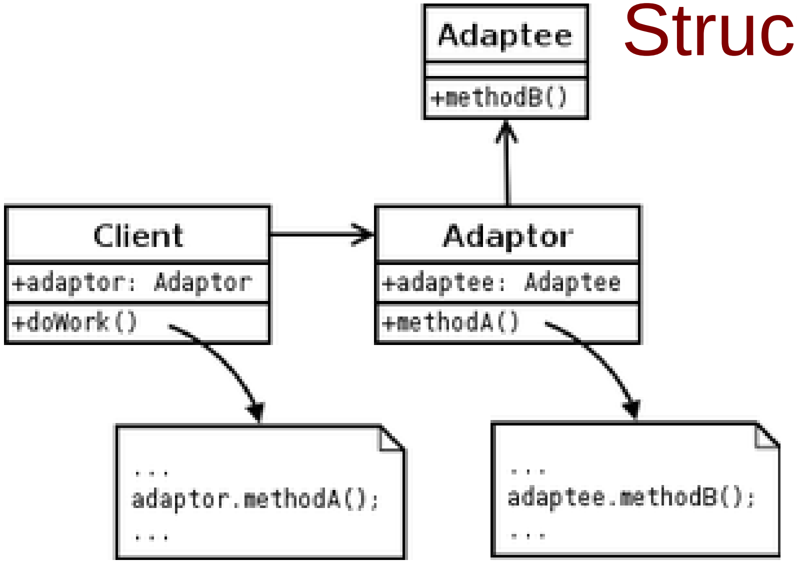
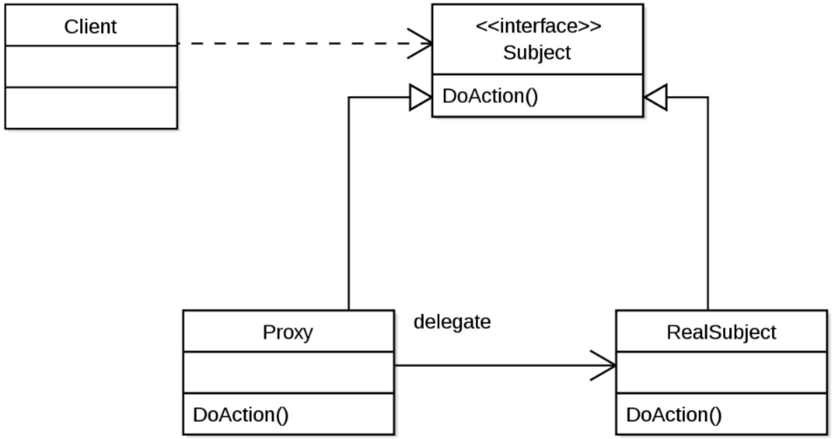
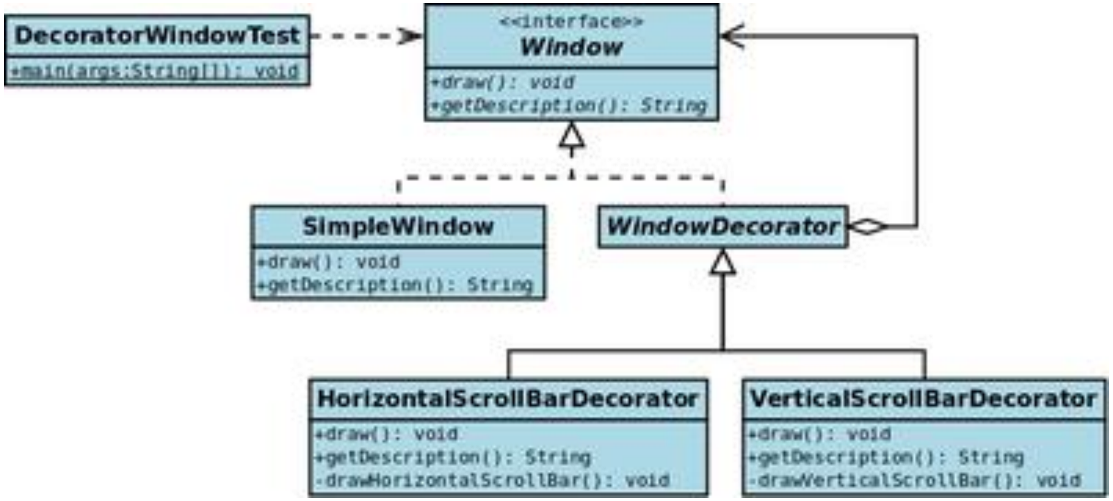
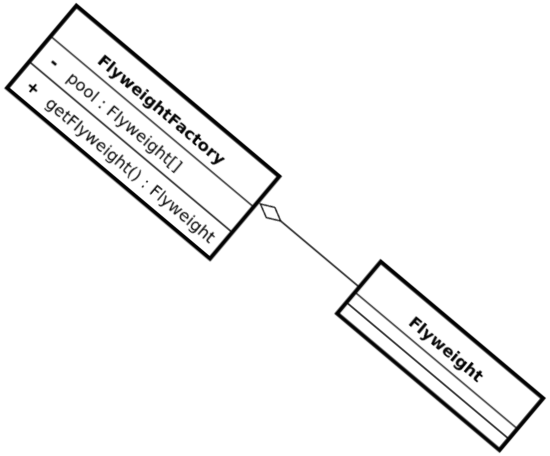
Software Design Patterns: Structural Patterns






DecoratorWindowTest
+main(args:String[]): void
<<interface>>
Window
+draw(): void
+getDescription(): String
SimpleWindow
+draw(): void
+getDescription(): String
WindowDecorator
HorizontalScrollBarDecorator
+draw(): void
+getDescription(): String
-drawHorizontalScrollBar(): void
VerticalScrollBarDecorator
+draw(): void
+getDescription(): String
-drawVerticalScrollBar(): void



Structural Design Patterns:

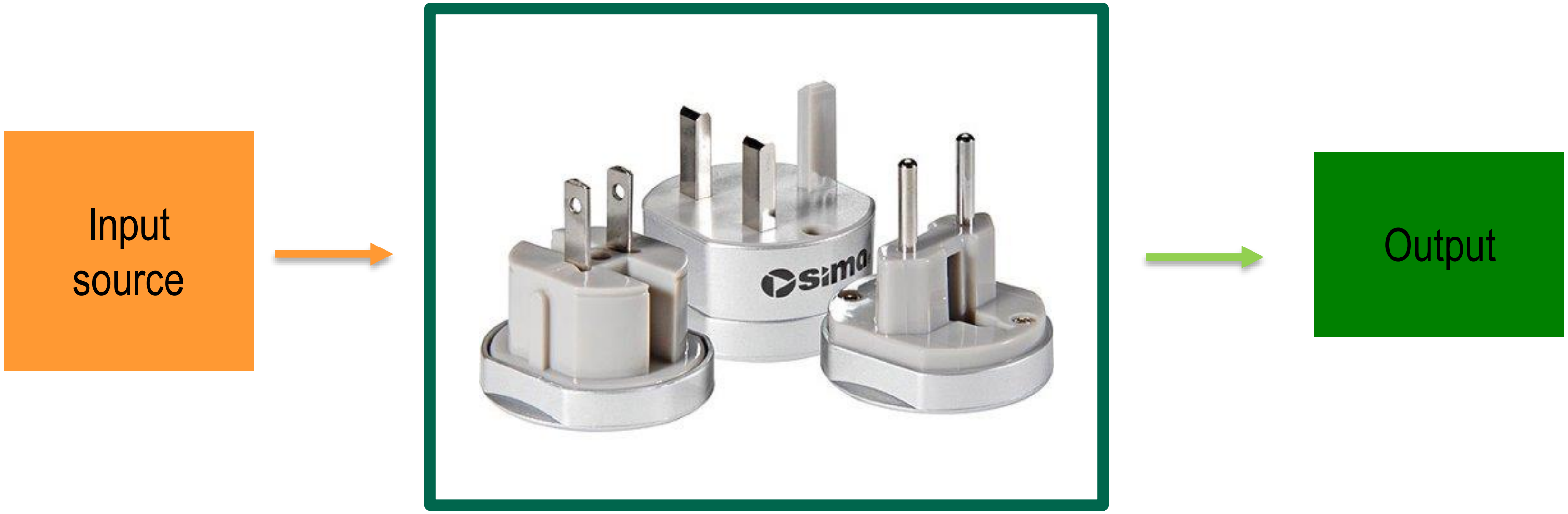
as defined in Elements of Reusable OO Software

- Structural patterns are concerned with how classes and objects are composed to form larger structures.
  - Structural *class* patterns use inheritance to compose interfaces or implementations. An example of this is multiple inheritance, where a derived class has the combined properties of its parent or base classes.
  - Structural ***object* patterns** describe ways to compose to objects to realize new functionality. The added flexibility here is that you can change the composition at run time as opposed to a static class composition.

This will be our focus!

Adapter Pattern

Intent*:* Convert an interface of a class into another interface clients expect. *Adapter lets classes work together that couldn't otherwise because of* ***incompatible*** *interfaces*.

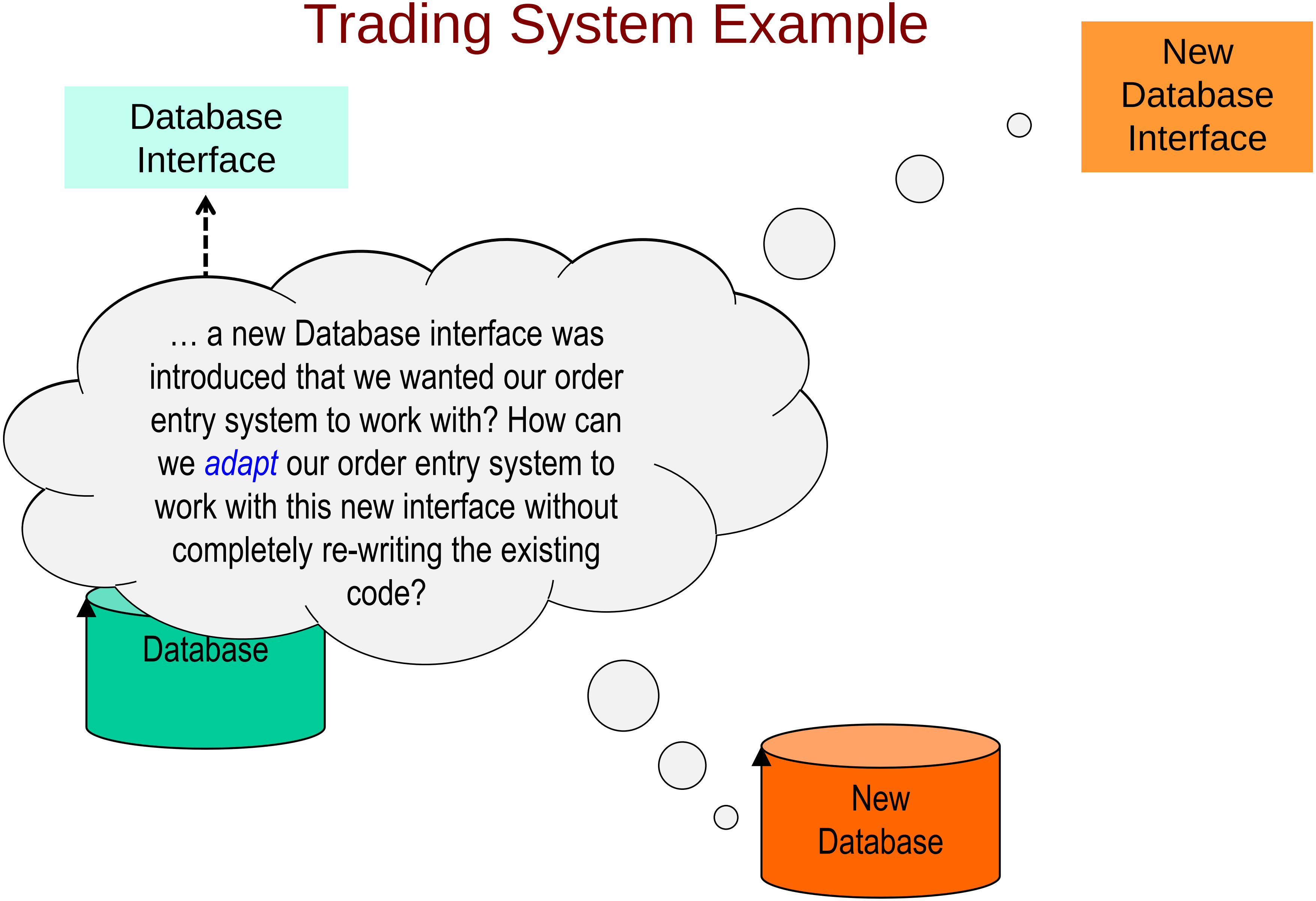


Input source

Output

Outlet adapters

Trading System Example



Database Interface

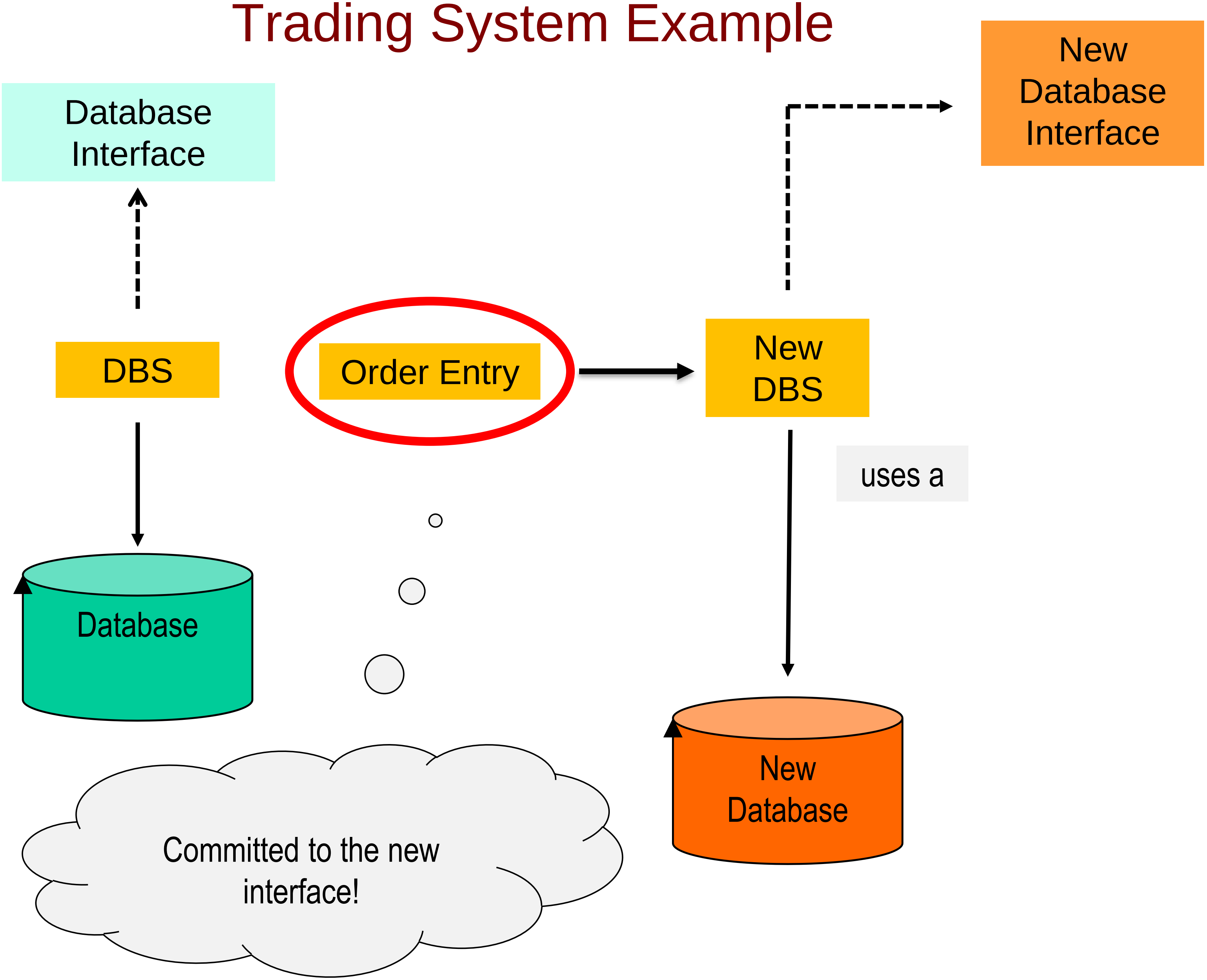
New Database Interface

… a new Database interface was introduced that we wanted our order entry system to work with? How can we *adapt* our order entry system to work with this new interface without completely re-writing the existing code?

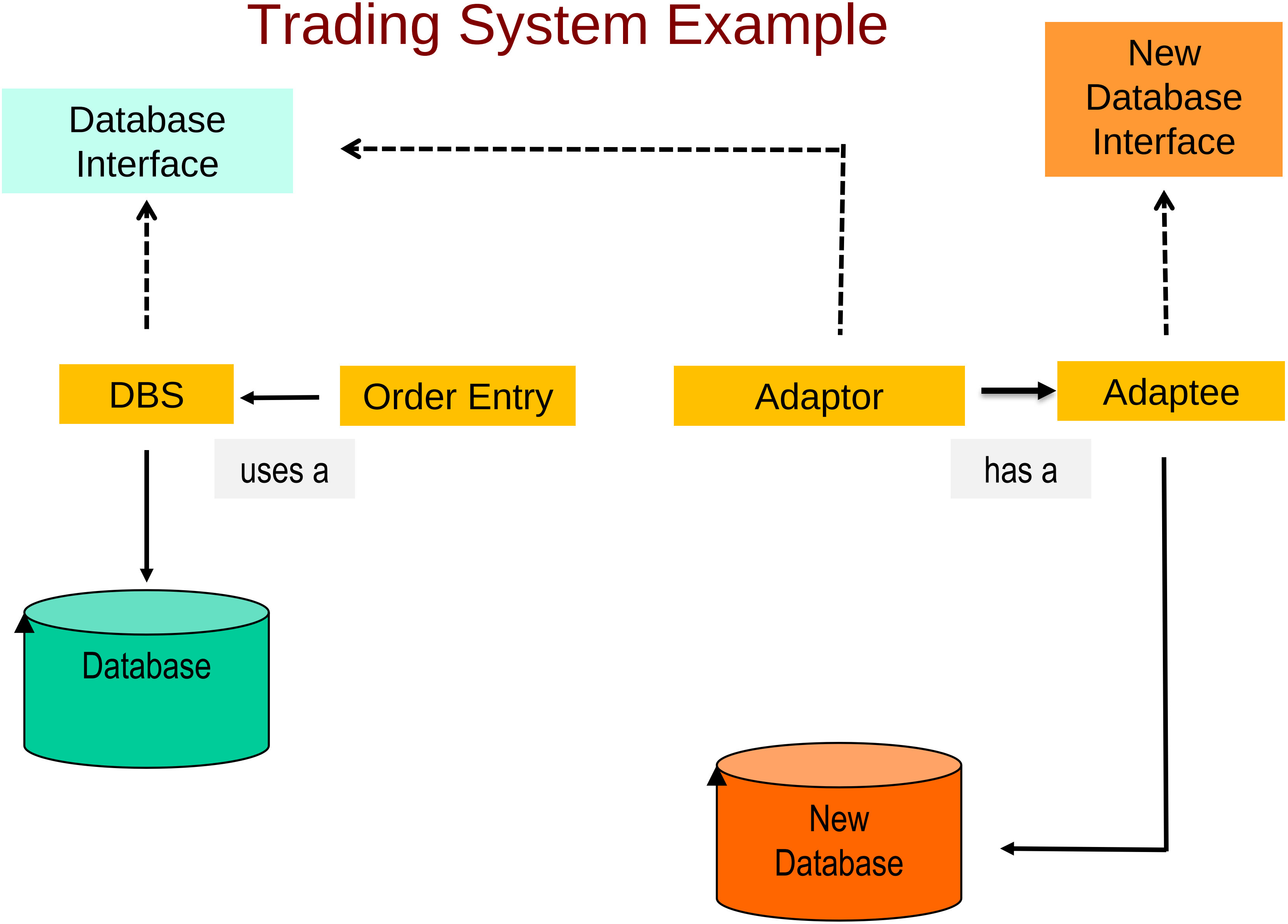
Database

New Database

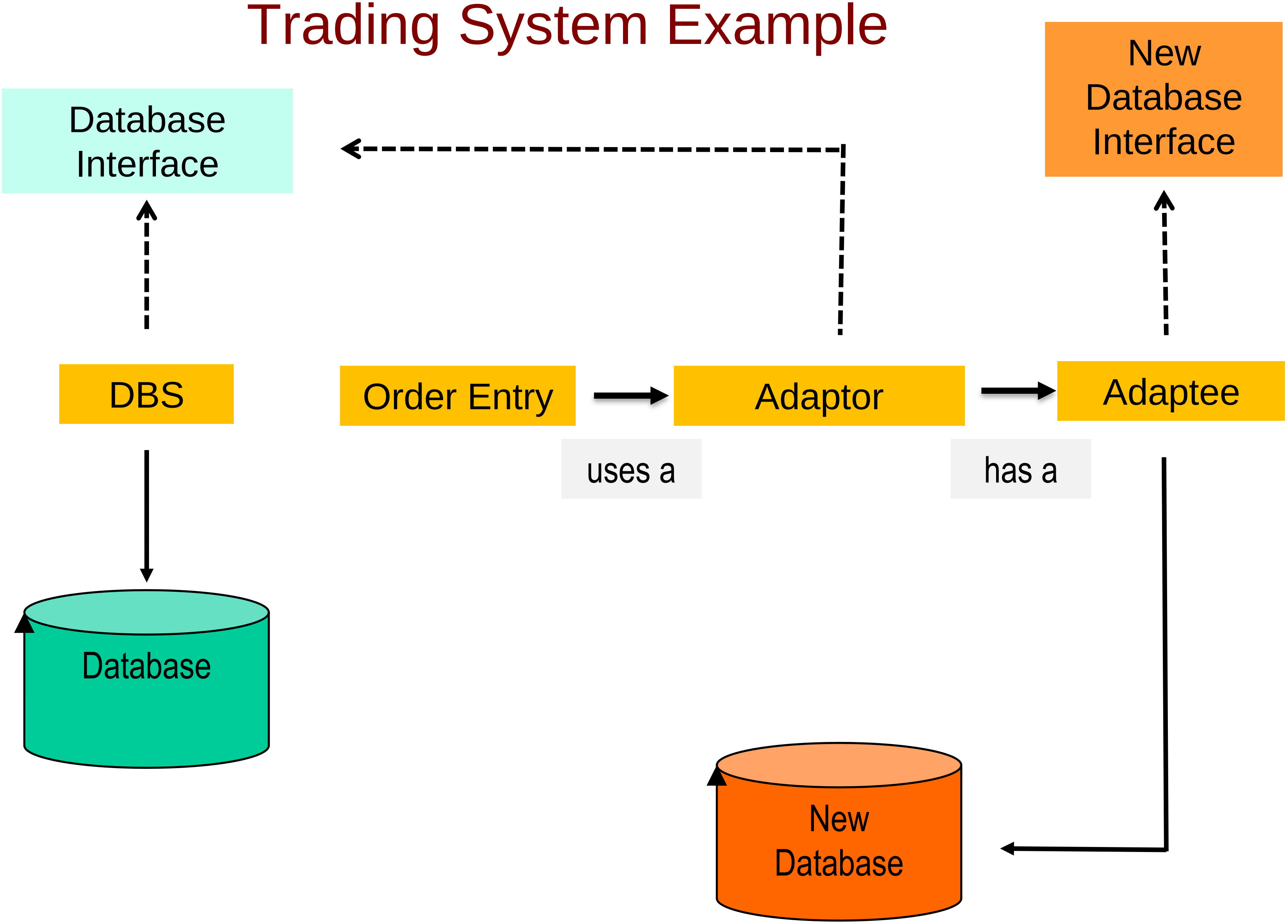
Trading System Example



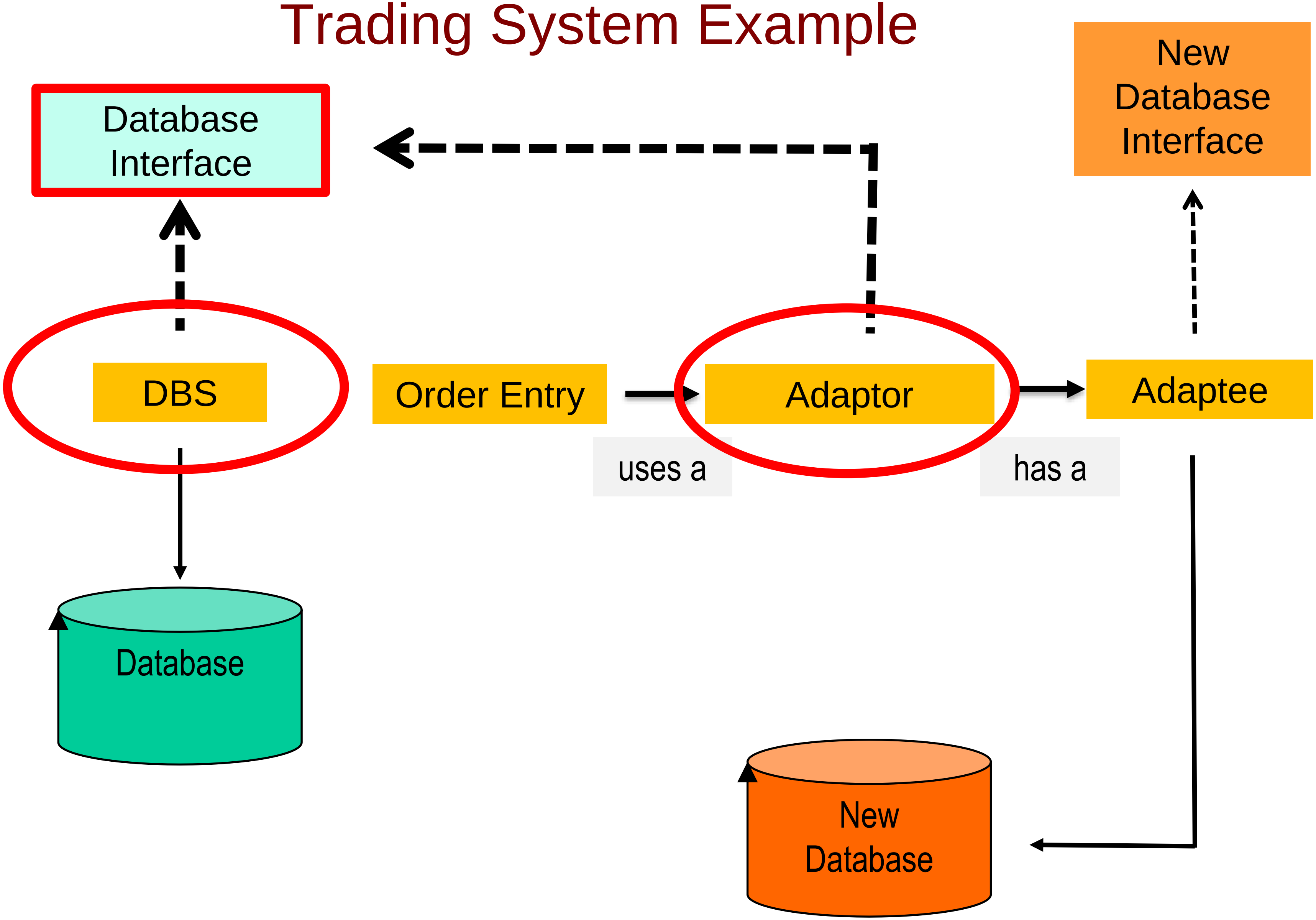
Trading System Example



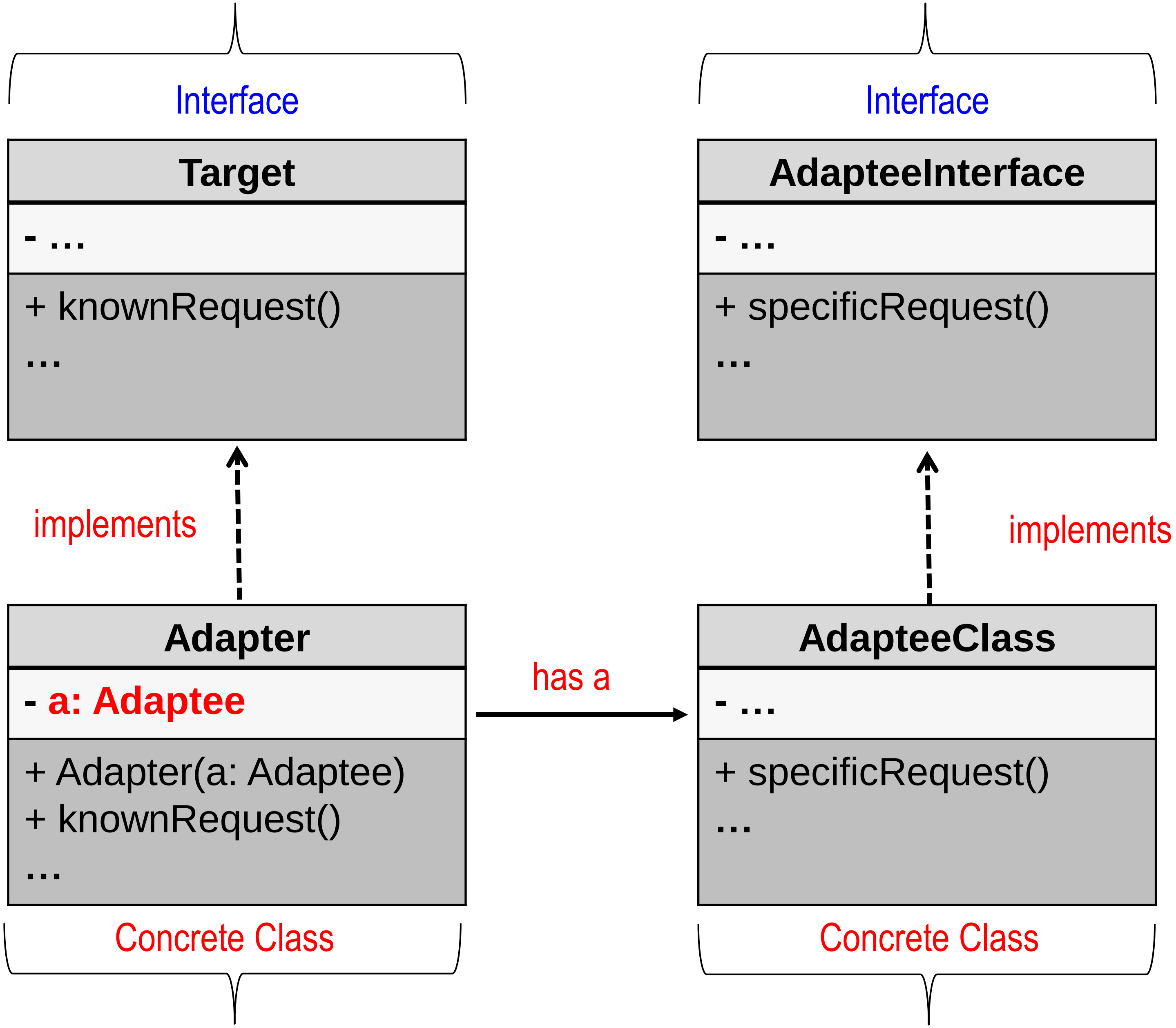
Trading System Example



Trading System Example



Adapter Pattern



Interface

Target

- …

+ knownRequest()

…

Interface

AdapteeInterface

- …

+ specificRequest()

…

implements

implements

Adapter

- a: Adaptee

+ Adapter(a: Adaptee)

+ knownRequest()

…

has a

AdapteeClass

- …

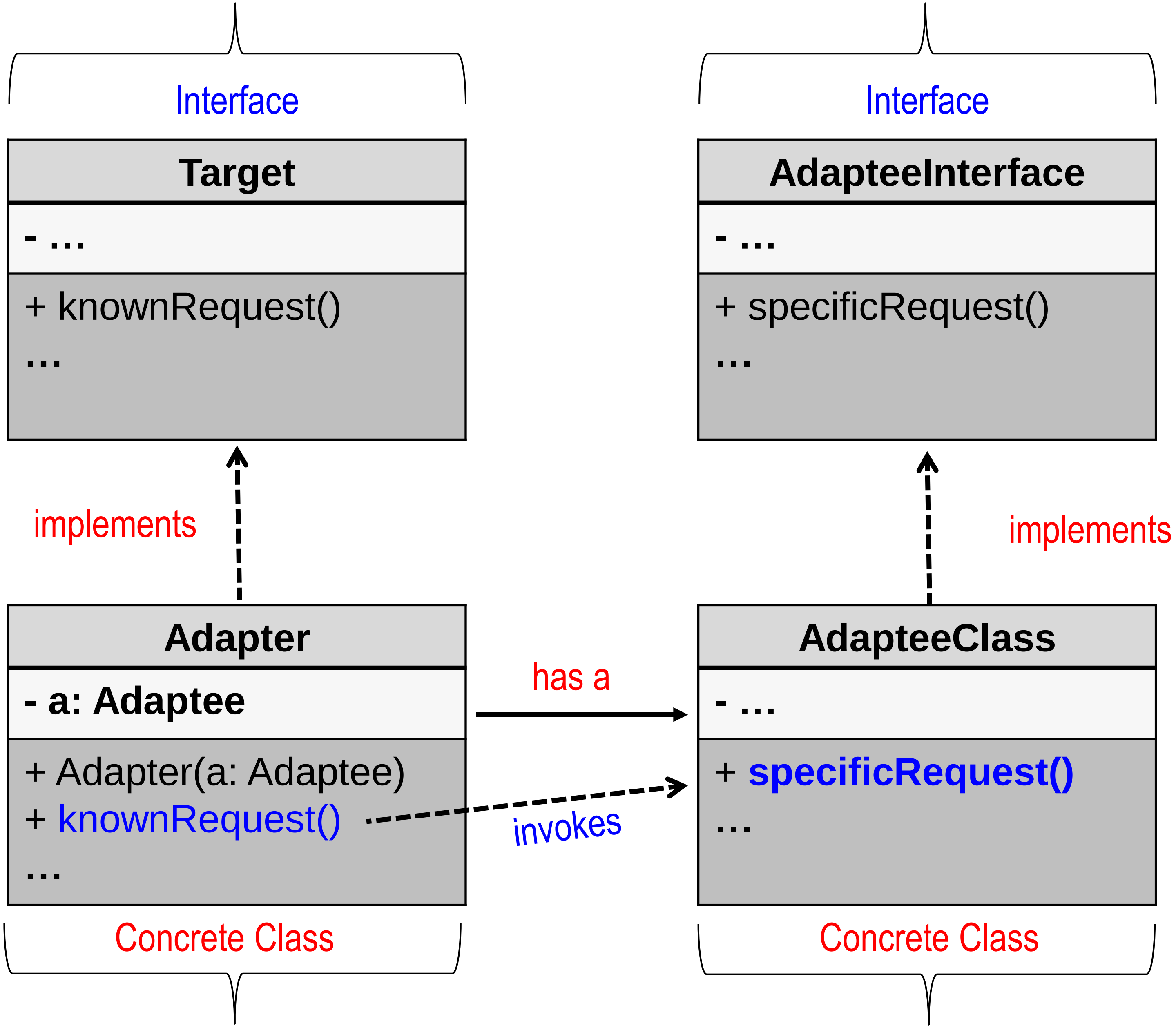
+ specificRequest()

…

Concrete Class

Concrete Class

Adapter Pattern



Interface

Target

- ...

