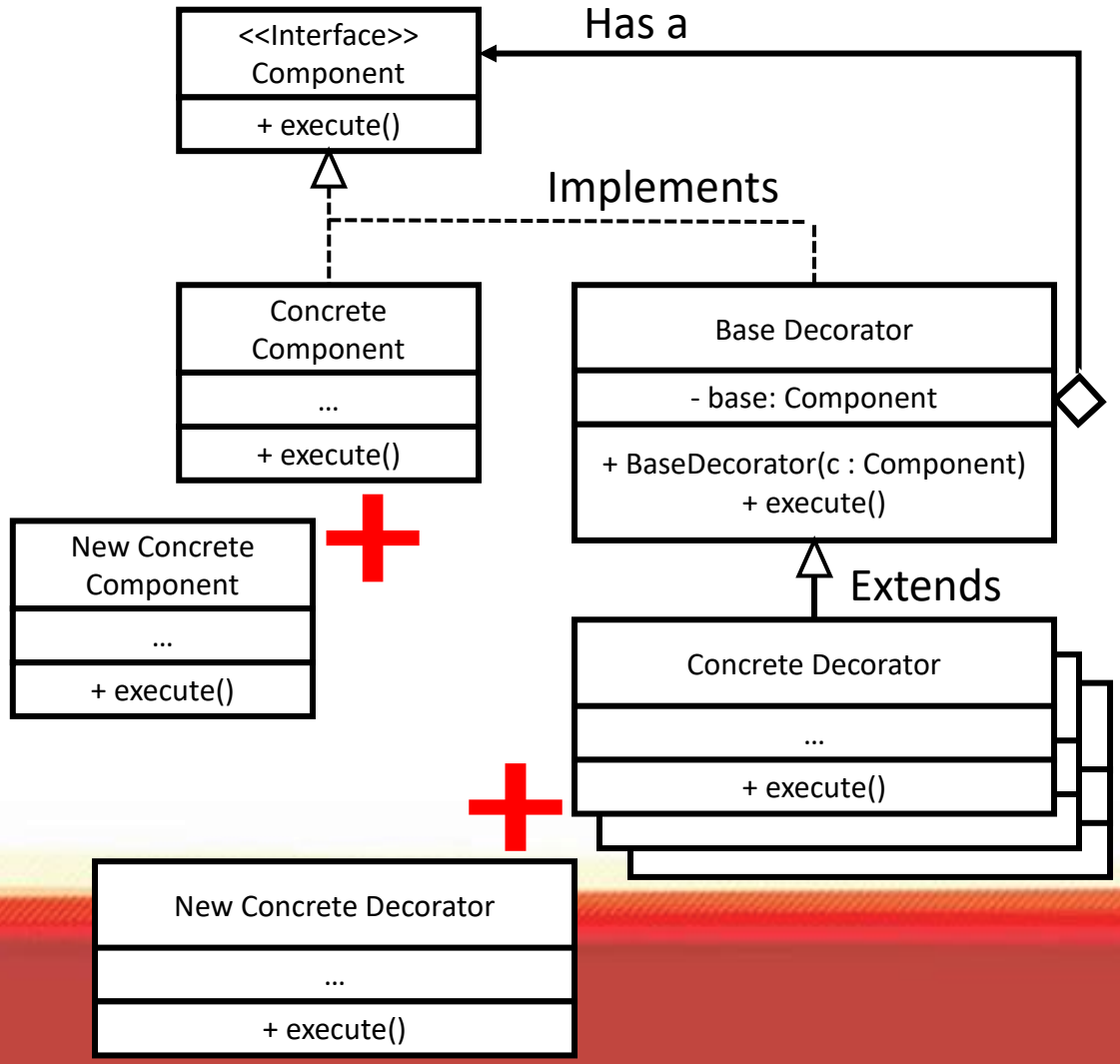
Decorator Pattern UML



Concrete Decorator

- These are the concrete Decorators that **extend** the Base Decorator
- It overwrites the execute() function of the Base Decorator with the extra functionality of the Concrete Decorator
- The overwritten execute() function contains a call super.execute(), which simply calls the execute() function of the extended object, in this case the Base Decorator, thus calling the execute() function of the stored base Component.
- The remainder of the overwritten execute() function is the addition of the Decorator
  - E.g. adding a topping of 35 cents to the rest of the pizza
  - cost() {return super.cost() + 0.35}

Decorator Pattern UML



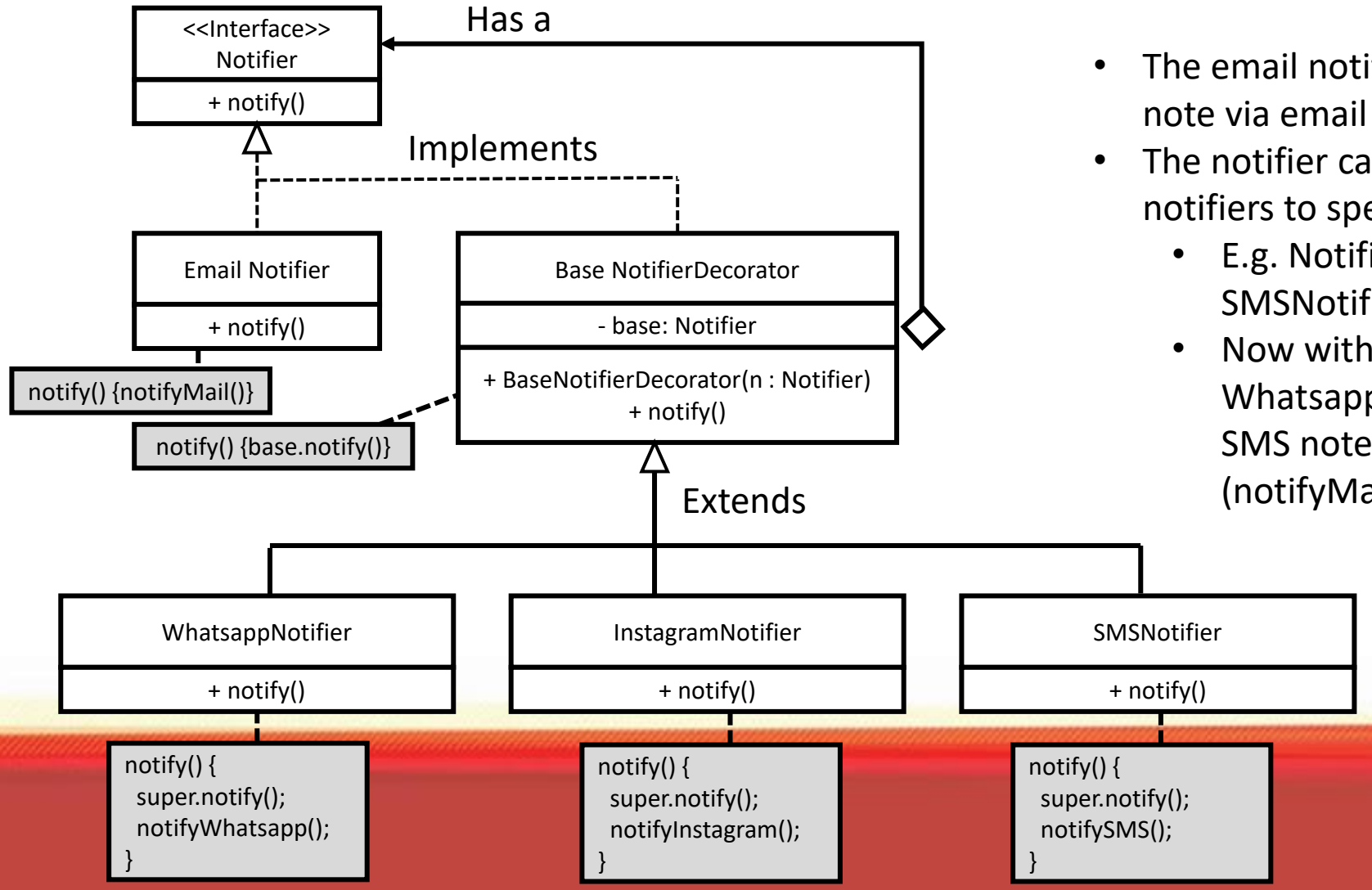
Adding new Concrete Component

- Only the execute() function of this new Component should be implemented
  - E.g. the cost of a different base for a pizza
- It can immediately be used in the whole system while not changed any code in any other class!

Adding new Concrete Decorator

- Similarly, only the execute() function of this new Decorator should be implemented
  - E.g the cost of a new topping for a pizza
- It can immediately be used in the whole system while not changed any code in any other class!

Application on the previous problem



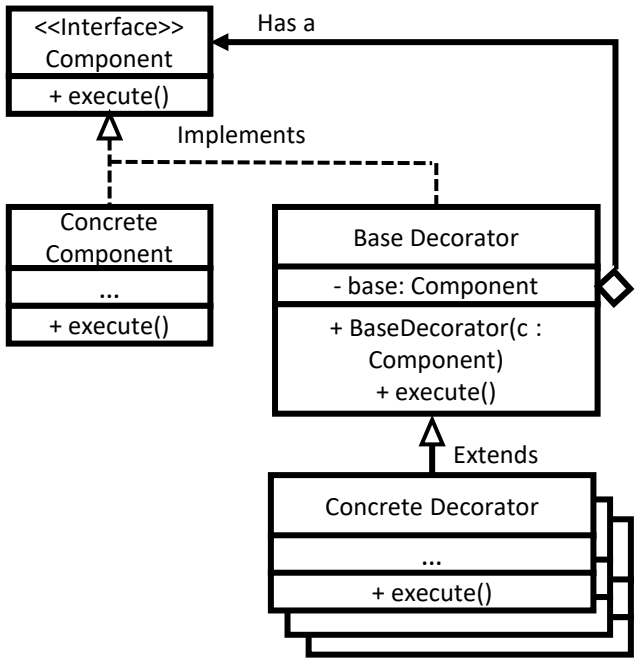
- The email notifier is the base, as everyone gets a note via email
- The notifier can then be wrapped with other notifiers to specify the desired notifiers
  - E.g. Notifier n = new WhatsappNotifier(new SMSNotifier(new EmailNotifier));
  - Now with n.notify, the user receives a Whatsapp note (notifyWhatsapp() is called), a SMS note (notifySMS() is called) and an email (notifyMail() is called)

Live coding session

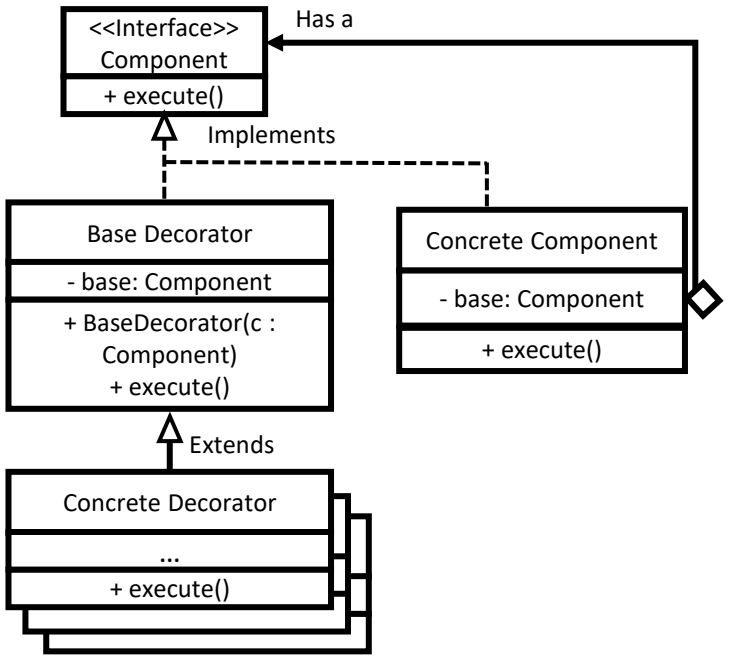
- Lets take a look at the program from the previous example
- This can be found in DecoratorPatternExample.zip
- Take a look at it in your own time as well and experiment!

Question 9: Correct UML for Decorator Pattern?