+ knownRequest()
...

Interface

AdapteeInterface

- ...

+ specificRequest()
...

implements

implements

Adapter

- a: Adaptee

+ Adapter(a: Adaptee)
+ knownRequest()
...

has a

invokes

AdapteeClass

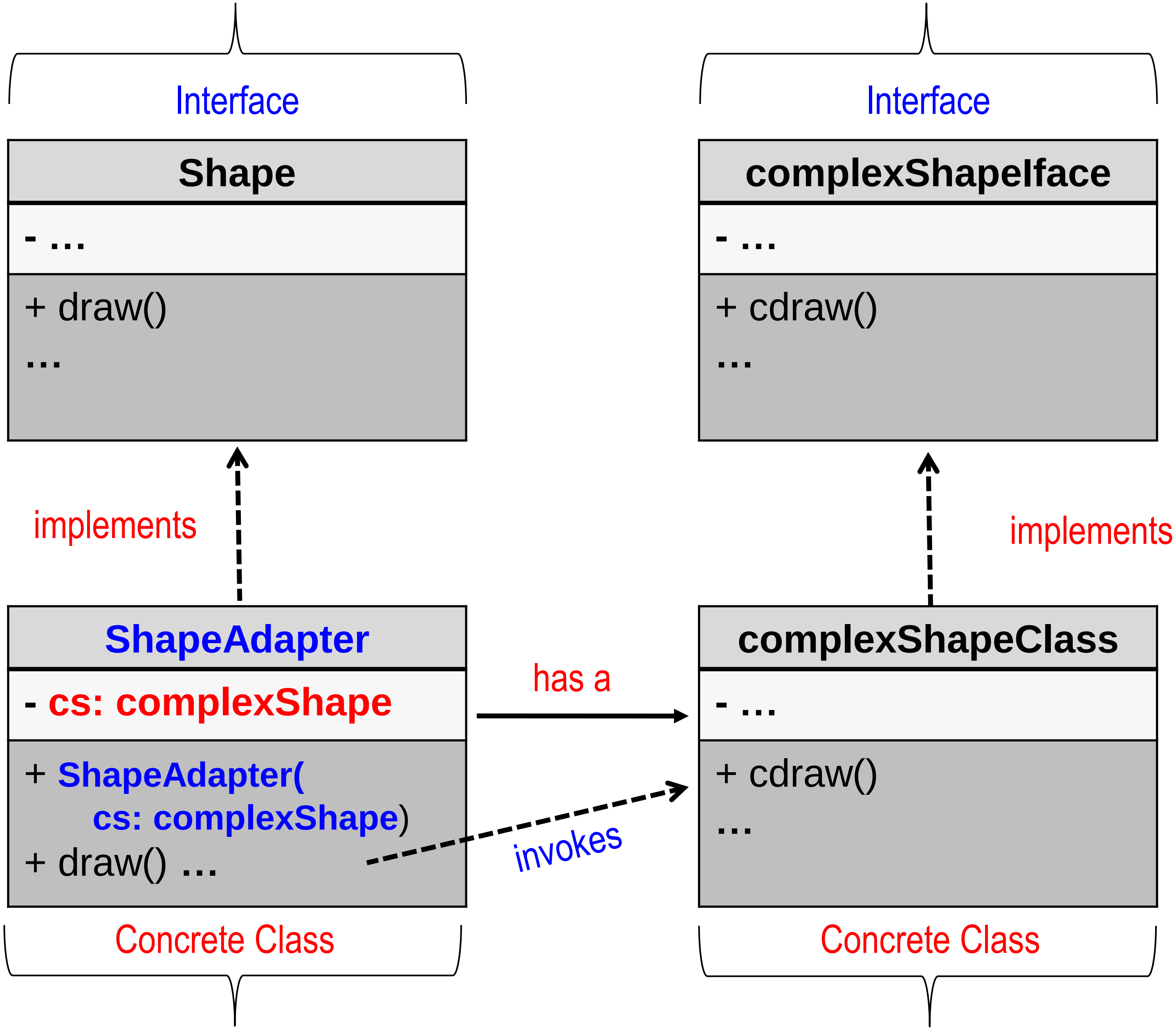
- ...

+ **specificRequest()**
...

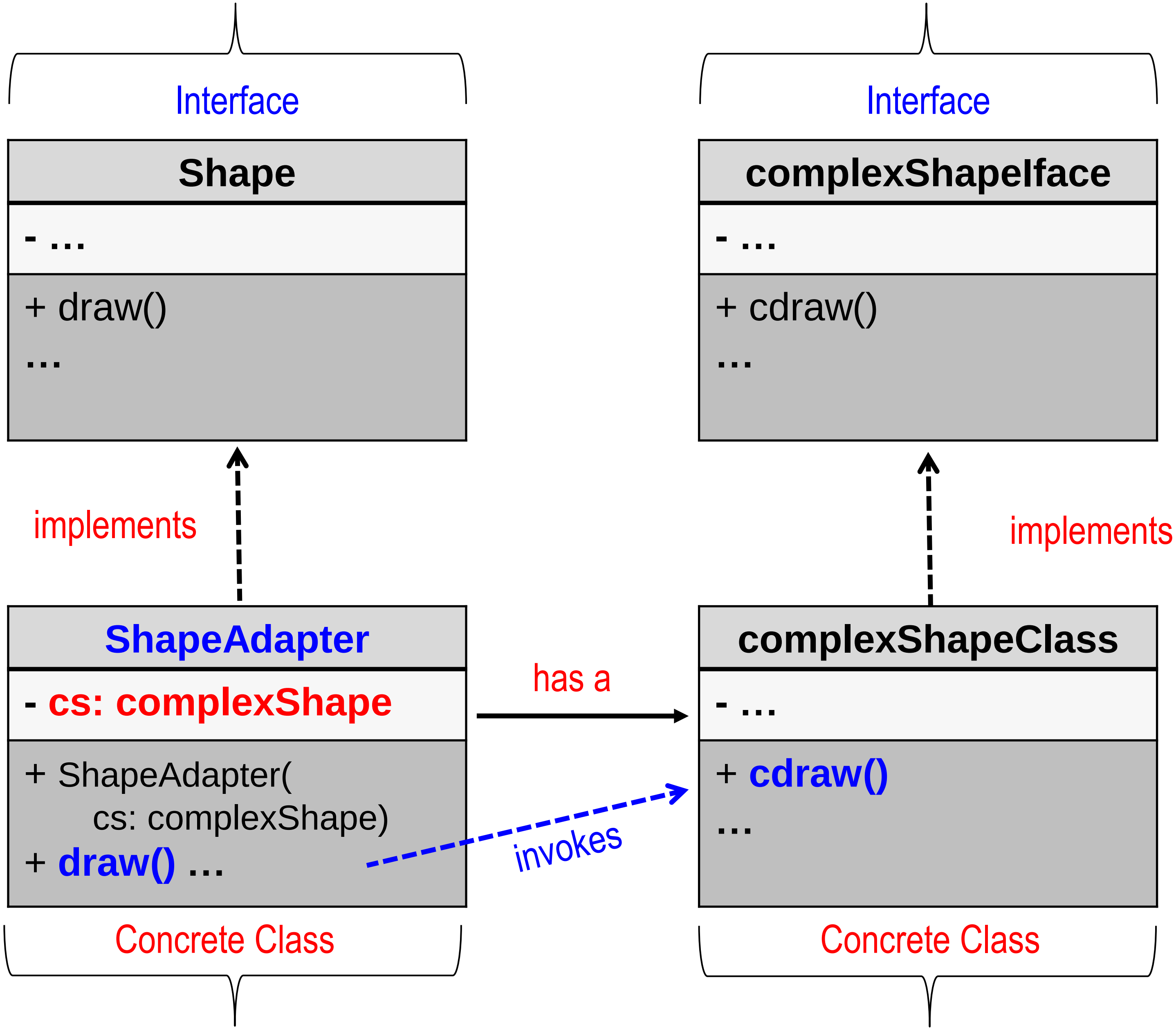
Concrete Class

Concrete Class

Adapter Pattern



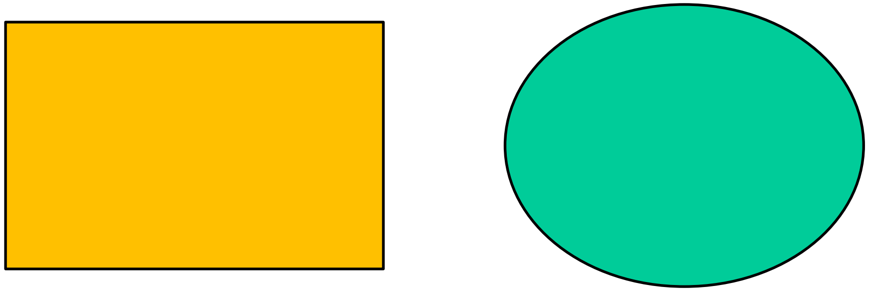
Adapter Pattern



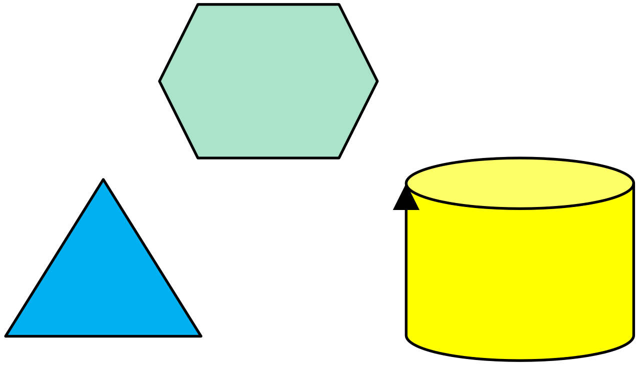
Adapter Pattern:

simple shape example

Shape



complexShape

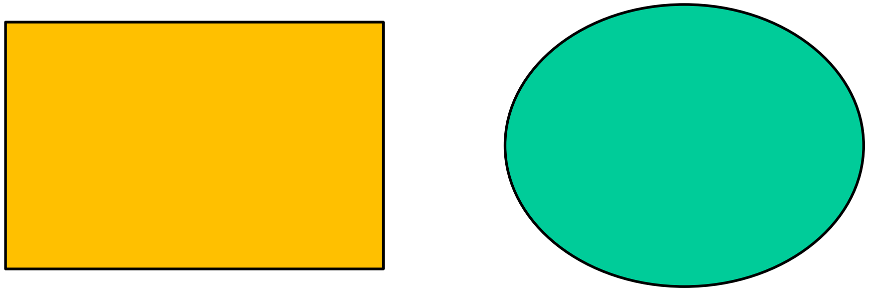


```
public class shapeAdapter implements Shape {
   private complexShape cs;

   public shapeAdapter( complexShape cs ) {
      this.cs = cs;
   }
   // draw method expected by Shape Interface
   public void draw() {
      cs.cdraw();
   }
   ...
} // class
```

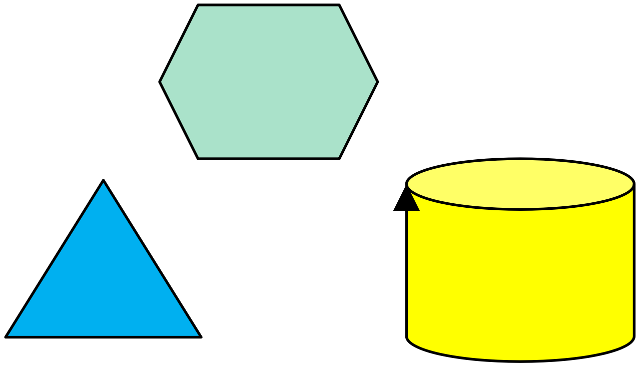
Adapter Pattern:

simple shape example

Shape



complexShape

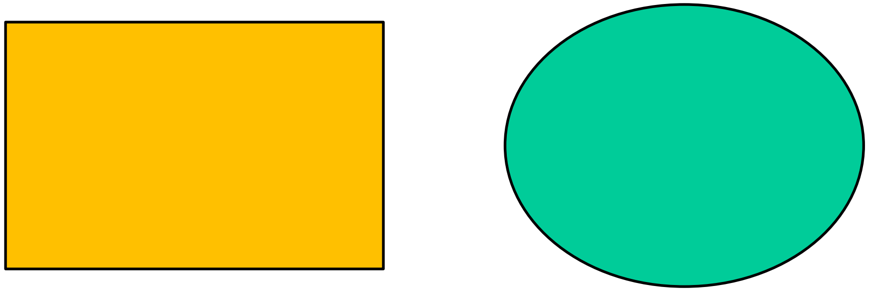


```
public class shapeAdapter implements Shape {
   private complexShape cs;

   public shapeAdapter( complexShape cs ) {
      this.cs = cs;
   }
   // draw method expected by Shape Interface
   public void draw() {
      cs.cdraw(); // call the cdraw method of cs object
   }
   ...
} // class
```

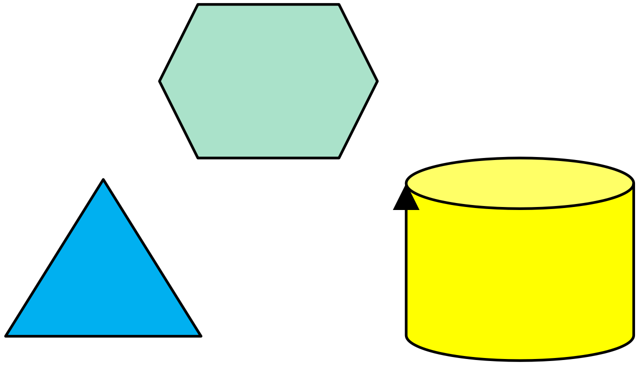
Adapter Pattern:

simple shape example

Shape



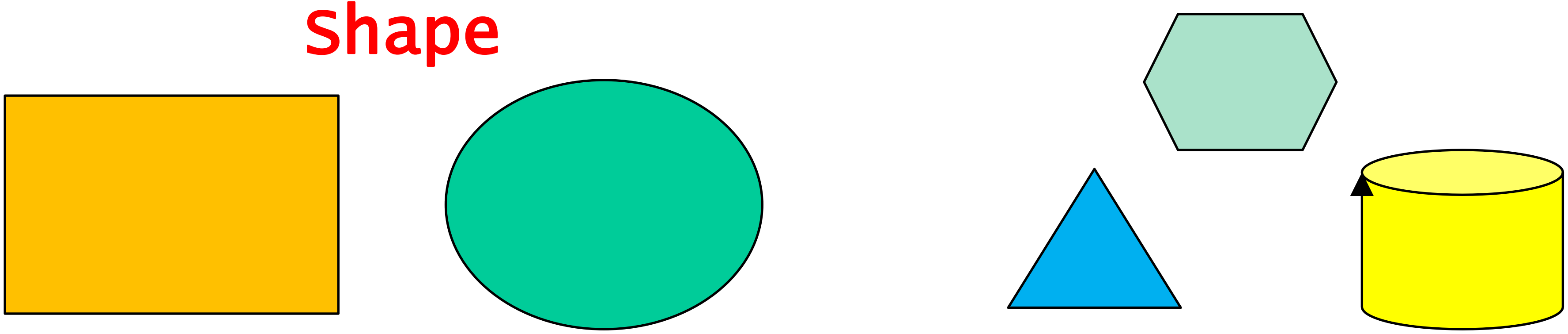
complexShape



```
public class DrawingTest {

  public static void main( ... ) {
    ShapeDrawer sd = new ShapeDrawer();

    sd.addShape( new Rectanlge() );
    sd.addShape( new Circle() );
    sd.addShape( new shapeAdapter(Drum()) );
    sd.draw();

  }
} // class
```

Adapter Pattern:

simple shape example



```
public class DrawingTest {

   public static void main( ... ) {
      ShapeDrawer sd = new ShapeDrawer()

      sd.addShape( new Rectanlge() );
      sd.addShape( new Circle() );
      sd.addShape( new shapeAdapter(Drum()) );
      sd.draw();

   }
} // class
```

ShapeAdapter is a Shape!

Adapter Pattern:

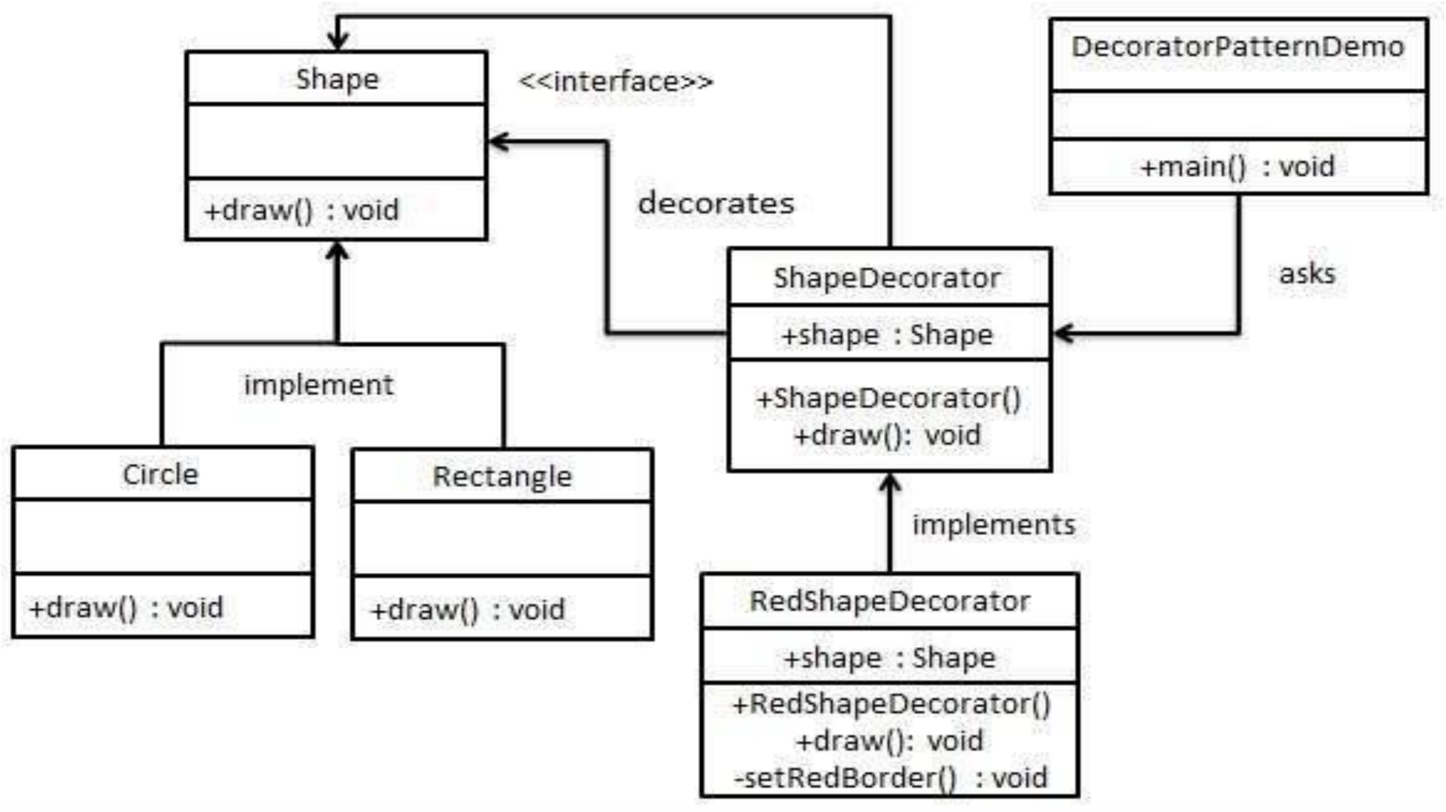
As defined in Elements of Reusable OO Software

- **Consequences (Advantages/Disadvantages):**
  - Allows a single Adapter to work with many concrete Adaptees.
  - The Adap
  - Difficult to s
    the Adap
    with the i

This is a very useful design patter to allow to test out new interfaces without changing existing implementation. Also very useful when you when you want to be able to alternate between multiple interfaces.

Decorator Pattern

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



Tutorials Point

Decorator Pattern:

Elements of Reusable OO Software

- Motivation **and Applicability**: Sometimes we want to add additional responsibilities to individual objects, but not necessarily to an entire class. A UI fo properties lik component,
  - To be abl
  - **To add r and trans**
  - For respo
  - **When ex al behaviors is impractical and would produce an explosion of subclasses.**

What if you wanted objects to “increase” in power dynamically?

This pattern is designed to allow objects to gain in responsibility to fit the needs of your program…

Decorator Pattern:

Elements of Reusable OO Software

- Motivation **and Applicability**: Sometimes we want to add additional responsibilities to individual objects, but not necessarily to an entire class. A UI fo
properties lik
component, a
  - To be able
  - **To add re**
and trans
  - For respon
  - **When ext**
behaviors is impractical and would produce an explosion of subclasses.

Consider a typical Starbucks coffee order:

- tall skinny decaf no foam latte
- skinny venti mocha
- half caramel, half vanilla latte, decaf espresso heated only to 100° with nonfat milk and caramel drizzle on top

Decorator Pattern:

Elements of Reusable OO Software

- Motivation **and Applicability**: Sometimes we want to add additional responsibilities to individual objects, but not necessarily to an entire class. A UI fo
properties lik
component, a
  - To be able
  - **To add re**
and trans
  - For responsibilities that can be withdrawn.
  - **When extending a class to represent all possible additional behaviors is impractical and would produce an explosion of subclasses.**

Consider a typical Starbucks coffee order:

- tall skinny decaf no foam latte

- skinny venti mocha

- half caramel, half vanilla latte, decaf espresso heated only to 100° with nonfat milk and caramel drizzle on top

Decorator Pattern:

Elements of Reusable OO Software

- Motivation **and Applicability**: Sometimes we want to add additional responsibilities to individual objects, but not necessarily to an entire
class. A UI fo
properties lik
component, a
    - To be able
    - **To add re**
    and trans
    - For responsibilities that can be withdrawn.
    - **When extending a class to represent all possible additional behaviors is impractical and would produce an explosion of subclasses.**

Consider a typical Starbucks coffee order:

- tall skinny decaf no foam latte

- skinny venti mocha

- half caramel, half vanilla latte, decaf espresso heated only to 100° with nonfat milk and caramel drizzle on top

Decorator Pattern:

Elements of Reusable OO Software

- Motivation **and Applicability**: Sometimes we want to add additional responsibilities to individual objects, but not necessarily to an entire class. A UI fo
  properties lik
  component, a
  - To be able
  - **To add re**
    and trans
  - For respon sibilities that can be withdrawn.
  - **When extending a class to represent all possible additional behaviors is impractical and would produce an explosion of subclasses.**

Consider a typical Starbucks coffee order:

- tall skinny decaf no foam latte

- skinny venti mocha

- half caramel, half vanilla latte, decaf espresso heated only to 100° with nonfat milk and caramel drizzle on top

Decorator Pattern:

Elements of Reusable OO Software

- Motivation **and Applicability**: Sometimes we want to add additional responsibilities to individual objects, but not necessarily to an entire class. A UI fo properties lik component, a
    - To be able
    - **To add re and trans**
    - For respon sibilities that can be withdrawn.
    - **When extending a class to represent all possible additional behaviors is impractical and would produce an explosion of subclasses.**

Consider a typical Starbucks coffee order:

- tall skinny decaf no foam latte

- skinny venti mocha

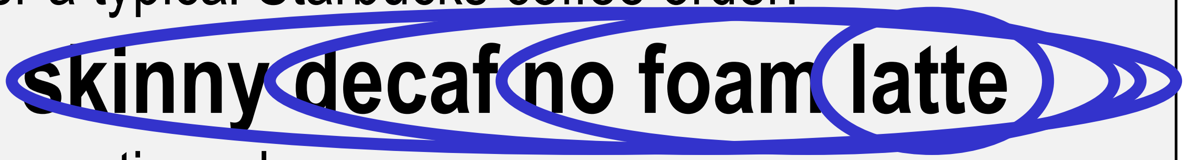
- half caramel, half vanilla latte, decaf espresso heated only to 100° with nonfat milk and caramel drizzle on top

Decorator Pattern:

Elements of Reusable OO Software

- Motivation **and Applicability**: Sometimes we want to add additional responsibilities to individual objects, but not necessarily to an entire class. A UI fo properties lik component, a
  - To be able
  - **To add re and trans**
  - For responsibilities that can be withdrawn.
  - **When extending a class to represent all possible additional behaviors is impractical and would produce an explosion of subclasses.**

Consider a typical Starbucks coffee order:



- tall skinny decaf no foam latte

- skinny venti mocha

- half caramel, half vanilla latte, decaf espresso heated only to 100° with nonfat milk and caramel drizzle on top

Decorator Pattern:
Elements of Reusable OO Software

- Motivation **and Applicability**: Sometimes we want to add additional responsibilities to individual objects, but not necessarily to an entire class. A UI fo properties lik component, a
  - To be able
  - **To add re and trans**
  - For responsibilities that can be withdrawn.
  - **When extending a class to represent all possible additional behaviors is impractical and would produce an explosion of subclasses.**

Consider a typical Starbucks coffee order:

- tall skinny decaf no foam latte

- skinny venti mocha

- half caramel, half vanilla latte, decaf espresso heated only to 100° with nonfat milk and caramel drizzle on top

Decorator Pattern:

Elements of Reusable OO Software

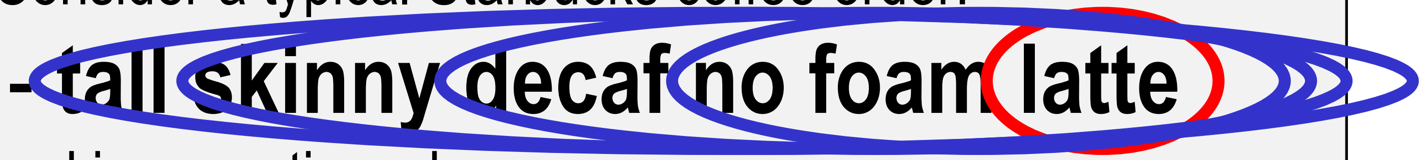
- Motivation **and Applicability**: Sometimes we want to add additional responsibilities to individual objects, but not necessarily to an entire class. A UI fo

  properties lik

  component, a
  - To be able
  - **To add re**

    and trans
  - For responsibilities that can be withdrawn.
  - **When extending a class to represent all possible additional behaviors is impractical and would produce an explosion of subclasses.**

Consider a typical Starbucks coffee order:



- tall skinny decaf no foam latte

- skinny venti mocha

- half caramel, half vanilla latte, decaf espresso heated only to 100° with nonfat milk and caramel drizzle on top

Decorator Pattern:

classic Ice Cream example

Decorator Pattern:

classic Ice Cream example

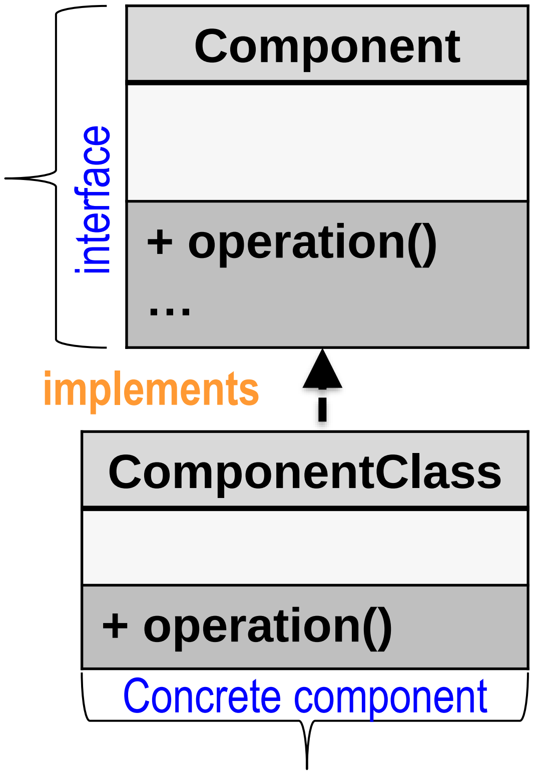
Decorator Pattern:

classic Ice Cream example

Decorator Pattern:

classic Ice Cream example

Decorator Pattern



Component

+ operation()

…

interface

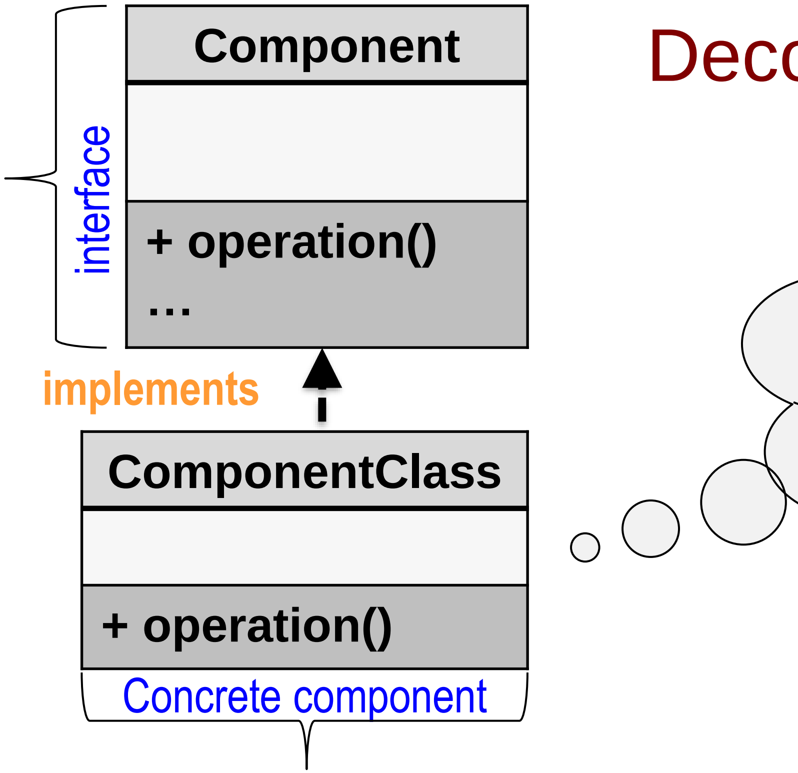
implements

ComponentClass

+ operation()

Concrete component

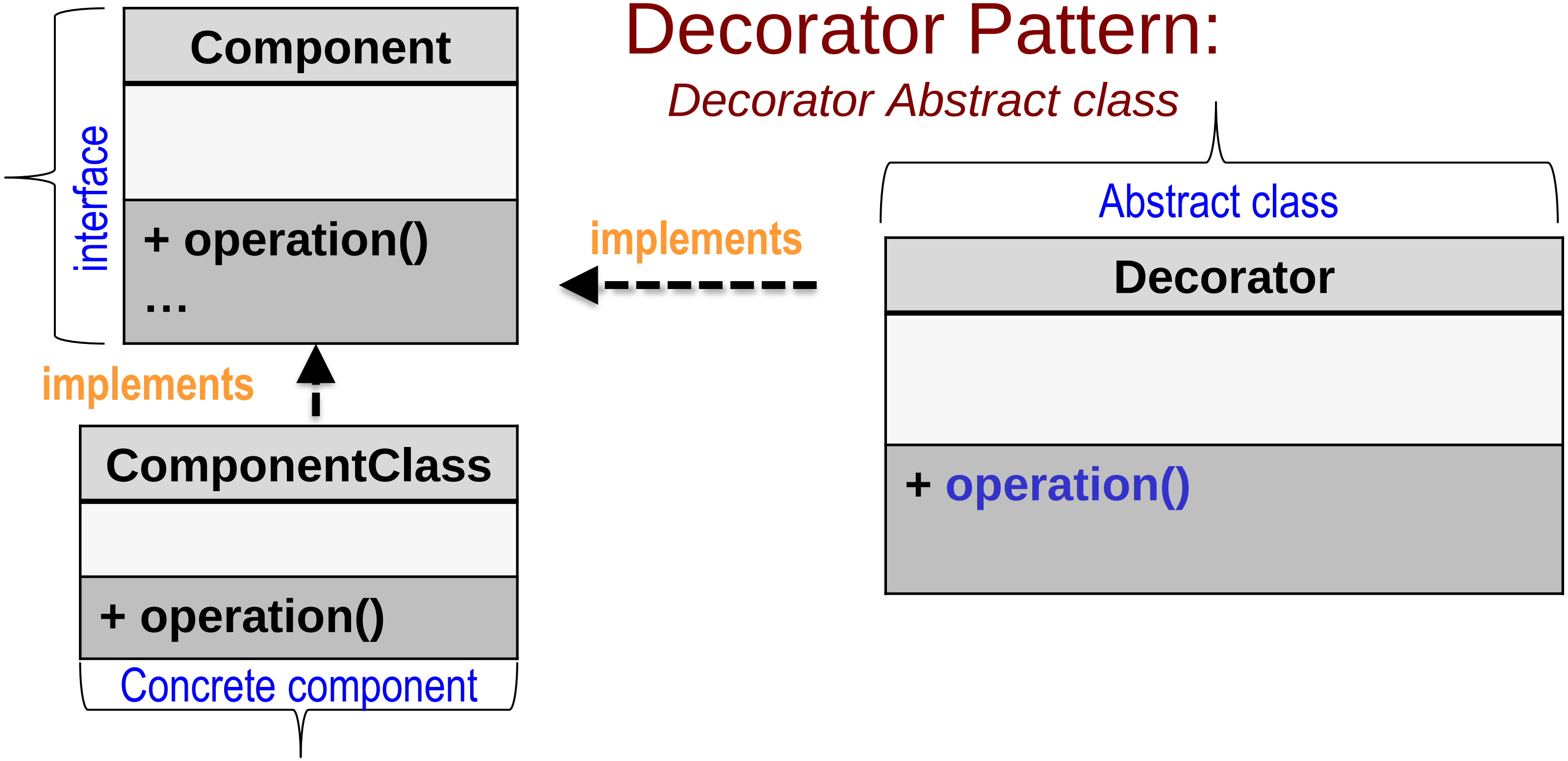
Decorator Pattern



Base component (that) implements the base behavior of the component. *Additional* functionality will be added via Decorators!