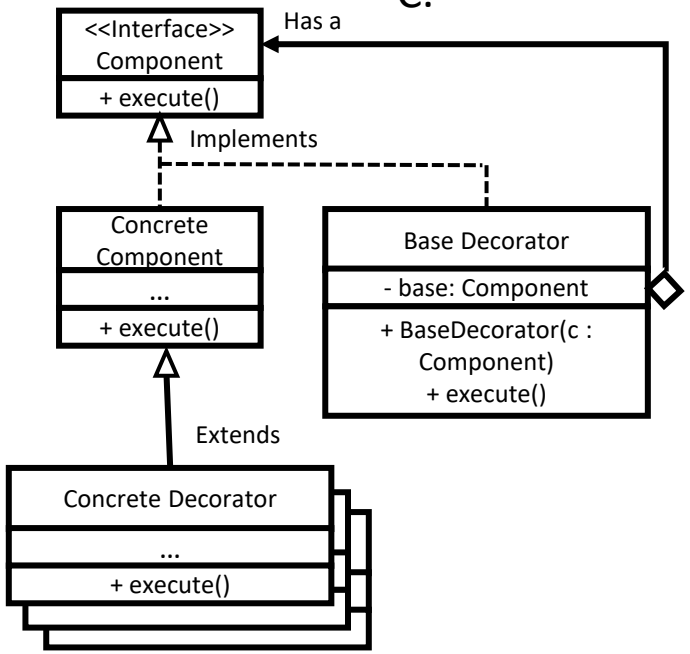
A.



B.

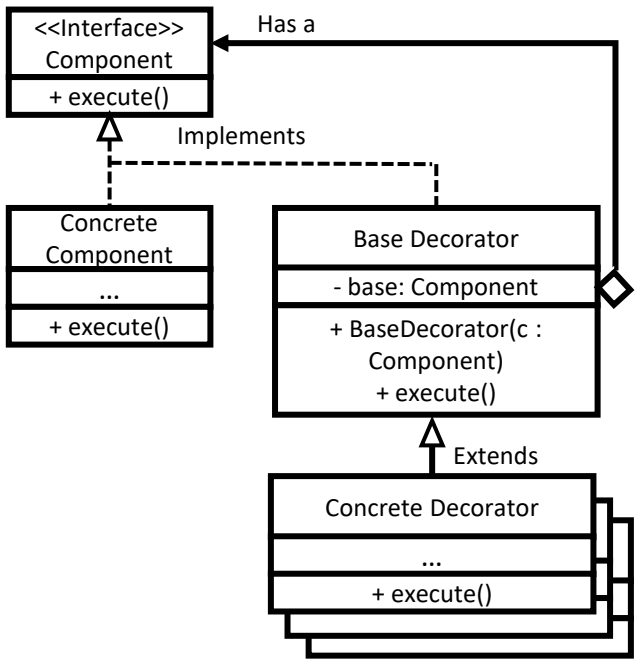


C.

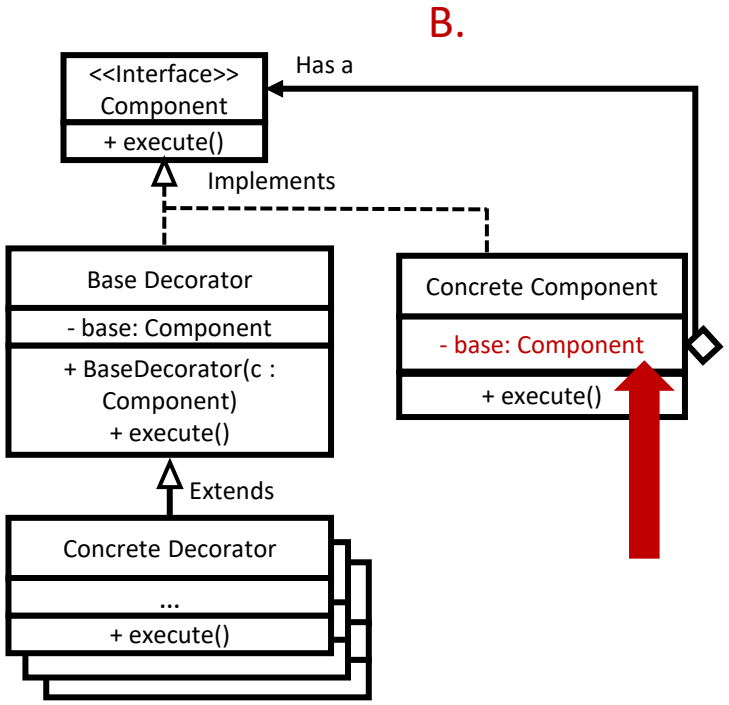


Question 9: Correct UML for Decorator Pattern?

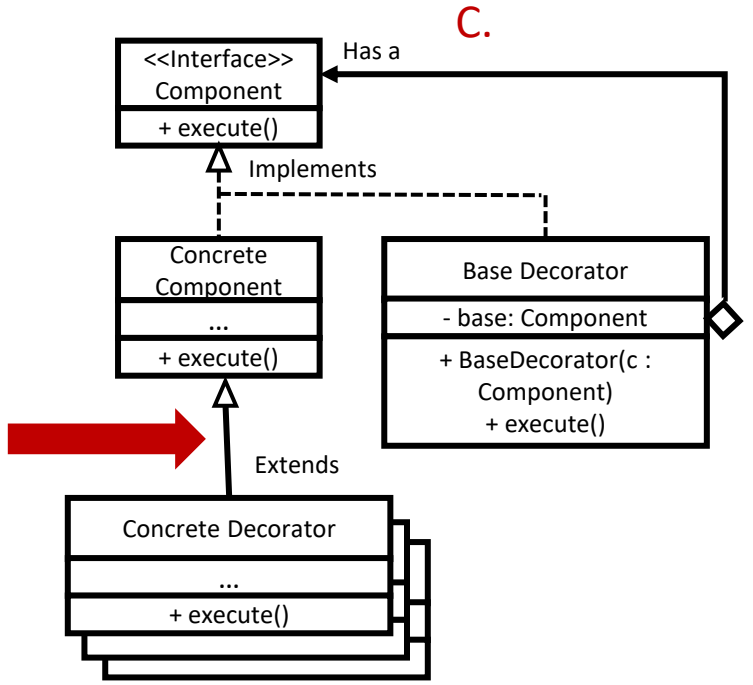
A.



B.



C.



A is correct. The concrete component in B also has a base Component as attribute, but this is not allowed as it can only be a Base. In C, the Concrete Decorators are extending the Concrete Component, but they should be extending the Base Decorators.

Question 10

Suppose we have a data structure representing multiple vehicles (e.g. car, bus, truck etc) with the customization option to have extra large tires. There are no further customization options.

Should we use the Decorator Pattern here and how?

A. Yes, the vehicles are the Concrete Components and the extra large tires the Decorators

B. Yes, the tires are the Concrete Components and the vehicles the Decorators

C. No, it is not necessary to apply the pattern

Question 10

Suppose we have a data structure representing multiple vehicles (e.g. car, bus, truck etc) with the customization option to have extra large tires. There are no further customization options.

Should we use the Decorator Pattern here and how?

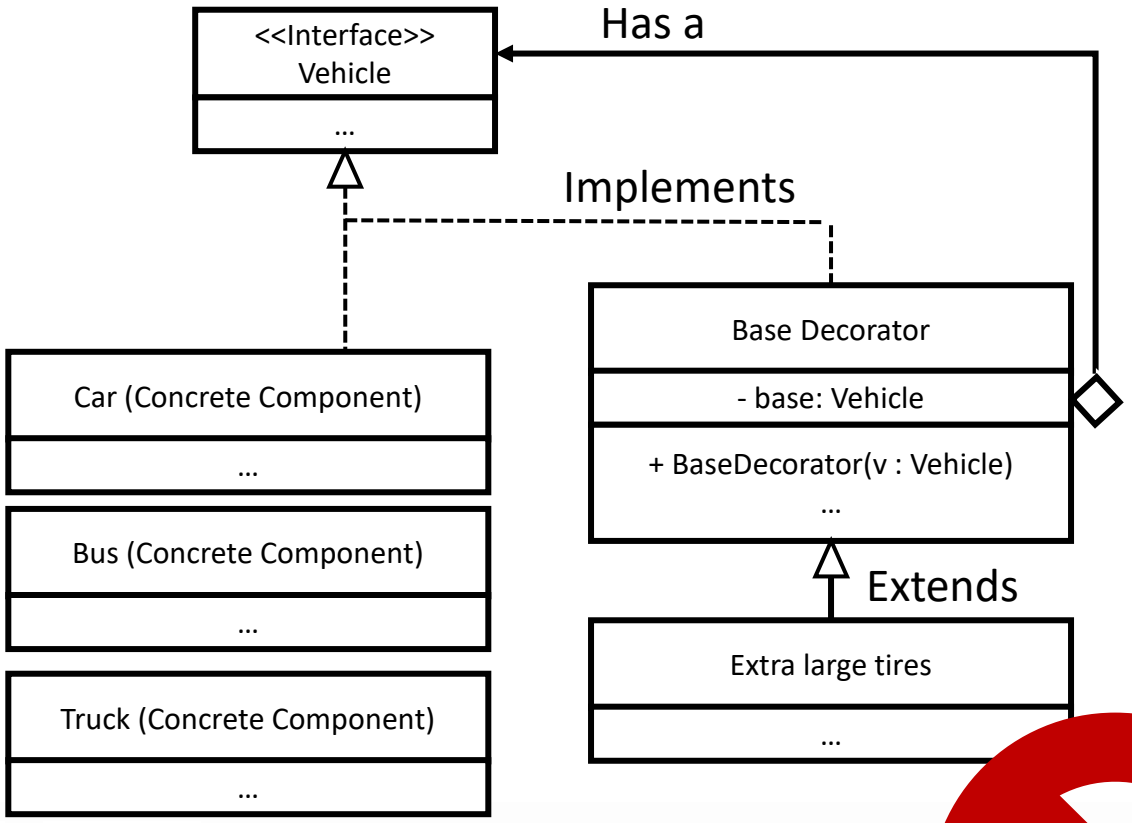
A. Yes, the vehicles are the Concrete Components and the extra large tires the Decorators

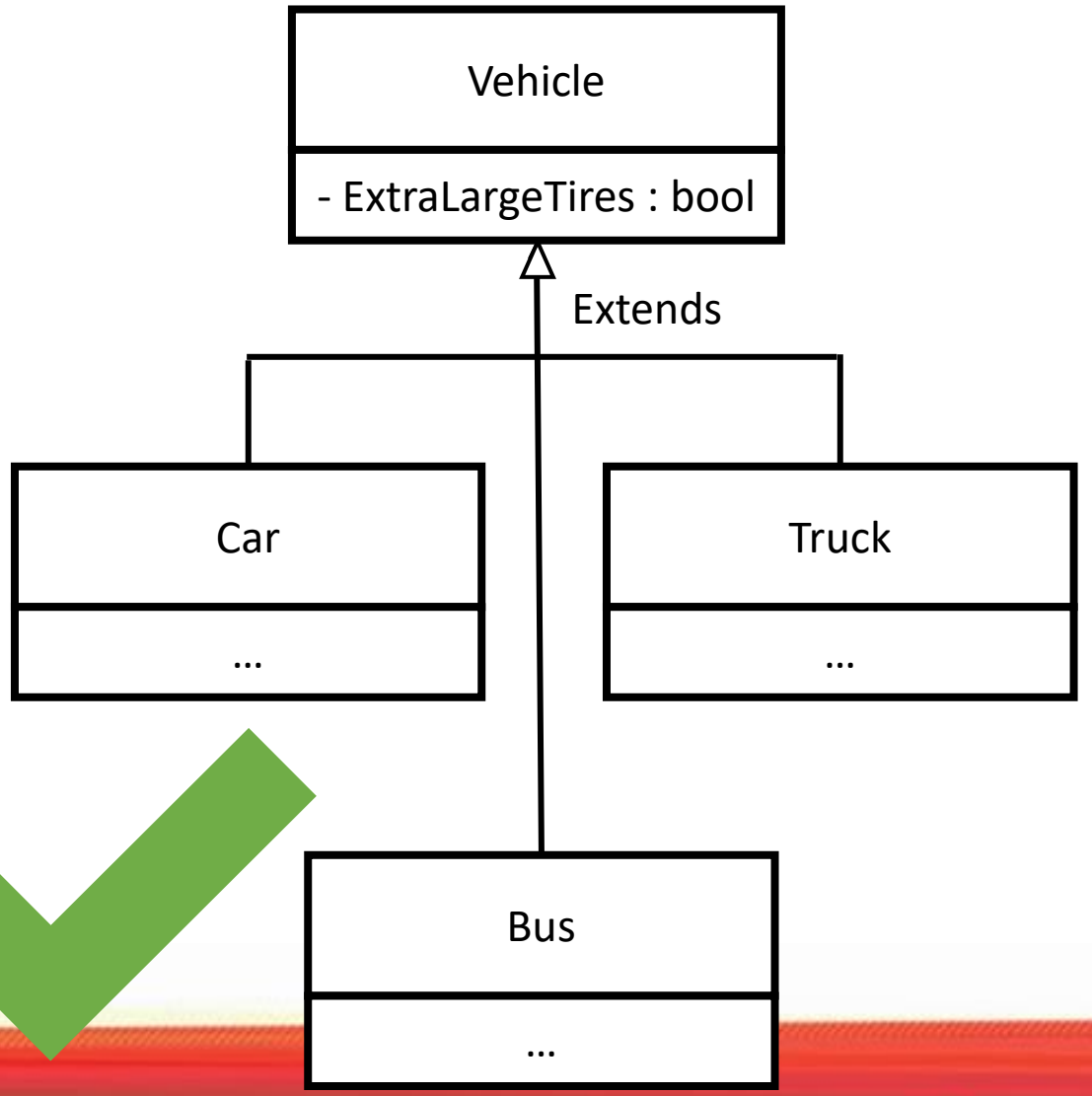
B. Yes, the tires are the Concrete Components and the vehicles the Decorators

C. No, it is not necessary to apply the pattern

We have many different Concrete Components (the vehicles) and only one Decorator (extra large tires). It is better to simply add ExtraLargeTires as a Boolean attribute to all the vehicles which is either true or false

Question 10


<<Interface>>
Vehicle
...
Has a
Implements
Base Decorator
- base: Vehicle
+ BaseDecorator(v : Vehicle)
...
Extends
Extra large tires
...
Car (Concrete Component)
...
Bus (Concrete Component)
...
Truck (Concrete Component)
...