Decorator Pattern:

Decorator Abstract class



interface

Component

+ operation()

…

implements

ComponentClass

+ operation()

Concrete component

implements

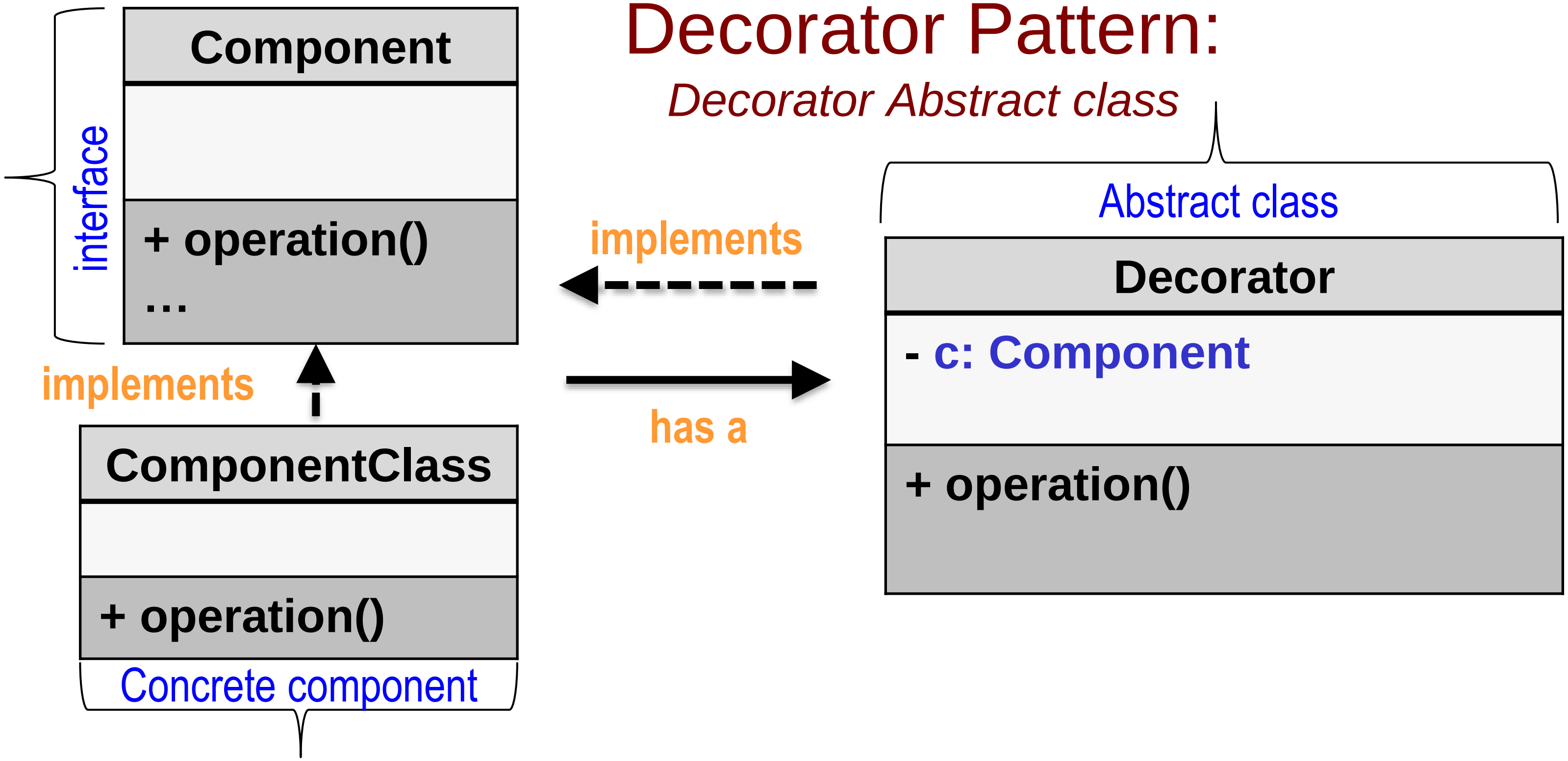
Abstract class

Decorator

+ operation()

Decorator Pattern:

Decorator Abstract class



interface

Component

+ operation()

…

implements

ComponentClass

+ operation()

Concrete component

implements

has a

Abstract class

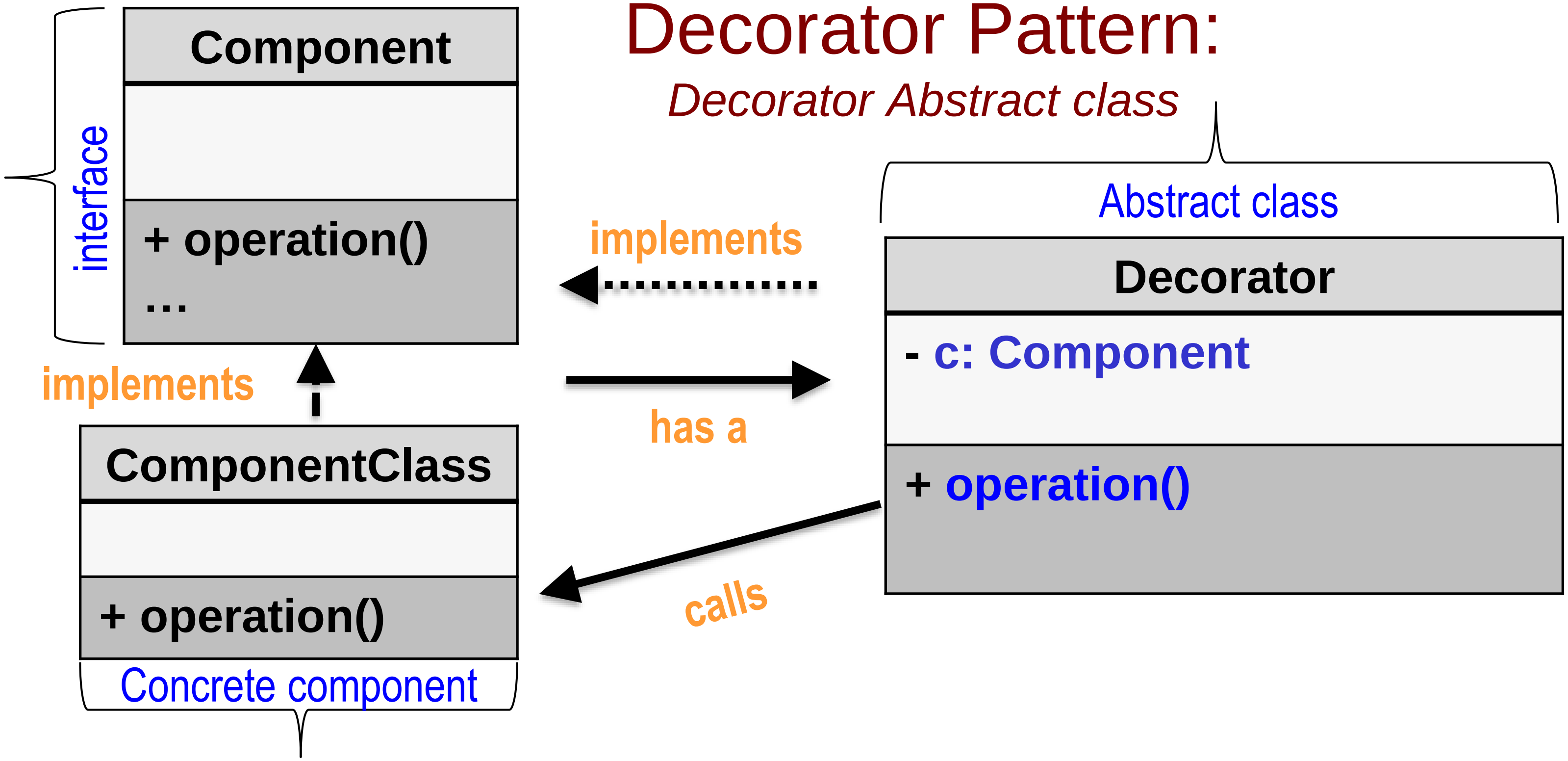
Decorator

- c: Component

+ operation()

Decorator Pattern:

Decorator Abstract class



interface

Component

+ operation()

…

implements

ComponentClass

+ operation()

Concrete component

implements

has a

calls

Abstract class

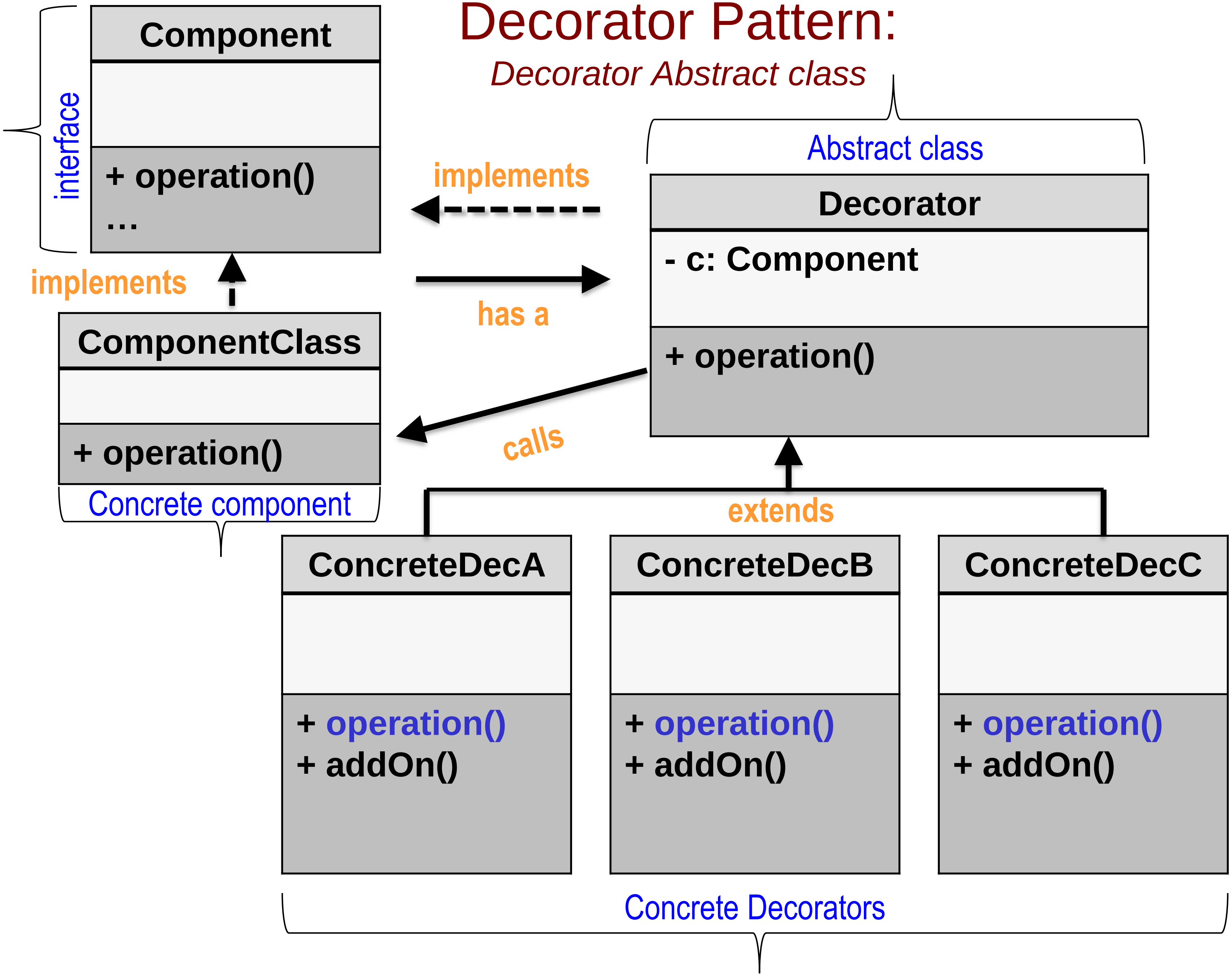
Decorator

- c: Component

+ operation()

Decorator Pattern:

Decorator Abstract class



interface

Component

+ operation()
…

implements

ComponentClass

+ operation()

Concrete component

implements

has a

calls

Abstract class

Decorator

- c: Component

+ operation()

extends

ConcreteDecA

+ operation()
+ addOn()

ConcreteDecB

+ operation()
+ addOn()

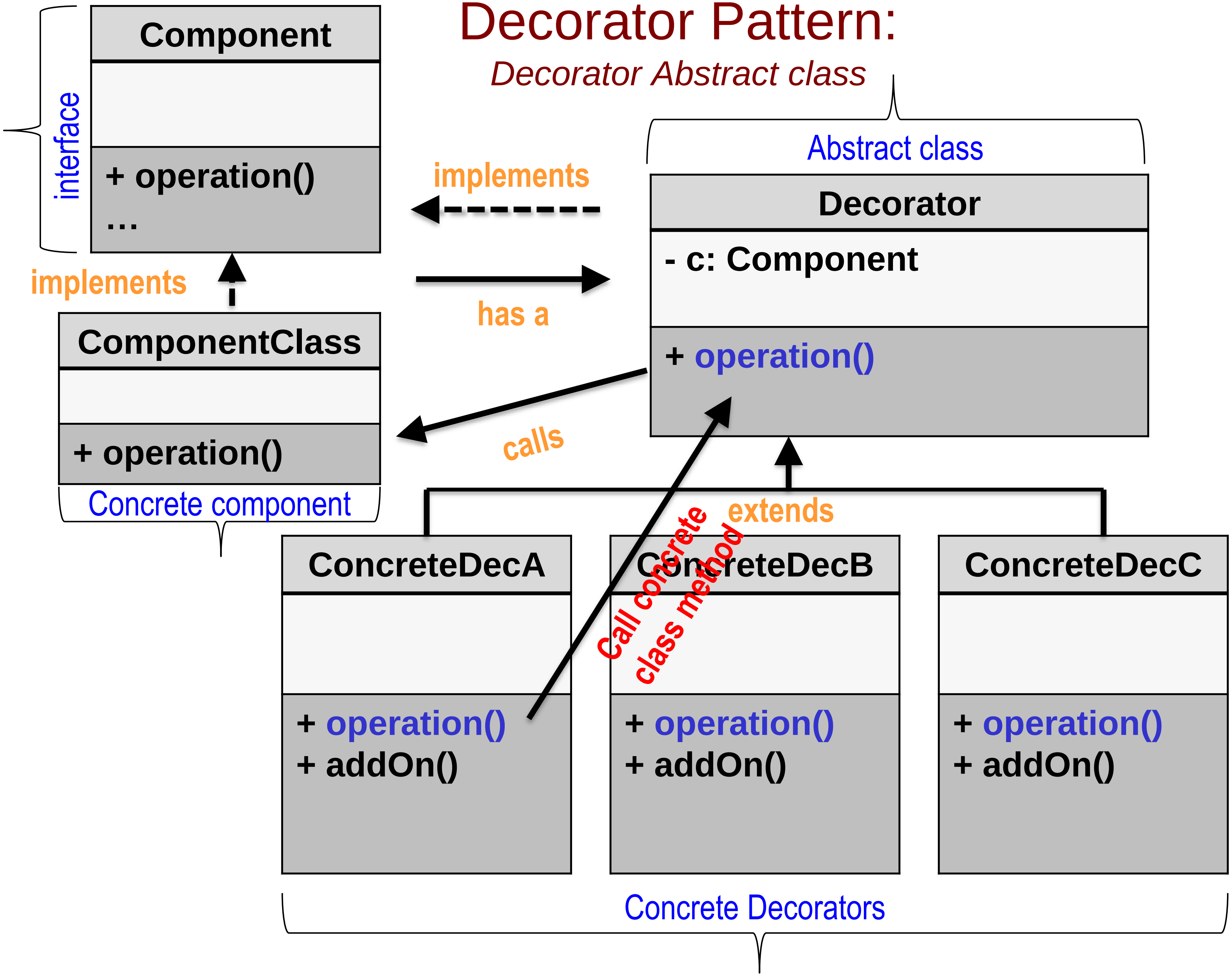
ConcreteDecC

+ operation()
+ addOn()

Concrete Decorators

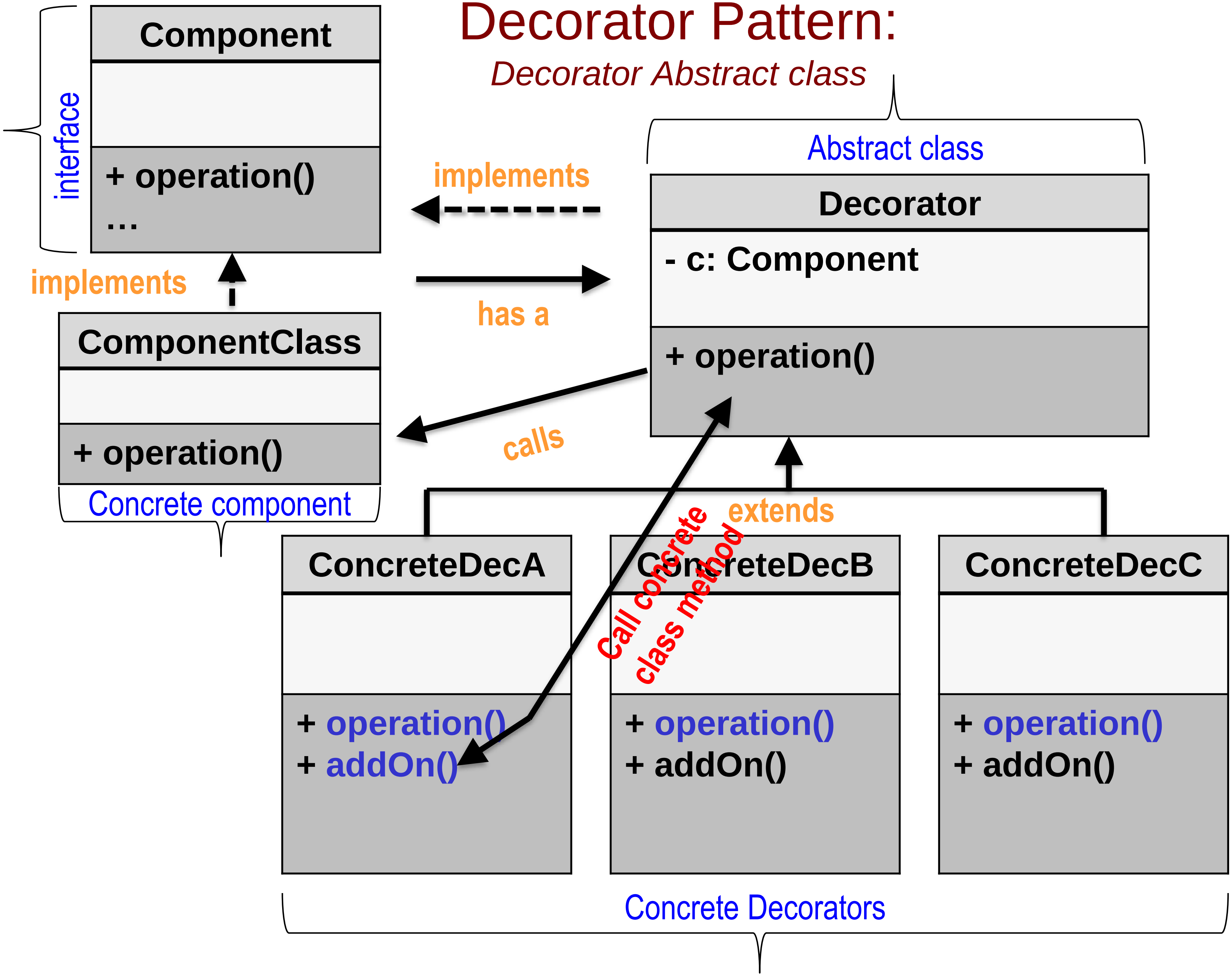
Decorator Pattern:

Decorator Abstract class



Decorator Pattern:

Decorator Abstract class



Component

+ operation()

…

interface

implements

ComponentClass

+ operation()

Concrete component

implements

has a

calls

Abstract class

Decorator

- c: Component

+ operation()

extends

Call concrete class method

ConcreteDecA

+ operation()

+ addOn()

ConcreteDecB

+ operation()

+ addOn()

ConcreteDecC

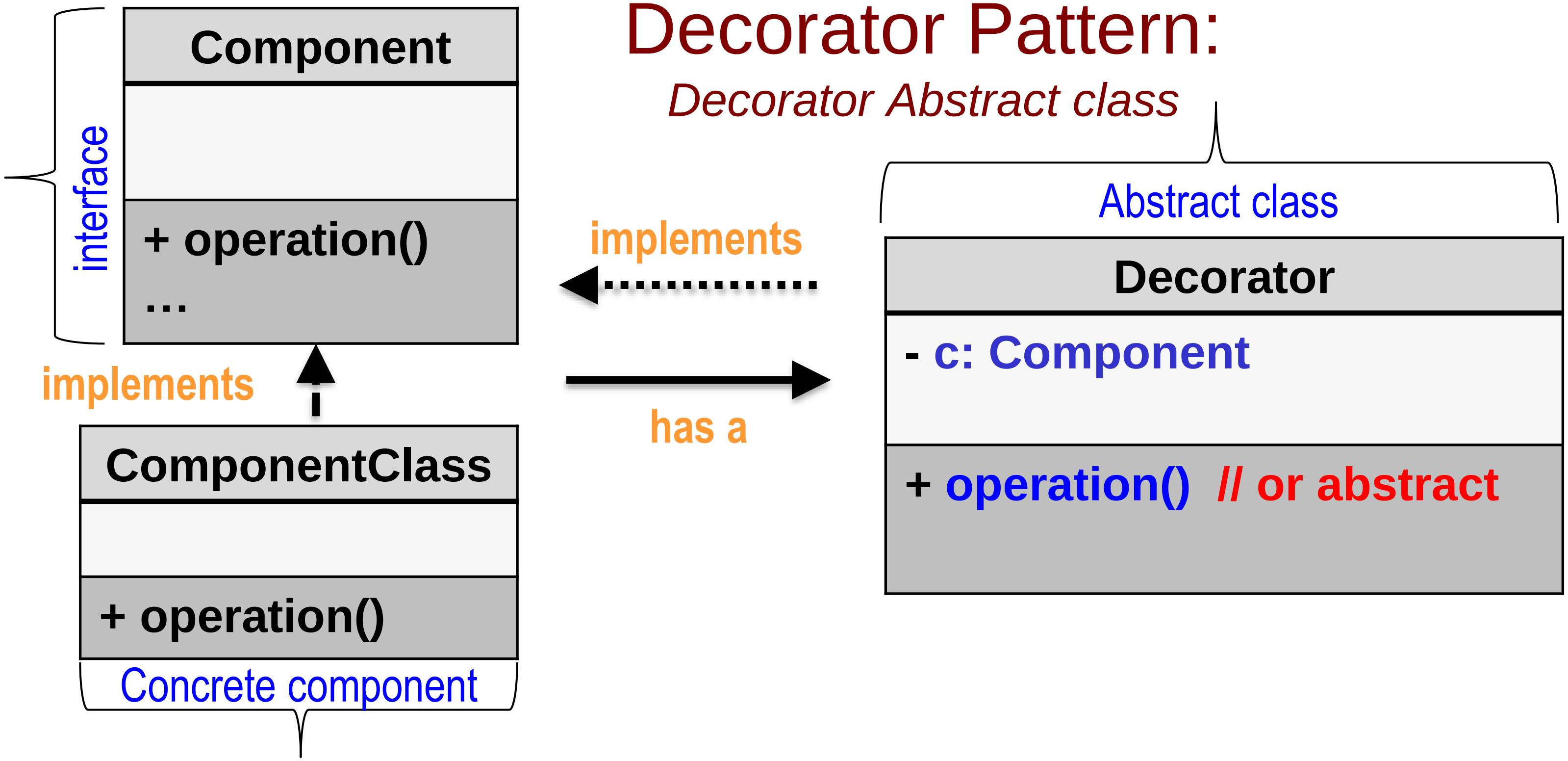
+ operation()

+ addOn()

Concrete Decorators

Decorator Pattern:

Decorator Abstract class



interface

Component

+ operation()

…

implements

ComponentClass

+ operation()

Concrete component

implements

has a

Abstract class

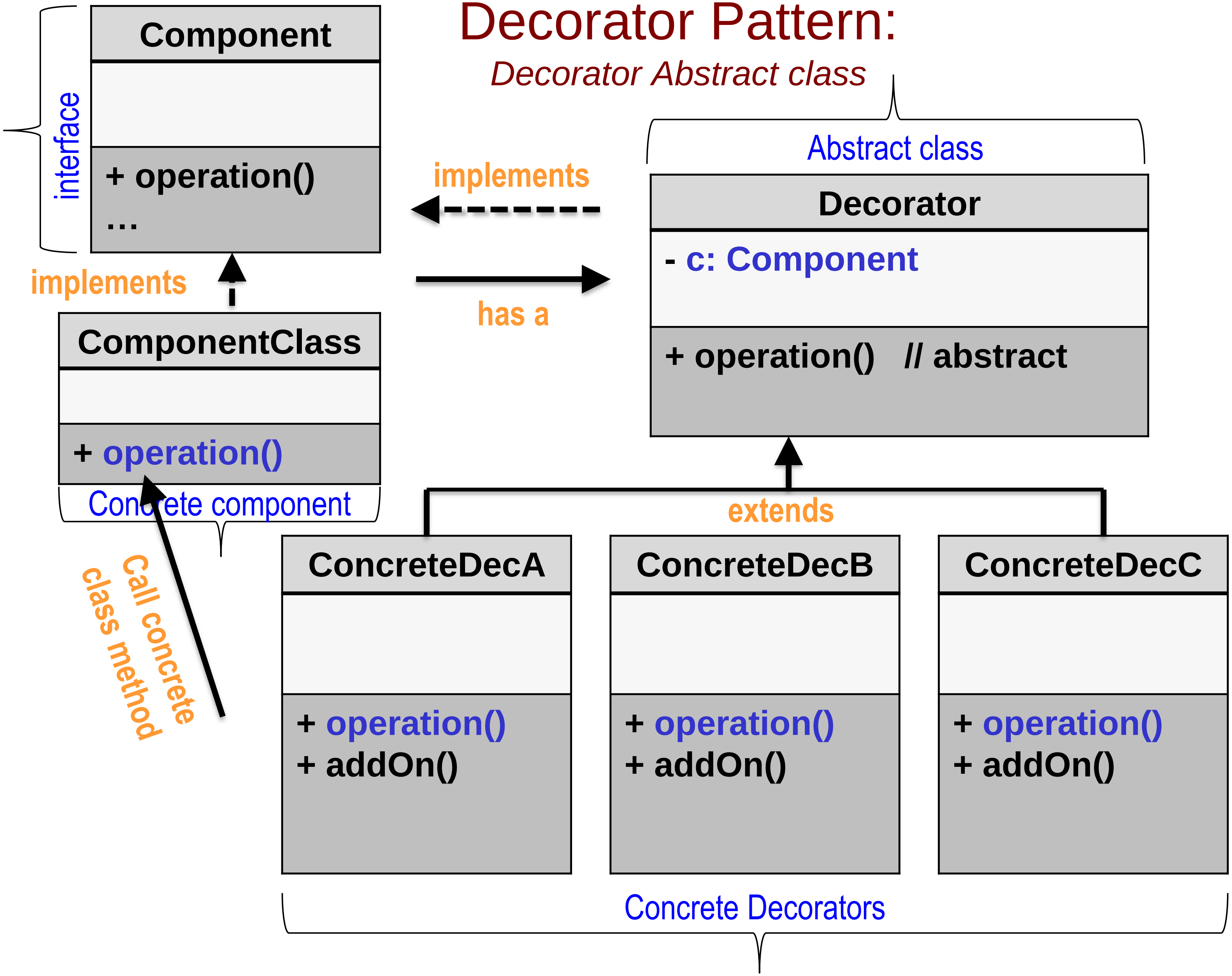
Decorator

- c: Component

+ operation() // or abstract

Decorator Pattern:

Decorator Abstract class



interface

Component

+ operation()

…

implements

ComponentClass

+ operation()

Concrete component

Call concrete class method

implements

has a

Abstract class

Decorator

- c: Component

+ operation() // abstract

extends

ConcreteDecA

+ operation()

+ addOn()

ConcreteDecB

+ operation()

+ addOn()