Vehicle
- ExtraLargeTires : bool
Extends
Car
...
Truck
...
Bus
...

Question 11

Suppose we have a data structure representing a tech shop. It is possible to buy either a desktop or laptop with multiple accessories, like a mouse, mouse pad, keyboard etc. Moreover, multiple mouses or keyboards can be added to the order.

Should we use the Decorator Pattern here and how?

A. Yes, the laptop/desktop are the Concrete Components and the accessories are the Decorators

B. Yes, the accessories are the Concrete Components and the laptop/desktop are the Decorators

C. No, it is not necessary to apply the pattern

Question 11

Suppose we have a data structure representing a tech shop. It is possible to buy either a desktop or laptop with multiple accessories, like a mouse, mouse pad, keyboard etc. Moreover, multiple mouses or keyboards can be added to the order.

Should we use the Decorator Pattern here and how?

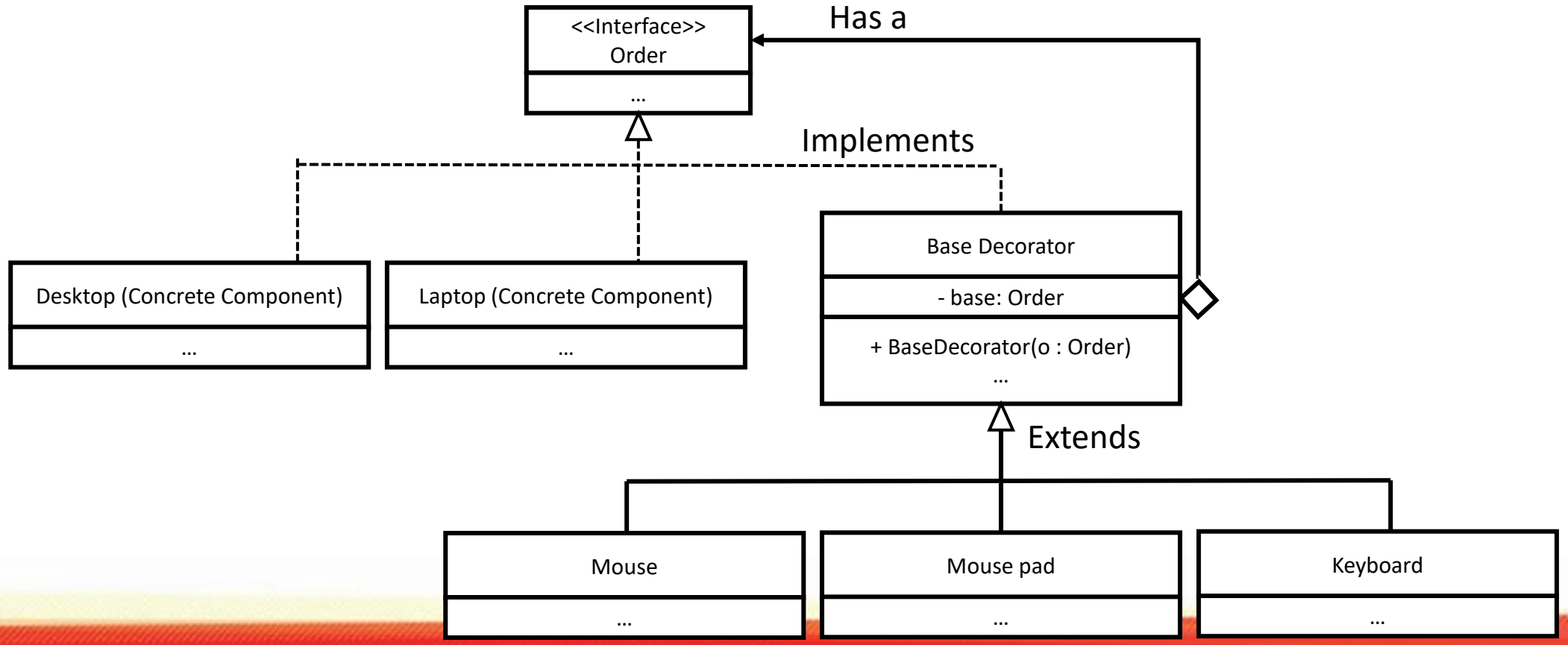
A. Yes, the laptop/desktop are the Concrete Components and the accessories are the Decorators

B. Yes, the accessories are the Concrete Components and the laptop/desktop are the Decorators

C. No, it is not necessary to apply the pattern

The Decorator pattern fits perfectly for this use case, see the next slide

Question 11


<<Interface>>
Order
...
Has a
Implements
Desktop (Concrete Component)
...
Laptop (Concrete Component)
...
Base Decorator
- base: Order
+ BaseDecorator(o : Order)
...
Extends
Mouse
...
Mouse pad
...
Keyboard
...

Question 12

Suppose we have a data structure representing a GUI window with the option to have a border around it. This border can be exactly one of many colors. There are no other customizable options.

Should we use the Decorator Pattern here and how?

A. Yes, the window is the Concrete Component and the border option and colors are the Decorators

B. Yes, the border option and the colors are the Concrete Components and the window is the Decorator

C. No, it is not necessary to apply the pattern

Question 12

Suppose we have a data structure representing a GUI window with the option to have a border around it. This border can be exactly one of many colors. There are no other customizable options.

Should we use the Decorator Pattern here and how?

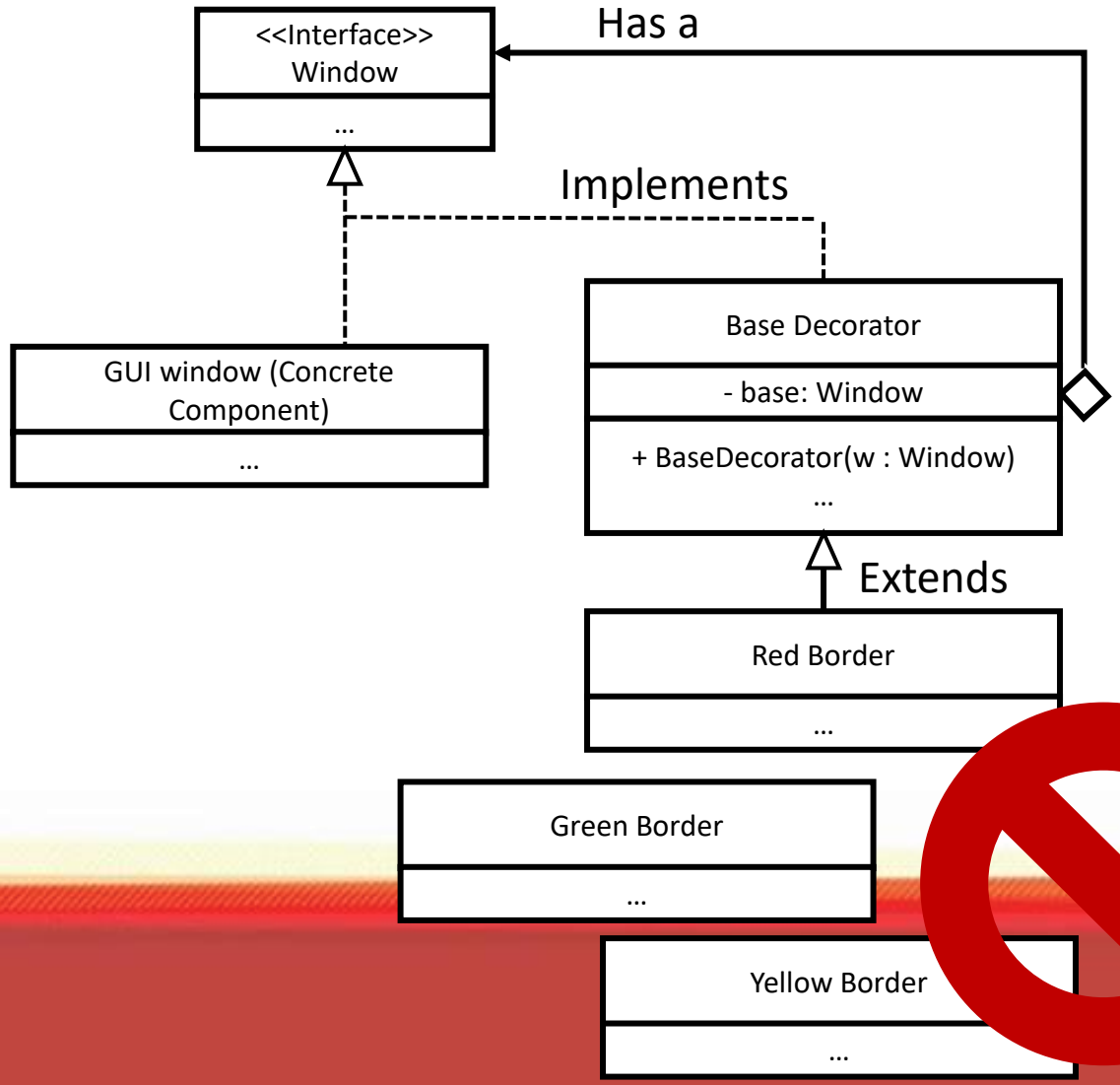
A. Yes, the window is the Concrete Component and the border option and colors are the Decorators

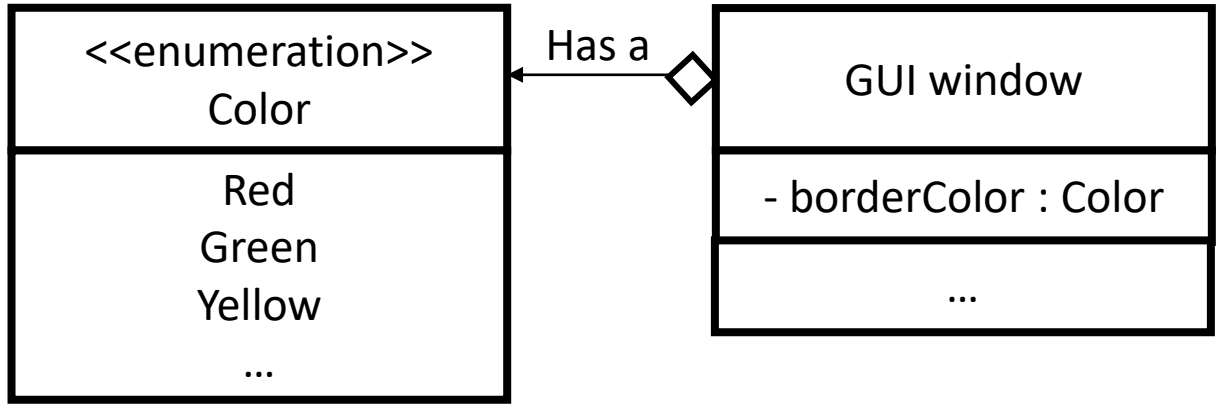
B. Yes, the border option and the colors are the Concrete Components and the window is the Decorator

C. No, it is not necessary to apply the pattern

The different colors can work as Decorators and the GUI window as a Concrete Component, but it is a bit too much as there is exactly one color used. The different colors can better be represented as an enum and the color of the border as an attribute inside the GUI window object.