ConcreteDecC

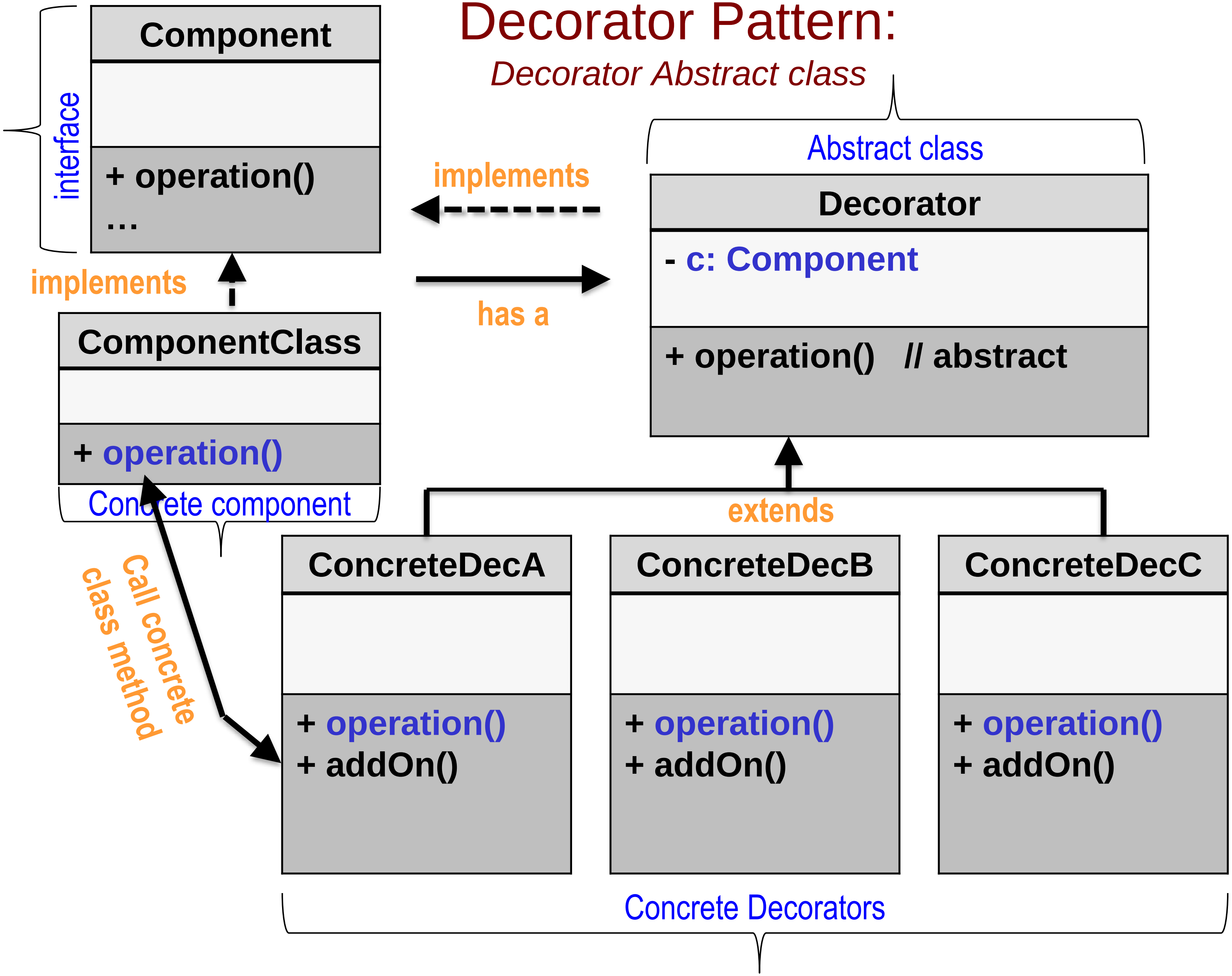
+ operation()

+ addOn()

Concrete Decorators

Decorator Pattern:

Decorator Abstract class



interface

Component

+ operation()
…

implements

ComponentClass

+ operation()

Concrete component

implements

has a

Abstract class

Decorator

- c: Component

+ operation() // abstract

extends

ConcreteDecA

+ operation()
+ addOn()

ConcreteDecB

+ operation()
+ addOn()

ConcreteDecC

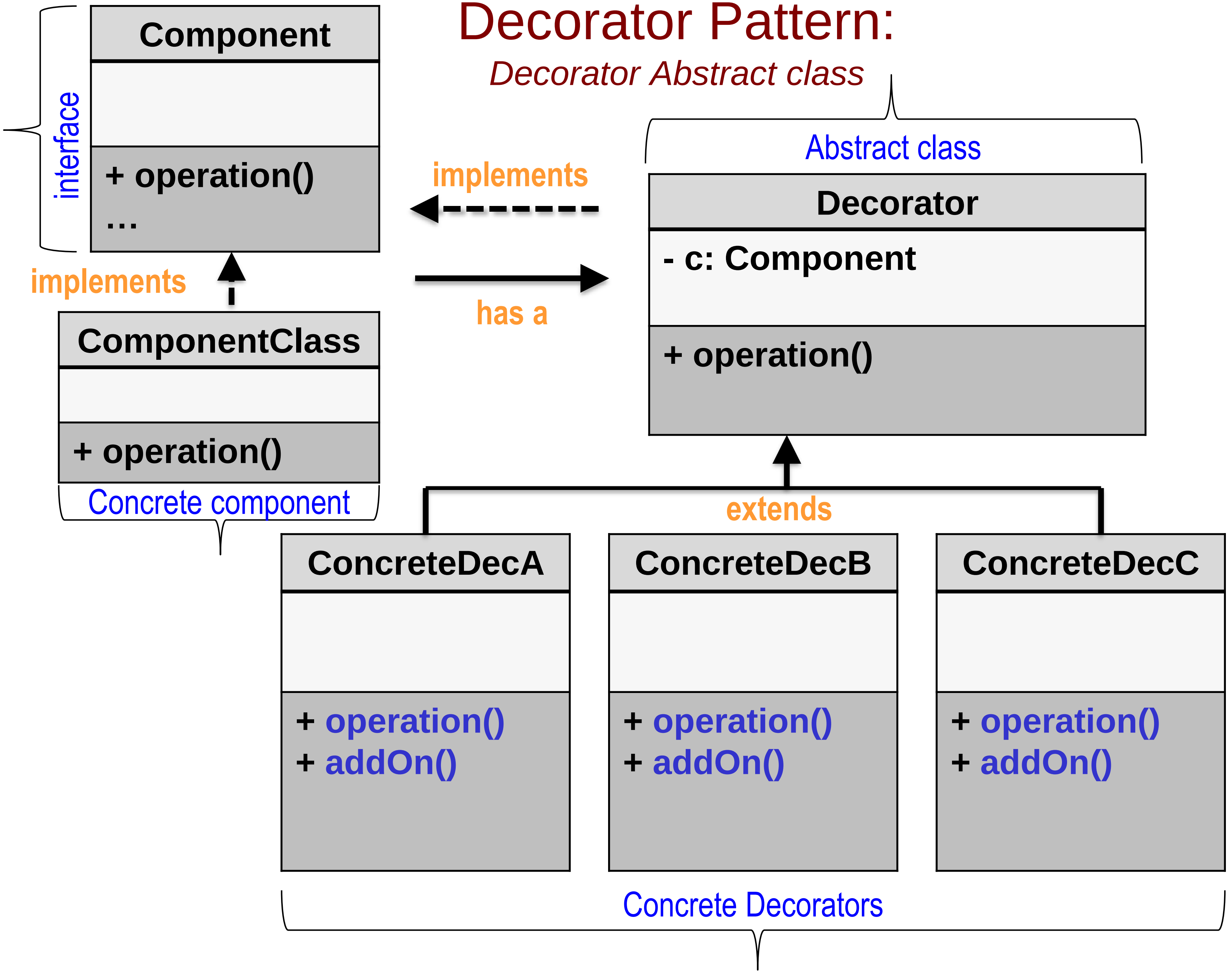
+ operation()
+ addOn()

Call concrete class method

Concrete Decorators

Decorator Pattern:

Decorator Abstract class



interface

Component

+ operation()

…

implements

ComponentClass

+ operation()

Concrete component

implements

has a

Abstract class

Decorator

- c: Component

+ operation()

extends

ConcreteDecA

+ operation()
+ addOn()

ConcreteDecB

+ operation()
+ addOn()

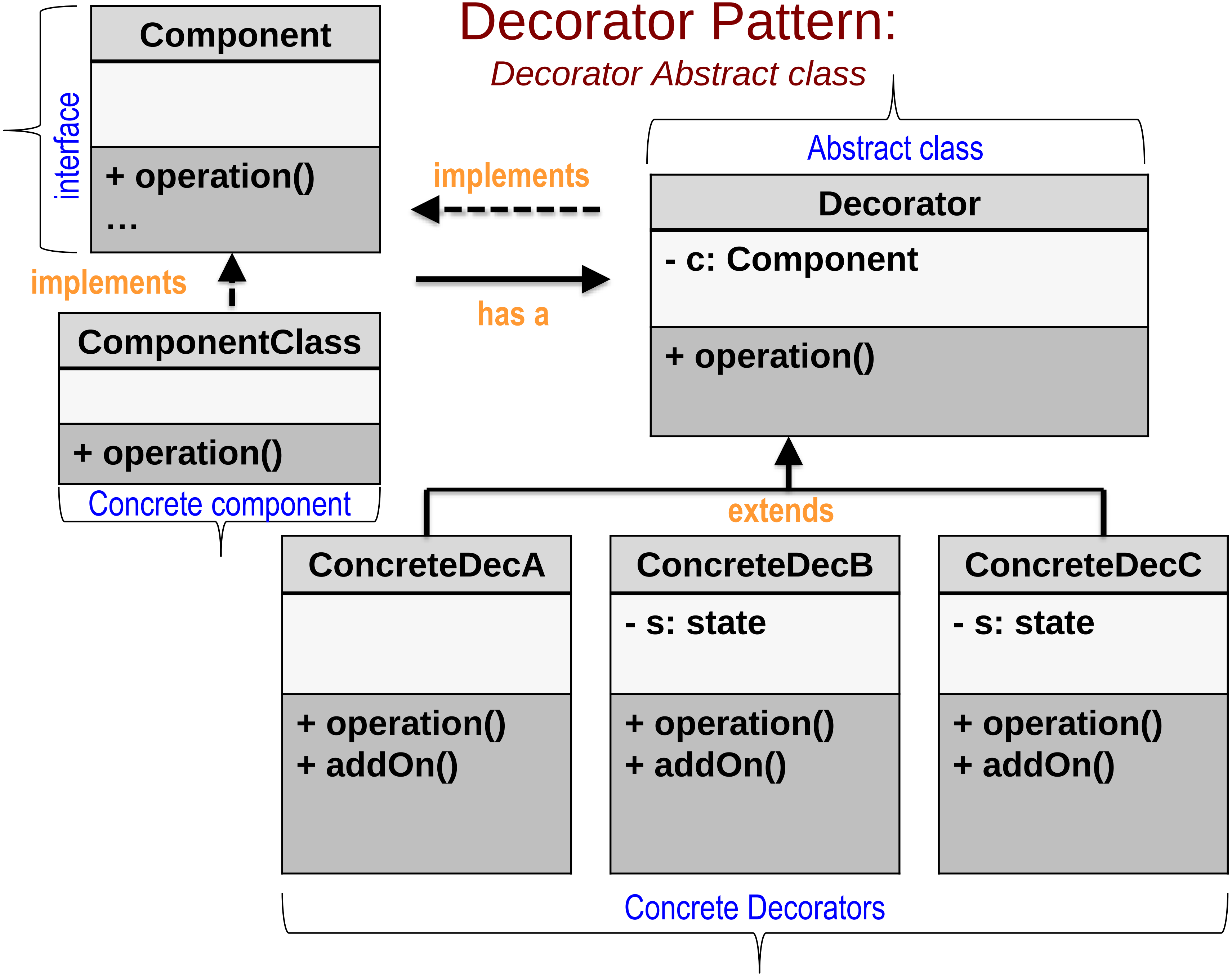
ConcreteDecC

+ operation()
+ addOn()

Concrete Decorators

Decorator Pattern:

Decorator Abstract class



interface

Component

+ operation()
…

implements

ComponentClass

+ operation()

Concrete component

implements

has a

Abstract class

Decorator

- c: Component

+ operation()

extends

ConcreteDecA

+ operation()
+ addOn()

ConcreteDecB

- s: state

+ operation()
+ addOn()

ConcreteDecC

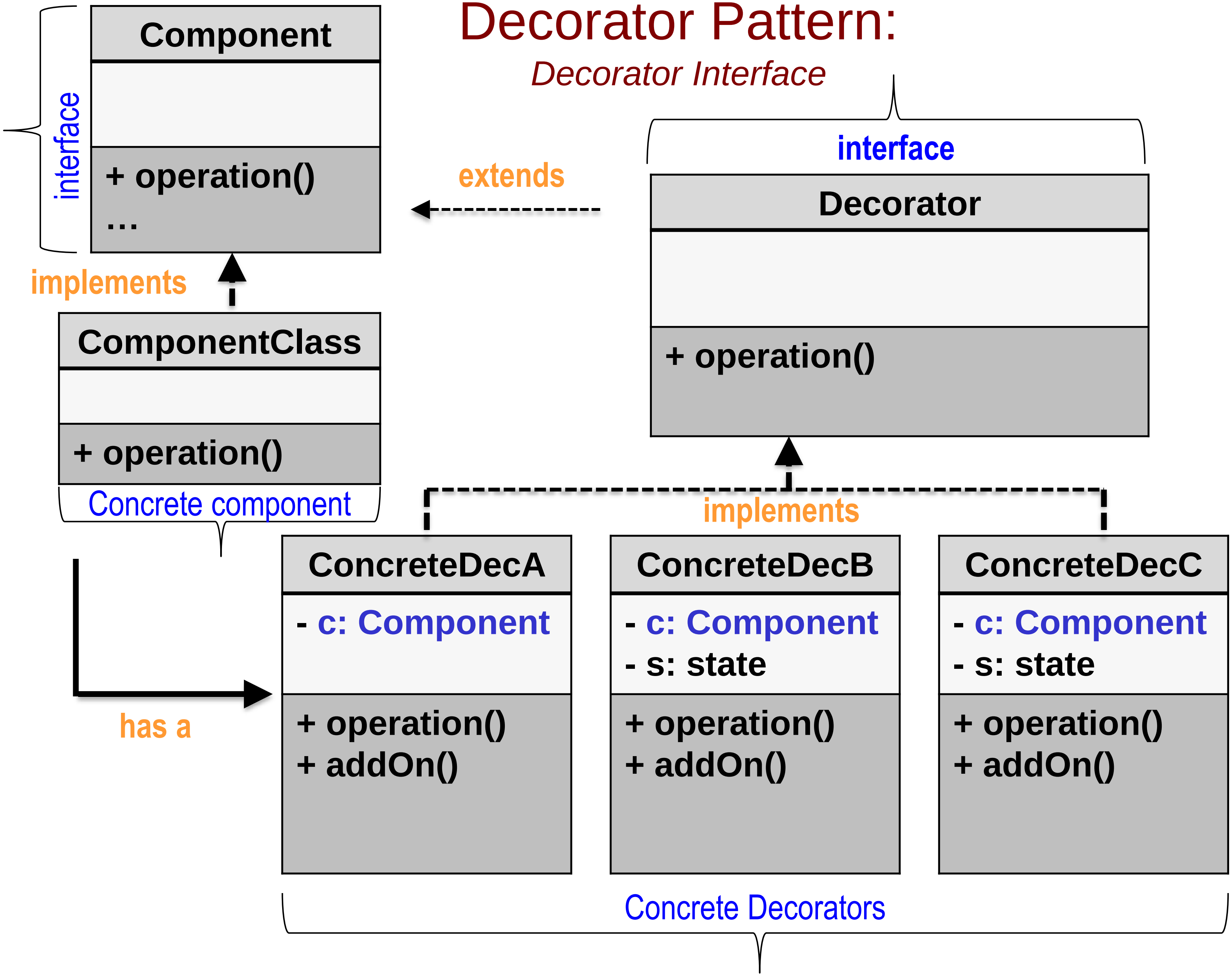
- s: state

+ operation()
+ addOn()

Concrete Decorators

Decorator Pattern:

Decorator Interface



interface

Component

+ operation()

…

implements

ComponentClass

+ operation()

Concrete component

extends

interface

Decorator

+ operation()

implements

has a

ConcreteDecA

- c: Component

+ operation()
+ addOn()

ConcreteDecB

- c: Component
- s: state

+ operation()
+ addOn()

ConcreteDecC

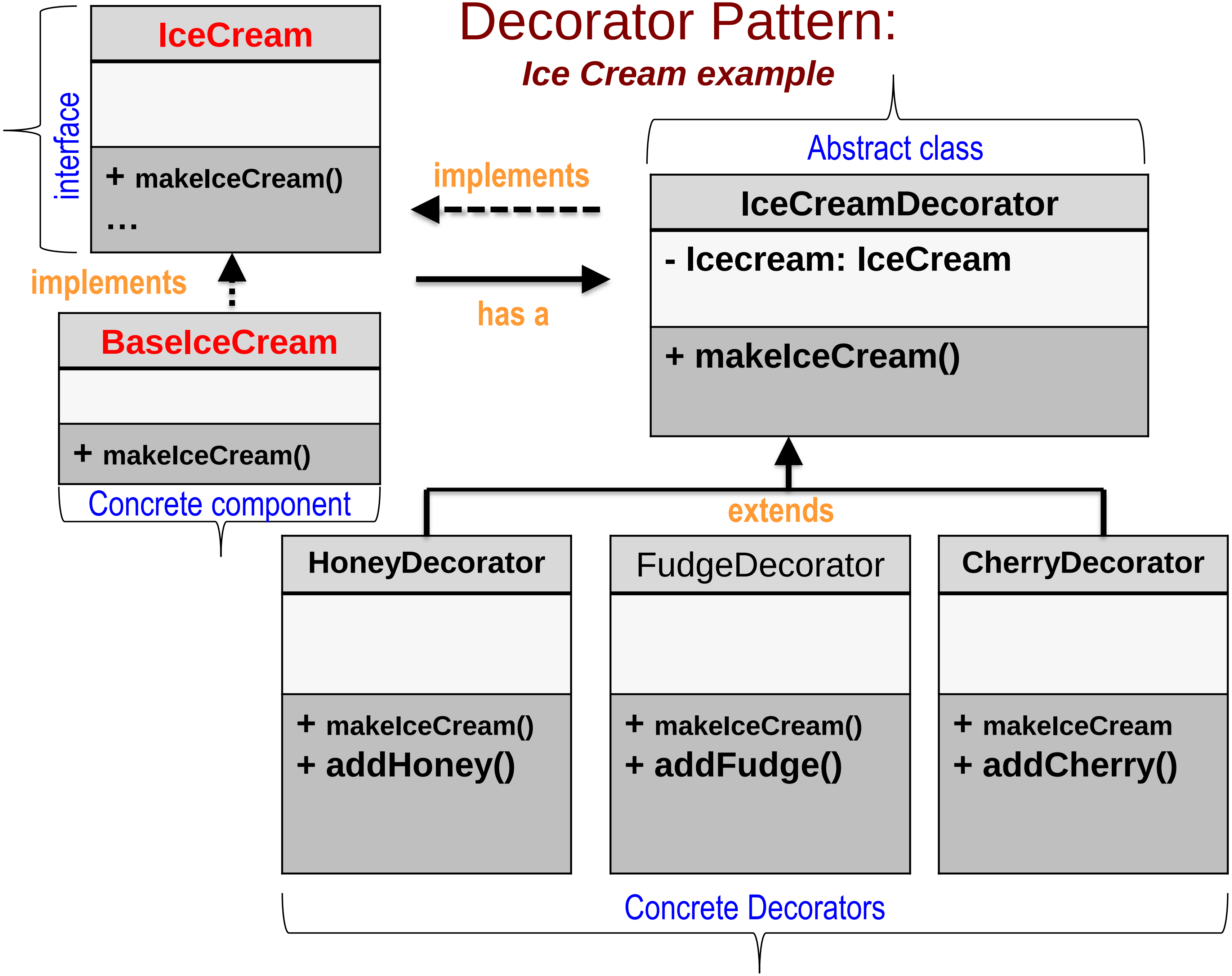
- c: Component
- s: state

+ operation()
+ addOn()

Concrete Decorators

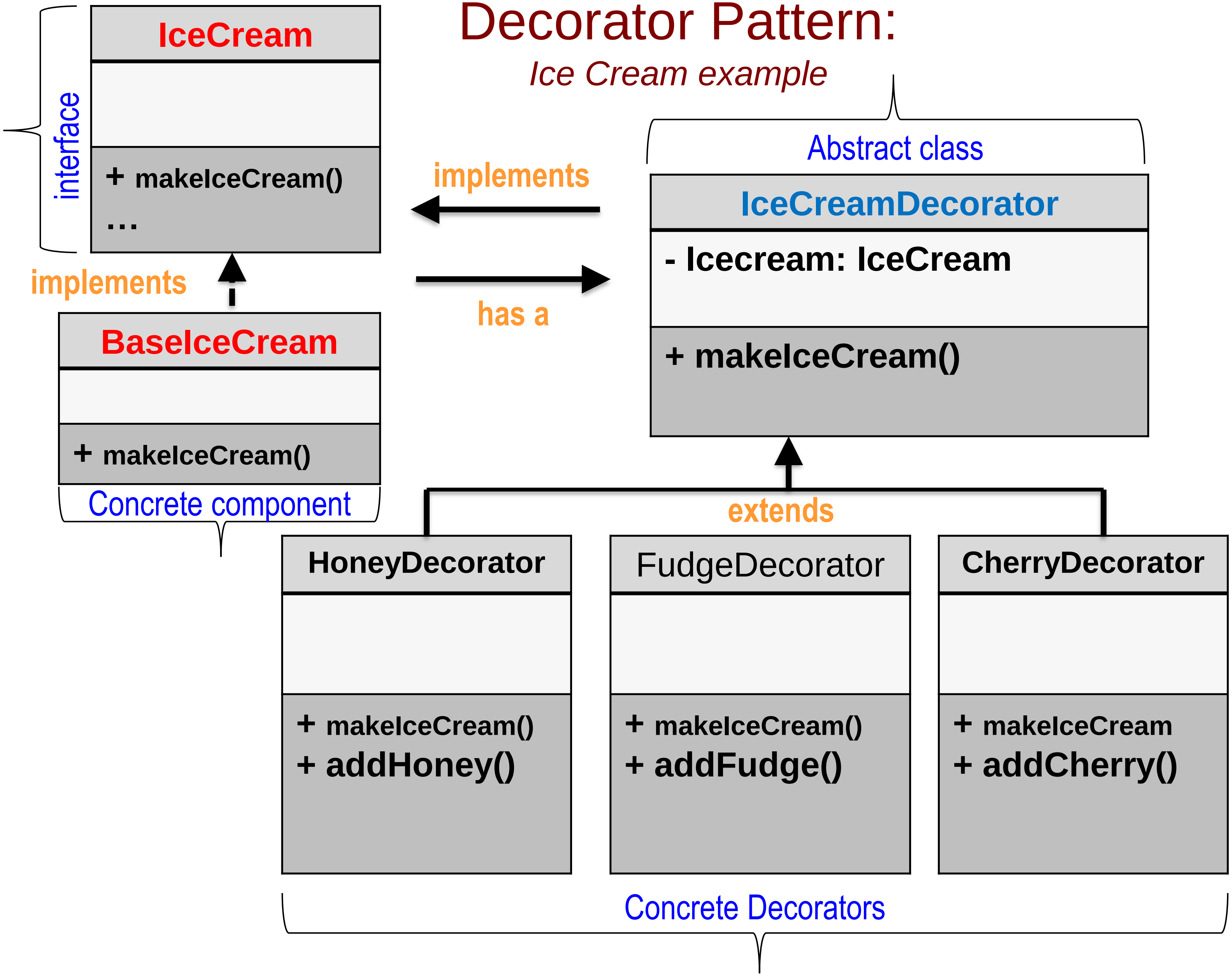
Decorator Pattern:

Ice Cream example



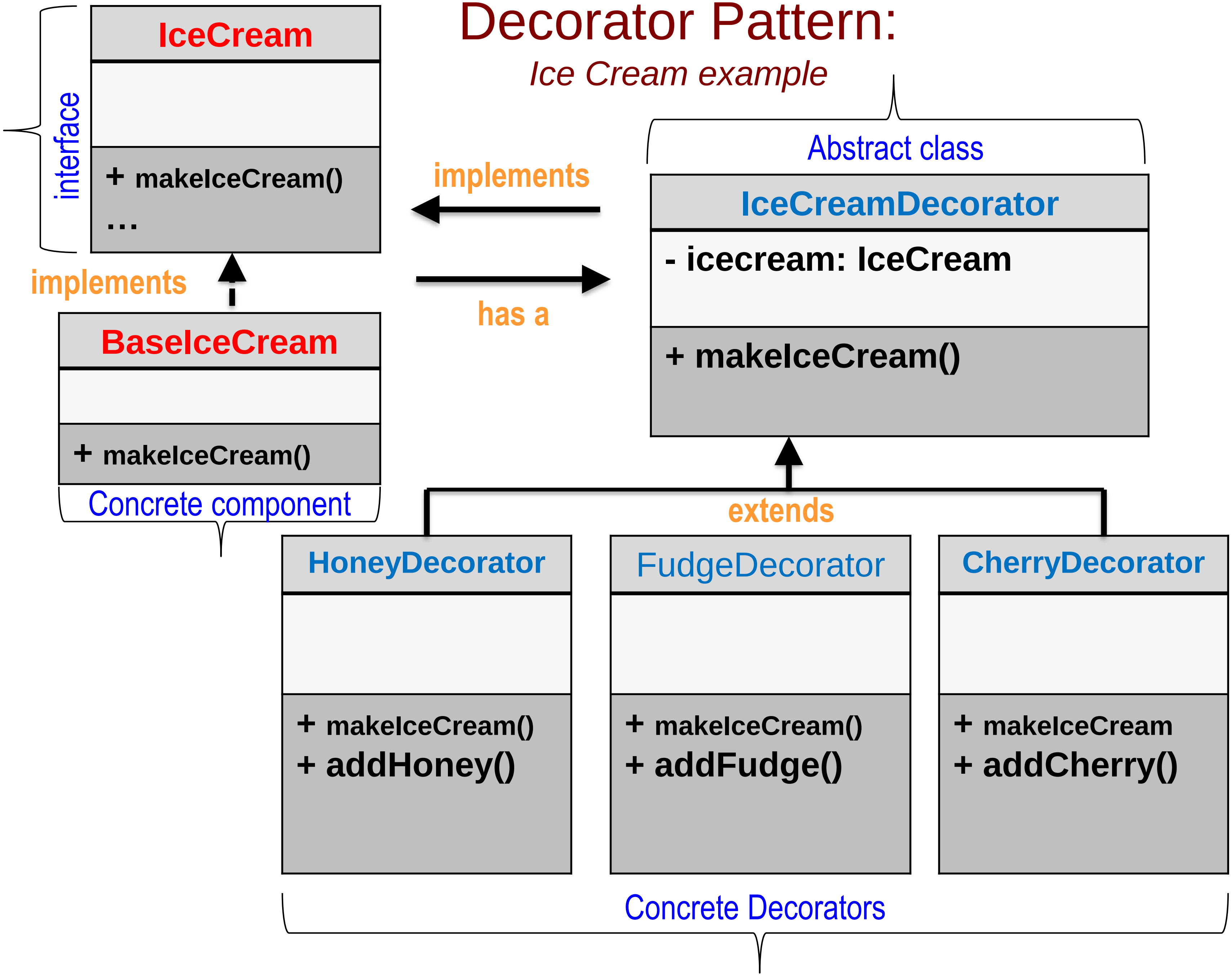
Decorator Pattern:

Ice Cream example



Decorator Pattern:

Ice Cream example



interface

IceCream

+ makeIceCream()

…

implements

BaseIceCream

+ makeIceCream()

Concrete component

implements

has a

Abstract class

IceCreamDecorator

- icecream: IceCream

+ makeIceCream()

extends

HoneyDecorator

+ makeIceCream()

+ addHoney()

FudgeDecorator

+ makeIceCream()

+ addFudge()

CherryDecorator

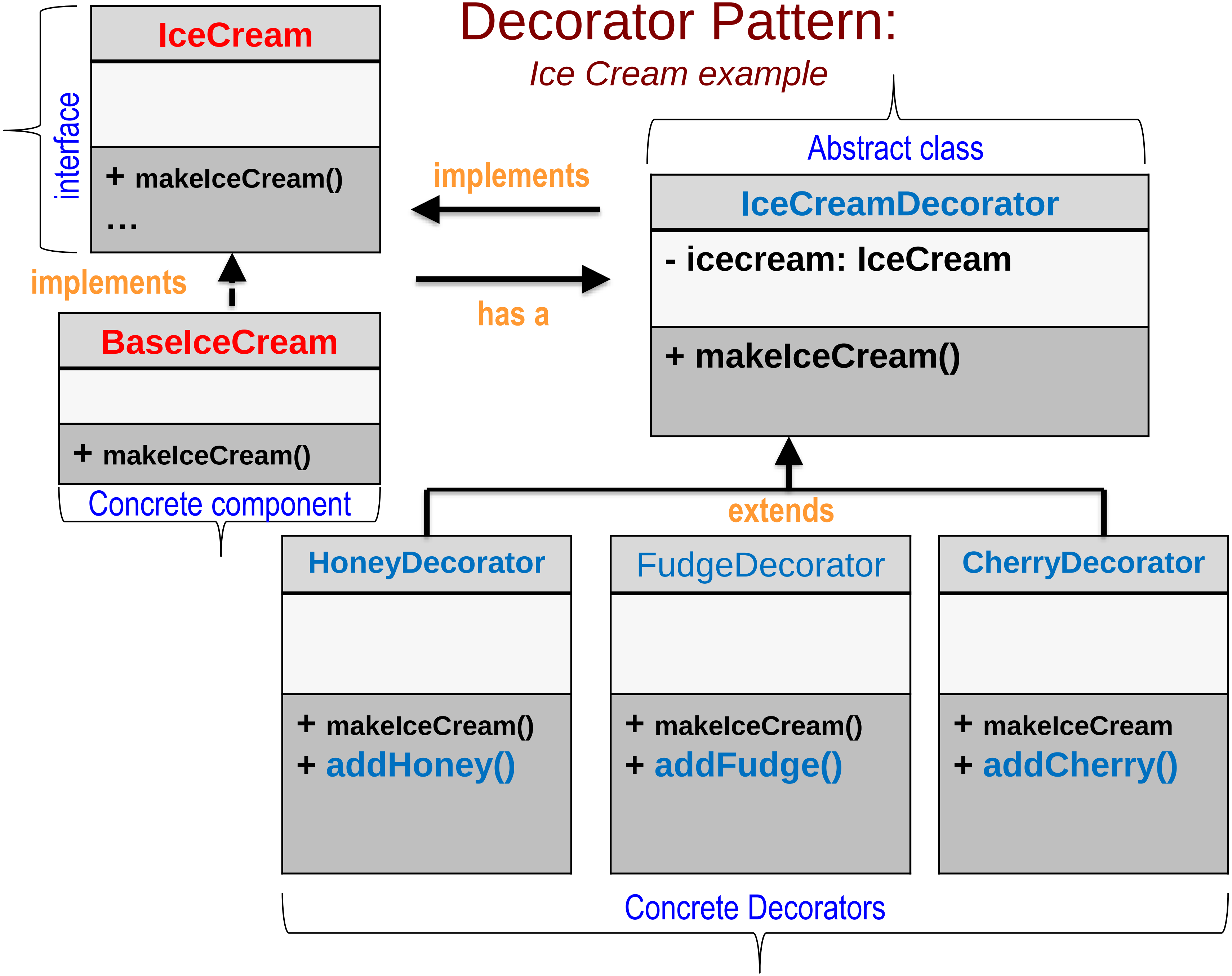
+ makeIceCream

+ addCherry()

Concrete Decorators

Decorator Pattern:

Ice Cream example



interface

IceCream

+ makeIceCream()
…

implements

BaseIceCream

+ makeIceCream()

Concrete component

implements

has a

Abstract class

IceCreamDecorator

- icecream: IceCream

+ makeIceCream()

extends

HoneyDecorator

+ makeIceCream()
+ addHoney()

FudgeDecorator

+ makeIceCream()
+ addFudge()

CherryDecorator

+ makeIceCream
+ addCherry()

Concrete Decorators

Implementation

```
abstract class IceCreamDecorator implements IceCream
{
  protected IceCream decoratedIceCream;

  public IceCreamDecorator(Icecream iceCream) {
    decoratedIceCream = iceCream;
  }

  public String makeIceCream() {
    return( decoratedIceCream.makeIceCream() );
  }

} // class
```

Implementation

```
abstract class IceCreamDecorator implements IceCream
{
  protected IceCream decoratedIceCream;

  public IceCreamDecorator(Icecream iceCream) {
    decoratedIceCream = iceCream;
  }

  public String makeIceCream() {
    return( decoratedIceCream.makeIceCream() );
  }

} // class
```

Implementation

```
abstract class IceCreamDecorator implements IceCream
{
  protected IceCream decoratedIceCream;

  public IceCreamDecorator(Icecream iceCream) {
    decoratedIceCream = iceCream;
  }

  public String makeIceCream() {
    return( decoratedIceCream.makeIceCream() );
  }

} // class
```

Implementation

```
abstract class IceCreamDecorator implements IceCream
{
  protected IceCream decoratedIceCream;

  public IceCreamDecorator(Icecream iceCream) {
    decoratedIceCream = iceCream;
  }

  public String makeIceCream() {
    return( decoratedIceCream.makeIceCream() );
  }

} // class
```

Implementation

```
public class FudgeDecorator extends IcecreamDecorator
{

  public FudgeDecorator(Icecream icecream) {
    super(icecream);
  }

  public String makeIcecream() {

    return super.makeIcecream() +
           addFudge();
  }

  private String addFudge() {
     ...
  }

}
```

Implementation

```
public class FudgeDecorator extends IcecreamDecorator
{

  public FudgeDecorator(Icecream icecream) {
    super(icecream);
  }

  public String makeIcecream() {

    return super.makeIcecream() +
           addFudge();
  }

  private String addFudge() {
     ...
  }

}
```

Implementation

```
public class FudgeDecorator extends IcecreamDecorator
{

  public FudgeDecorator(Icecream icecream) {
    super(icecream);
  }

  public String makeIcecream() {

    return super.makeIcecream() +
           addFudge();
  }

  private String addFudge() {
     ...
  }

}
```

Implementation

```
public class FudgeDecorator extends IcecreamDecorator
{

  public FudgeDecorator(Icecream icecream) {
    super(icecream);
  }

  public String makeIcecream() {

    return super.makeIcecream() +
           addFudge();
  }

  private String addFudge() {
     ...
  }

}
```

Implementation

```
public class FudgeDecorator extends IcecreamDecorator
{

   public FudgeDecorato
      super(icecream);
   }

   public String makeIcecre    () {

      return super.makeIcecream() +
             addFudge();
   }

   private String addFudge() {
       ...
   }

}
```

In the case that the IceCreamDecorator is an abstract class … and the method is implemented in the super class.