Question 12


<<Interface>>
Window
...
Has a
Implements
GUI window (Concrete Component)
...
Base Decorator
- base: Window
+ BaseDecorator(w : Window)
...
Extends
Red Border
...
Green Border
...
Yellow Border
...

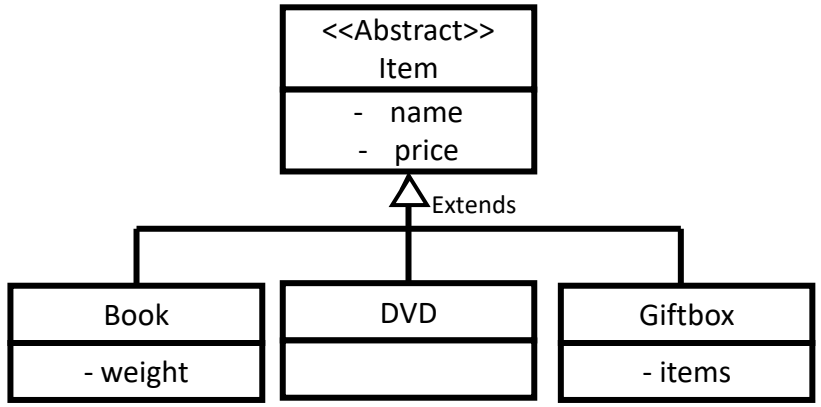

<<enumeration>>
Color
Red
Green
Yellow
...
Has a
GUI window
- borderColor : Color
...

Recap

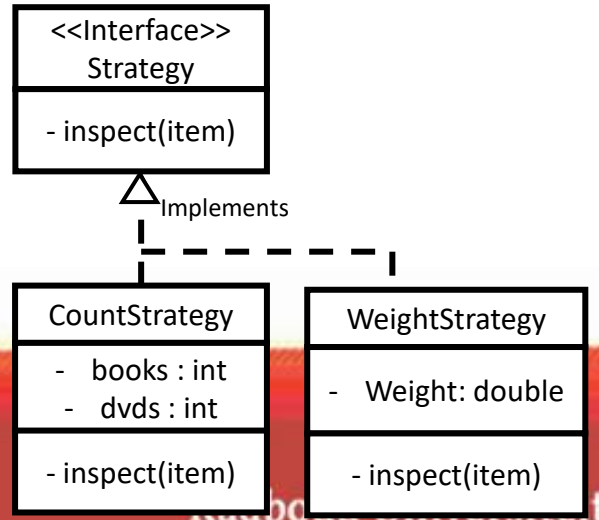
- The Decorator Pattern can be used if there are multiple properties (Decorators) that can be “wrapped” around a base object (Concrete Component)
  - E.g. options for a GUI window, accessories for products etc.
- It splits the Concrete Component and Decorators
- Multiple Decorators can be “wrapped” around a Concrete Component to create a new Component
- With this Pattern, one can easily
  - Add new Decorators
  - Add new Concrete Components

Visitor Pattern

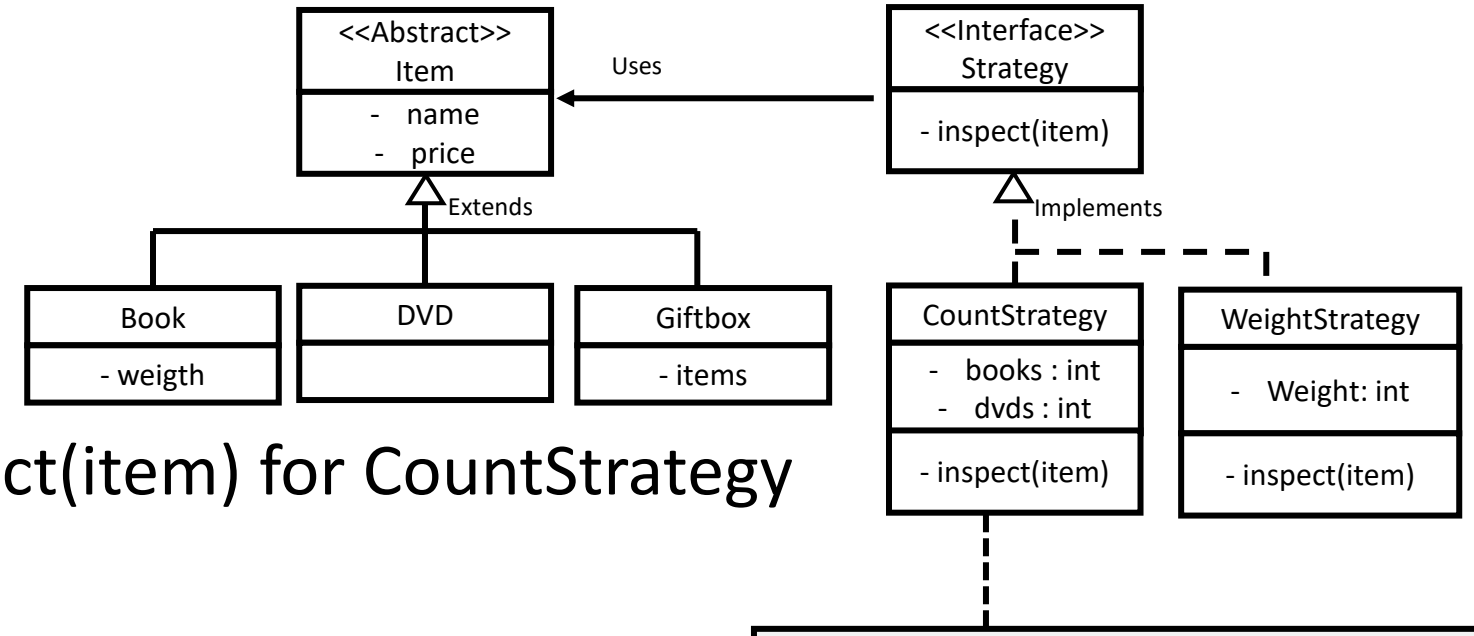
Problem I



- Suppose we have a shop with different items:
  - DVD's (name and price)
  - Books (name, price and unique weight)
  - Giftbox (name, price and list of items)
- There are different operations that can be performed on items
  - Count all the books and DVDs
  - Get the total weight of the item
- These can be seen as different "strategies"



Problem II

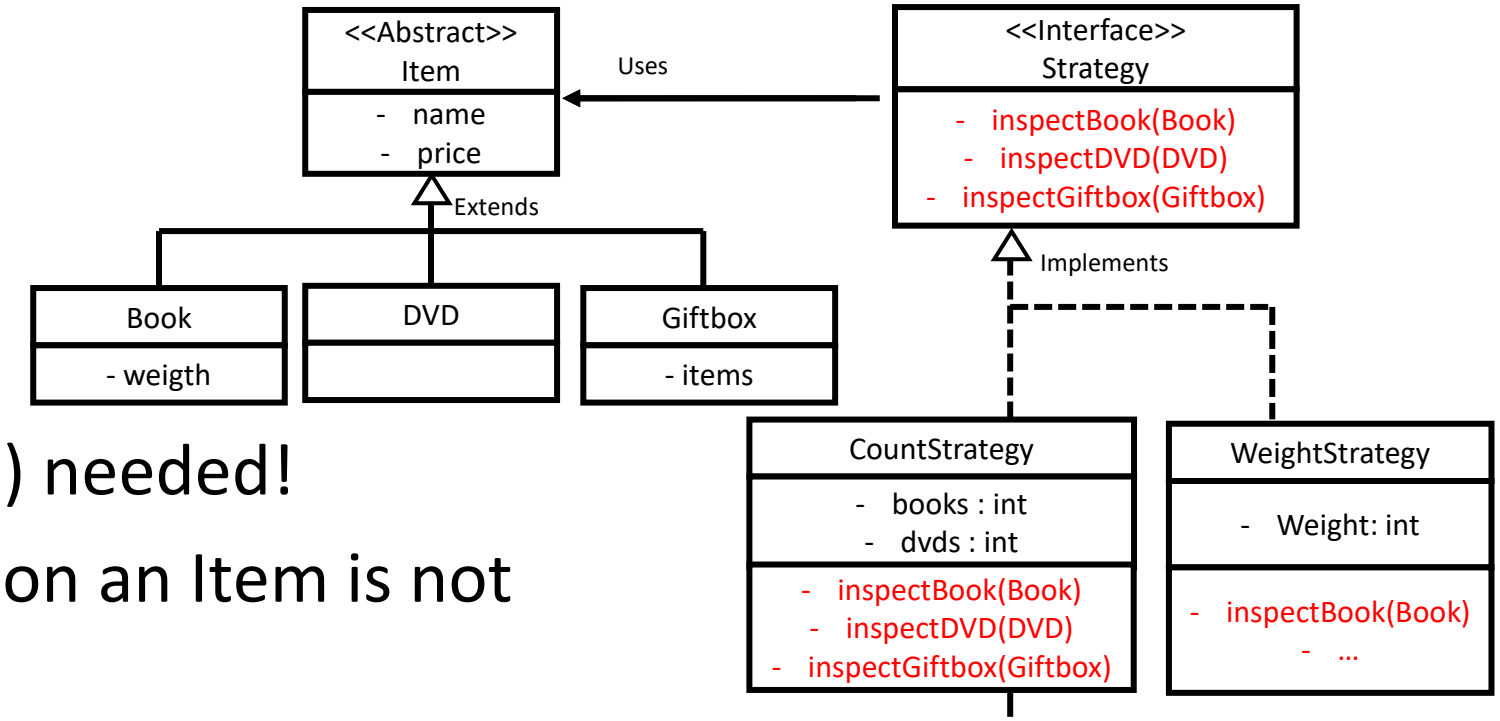


- At first glance this look OK
- However, what does inspect(item) for CountStrategy looks like?
- Using getClass() or instanceof is not something we want in OO…
- Solution: make an inspect function for every item!

```
Void inspect(item) {
   if (item.getClass() == Book.class) {
        books += 1;
   } else if (item.getClass() == DVD.class) {
        dvds += 1;
   } else if (item.getClass() == Giftbox.class) {
        for (Item subItem: item.getItems()) {
             inspect(subItem);
         }
   }
}
```

Problem III

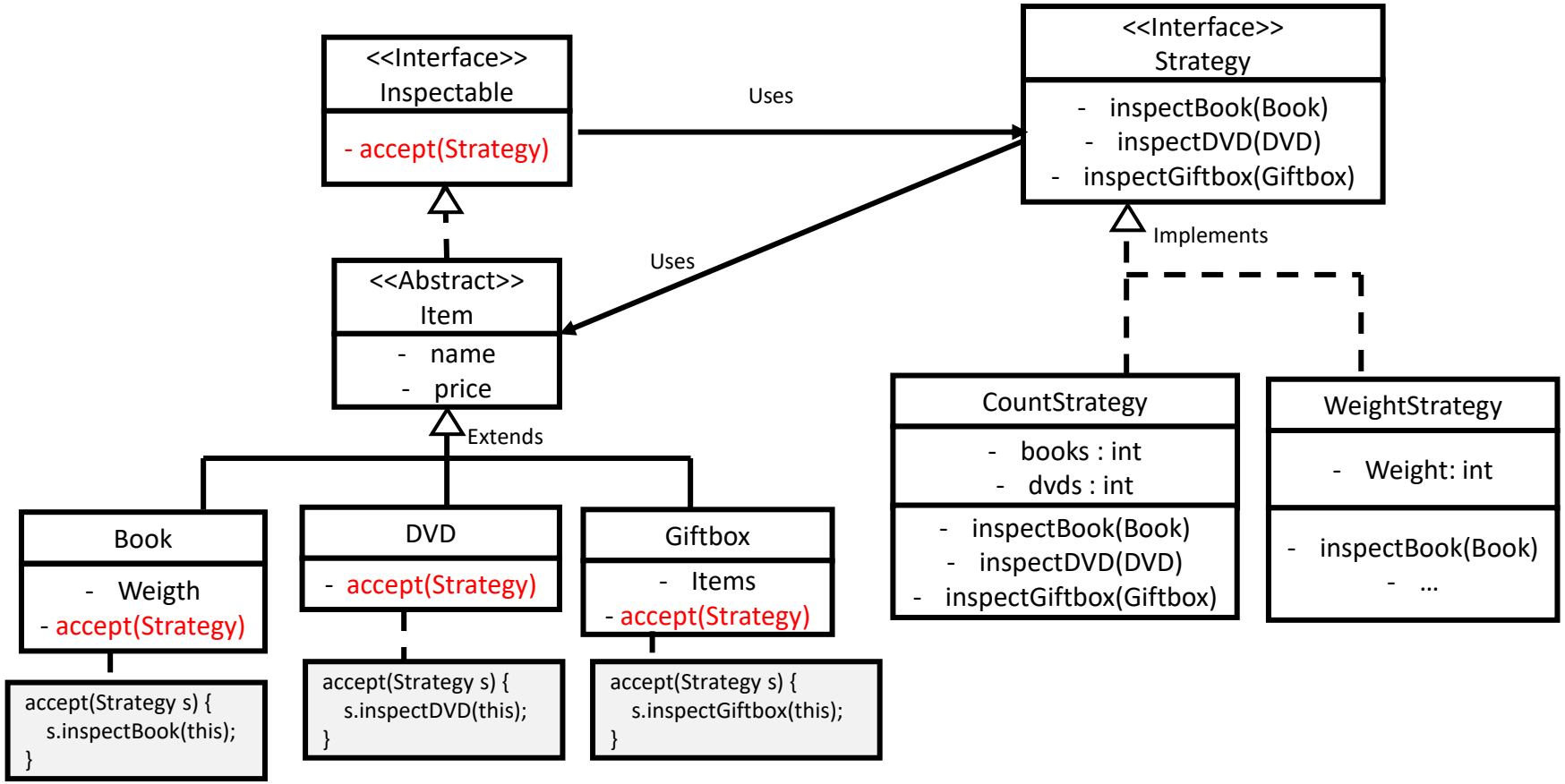
- New implementation:
- No instanceof or getClass() needed!
- Problem: using a Strategy on an Item is not abstract anymore…
- Solution: Make a new function for all items which
  - Takes a strategy as argument
  - Chooses the correct inspect function for the current Item




```
void inspectBook(Book b) {
    books += 1;
}

void inspectDVD(DVD d) {
    dvds += 1;
}

Void inspectGiftbox(Giftbox gb) {
   for (Item subItem: item.getItems()) {
        inspect(subItem);
   }
}
```