Implementation

```
public class FudgeDecorator extends IcecreamDecorator
{

  public FudgeDecorator(Icecream icecream) {
    super(icecream);
  }

  public String makeIcecream() {

    return decoratedIcecream.makeIcecream() +
           addFudge();
  }

  private String addF
     ...
  }

}
```

In the case that the IceCreamDecorator is an interface or the method is abstract … and the method is implemented in the sub class.

Implementation

```
public class FudgeDecorator extends IcecreamDecorator
{

  public FudgeDecorator(Icecream icecream) {
    super(icecream);
  }

  public String makeIcecream() {

    return super.makeIcecream() +
           addFudge();
  }

  private String addFudge() {
     ...
  }

}
```

Implementation

```
public class HoneyDecorator extends IcecreamDecorator
{















}
```

Implementation

```
public class CherryDecorator extends IcecreamDecorator
{














}
```

Implementation

```
public class MakeIceCream
{

   public static void main( String args[] ) {

      IceCream plain = new BaseIceCream();
      IceCream cherry = new CherryDecorator(plain));

      IceCream fancy = new CherryDecorator(
                         new HoneyDecorator(
                           new NuttyDecorator(
                             new BaseIceCream() ) ) );




   }
}
```

Implementation

```
public class MakeIceCream
{

   public static void main( String args[] ) {

      IceCream plain = new BaseIceCream();
      IceCream cherry = new CherryDecorator(plain));

      IceCream fancy = new CherryDecorator(
                         new HoneyDecorator(
                           new NuttyDecorator(
                             new BaseIceCream() ) ) );




   }
}
```

Implementation

```
public class MakeIceCream
{

   public static void main( String args[] ) {

      IceCream plain = new IceCream();
      IceCream cherry = new CherryDecorator(plain));

      IceCream fancy = new CherryDecorator(
                         new HoneyDecorator(
                           new NuttyDecorator(
                             new BaseIceCream() ) ) );




   }
}
```

Implementation

```
public class MakeIceCream
{

   public static void main( String args[] ) {

      IceCream plain = new IceCream();
      IceCream cherry = new CherryDecorator(plain));

      IceCream fancy = new CherryDecorator(
                         new HoneyDecorator(
                           new NuttyDecorator(
                             new BaseIceCream() ) ) );




   }
}
```

Implementation

```
public class MakeIceCream
{

   public static void main( String args[] ) {

      IceCream plain = new IceCream();
      IceCream cherry = new CherryDecorator(plain));

      IceCream fancy = new CherryDecorator(
                         new HoneyDecorator(
                           new NuttyDecorator(
                             new BaseIceCream() ) ) );




   }
}
```

Implementation

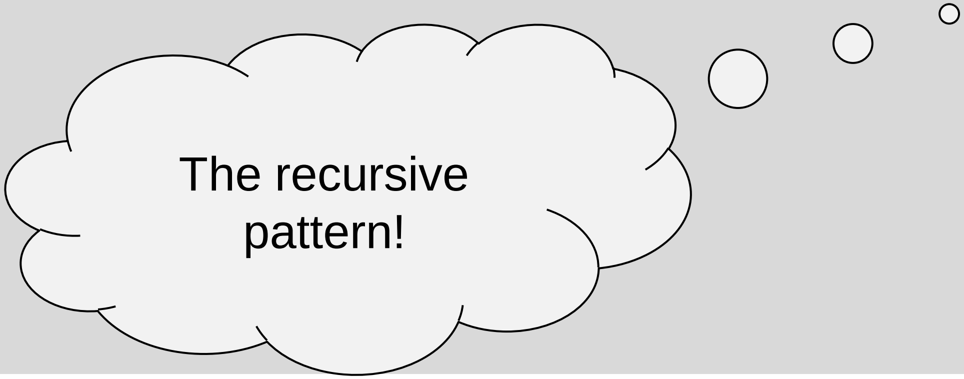
```
public class MakeIceCream
{

   public static void main( String args[] ) {

      IceCream plain = new IceCream();
      IceCream cherry = new CherryDecorator(plain));

      IceCream fancy = new CherryDecorator(
                         new HoneyDecorator(
                           new NuttyDecorator(
                             new BaseIceCream() ) ) );




   }
}
```

Implementation

```
public class MakeIceCream
{

   public static void main( String args[] ) {

      IceCream plain = new IceCream();
      IceCream cherry = new CherryDecorator(plain));

      IceCream fancy = new CherryDecorator(
                         new HoneyDecorator(
                           new NuttyDecorator(
                             new BaseIceCream() ) ) );




   }
}
```

Implementation

```
public class MakeIceCream
{

   public static void main( String args[] ) {

      IceCream plain = new IceCream();
      IceCream cherry = new CherryDecorator(plain));

      IceCream fancy = new CherryDecorator(
                         new HoneyDecorator(
                           new NuttyDecorator(
                             new BaseIceCream() ) ) );



   }
}
```



The recursive pattern!

Implementation

```
public class MakeIceCream
{

   public static void main( String args[] ) {

      IceCream plain = new IceCream();
      IceCream cherry = new CherryDecorator(plain));

      IceCream fancy = new CherryDecorator(
                         new HoneyDecorator(
                           new NuttyDecorator(
                             new BaseIceCream() ) ) );

      // Make the ice cream
      fancy.makeIceCream();

   }
}
```

Decorator Pattern:

Elements of Reusable OO Software

- **Consequences (Advantages/Disadvantages)**: The Decorator pattern provides more flexibility to add responsibility to objects than multiple inhe
  - Responsi
  - Object be
  - Avoids de
  - A decora y perspecti compone
  - End up with a lot of little objects.

This pattern allows object behavior to be enhanced and augmented during run-time!

Decorator Pattern:

Elements of Reusable OO Software

- Consequences (Advantages/Disadvantages): The Decorator pattern provides more flexibility to add responsibility to objects than multiple inheritance.
  - responsib
  - Object be
  - Avoids de
  - **A decora identity p ith the comp**
  - **End up with a lot of little objects.**

This pattern allows object behavior to be enhanced and augmented during run-time!

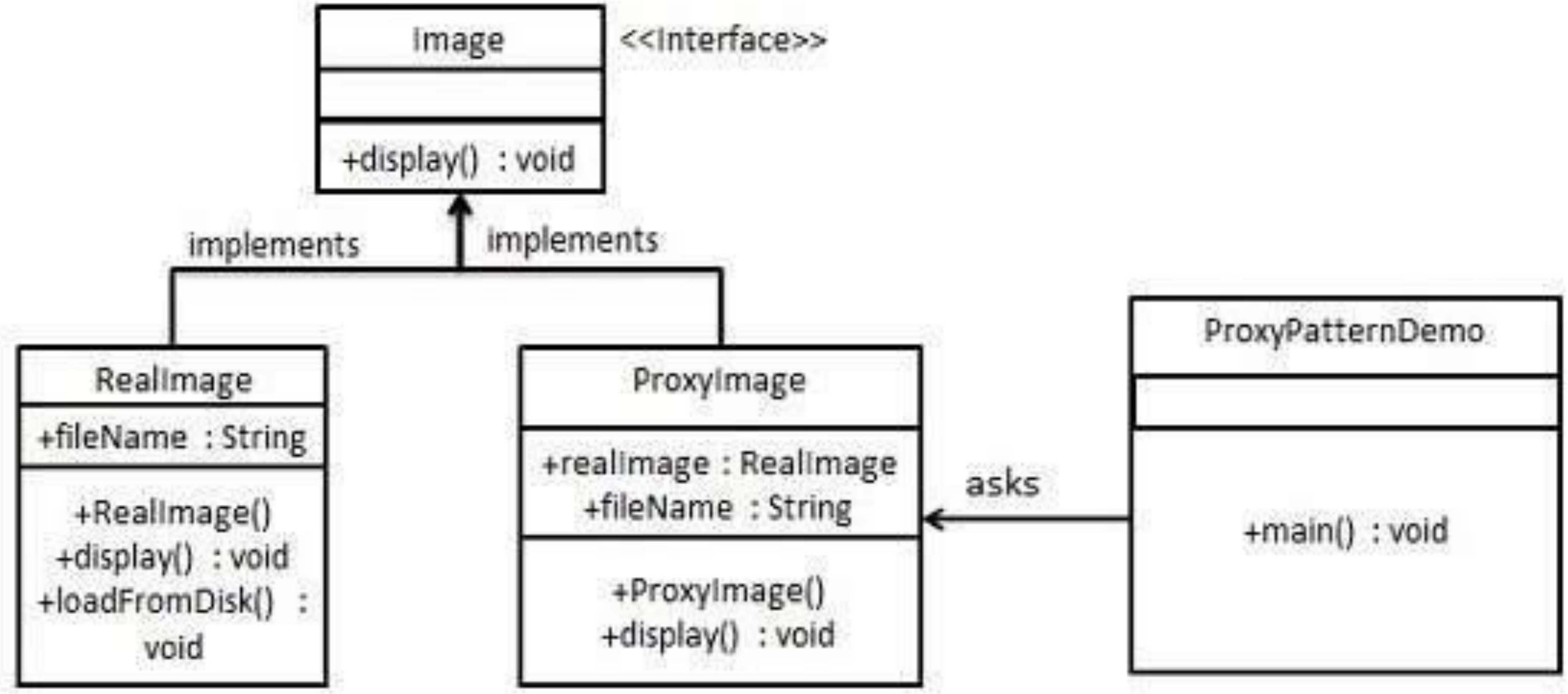
You end up with a lot of little objects….

Proxy Pattern

Intent*:* Provide a *surrogate* or placeholder for another object to control access to the target object.

Proxy Pattern

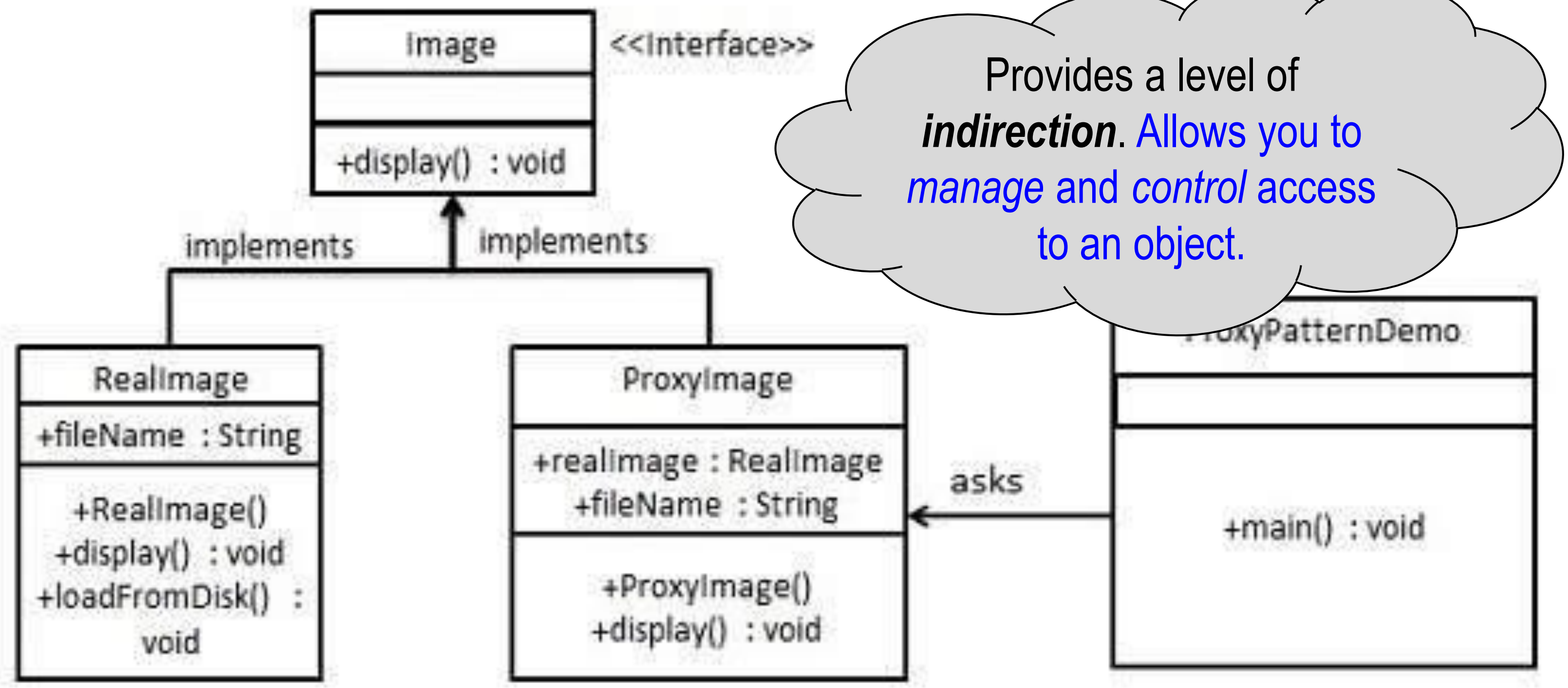
Intent*:* Provide a *surrogate* or placeholder for another object to control access to the target object.



TutorialsPoint

Proxy Pattern

Intent*:* Provide a *surrogate* or placeholder for another object to control access to the *target* object.



Provides a level of ***indirection***. Allows you to *manage* and *control* access to an object.

TutorialsPoint

Proxy Pattern:

Elements of Reusable OO Software

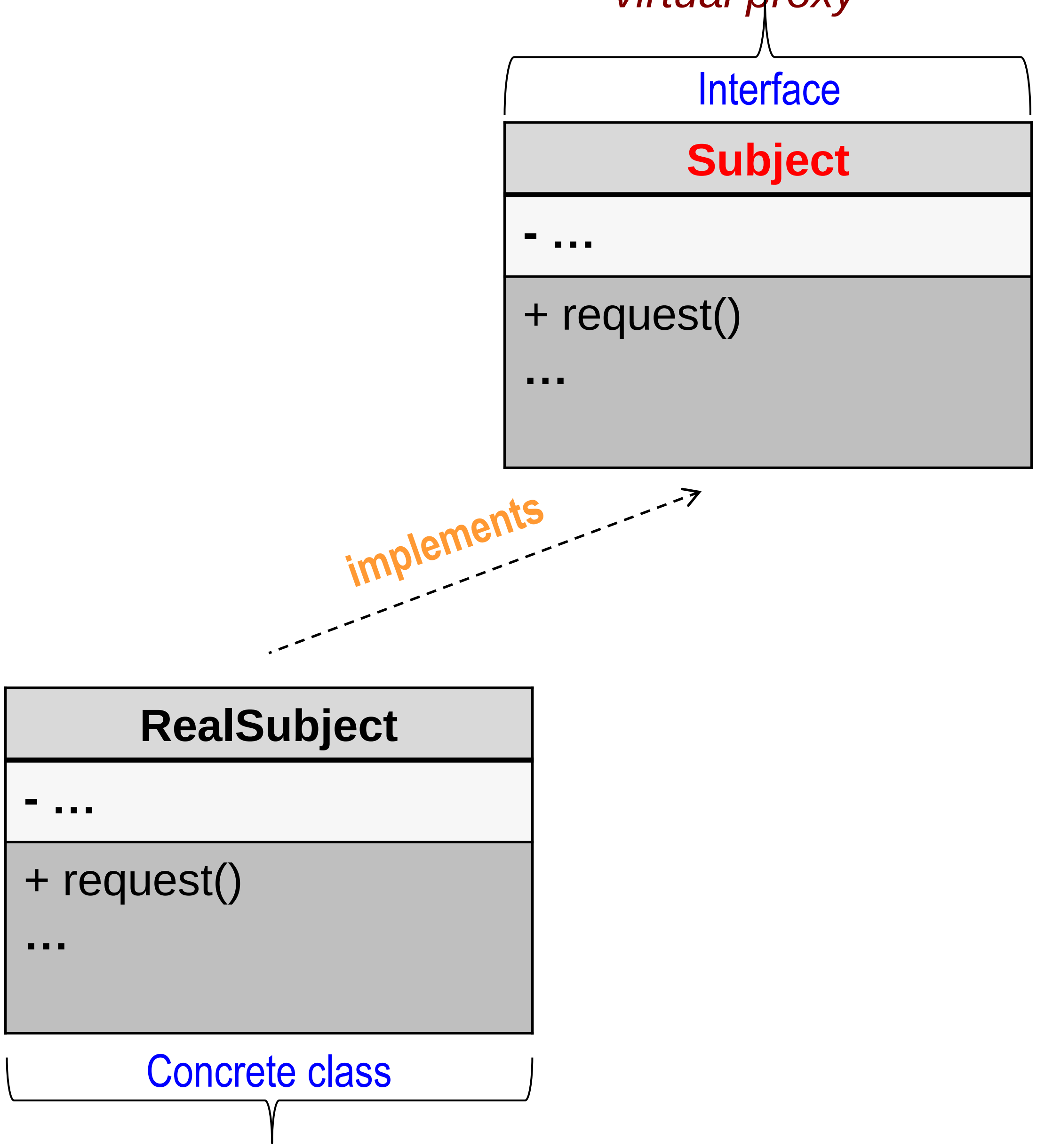
- Motivation **and Applicabilit** for controlling access to an object is to defe and initialization until we actually nee
  - Provide a leve g a surrogate object to control access to th ject.

All access related!

  - There are several common situations in which the Proxy pattern is useful:
    1. A ***remote*** proxy provides a local object representative for an object in a different address space.
    2. A ***virtual*** proxy creates expensive objects on demand (e.g. caching).
    3. A ***protection*** proxy controls access to the original object. Protection proxies are useful when objects should have different access rights.
    4. A ***smart*** proxy performs additional actions when an object is accessed (i.e. accounting, accessibility locking to ensure that only on client at a time has access to the object).

Proxy Pattern:

virtual proxy



Interface

Subject

- …

+ request()

…

implements

RealSubject

- …

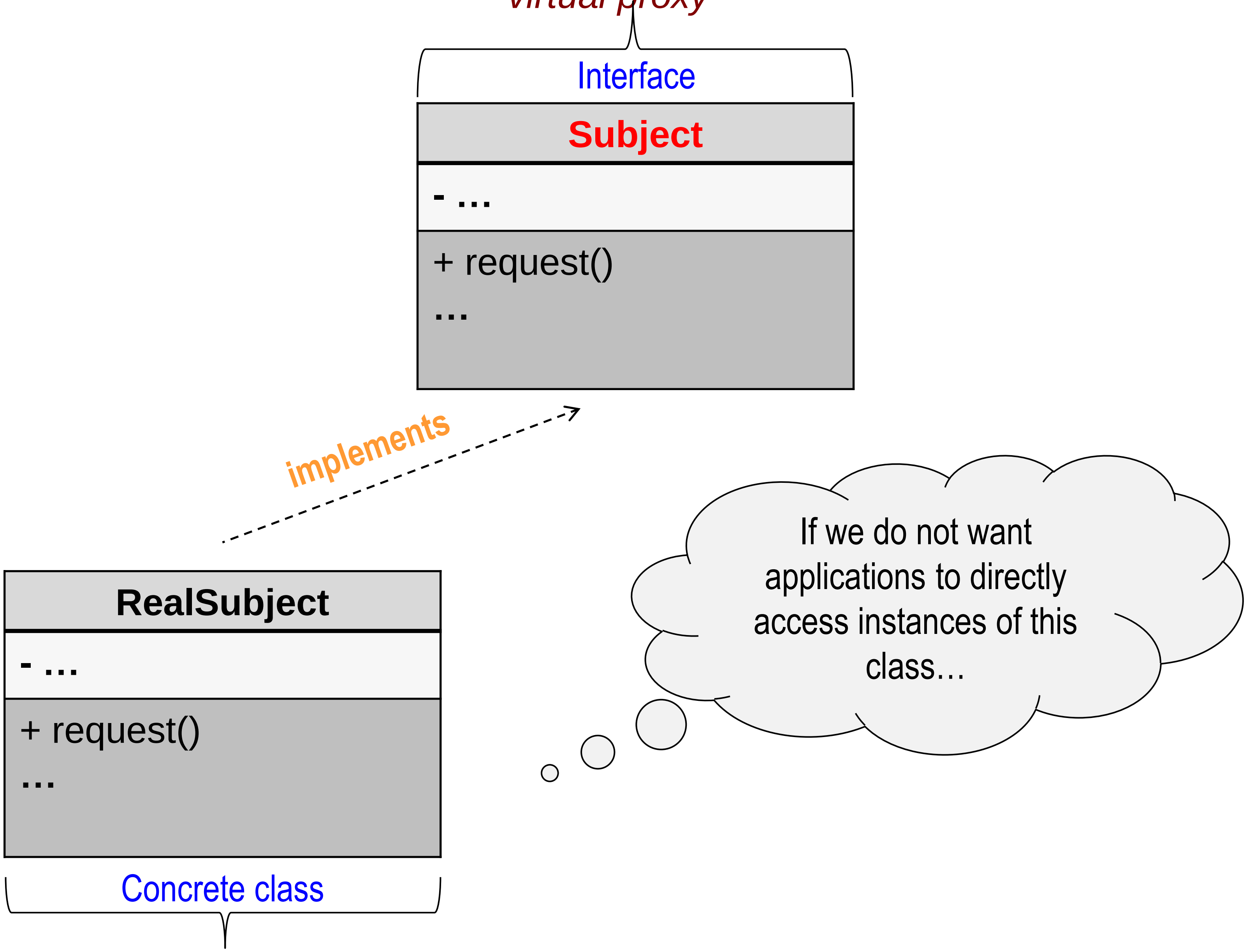
+ request()

…

Concrete class

Proxy Pattern:

virtual proxy



Interface

Subject

- …

+ request()

…

implements

RealSubject

- …

+ request()

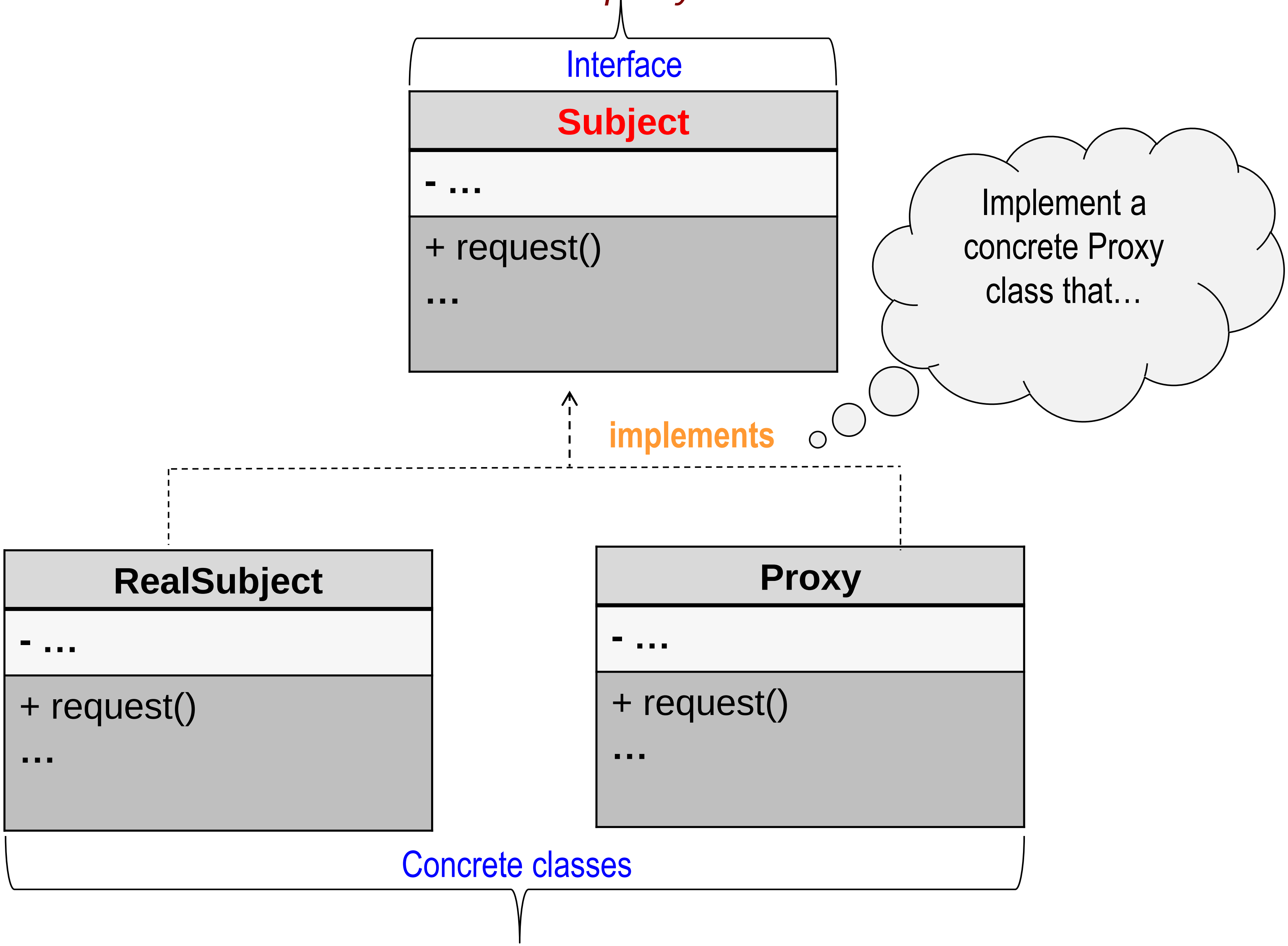
…

Concrete class

If we do not want applications to directly access instances of this class…

Proxy Pattern:

virtual proxy



Interface

Subject

- ...

+ request()

...

implements

Implement a concrete Proxy class that…

RealSubject

- ...

+ request()

...

Proxy

- ...

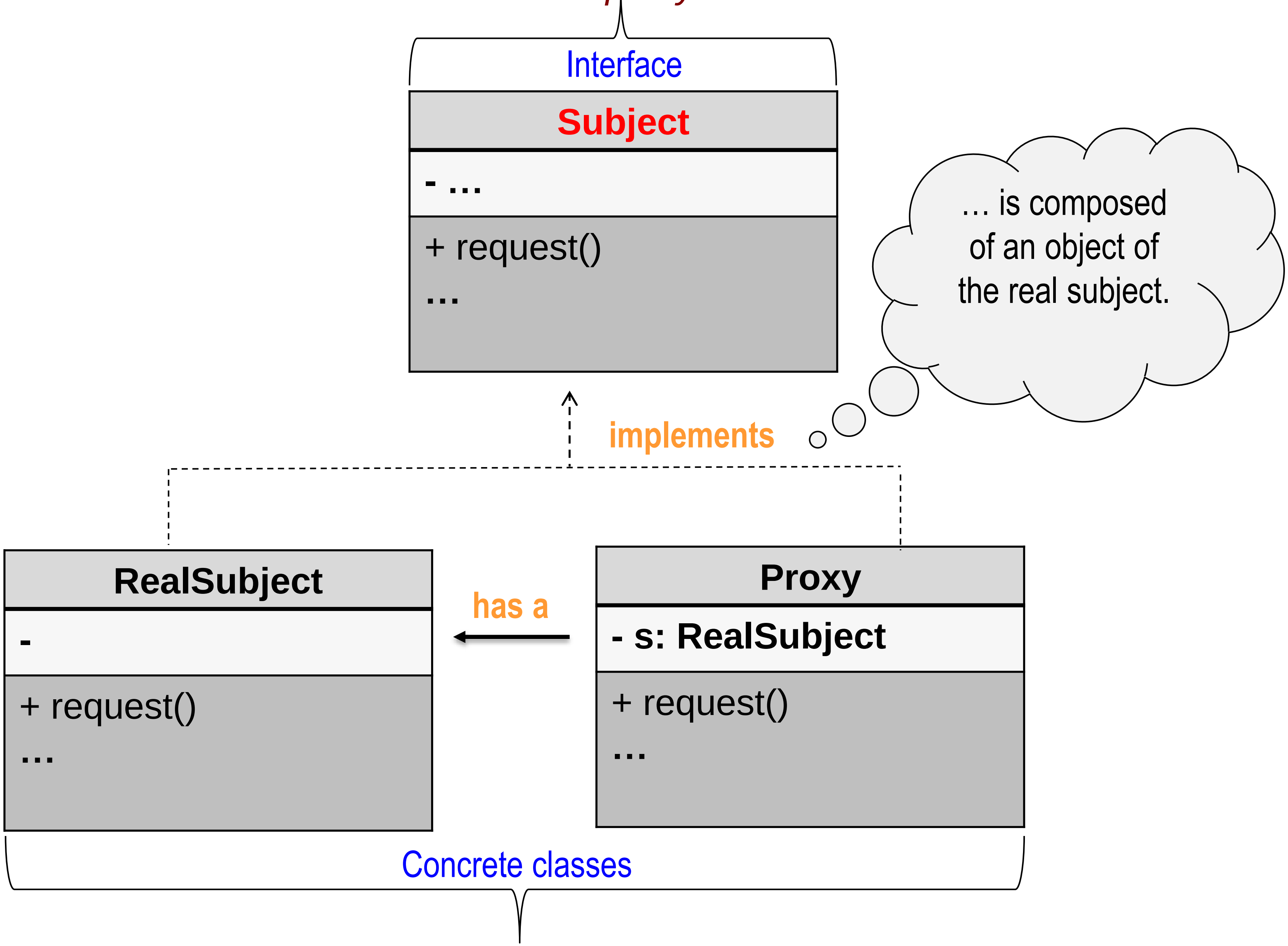
+ request()

...

Concrete classes

Proxy Pattern:

virtual proxy



Interface

Subject

- …

+ request()

…

… is composed of an object of the real subject.

implements

RealSubject

-

+ request()

…

has a

Proxy

- s: RealSubject

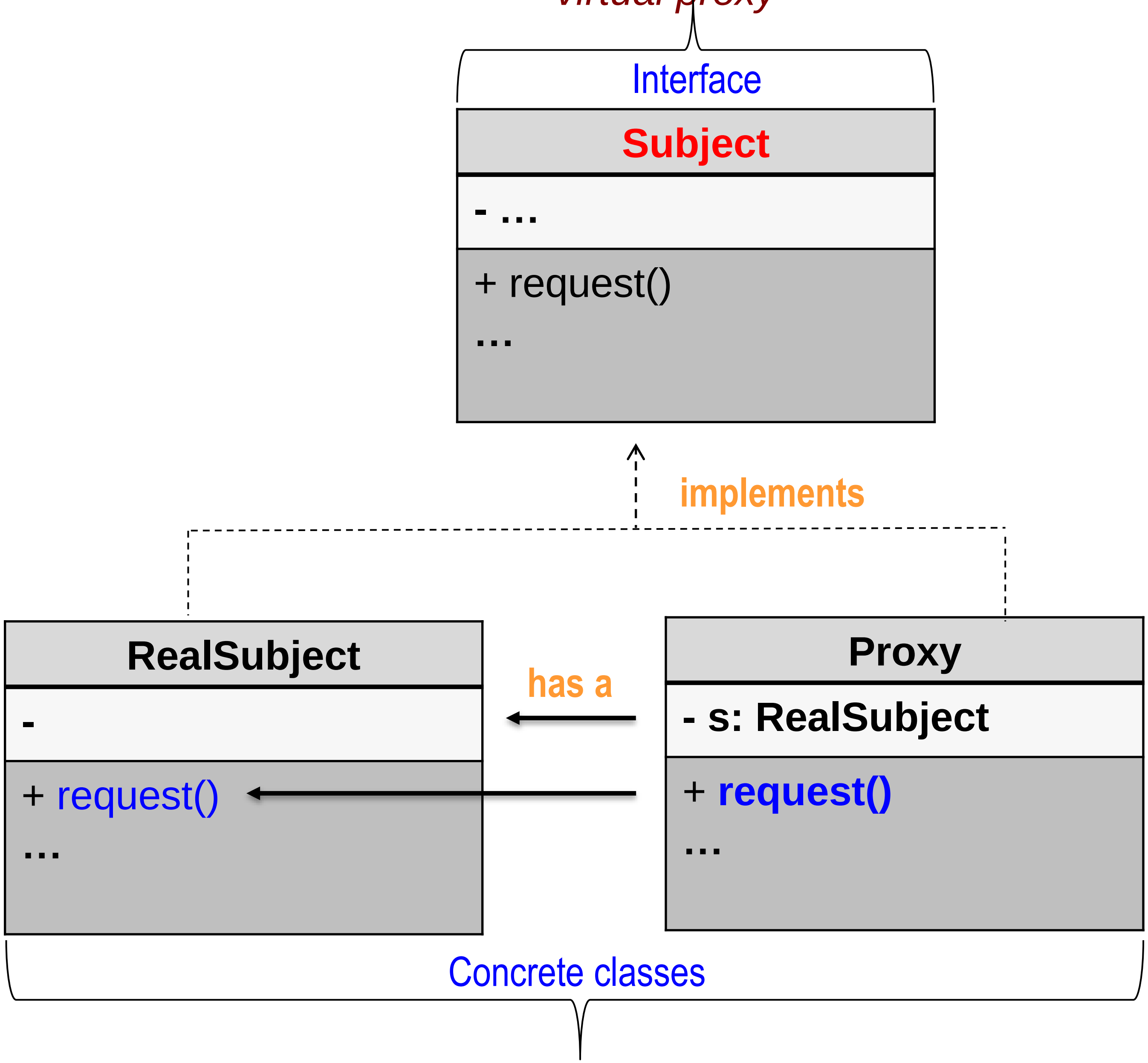
+ request()

…

Concrete classes

Proxy Pattern:

virtual proxy



Interface

Subject

- …

+ request()

…

implements

RealSubject

-

+ request()

…

has a

Proxy

- s: RealSubject

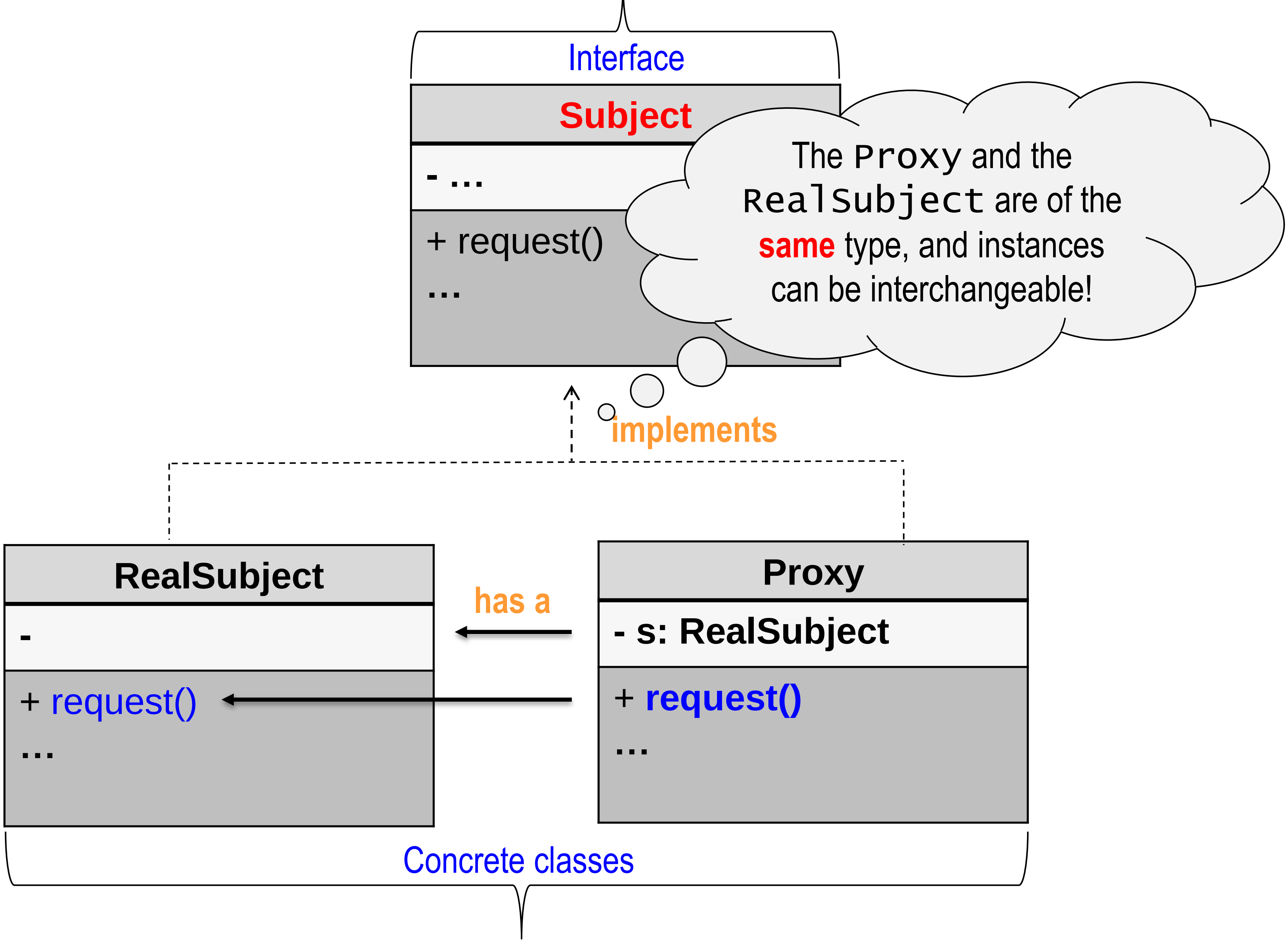
+ **request()**

…

Concrete classes

Proxy Pattern:

virtual proxy



Interface

Subject

- …

+ request()

…

The `Proxy` and the `RealSubject` are of the **same** type, and instances can be interchangeable!

implements

RealSubject

-

+ request()

…

has a

Proxy

- s: RealSubject

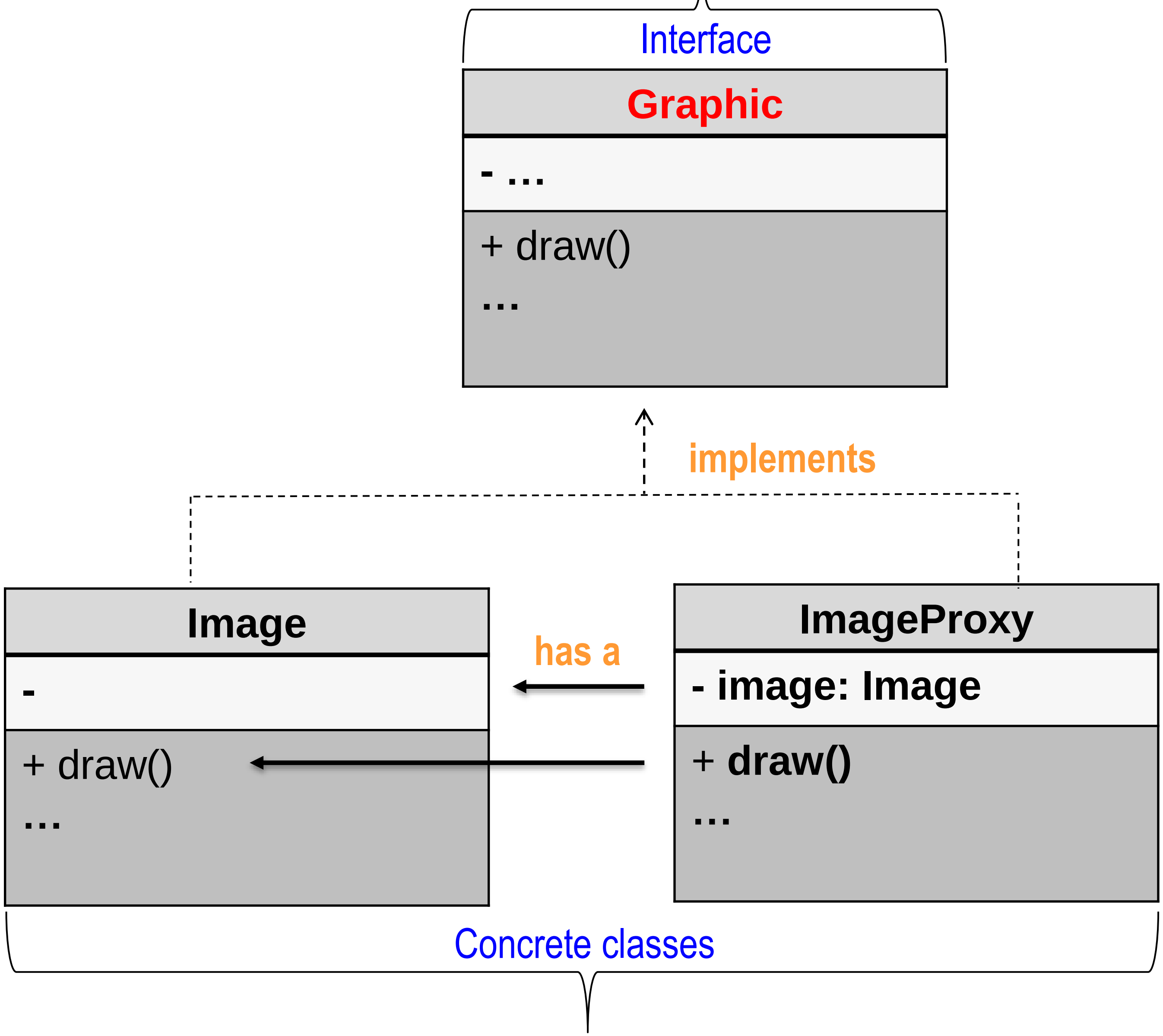
+ **request()**

…

Concrete classes

Proxy Pattern:

virtual proxy



Interface

Graphic

- …

+ draw()

…

implements

Image

-

+ draw()

…

has a

ImageProxy

- image: Image

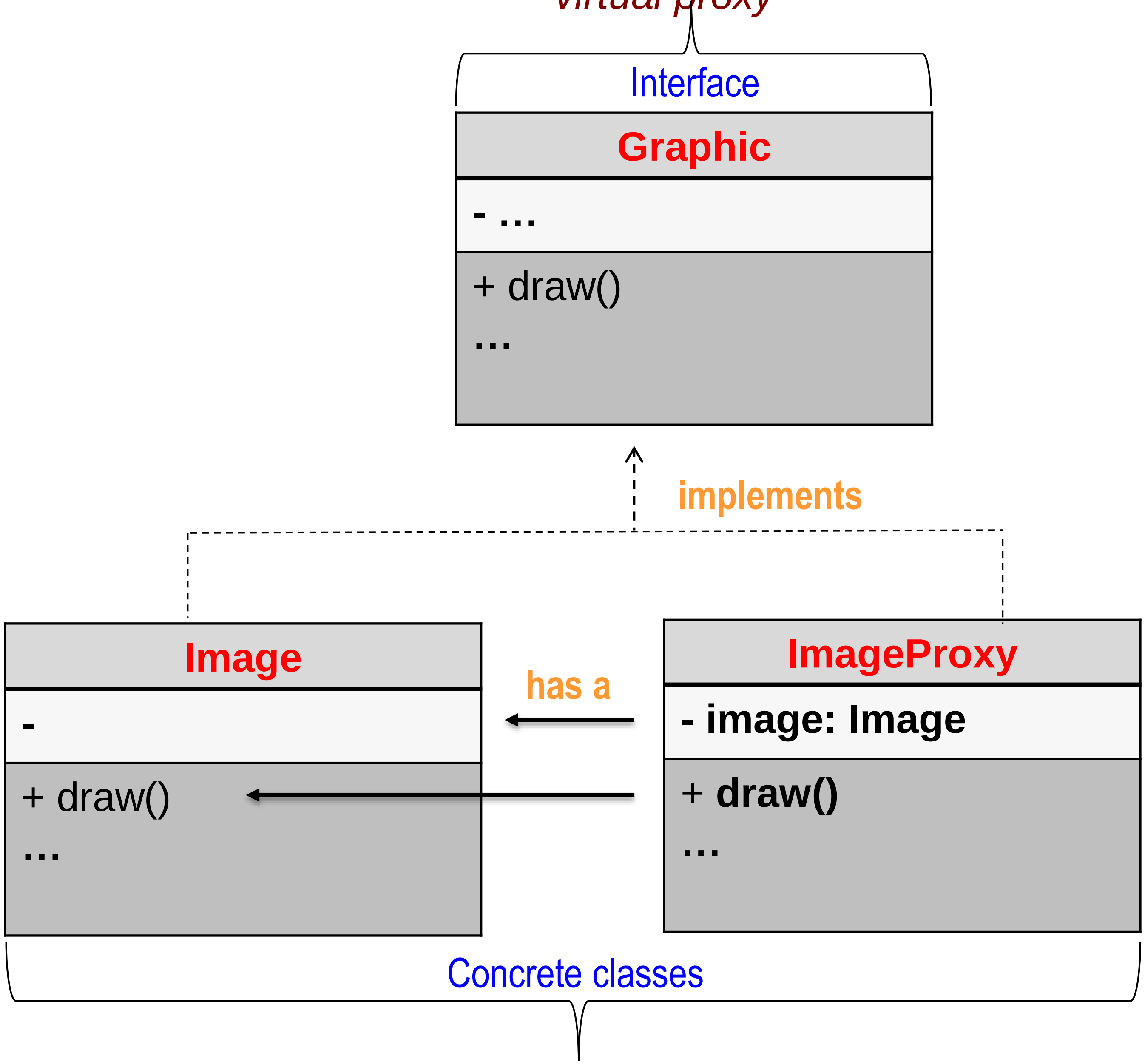
+ **draw()**

…

Concrete classes

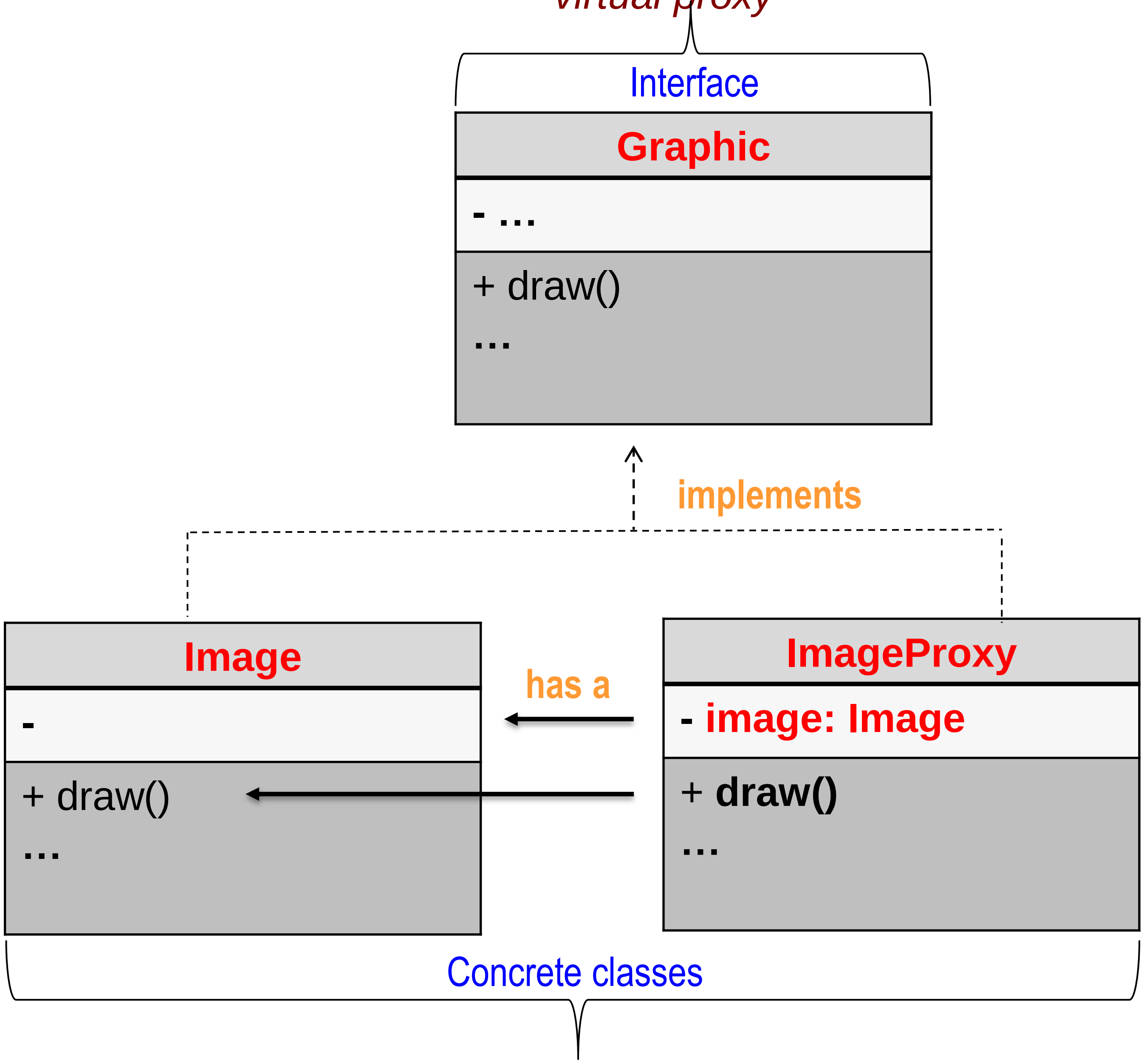
Proxy Pattern:

virtual proxy



Proxy Pattern:

virtual proxy



Interface

Graphic

- …

+ draw()

…

implements

Image

-

+ draw()

…

has a

ImageProxy

- image: Image

+ draw()

…

Concrete classes

Implementation:

Image

```
public interface Graphic {

   void display();
} // interface
```

Implementation:

Image

```
public interface Graphic {

   void display();
} // interface
```

```
public class Image implements Graphic {
   private String fileName;
   // reference to an image
   public Image(String fileName){
      this.fileName = fileName;
      loadFromDisk(fileName);
   }

   public void display() {
      ...
   }

   private void loadFromDisk(String fileName){
      ...
   }
} // class
```

Implementation:

Image

```
public interface Graphic {

   void display();
} // interface
```

```
public class Image implements Graphic {
   private String fileName;
   // reference to an image
   public Image(String fileName){
      this.fileName = fileName;
      loadFromDisk(fileName);
   }

   public void display() {
      ...
   }

   private void loadFromDisk(String fileName){
      ...
   }
} // class
```

Implementation:

Image

```
public interface Graphic {

   void display();
} // interface
```

```
public class Image implements Graphic {
   private String fileName;
   // reference to an image
   public Image(String fileName){
      this.fileName = fileName;
      loadFromDisk(fileName);
   }

   public void display() {
      ...
   }

   private void loadFromDisk(String fileName){
      ...
   }
} // class
```

Implementation:

Image Proxy

```
public interface Graphic {

   void display();
} // interface
```

```
public class ImageProxy implements Graphic {
   private String fileName;
   private Image realImage = null;

   public ImageProxy(String fileName){
      this.fileName = fileName;
   }

   public void display() {
      if (!realImage)
         realImage = new Image(filename);
      realImage.display();
   }

} // class
```

Implementation:

Image Proxy

```
public interface Graphic {

   void display();
} // interface
```

```
public class ImageProxy implements Graphic {
   private String fileName;
   private Image realImage = null;

   public ImageProxy(String fileName){
      this.fileName = fileName;
   }

   public void display() {
      if (!realImage)
         realImage = new Image(filename);
      realImage.display();
   }

} // class
```

Implementation:

Image Proxy

```
public interface Graphic {

   void display();
} // interface
```

```
public class ImageProxy implements Graphic {
   private String fileName;
   private Image realImage = null;

   public ImageProxy(String fileName){
      this.fileName = fileName;
   }

   public void display() {
      if (!realImage)
         realImage = new Image(filename);
      realImage.display();
   }

} // class
```

Implementation:

Image Proxy

```
public interface Graphic {

   void display();
} // interface
```

```
public class ImageProxy implements Graphic {
   private String fileName;
   private Image realImage = null;

   public ImageProxy(String fileName){
      this.fileName = fileName;
   }

   public void display() {
      if (!realImage)
         realImage = new Image(filename);
      realImage.display();
   }

} // class
```

Implementation:

Image Proxy

```
public interface Graphic {

   void display();
} // interface
```

```
public class ImageProxy implements Graphic {
   private String fileName;
   private Image realImage = null;

   public ImageProxy(String fileName){
      this.fileName = fileName;
   }

   public void display() {
      if (!realImage)
         realImage = new Image(filename);
      realImage.display();
   }

} // class
```

Implementation

```
public class ProxyDemo {

   public static void main ( ... ) {
      Image image = new ImageProxy("someFile.jpg");

      image.display();     // will load and display

      image.display();     // will display

   }

} // class
```

Implementation

```
public class ProxyDemo {

   public static void main ( ... ) {
      Image image = new ImageProxy("someFile.jpg");

      image.display();    // will load and display

      image.display();    // will display

   }

} // class
```

Implementation

```
public class ProxyDemo {

   public static void main ( ... ) {
      Image image = new ImageProxy("someFile.jpg");

      image.display();    // will load and display

      image.display();    // will display

   }

} // class
```

Implementation

```
public class ProxyDemo {

   public static void main ( ... ) {
      Image image = new ImageProxy("someFile.jpg");

      image.display();    // will load and display

      image.display();    // will display

   }

} // class
```

Implementation

```
public class ProxyDemo {

   public static void main ( ... ) {
      Image image = new ImageProxy("someFile.jpg");

      // image.display()



   }

} // class
```

The image is never loaded!

Proxy Pattern:
Elements of Reusable OO Software

- Consequences (Advantages/Disadvantages): The Proxy pattern introduces a level of indirection when accessing an object. The additional in of the proxy cla
  - A remote different a
  - A virtual object on
  - Both prot housekee accessed.
  - Have to add layering if trying to hide the real objects from being created.

Depending on the proxy, this added layer can provide a level of protection, indirection, or housekeeping.

Proxy Pattern:

Elements of Reusable OO Software

- Consequences (Advantages/Disadvantages): The Proxy pattern introduces a level of indirection when accessing an object. The
additional in f
the proxy cla
  - A remote
different a
  - A virtual
object on
  - Both prot
housekee
accessed.
  - Have to add layering if trying to hide the real objects from being created.

Depending on the proxy, this added layer can provide a level of protection, indirection, or housekeeping.

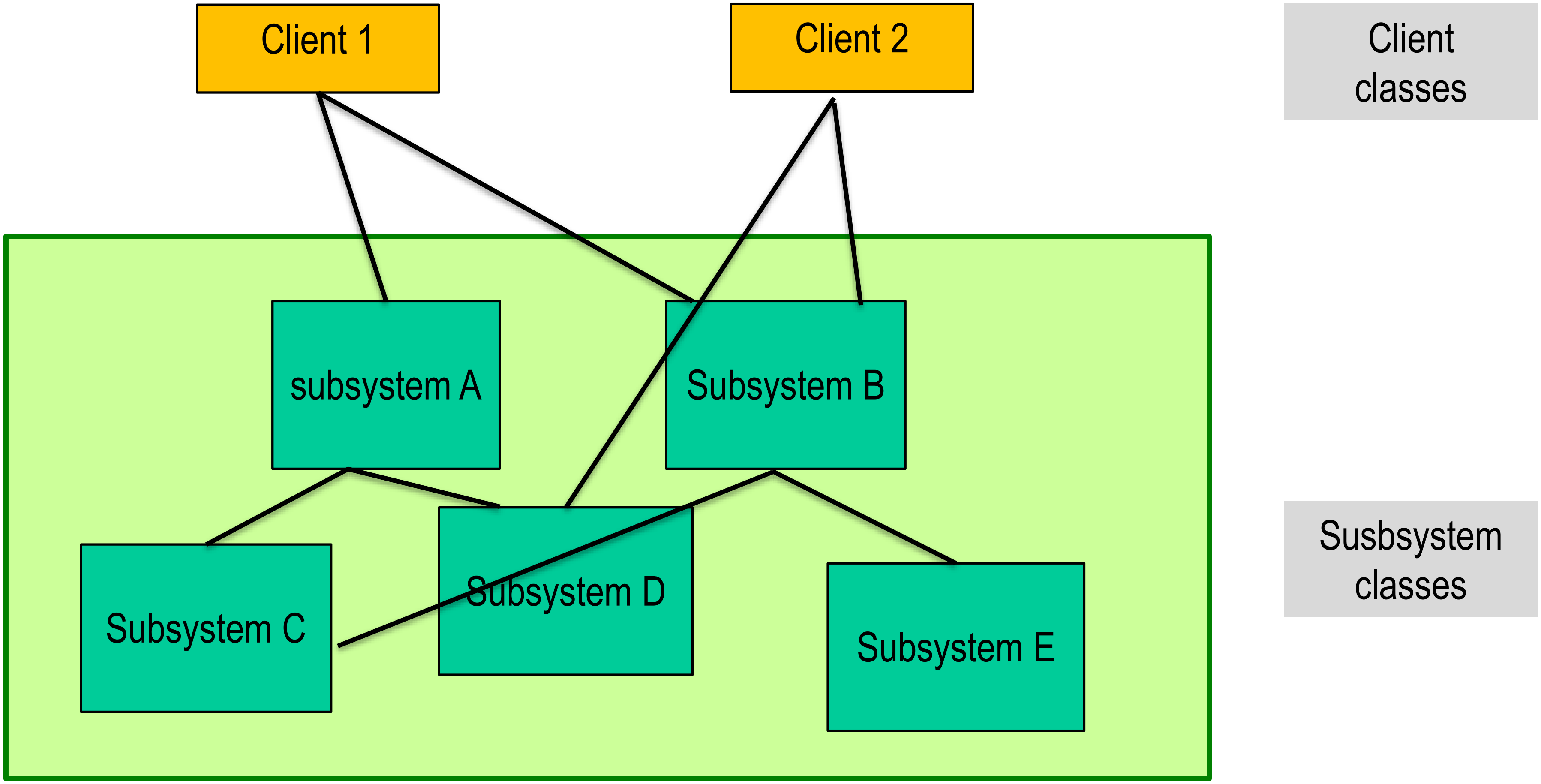
But, it requires adding a layer….

Facade Pattern

Intent*:* Provide a *unified interface* to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.

Facade Pattern

Intent*:* Provide a *unified interface* to a set of interfaces in a subsystem. Facade defines a *higher-level interface* that makes the subsystem easier to use.



Client 1

Client 2

Client classes

subsystem A

Subsystem B

Subsystem C

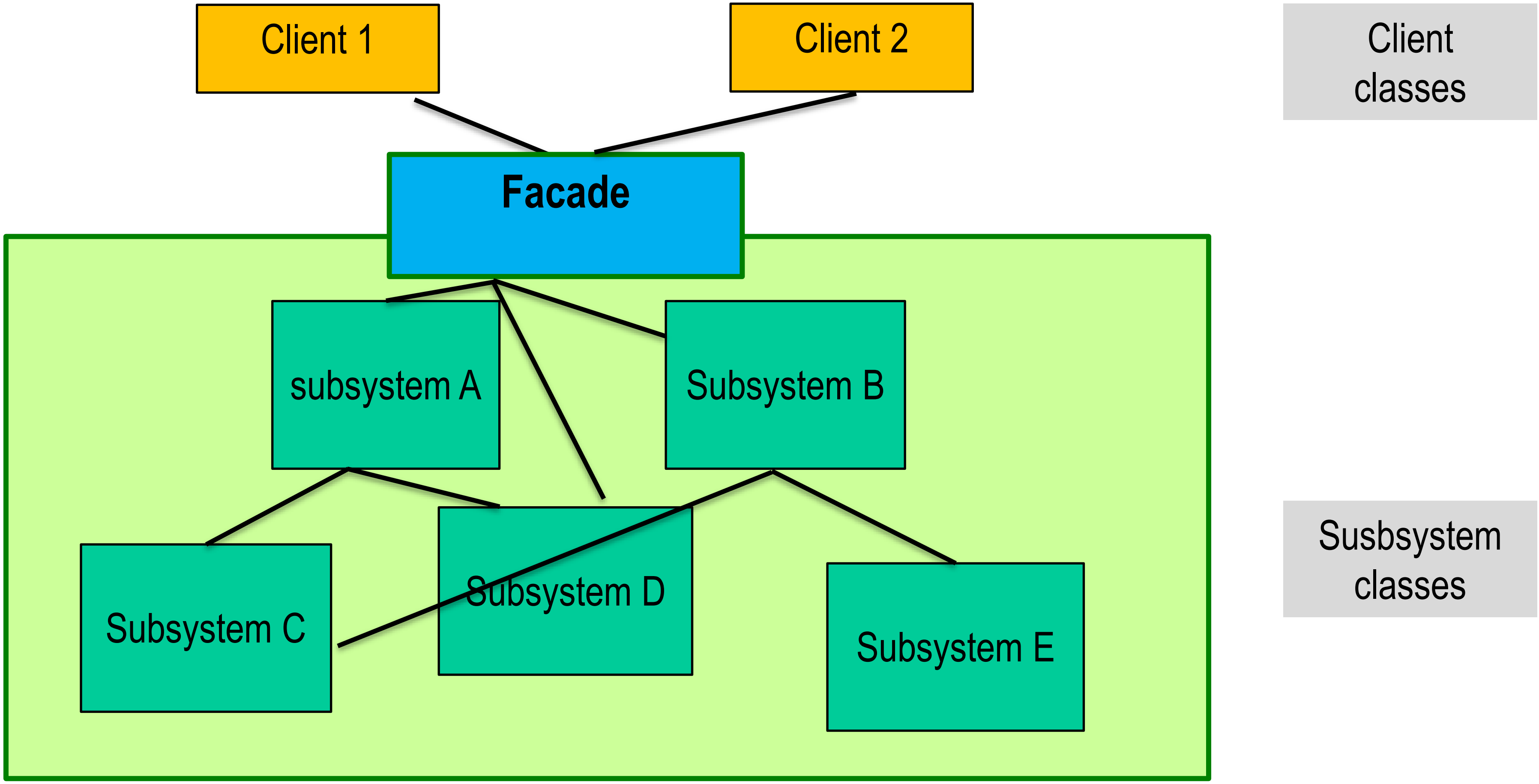
Subsystem D

Subsystem E

Susbsystem classes

Facade Pattern

Intent: Provide a *unified interface* to a set of interfaces in a subsystem. Facade defines a *higher-level interface* that makes the subsystem easier to use.