Problem IV

- Implementation:

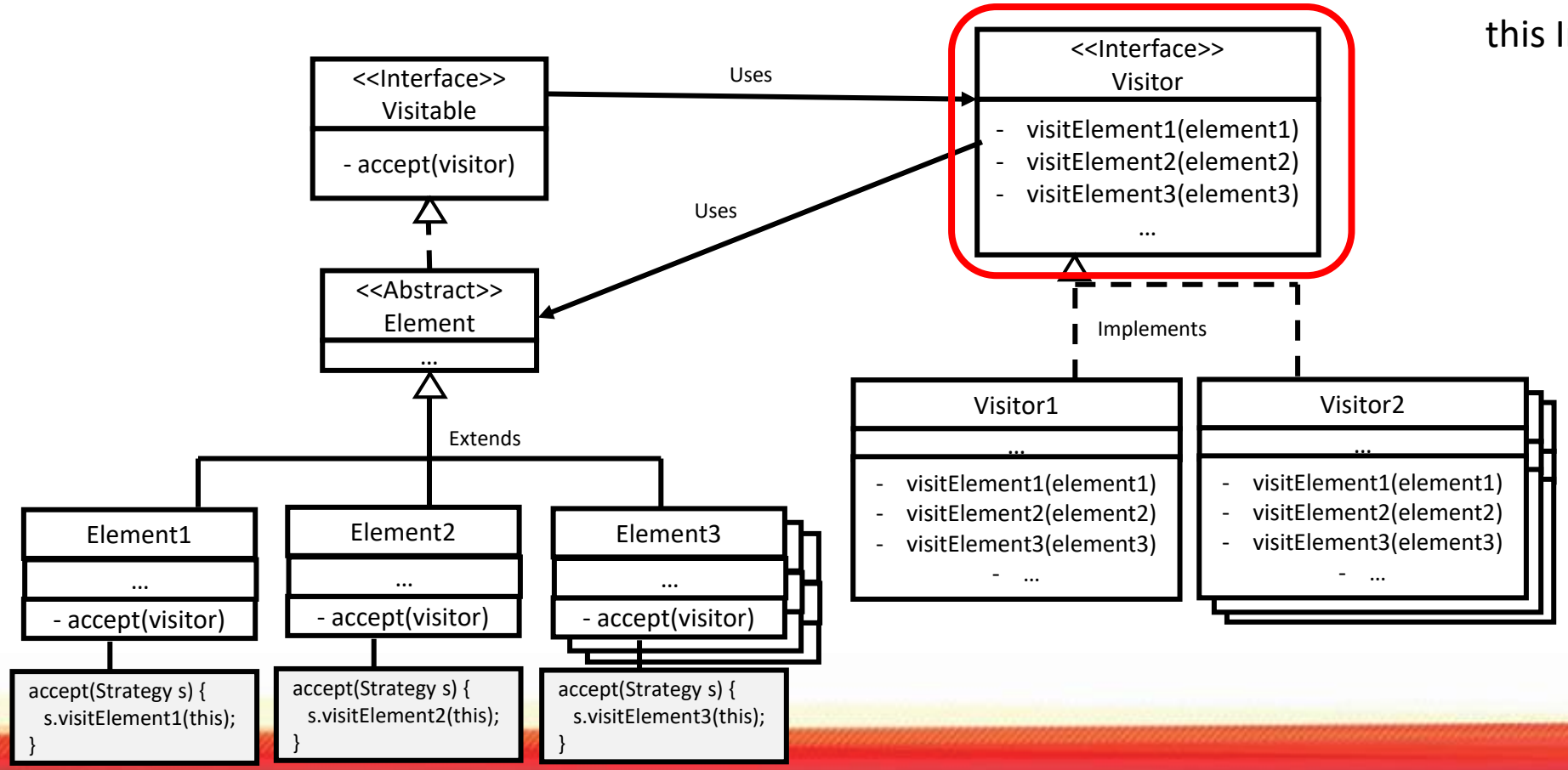


- The question probably arises: why would we want this complex pattern?
  - For every item and strategy, we can use item.accept(strategy) and it will work
  - No usage of instanceof or getClass()!
  - The Inspectable interface doesn't have to be touched anymore, but many strategies can be added!

Visitor Pattern

- Visitor Pattern is very similar to the Strategy Pattern
- But also works with multiple objects!
  - While keeping the code clean
- Objects are **Visitable** and contain the accept(strategy) function
- Strategies are **Visitors** and contain a visitObject(object) function for every Object
  - These are linked with each other
- Really useful when the Visitable Objects do not change often
- Many Visitors can be added and the code for the Visitables is unchanged!

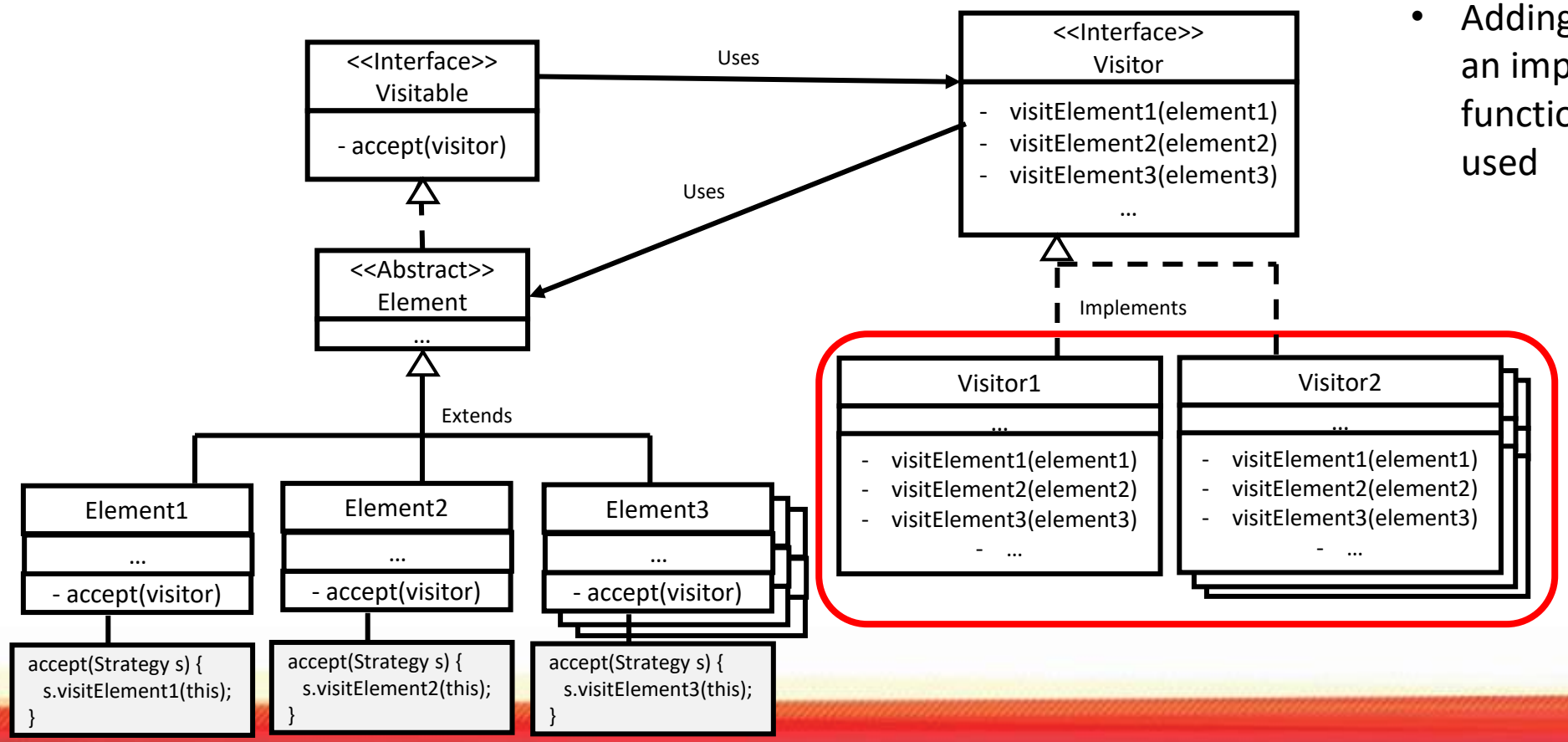
Visitor Pattern UML



<<Interface>> Visitor

- The interface for the Visitors, which contain a visit function for every element
- All the concrete Visitors implement this Interface

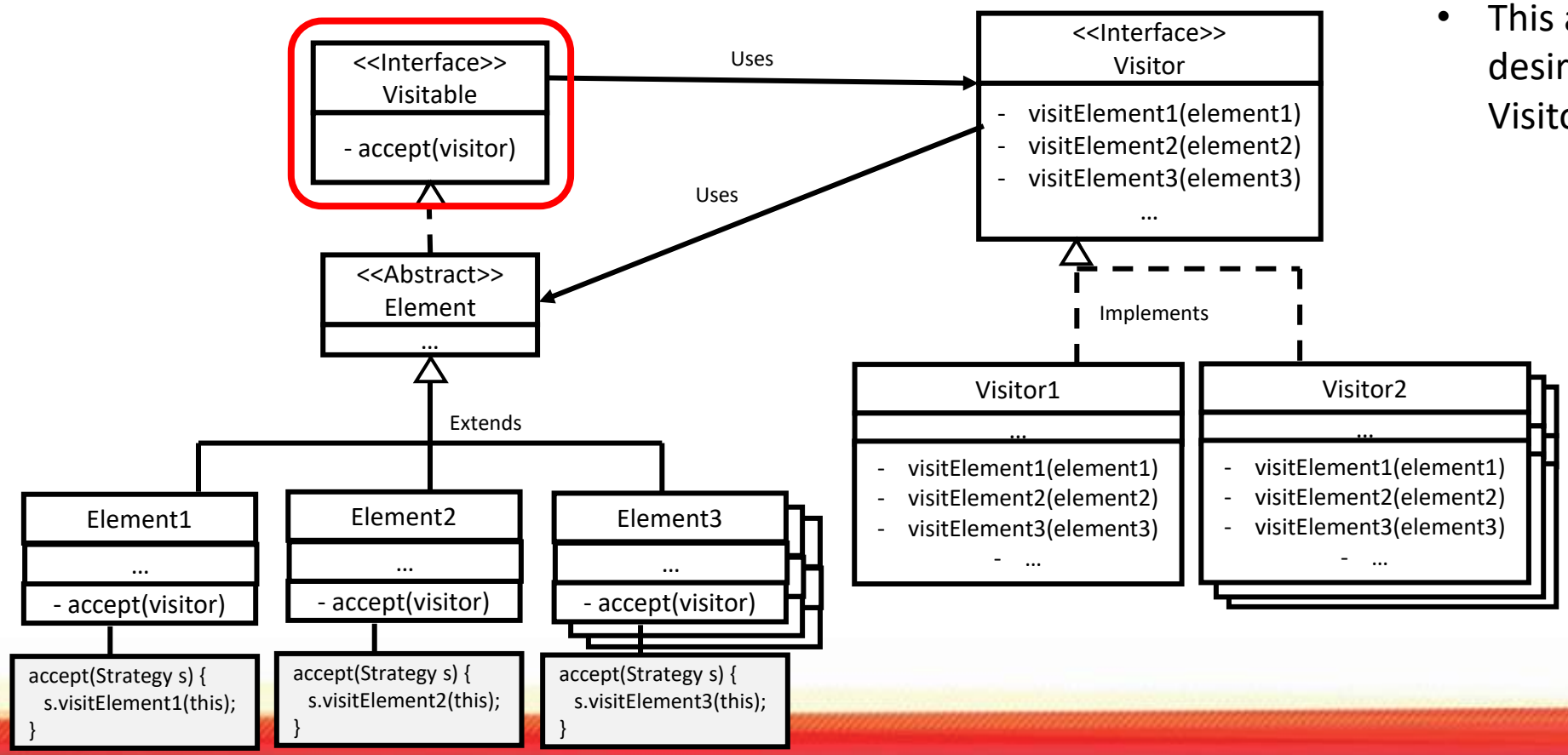
Visitor Pattern UML



Concrete Visitors

- These are the concrete Visitors with their own implementation of the visit function for all different elements
- Adding a new Visitor only requires an implementation of these visit functions and it can immediately be used

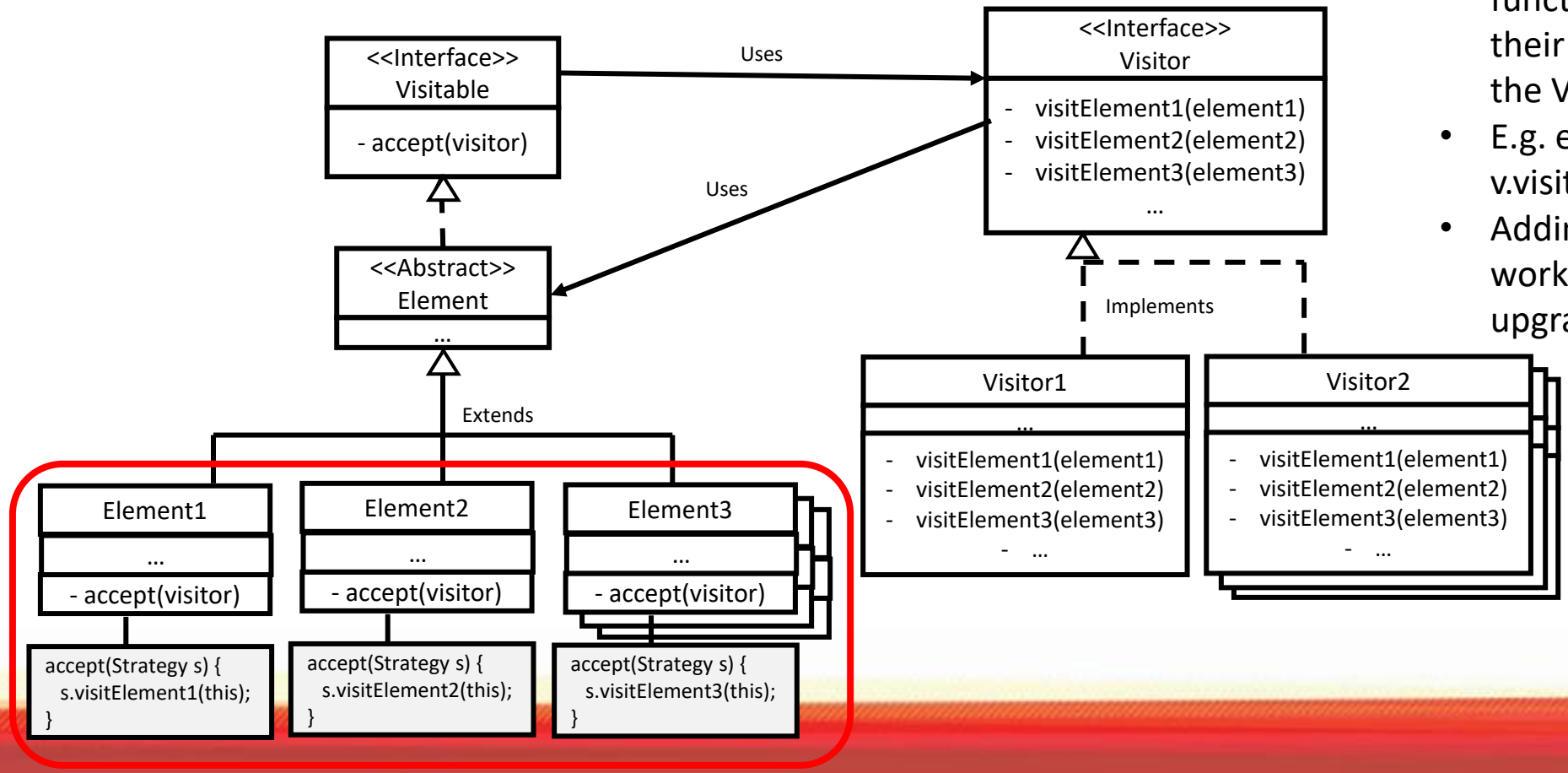
Visitor Pattern UML



<<Interface>> Visitable

- The interface for all the Visitable Elements.
- All implementations have an accept function that takes a Visitor
- This accept function executes the desired behavior from the passed Visitor on the Visitable Element

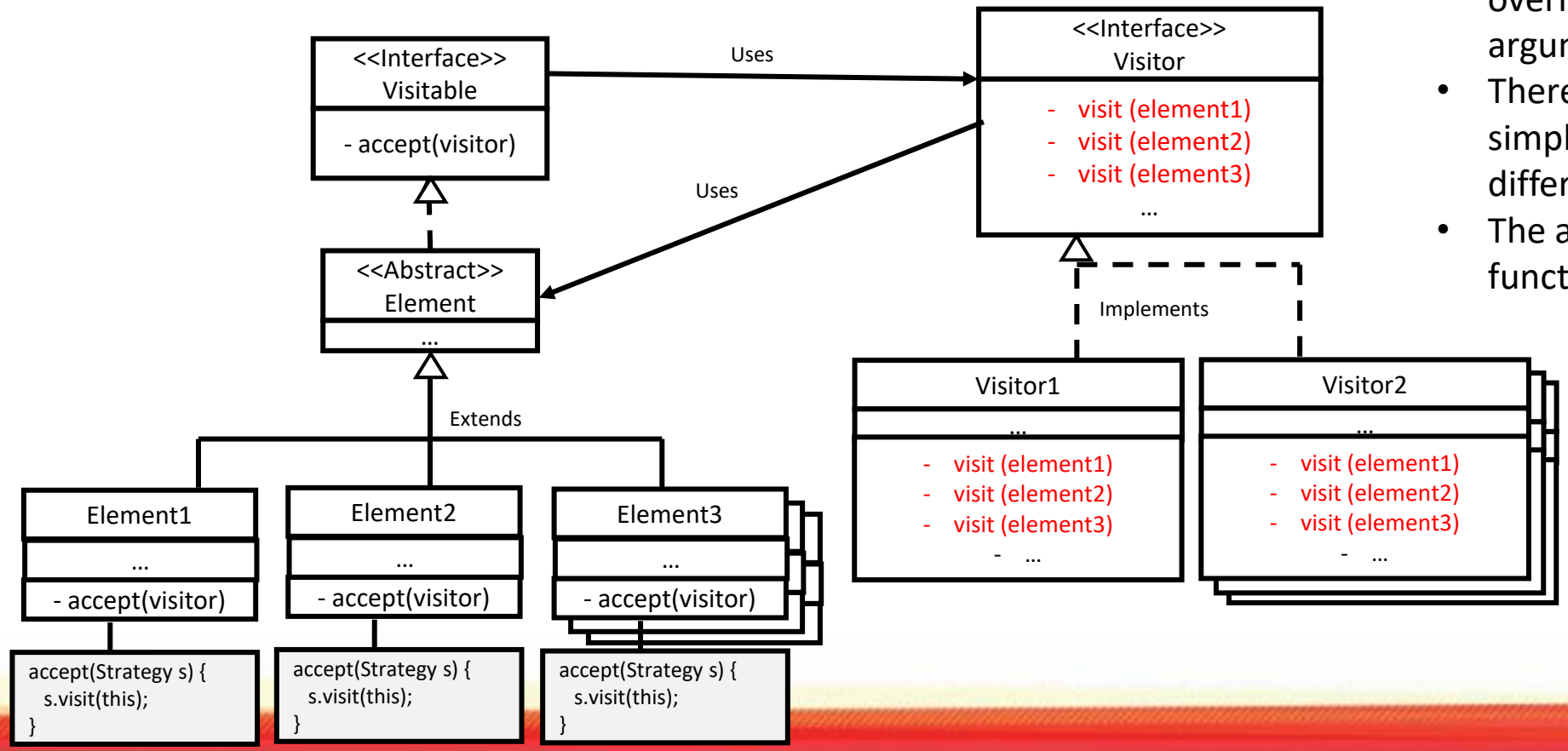
Visitor Pattern UML



Concrete Elements

- The concrete elements used in the program which extend the Abstract Element
- These all implement an accept function which is directly linked to their corresponding visit function in the Visitor
- E.g. el3.accept(visitor v) calls v.visitElement3(el3)
- Adding a new Element is a bit more work, as all Visitors should be upgraded with a new visit function

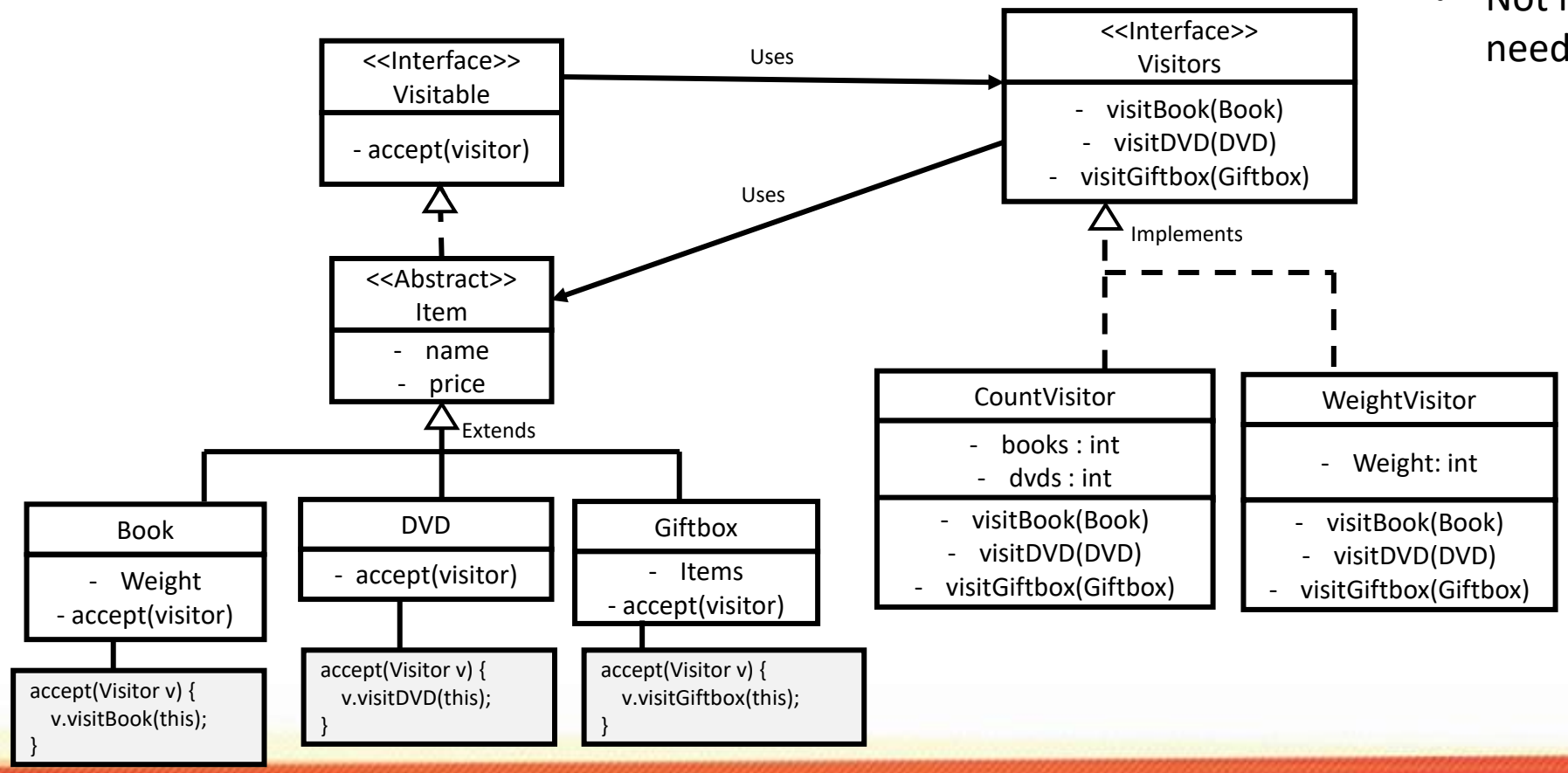
Visitor Pattern UML



Overloading

- Before, we used visitElement1(element1), visitElement2(element2) etc
- However, these functions can be overloaded, as the type of the arguments are different
- Therefore, it is common practice to simply use visit as the name of the different functions
- The accept(strategy) is the same function everywhere!

Application on previous problem



- As can be seen, for every item and every Visitor v, item.accept(v) can immediately be applied
- Not instanceof or getClass() is needed anymore in the Visitors!

Live coding session

- Lets take a look at the program from the previous example
- This can be found in VisitorPatternExample.zip
- Take a look at it in your own time as well and experiment!

Question 13

Suppose we have a program for an electrical business that sells keyboards, mouses, desktops etc and a complete set of these attributes. Different operations can be performed on these objects, e.g. getting the price, the operation system requirements etc.

Should we use the Visitor pattern and how?

A. Yes, the products are the Visitables and the operations the Visitors

B. Yes, the operations are the Visitables and the products the Visitables

C. No, it is not necessary to apply the pattern

Question 13

Suppose we have a program for an electrical business that sells keyboards, mouses, desktops etc and a complete set of these attributes. Different operations can be performed on these objects, e.g. getting the price, the operation system requirements etc.

Should we use the Visitor pattern and how?

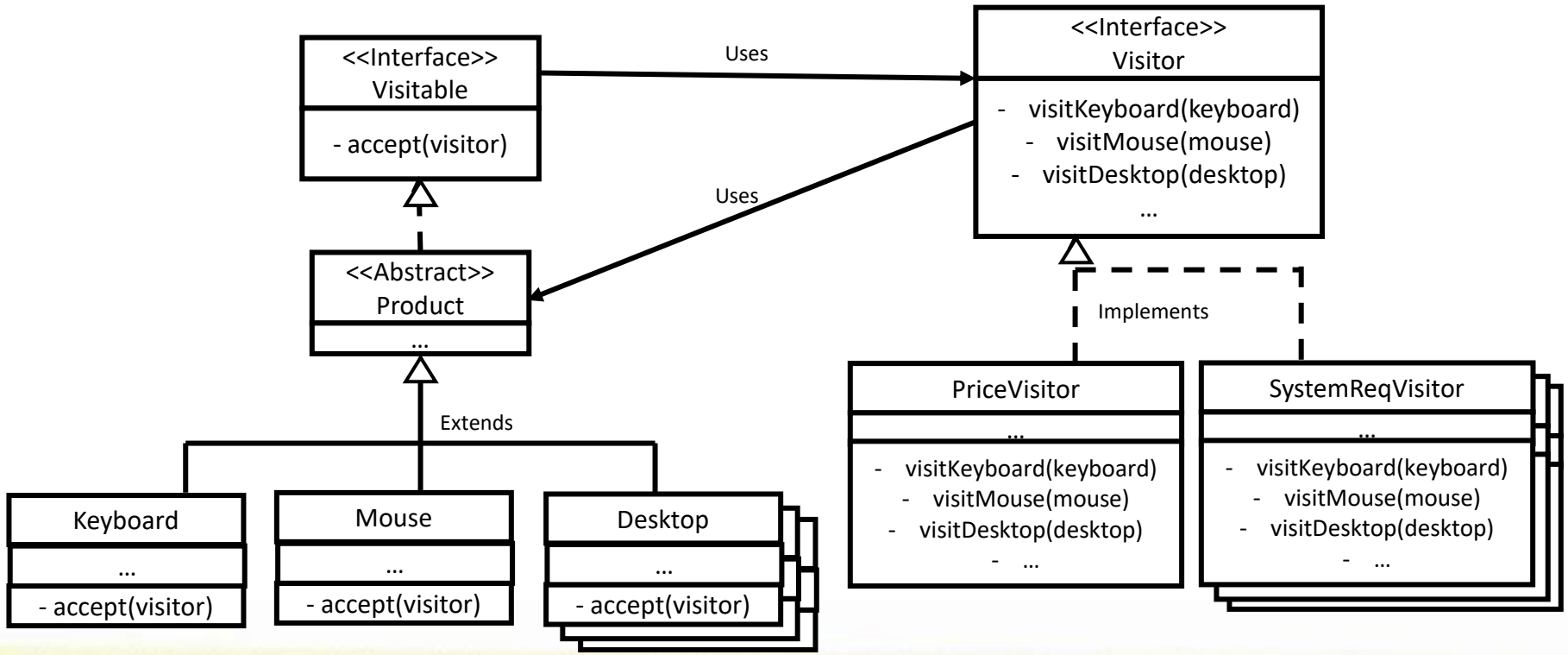
A. Yes, the products are the Visitables and the operations the Visitors

B. Yes, the operations are the Visitables and the products the Visitables

C. No, it is not necessary to apply the pattern

The Visitor pattern fits perfectly in this use case, see the next slide

Question 13


<<Interface>>
Visitable
- accept(visitor)
Uses
<<Interface>>
Visitor
- visitKeyboard(keyboard)
- visitMouse(mouse)
- visitDesktop(desktop)
...
Uses
<<Abstract>>
Product
...
Implements
Extends
Keyboard
...
- accept(visitor)
Mouse
...
- accept(visitor)
Desktop
...
- accept(visitor)
PriceVisitor
...
- visitKeyboard(keyboard)
- visitMouse(mouse)
- visitDesktop(desktop)
- ...
SystemReqVisitor
...
- visitKeyboard(keyboard)
- visitMouse(mouse)
- visitDesktop(desktop)
- ...

Question 14

Suppose we have a shop that sells bikes. All these bikes are in essence the same, but they differ in their attributes, e.g. different height, different tires, different colors. Several operations should be applied on these bikes, e.g. calculating the production cost or the total weight of the bike.

Should we use the Visitor pattern and how?

A. Yes, the bikes are the Visitables and the operations the Visitors

B. Yes, the operations are the Visitables and the bikes the Visitables

C. No, it is not necessary to apply the pattern

Question 14

Suppose we have a shop that sells bikes. All these bikes are in essence the same, but they differ in their attributes, e.g. different height, different tires, different colors. Several operations should be applied on these bikes, e.g. calculating the production cost or the total weight of the bike.

Should we use the Visitor pattern and how?

A. Yes, the bikes are the Visitables and the operations the Visitors

B. Yes, the operations are the Visitables and the bikes the Visitables

C. No, it is not necessary to apply the pattern

There is only one Visitable object, namely the bike which has multiple attributes. The Visitor pattern is not useful here, as the operations that should be performed on the bike should be implemented in that class. See the next slide

Question 14


Bike
- height : int
- tireType: Tire
- colorType: Color
…
- calculateCost ()
- calculateWeight ()
…

Question 15

Suppose we have a map with streets, roundabouts, bridges etc. Multiple calculations can be performed on these roads, e.g. calculating the length of a route or the fuel consumption. Other calculations could be added to the program, and different maps can be made and will extend the different road objects with roads specifically for bikes or trains for example.

Should we use the Visitor pattern and how?

A. Yes, the road objects are the Visitables and the calculations the Visitors

B. Yes, the calculations are the Visitables and the road objects the Visitors

C. No, it is not necessary to apply the pattern

Question 15

Suppose we have a map with streets, roundabouts, bridges etc. Multiple calculations can be performed on these roads, e.g. calculating the length of a route or the fuel consumption. Other calculations could be added to the program, and different maps can be made and will extend the different road objects with roads specifically for bikes or trains for example.

Should we use the Visitor pattern and how?

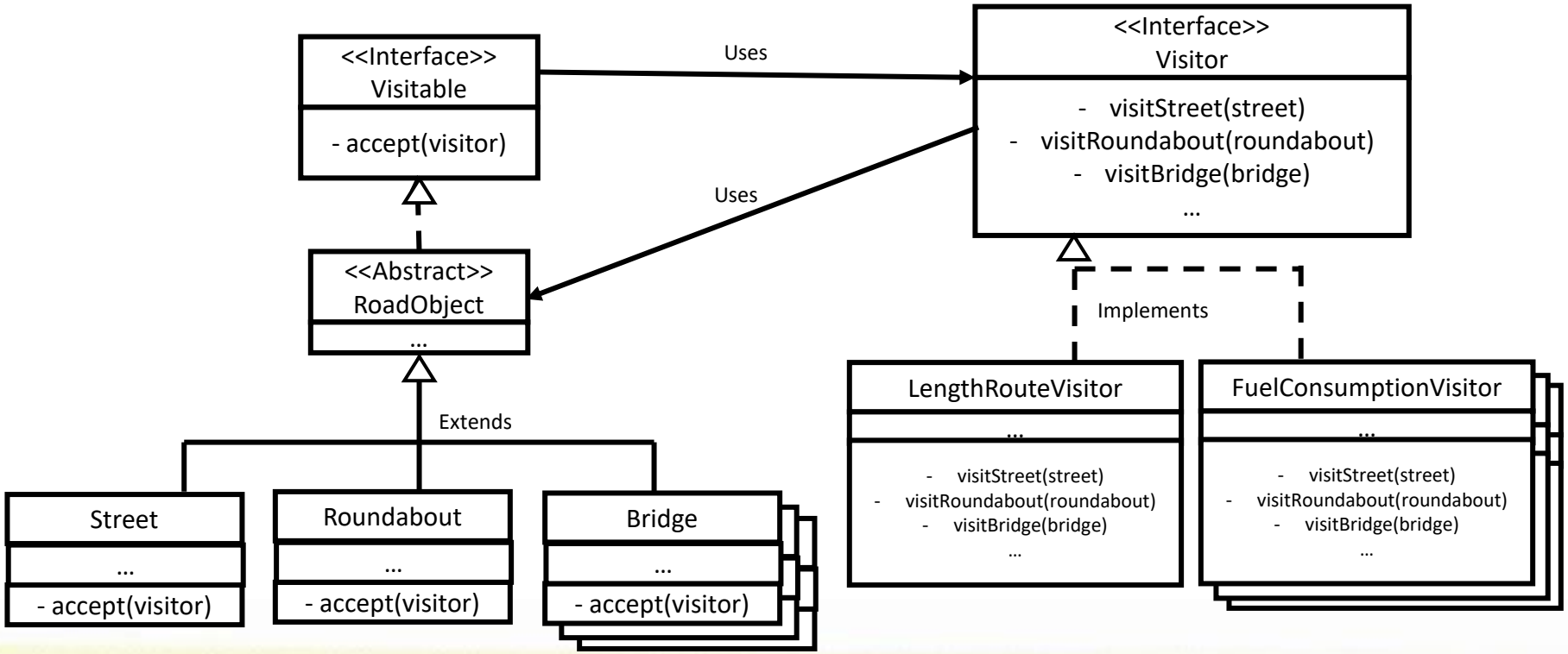
A. Yes, the road objects are the Visitables and the calculations the Visitors

B. Yes, the calculations are the Visitables and the road objects the Visitors

C. No, it is not necessary to apply the pattern

The Visitor pattern can be applied here. Note that it is a bit more work to add new road objects, but the pattern is still viable. See the next slide

Question 15


<<Interface>>
Visitable
- accept(visitor)
Uses
<<Interface>>
Visitor
- visitStreet(street)
- visitRoundabout(roundabout)
- visitBridge(bridge)
...
Uses
<<Abstract>>
RoadObject
...
Implements
Extends
Street
...
- accept(visitor)
Roundabout
...
- accept(visitor)
Bridge
...
- accept(visitor)
LengthRouteVisitor
...
- visitStreet(street)
- visitRoundabout(roundabout)
- visitBridge(bridge)
...
FuelConsumptionVisitor
...
- visitStreet(street)
- visitRoundabout(roundabout)
- visitBridge(bridge)
...

Recap

- The Visitor pattern can be used to split up several algorithms from a lot of different Elements with different properties
  - A lot of different problems fall into this category!
- It puts the Elements in a Visitable Interface and the algorithms in a Visitor interface
  - The visitors (algorithms) can visit the Visitables (Element)
- With this pattern, one can easily
  - Add new Visitors
- Note that it a bit more work to add new Elements
  - This pattern is used best when the Elements do not change a lot

Recognising Patterns

Recognising Patterns

- You have seen different patterns
  - But there are many more!
  - You probably used a lot of these subconsciously
- Learning these patterns by heart is not the goal
- Recognising reappearing structures in a program is the goal!
  - And we named these “reappearing structures”
- Using the design patterns benefits your code
  - Better readability
  - Better maintainability
- However, do not overuse (or misuse) these patterns if not necessary!

Question 16

In Brightspace, there are different roles: teachers, students, TA's etc. Each TA can have different rights in Brightspace; some can grade stuff, some can add material, some can only view all material. These different rights can be combined with each other. The teacher can assign rights to each individual TA.

Which design pattern can be recognised and applied here?

A. Strategy Pattern

B. Decorator Pattern

C. Visitor Pattern

D. No pattern or undiscussed pattern

Question 16

In Brightspace, there are different roles: teachers, students, TA's etc. Each TA can have different rights in Brightspace; some can grade stuff, some can add material, some can only view all material. These different rights can be combined with each other. The teacher can assign rights to each individual TA.

Which design pattern can be recognised and applied here?

A. Strategy Pattern

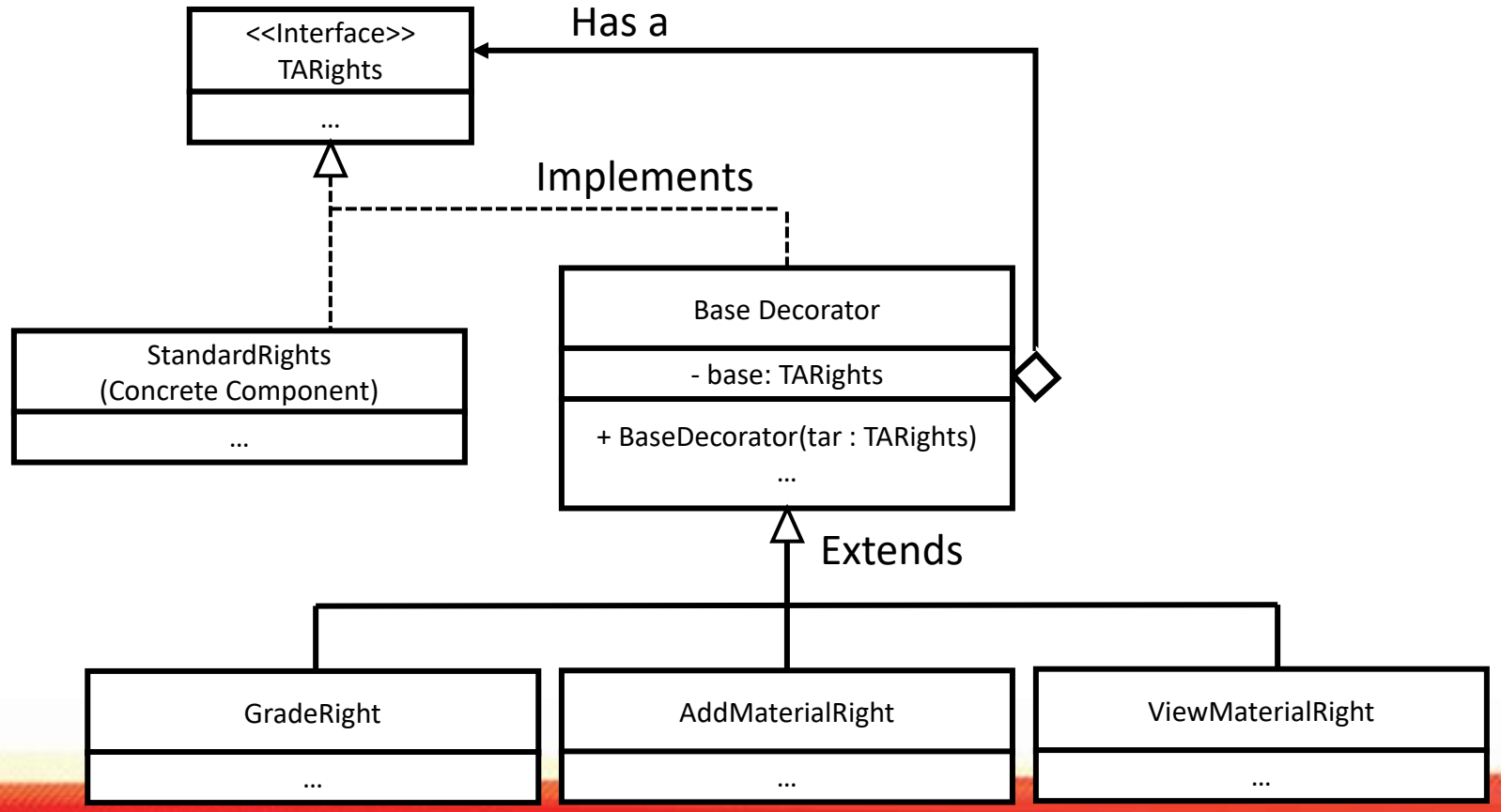
B. Decorator Pattern

C. Visitor Pattern

D. No pattern or undiscussed pattern

The standard rights for TA's is the Concrete Component and the other different rights which are not standard are the Decorators. It is also possible to implement the different rights as Boolean attributes. See the next slide

Question 16


<<Interface>>
TARights
...
Has a
Implements
StandardRights
(Concrete Component)
...
Base Decorator
- base: TARights
+ BaseDecorator(tar : TARights)
...
Extends
GradeRight
...
AddMaterialRight
...
ViewMaterialRight
...

Question 17

In Brightspace, we have different views for a course page, e.g. students can only see material that is published and TA's can see all material, but cannot see the menu for adding or removing material. Teachers can see everything, including this menu. We are looking at the part of the program that shows these different views/interfaces on the screen.

Which design pattern can be recognised and applied here?

A. Strategy Pattern

B. Decorator Pattern

C. Visitor Pattern

D. No pattern or undiscussed pattern

Question 17

In Brightspace, we have different views for a course page, e.g. students can only see material that is published and TA's can see all material, but cannot see the menu for adding or removing material. Teachers can see everything, including this menu. We are looking at the part of the program that shows these different views/interfaces on the screen.

Which design pattern can be recognised and applied here?

A. Strategy Pattern

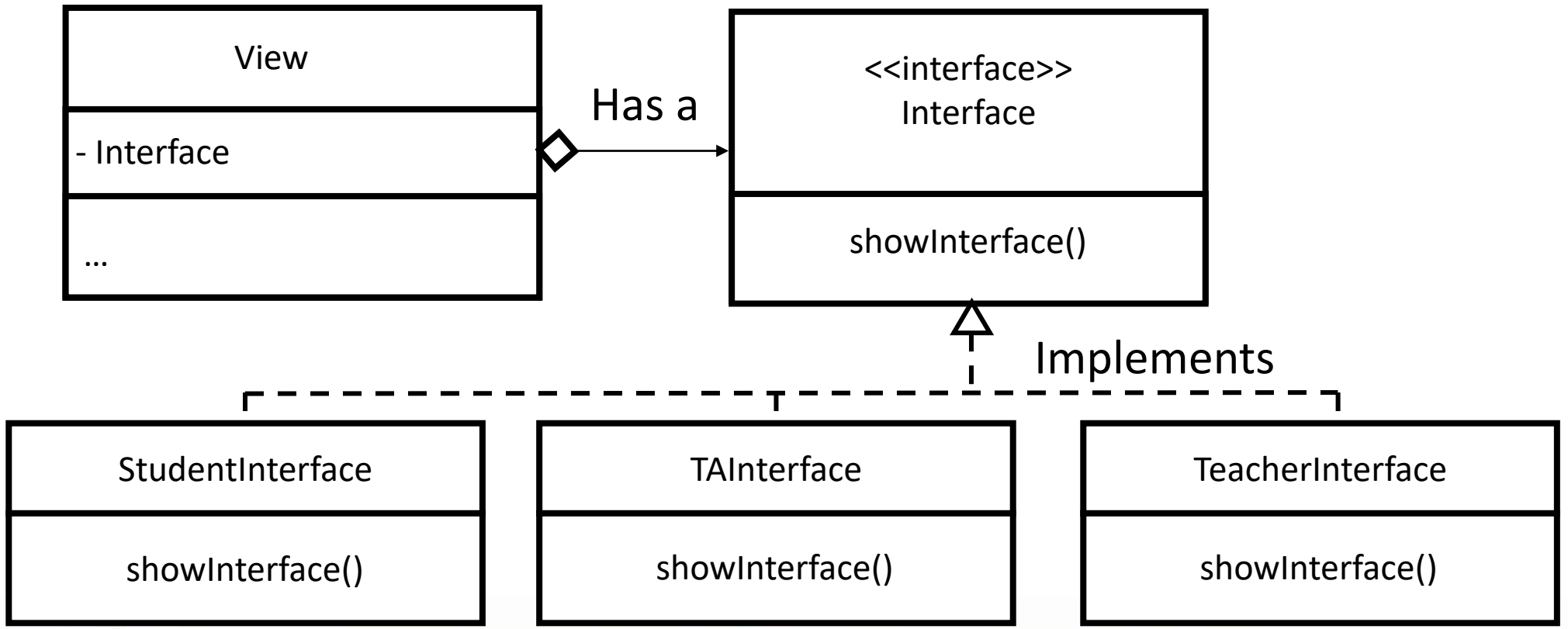
B. Decorator Pattern

C. Visitor Pattern

D. No pattern or undiscussed pattern

These different views can be seen as different strategies, which results in an application of the strategy pattern. See the next slide.

Question 17


View
- Interface
...
Has a
<<interface>>
Interface
showInterface()
Implements
StudentInterface
showInterface()
TAInterface
showInterface()
TeacherInterface
showInterface()

Question 18

In Brightspace, students should be notified when grades are published or the teacher places an announcement for all students. The other way around, the teacher should receive a message on Brightspace whenever a students asks something in a discussion thread or hands in an assignment.

Which design pattern can be recognised and applied here?

A. Strategy Pattern

B. Decorator Pattern

C. Visitor Pattern

D. No pattern or undiscussed pattern

Question 18

In Brightspace, students should be notified when grades are published or the teacher places an announcement for all students. The other way around, the teacher should receive a message on Brightspace whenever a students asks something in a discussion thread or hands in an assignment.

Which design pattern can be recognised and applied here?

A. Strategy Pattern

B. Decorator Pattern

C. Visitor Pattern

D. No pattern or undiscussed pattern

No pattern that we discussed in this lecture can be applied here, but this is a typical example of the Observer Pattern, which you probably already used in different programs. This Pattern waits till something specific happens (grades are published) and then executes a certain action (notify the students).

Recap

- We discussed 3 concrete design patterns
  - Strategy pattern
  - Decorator pattern
  - Visitor pattern
- For each pattern, we saw
  - How it works
  - An implementation in Java
  - The use cases
- In your assignment, you have to
  - Recognise a pattern in given code and refactor this code
  - Implement the visitor pattern

Questionnaire

- Please fill in this questionnaire for my research (takes at most 2 minutes, but it really helps me to research/improve education)
- Also if you are watching the recording or reading the slides!

References for the specific patterns

Strategy Pattern:

https://refactoring.guru/design-patterns/strategy

Decorator Pattern:

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

Visitor Pattern:

https://refactoring.guru/design-patterns/visitor

I used these sites as my main reference for the examples. They are very clear and easy to follow.

References for design patterns

Clear and simple explanations of patterns:

https://refactoring.guru/design-patterns/catalog

Short explanations of the patterns with one simple, elaborated example:

https://www.tutorialspoint.com/design_pattern/design_pattern_overview.htm

A bit longer and in-depth explanations of patterns:

https://www.oodesign.com/

Several design patterns discussed with a real-life example:

https://ronnieschaniel.medium.com/object-oriented-design-patterns-explained-using-practical-examples-84807445b092