Client 1

Client 2

Client classes

Facade

subsystem A

Subsystem B

Subsystem C

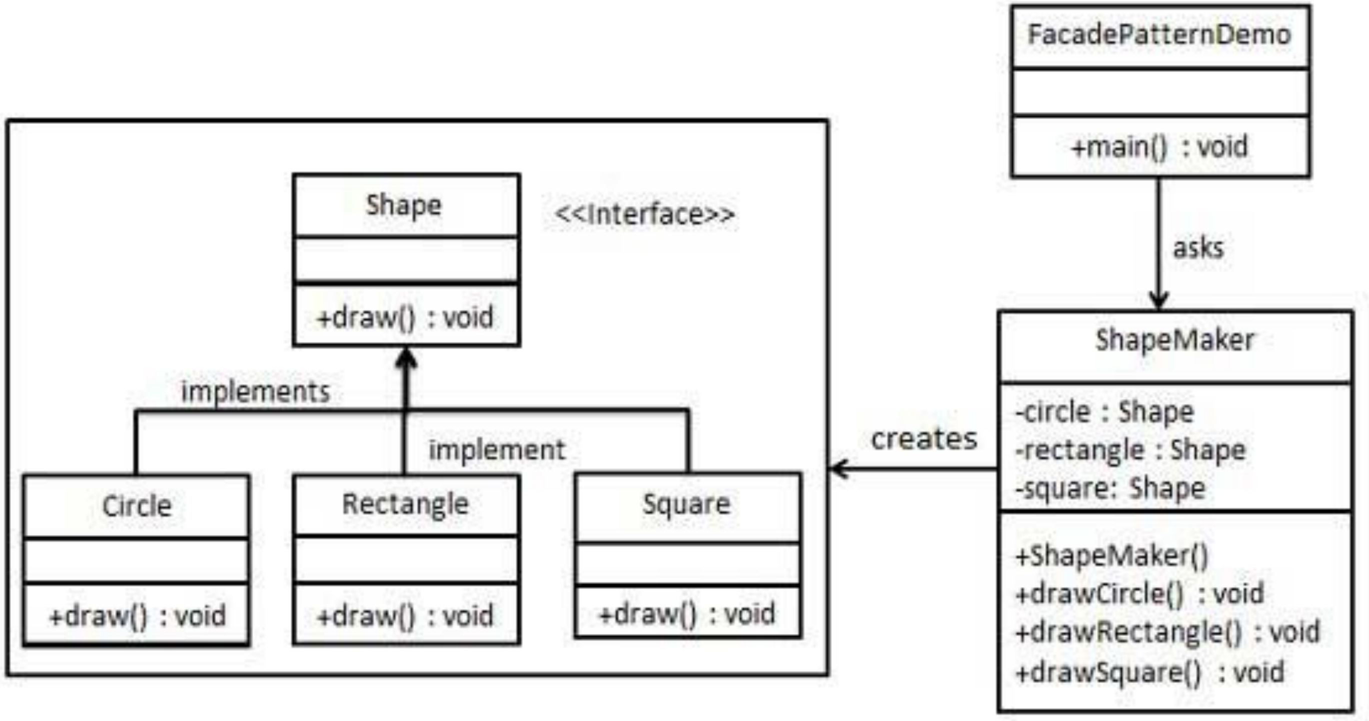
Subsystem D

Subsystem E

Susbsystem classes

Facade Pattern

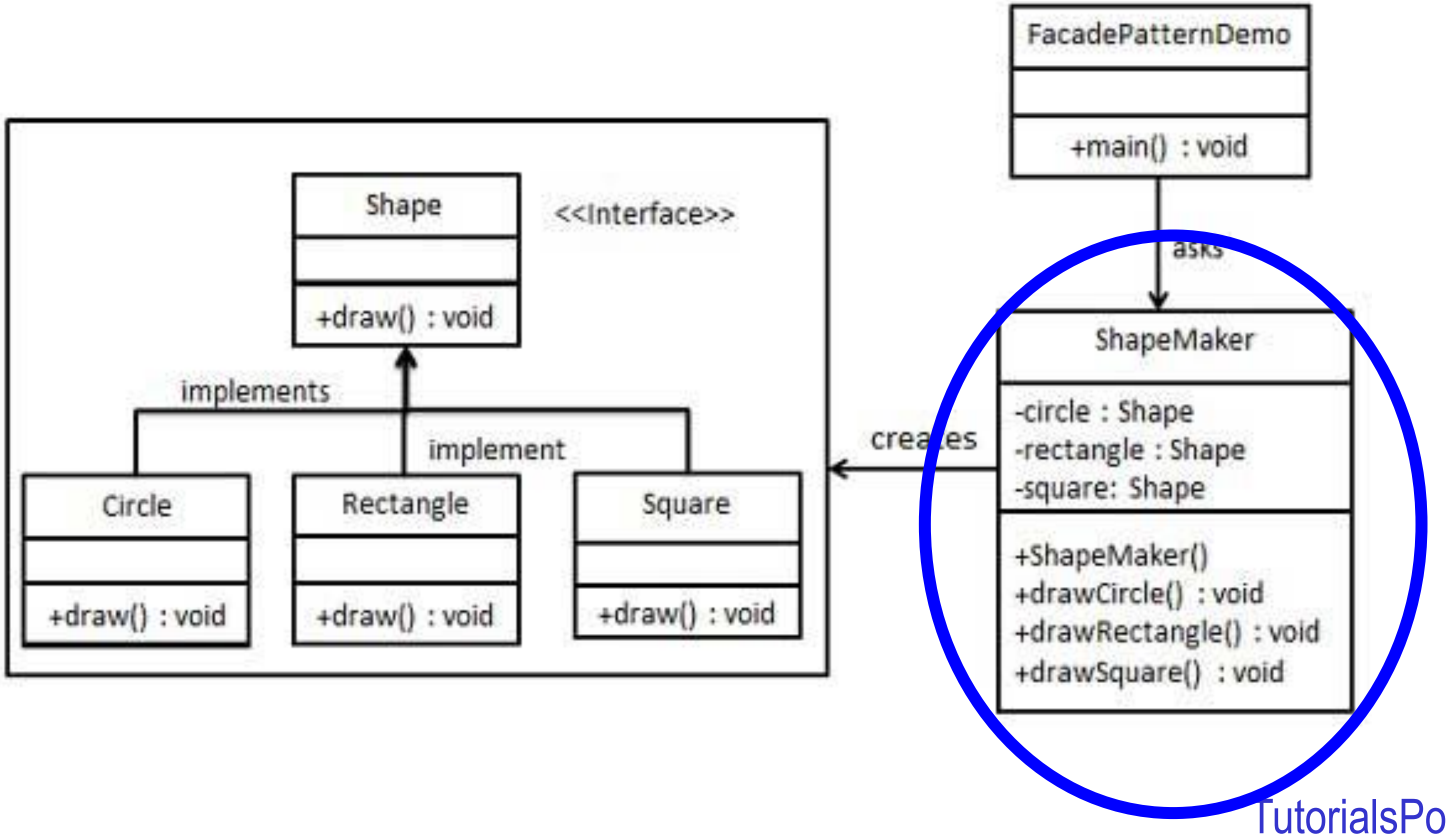
Intent*:* Provide a *unified interface* to a set of interfaces in a subsystem. Facade defines a *higher-level interface* that makes the subsystem easier to use.



TutorialsPoint

Facade Pattern

Intent*:* Provide a *unified interface* to a set of interfaces in a subsystem. Facade defines a *higher-level interface* that makes the subsystem easier to use.



TutorialsPoint

Facade Pattern:

Elements of Reusable OO Software

- **Motivation** and Applicability: Structuring or decomposing a system into sub-systems helps reduce the complexity and allows to better
understand t
dependencie d
an applicatio
dependencie
  - A façade re
general fa
  - Shields cl
know abo

What if you have a complicated set of program types and you want to simplify the interface that clients use?

Facade Pattern:

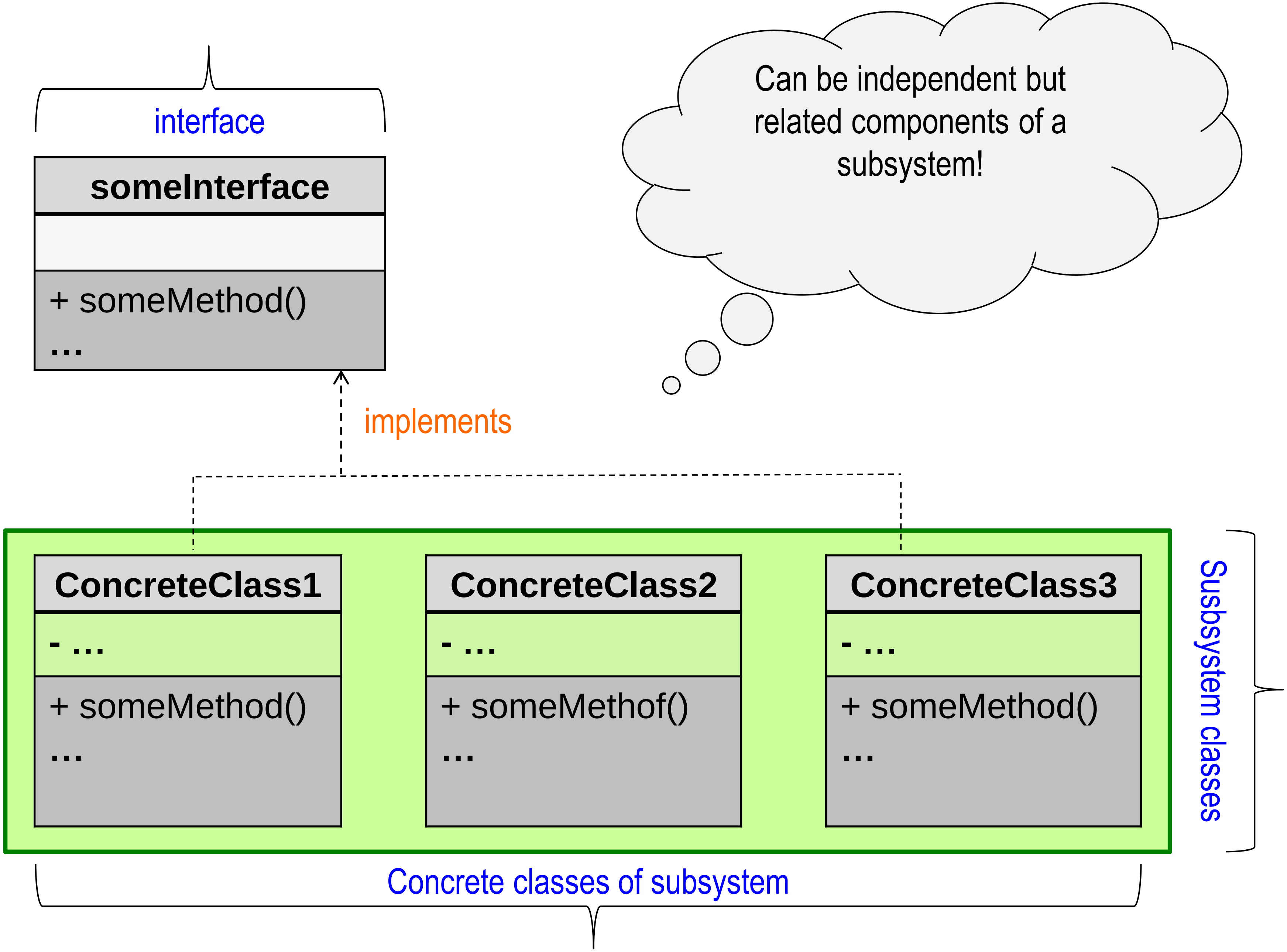
Elements of Reusable OO Software

- Motivation **and Applicability**: Structuring or decomposing a system in sub-systems helps reduce the complexity and allows to better understand t dependencie d an applicatio dependencie
  - A façade re general fa
  - Shields cl know abo
  - You want to provide a simple interface to a complex subsystem.
  - **Decouple the subsystem from clients and higher level applications.**
  - **Want to promote subsystem independence and portability.**
  - Create a layered subsystem, by providing a façade entry point to each subsystem.

What if you have a complicated set of program types and you want to simplify the interface that clients use?

Create a layered subsystem, and provide a façade entry point to each subsystem.

Facade Pattern



interface

someInterface

+ someMethod()
...

Can be independent but related components of a subsystem!

implements

ConcreteClass1

- ...

+ someMethod()
...

ConcreteClass2

- ...

+ someMethof()
...

ConcreteClass3

- ...

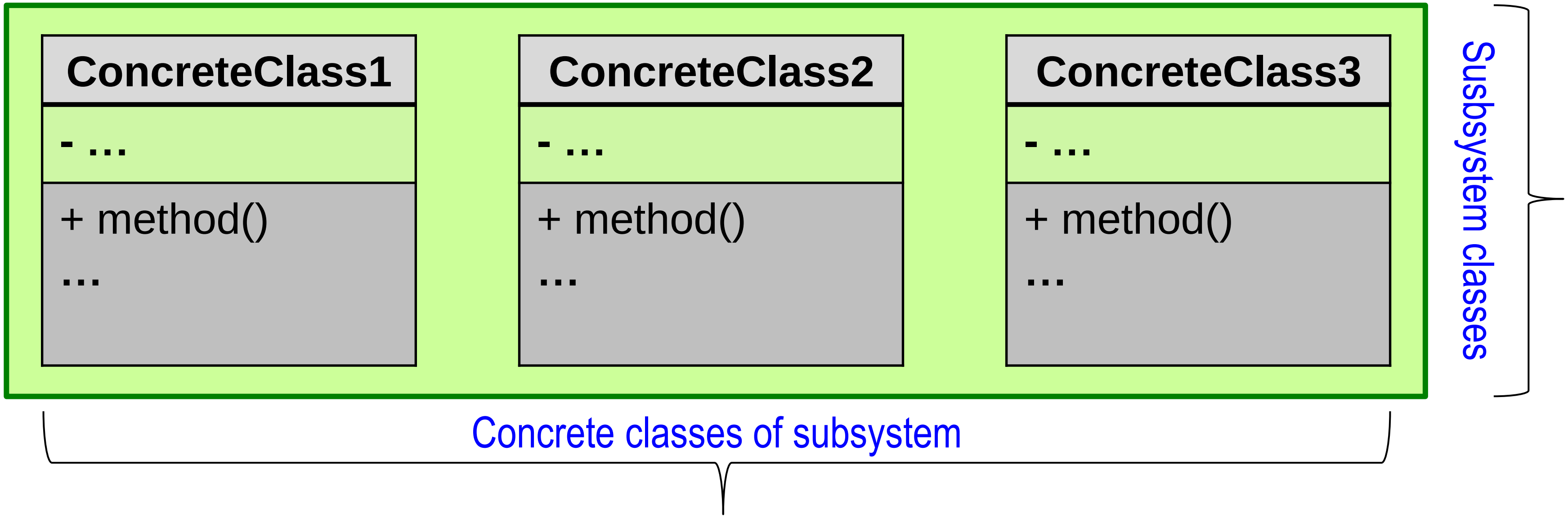
+ someMethod()
...

Susbsystem classes

Concrete classes of subsystem

Facade Pattern

Can also be independent but unrelated components of a subsystem!



ConcreteClass1

- ...

+ method()

...

ConcreteClass2

- ...

+ method()

...

ConcreteClass3

- ...

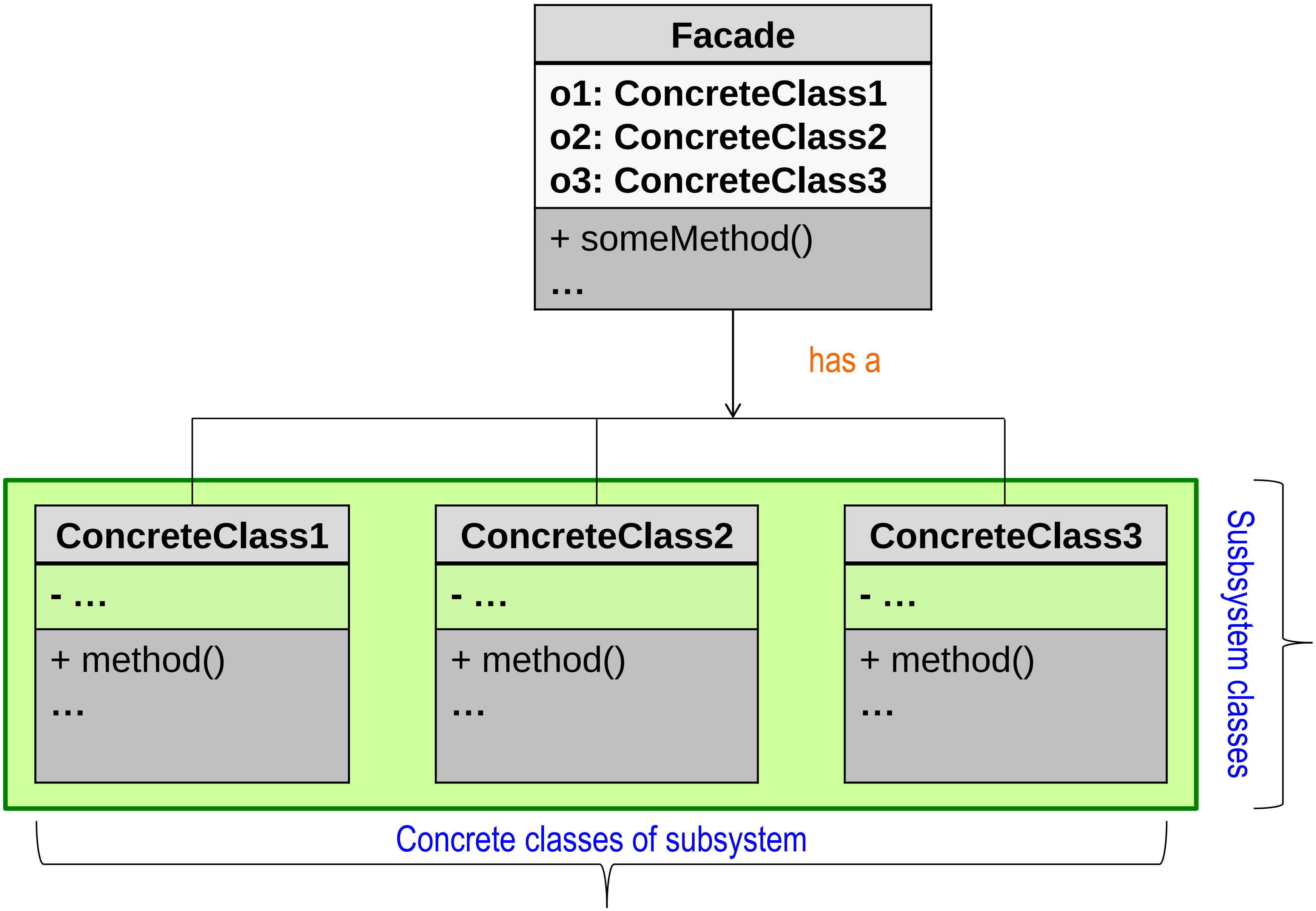
+ method()

...

Susbsystem classes

Concrete classes of subsystem

Facade Pattern



Facade

o1: ConcreteClass1
o2: ConcreteClass2
o3: ConcreteClass3

+ someMethod()
…

has a

ConcreteClass1

- …

+ method()
…

ConcreteClass2

- …

+ method()
…

ConcreteClass3

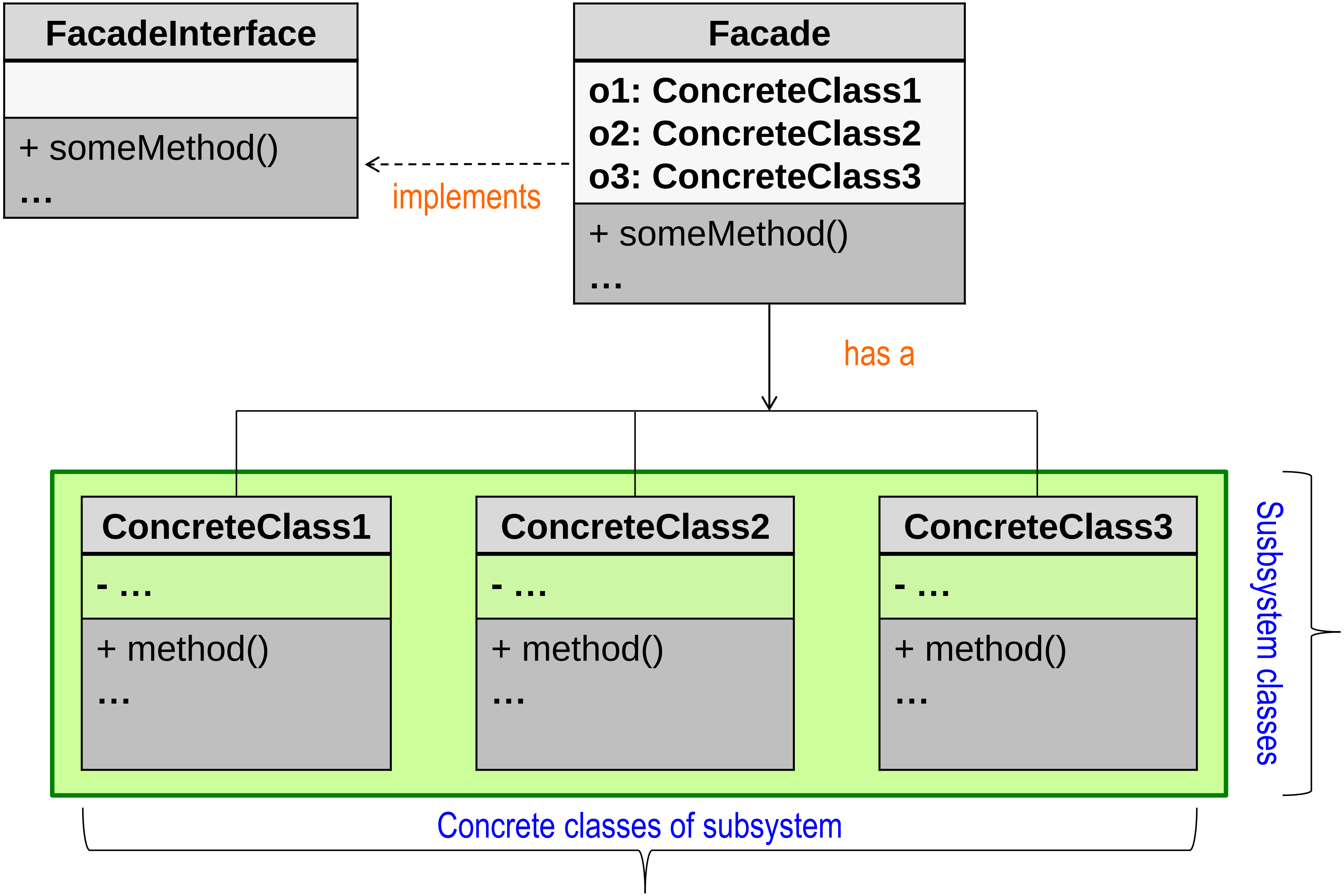
- …

+ method()
…

Susbsystem classes

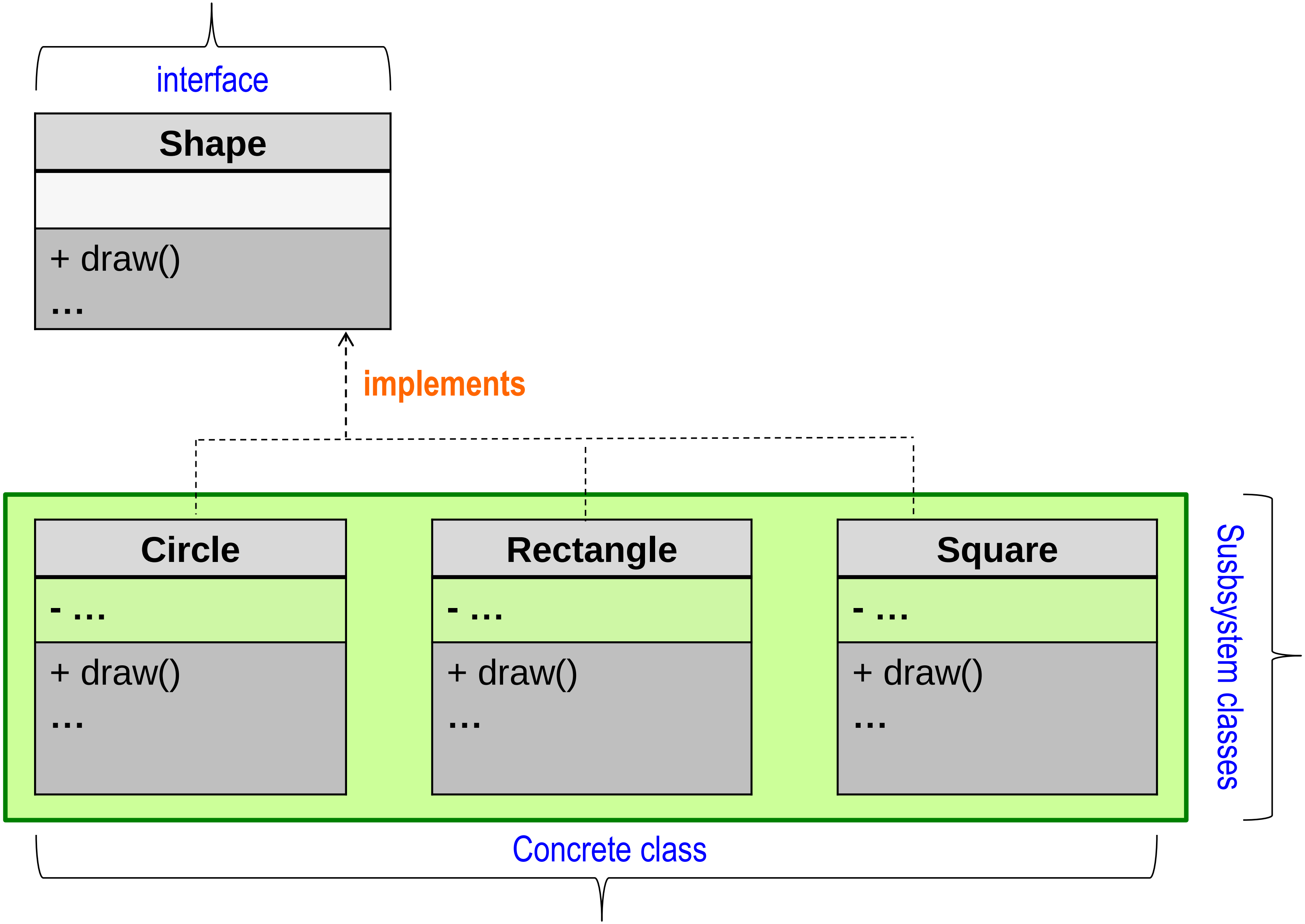
Concrete classes of subsystem

Facade Pattern



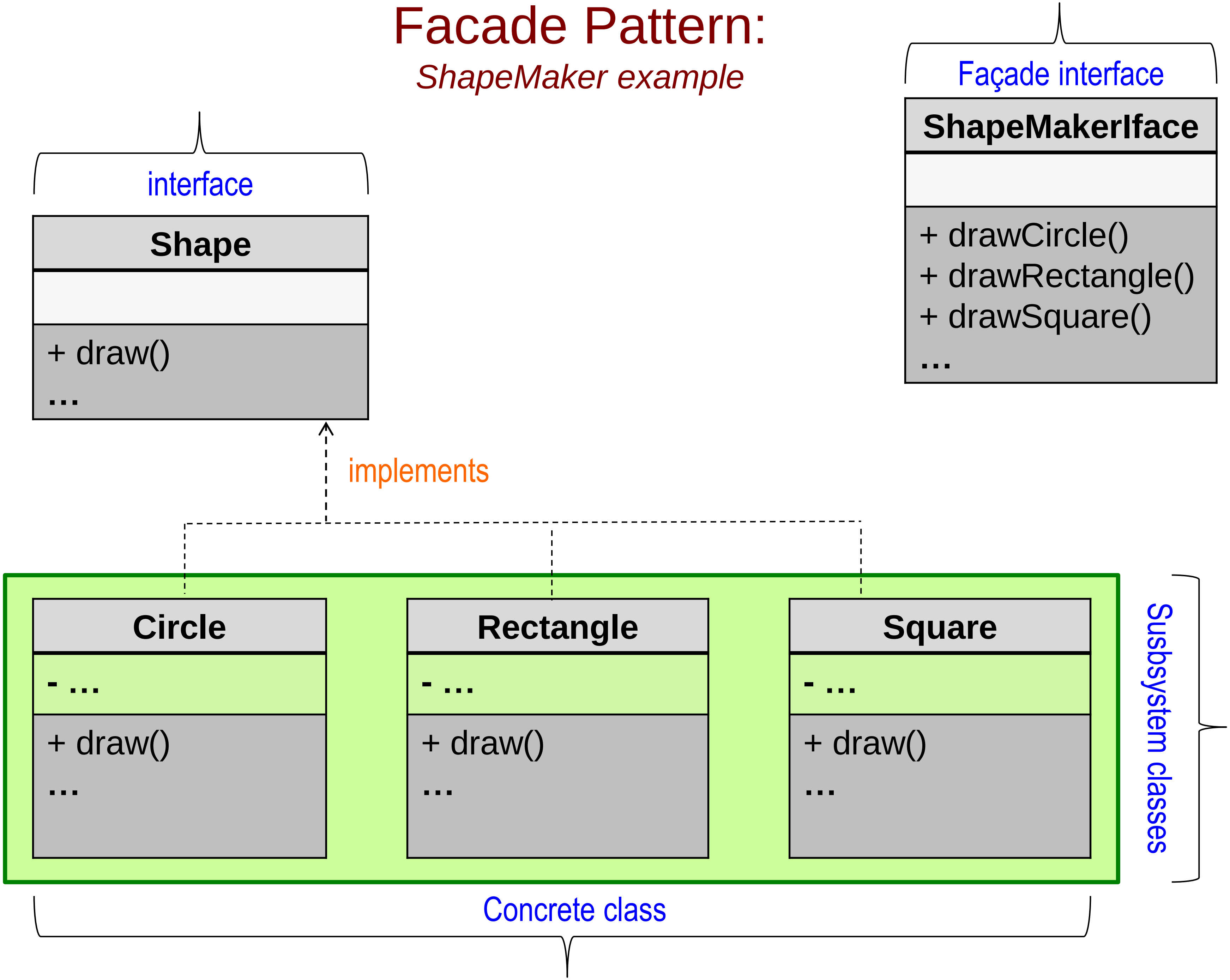
Facade Pattern:

ShapeMaker example



Facade Pattern:

ShapeMaker example



Façade interface

ShapeMakerIface

+ drawCircle()
+ drawRectangle()
+ drawSquare()
…

interface

Shape

+ draw()
…

implements

Circle

- …

+ draw()
…

Rectangle

- …

+ draw()
…

Square

- …

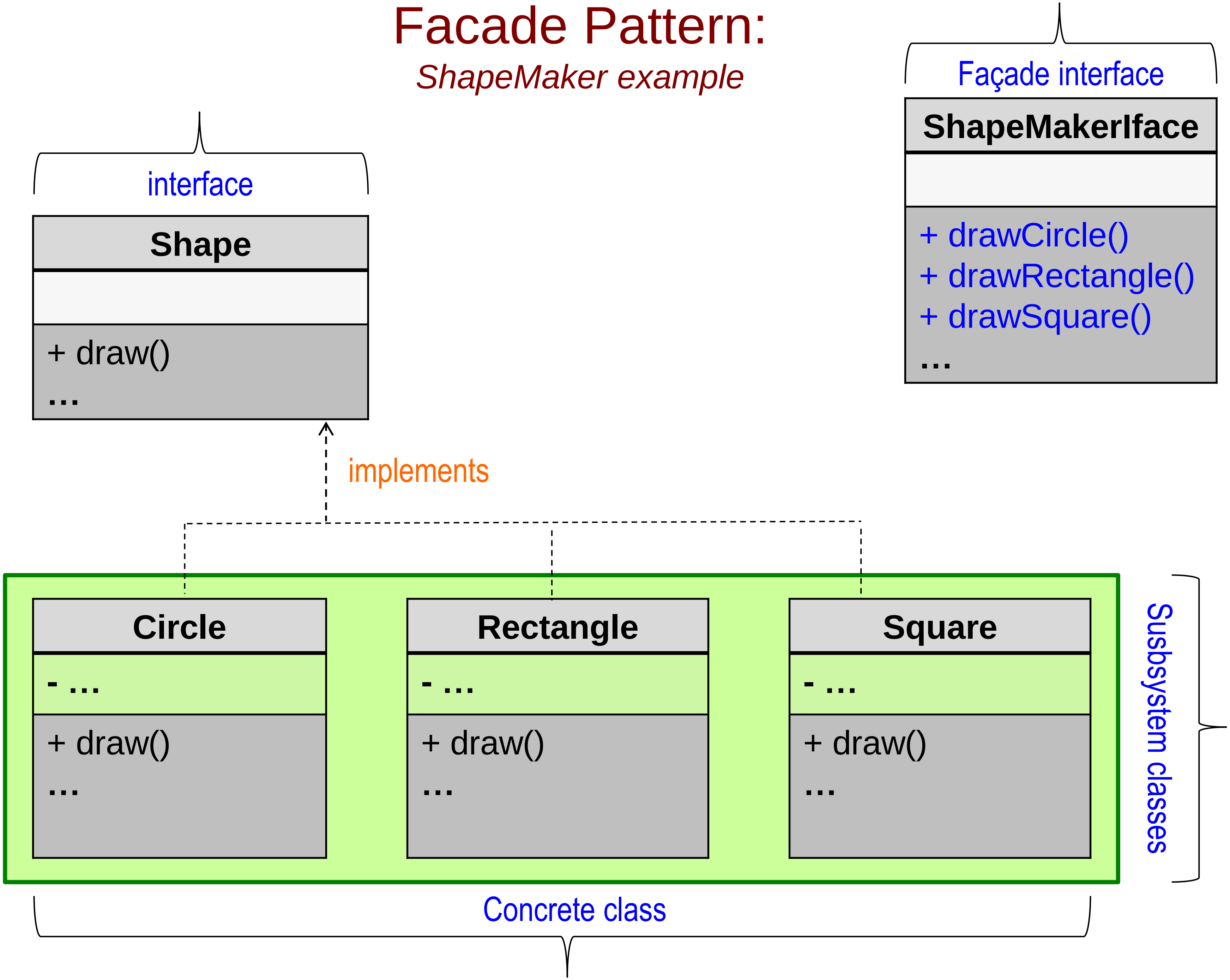
+ draw()
…

Susbsystem classes

Concrete class

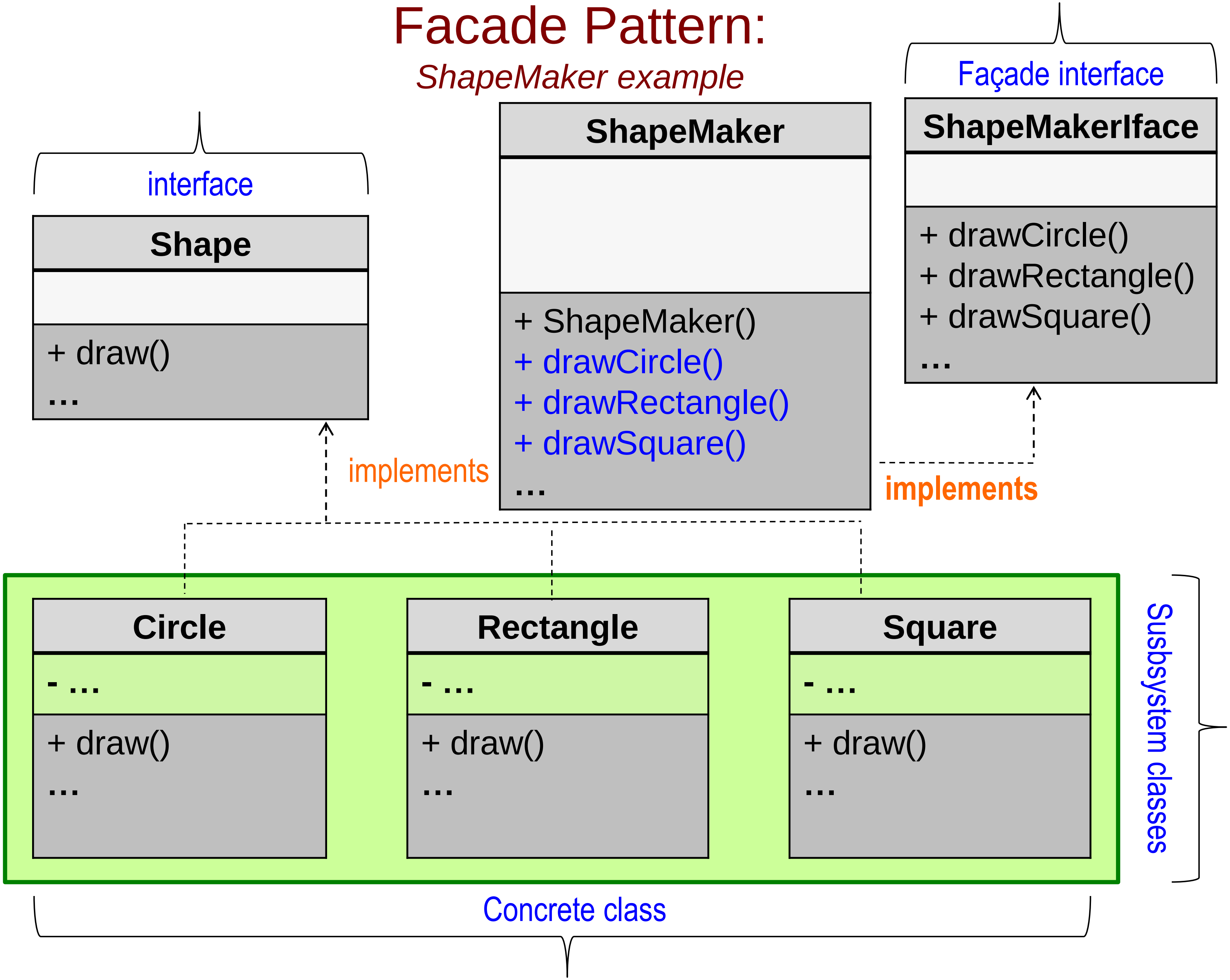
Facade Pattern:

ShapeMaker example



Facade Pattern:

ShapeMaker example



interface

Shape

+ draw()
…

ShapeMaker

+ ShapeMaker()
+ drawCircle()
+ drawRectangle()
+ drawSquare()
…

Façade interface

ShapeMakerIface

+ drawCircle()
+ drawRectangle()
+ drawSquare()
…

implements

implements

Circle

- …

+ draw()
…

Rectangle

- …

+ draw()
…

Square

- …

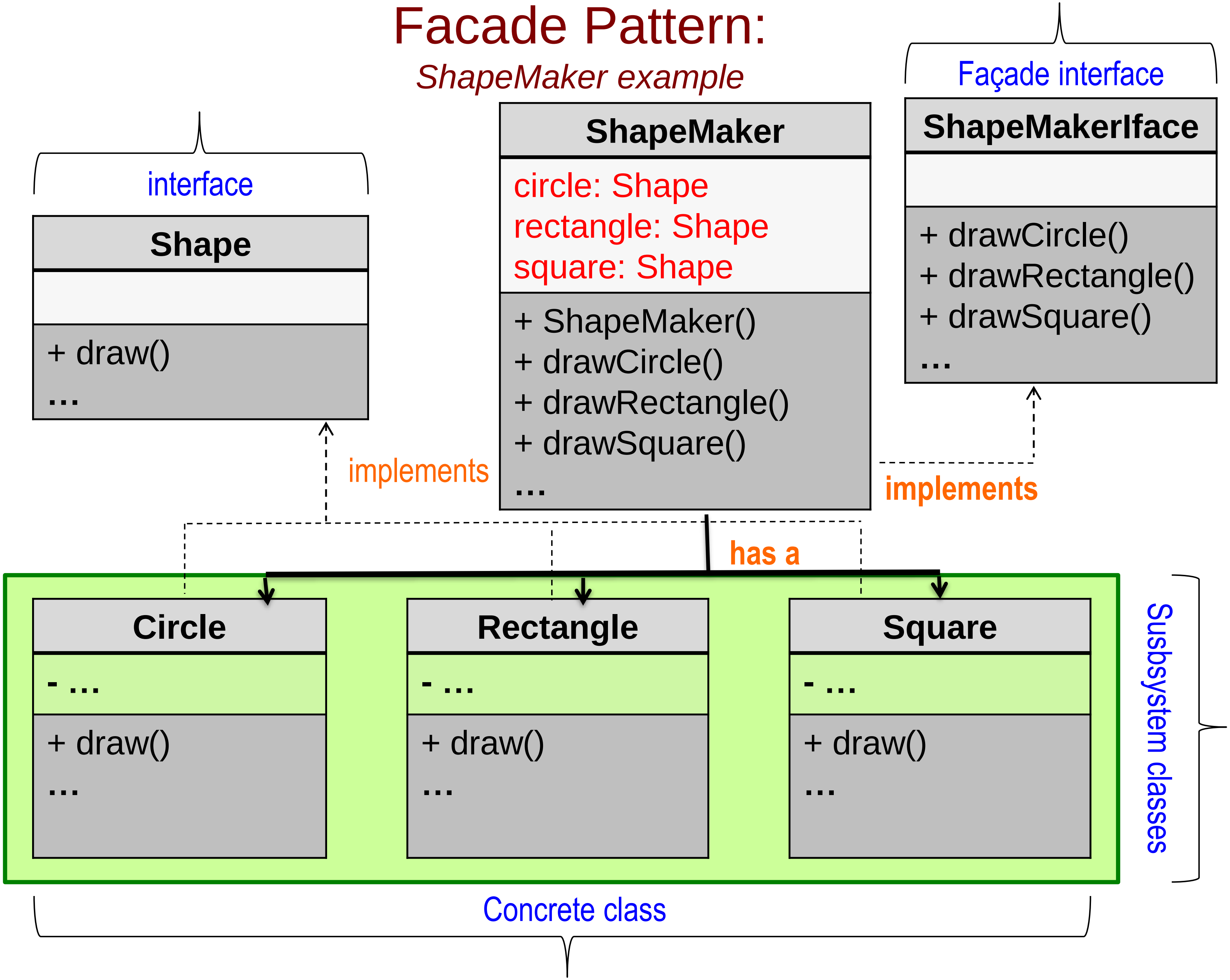
+ draw()
…

Susbsystem classes

Concrete class

Facade Pattern:

ShapeMaker example



interface

Shape

+ draw()
…

ShapeMaker

circle: Shape
rectangle: Shape
square: Shape

+ ShapeMaker()
+ drawCircle()
+ drawRectangle()
+ drawSquare()
…

Façade interface

ShapeMakerIface

+ drawCircle()
+ drawRectangle()
+ drawSquare()
…

implements

implements

has a

Circle

- …

+ draw()
…

Rectangle

- …

+ draw()
…

Square

- …

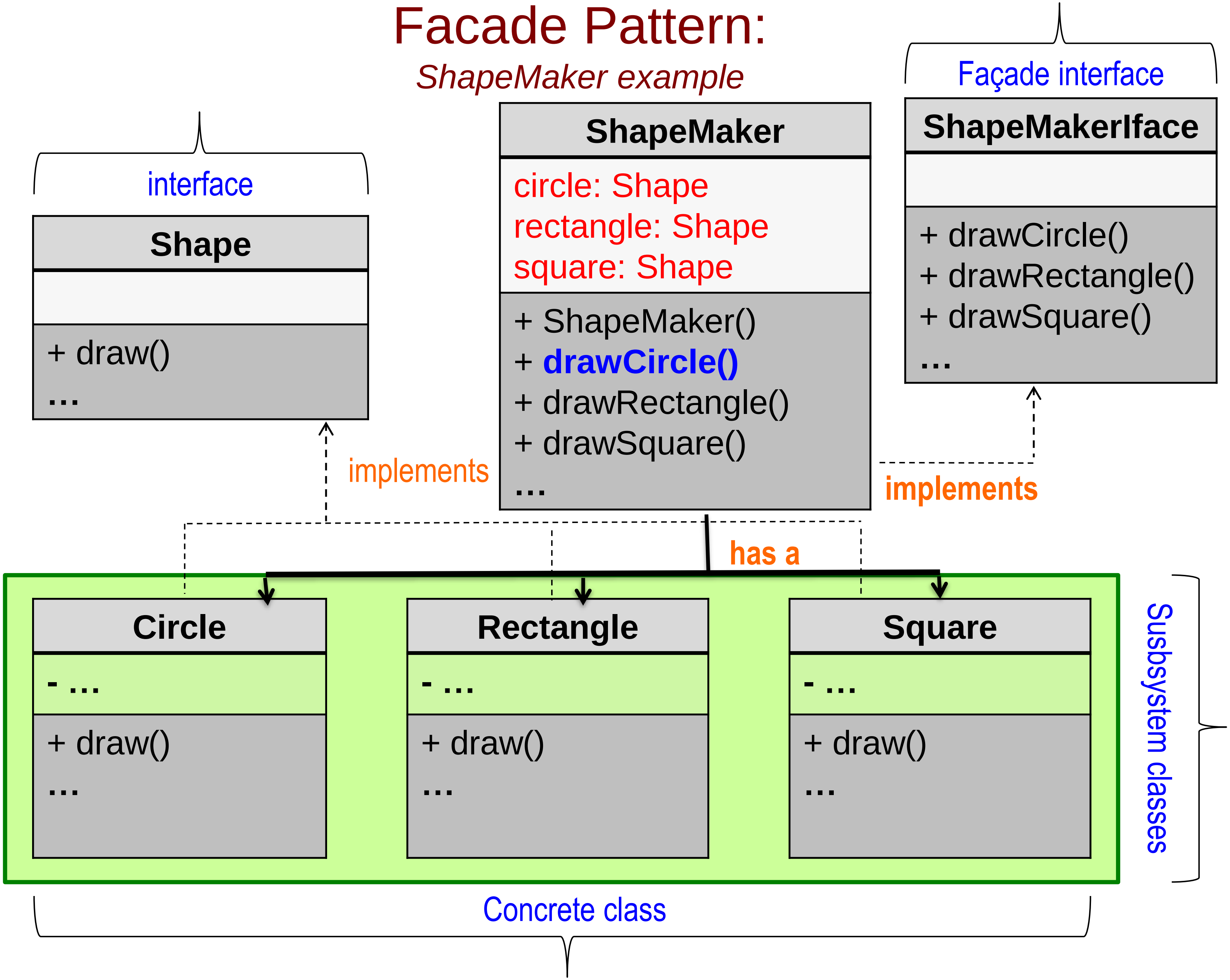
+ draw()
…

Susbsystem classes

Concrete class

Facade Pattern:

ShapeMaker example



interface

Shape

+ draw()

…

implements

ShapeMaker

circle: Shape
rectangle: Shape
square: Shape

+ ShapeMaker()
+ **drawCircle()**
+ drawRectangle()
+ drawSquare()
…

Façade interface

ShapeMakerIface

+ drawCircle()
+ drawRectangle()
+ drawSquare()
…

implements

has a

Circle

- …

+ draw()
…

Rectangle

- …

+ draw()
…

Square

- …

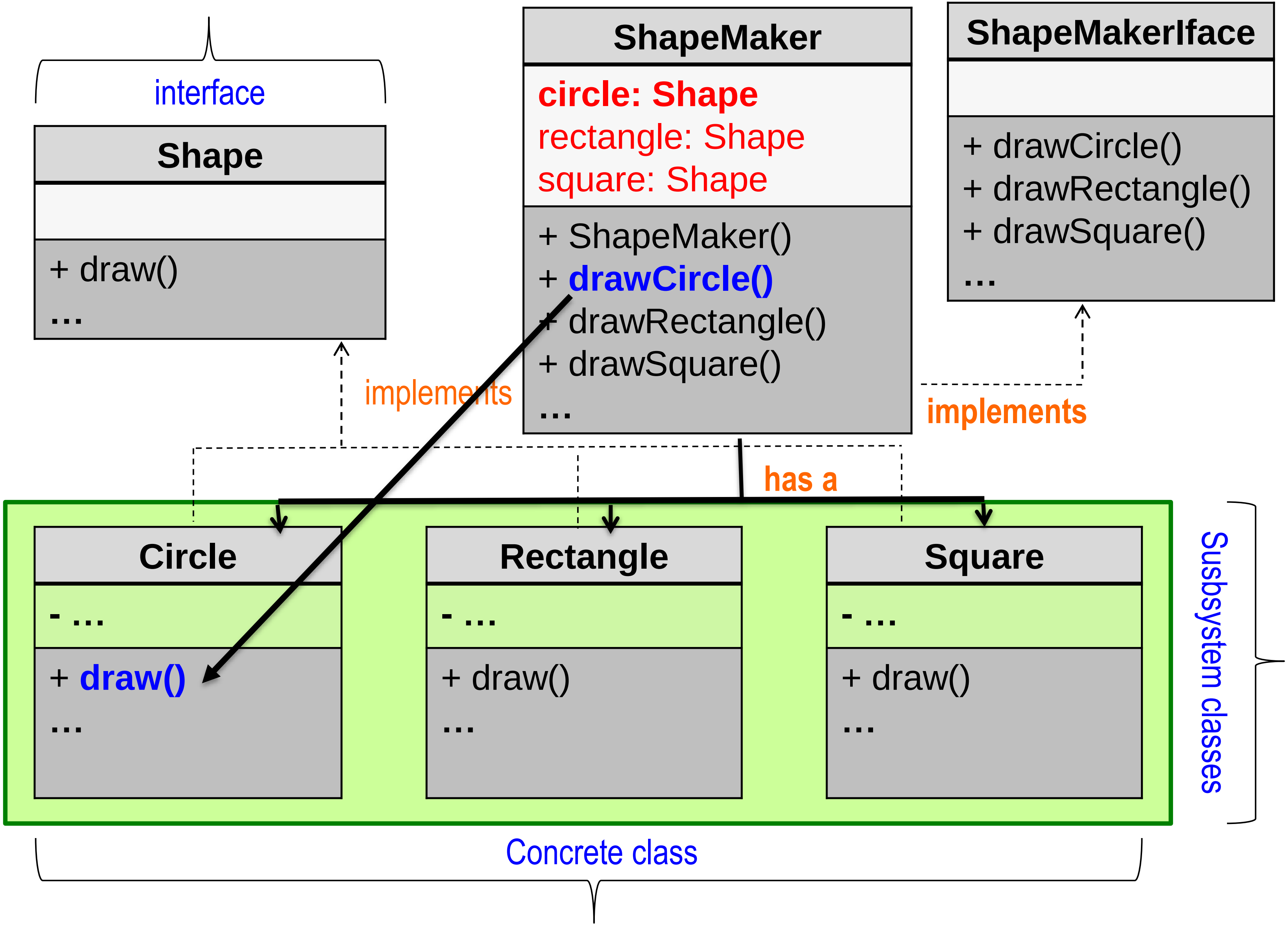
+ draw()
…

Susbsystem classes

Concrete class

Facade Pattern:

ShapeMaker example



interface

Shape

+ draw()
…

Façade interface

ShapeMakerIface

+ drawCircle()
+ drawRectangle()
+ drawSquare()
…

ShapeMaker

circle: Shape
rectangle: Shape
square: Shape

+ ShapeMaker()
+ **drawCircle()**
+ drawRectangle()
+ drawSquare()
…

implements

implements

has a

Circle

- …

+ **draw()**
…

Rectangle

- …

+ draw()
…

Square

- …

+ draw()
…

Susbsystem classes

Concrete class

Implementation

```
public class ShapeMaker implements ShapeMakerIface {
   private Shape circle;
   private Shape rectangle;
   private Shape square;

   public ShapeMaker() {
      circle = new Circle();
      rectangle = new Rectangle();
      square = new Square();
   }
   public void drawCircle() {
      circle.draw();
   }
   public void drawRectangle(){
      rectangle.draw();
   }
   public void drawSquare(){
      square.draw();
   }
} // class
```

Implementation

```
public class ShapeMaker implements ShapeMakerIface {
   private Shape circle;
   private Shape rectangle;
   private Shape square;

   public ShapeMaker() {
      circle = new Circle();
      rectangle = new Rectangle();
      square = new Square();
   }
   public void drawCircle() {
      circle.draw();
   }
   public void drawRectangle(){
      rectangle.draw();
   }
   public void drawSquare(){
      square.draw();
   }
} // class
```

Implementation

```
public class ShapeMaker implements ShapeMakerIface {
   private Shape circle;
   private Shape rectangle;
   private Shape square;

   public ShapeMaker() {
      circle = new Circle();
      rectangle = new Rectangle();
      square = new Square();
   }
   public void drawCircle() {
      circle.draw();
   }
   public void drawRectangle(){
      rectangle.draw();
   }
   public void drawSquare(){
      square.draw();
   }
} // class
```

Implementation

```
public class ShapeMakerTest {

   public static void main( ... ) {
      ShapeMakerIface shapemaker = new ShapeMaker();

      shapeMaker.drawCircle();
      shapeMaker.drawRectangle();
      shapeMaker.drawSquare();

   } // main
} // class
```

Implementation

```
public class ShapeMakerTest {

   public static void main( ... ) {
      ShapeMakerIface shapemaker = new ShapeMaker();

      shapeMaker.drawCircle();
      shapeMaker.drawRectangle();
      shapeMaker.drawSquare();

   } // main
} // class
```

Implementation

```
public class ShapeMakerTest {

   public static void main( ... ) {
     ShapeMakerIface shapemaker = new ShapeMaker();

      shapeMaker.drawCircle();
      shapeMaker.drawRectangle();
      shapeMaker.drawSquare();

   } // main
} // class
```

Facade Pattern:

Elements of Reusable OO Software

- Consequences (**Advantages**/Disadvantages):
  - Shields clients from subsystem components, thereby reducing the numb
    making th
  - Promotes ts.
  - It does n

Decouples the application from your system classes.

Facade Pattern:

Elements of Reusable OO Software

- Consequences (Advantages/Disadvantages):
  - Shields clients from subsystem components, thereby reducing the numb
    making th
  - Promotes ts.
  - It does no

Decouples the application from your system classes.

It does not stop applications from using the system classes directly.

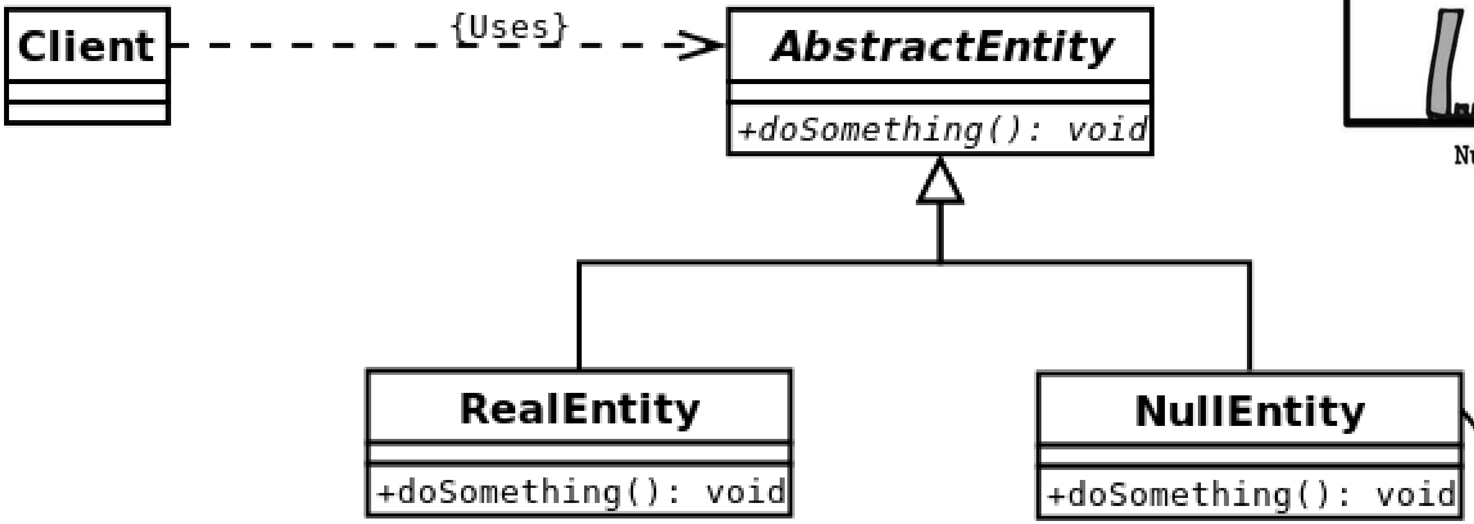
Discussion of Structural Patterns

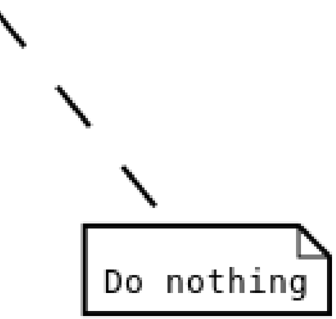
- There are overlapping similarities between many of the structural patterns because *they rely on the same set of*
  language
  namely: s
  patterns a

- *Need to f*
  appropria
  Decorato
  provides
  pattern is
  responsibilities dynamically. Its intent is to provide an indirect way to access an object when it is inconvenient or undesirable to access an object directly.

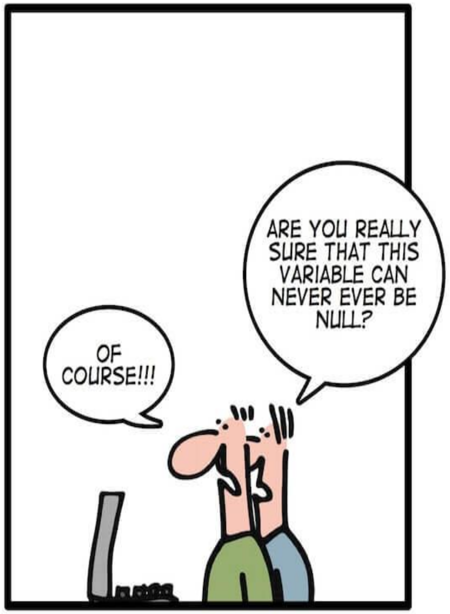
Always focus on the intent of the pattern as there are similarities across multiple pattern. But, what is the objective? That should distinguish between which pattern best applies.

Null Object Pattern





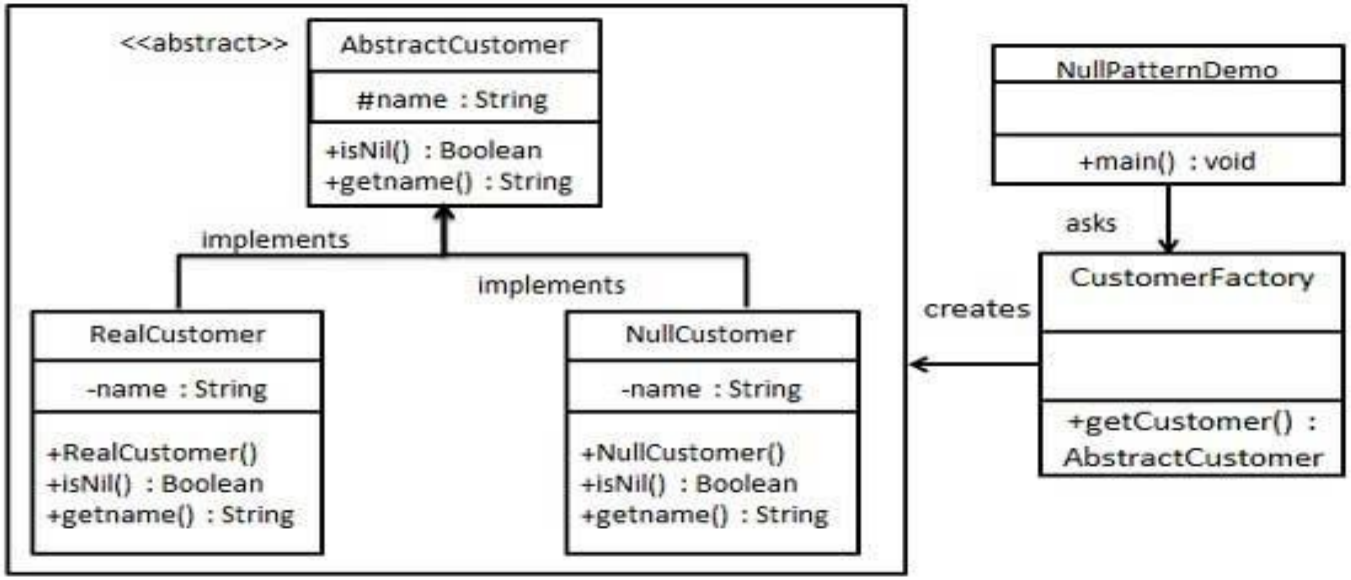
SIMPLY EXPLAINED


ARE YOU REALLY SURE THAT THIS VARIABLE CAN NEVER EVER BE NULL?
OF COURSE!!!

NullPointerException

Null Object Pattern

Intent: To simplify the use of dependencies that can be undefined. This is achieved by using instances of a concrete class that implements a known interface, instead of **null** references.



TutorialsPoint

Null Object Pattern

- **Motivation** and Applicability: Remove conditional checks and coding branches when dealing with the possibility of ***null*** references.
    - When you
      Polymorp

How to deal with null objects at run-time?

Null Object Pattern

- **Motivation** and Applicability: Remove conditional checks and coding branches when dealing with the possibility of ***null*** references.
    - When you
      Polymorp

How to deal with null objects at run-time?

null is an invention of British computer scientist Tony Hoare. He was knot to have later called his invention of null references as his **“billion dollar mistake”**.

Null Object Pattern

- Motivation **and Applicability**: Remove conditional checks and coding branches when dealing with the possibility of ***null*** references.
  - When you
    Polymorp

Replacing conditional logic and avoiding exception handling through…

Null Object Pattern

- Motivation **and Applicability**: Remove conditional checks and coding branches when dealing with the possibility of ***null*** references.
  - When you
    Polymorp

Replacing conditional logic and avoiding exception handling through…

Polymorphism.

Null Object Pattern

- Motivation **and Applicability**: Remove conditional checks and coding branches when dealing with the possibility of *null* references.
  - When you want to replace conditional checks with Polymorphism.

```
public class StudentClassDemo {
   public static void main(String[] args) {
      Student student1 MyStudents.getStudent("U33838");
      Student student2 MyStudents.getStudent("U48744");
      Student student3 MyStudents.getStudent("X48790");
      Student student1 MyStudents.getStudent("X68944");

      System.out.println(student1.getGPA());
      System.out.println(student2.getGPA());
      System.out.println(student3.getGPA());
      System.out.println(student4.getGPA());
   }
} // class
```

Null Object Pattern

- Motivation and Applicability: Remove conditional checks and coding branches when dealing with the possibility of *null* references.
  - When you want to replace conditional checks with Polymorphism.

```
public class StudentClassDemo {
   public static void main(String[] args) {
      Student student1 MyStudents.getStudent("U33838");
      Student student2 MyStudents.getStudent("U48744");
      Student student3 MyStudents.getStudent("X48790");
      Student student1 NyStudents.getStudent("X68944");

      System.out.println(student1.getGPA());
      System.out.println(student2.getGPA());
      System.out.println(student3.getGPA());
      System.out.println(student4.getGPA());
   }
} // class
```

Null Object Pattern

- Motivation and Applicability: Remove conditional checks and coding branches when dealing with the possibility of *null* references.
  - When you want to replace conditional checks with Polymorphism.

```
public class StudentClassDemo {
   public static void main(String[] args) {
      Student student1 MyStudents.getStudent("U33838");
      Student student2 MyStudents.getStudent("U48744");
      Student student3 MyStudents.getStudent("X48790");
      Student student1 MyStudents.getStudent("X68944");

      System.out.println(student1.getGPA());
      System.out.println(student2.getGPA());
      System.out.println(student3.getGPA());
      System.out.println(student4.getGPA());
   }
} // class
```

Null Object Pattern

- Motivation and Applicability: Remove conditional checks and coding branches when dealing with the possibility of *null* references.
  - When you want to replace conditional checks with Polymorphism.

```
public class StudentClassDemo {
   public static void main(String[] args) {
      Student student1 MyStudents.getStudent("U33838");
      Student student2 MyStudents.getStudent("U48744");
      Student student3 MyStudents.getStudent("X48790");
      Student student1 MyStudents.getStudent("X68944");
      if (student1 != null)
          System.out.println(student1.getGPA());
      System.out.println(student2.getGPA());
      System.out.println(student3.getGPA());
      System.out.println(student4.getGPA());
   }
} // class
```

Null Object Pattern

- Motivation and Applicability: Rem c ks and coding branches when de s.
  - When you
    Polymorphi

Can also use exception handling, but this is still just a different conditional block.

```
public class Studen
   public static void main(String[] args) {
      Student student1 MyStudents.getStudent("U33838");
      Student student2 MyStudents.getStudent("U48744");
      Student student3 MyStudents.getStudent("X48790");
      Student student1 MyStudents.getStudent("X68944");
      if (student1 != null)
          System.out.println(student1.getGPA());
      System.out.println(student2.getGPA());
      System.out.println(student3.getGPA());
      System.out.println(student4.getGPA());
   }
} // class
```

Null Object Pattern

- Motivation and Applicability: Remove conditional checks and coding branches when dealing with the possibility of *null* references.
  - When you want to replace conditional checks with Polymorphism.

```
public class StudentClassDemo {
   public static void main(String[] args) {
      Student student1 MyStudents.getStudent("U33838");
      Student student2 MyStudents.getStudent("U48744");
      Student student3 MyStudents.getStudent("X48790");
      Student student1 MyStudents.getStudent("X68944");
      if (student1 != null)
          System.out.println(student1.getGPA());
      System.out.println(student2.getGPA());
      System.out.println(student3.getGPA());
      System.out.println(student4.getGPA());
   }
} // class
```

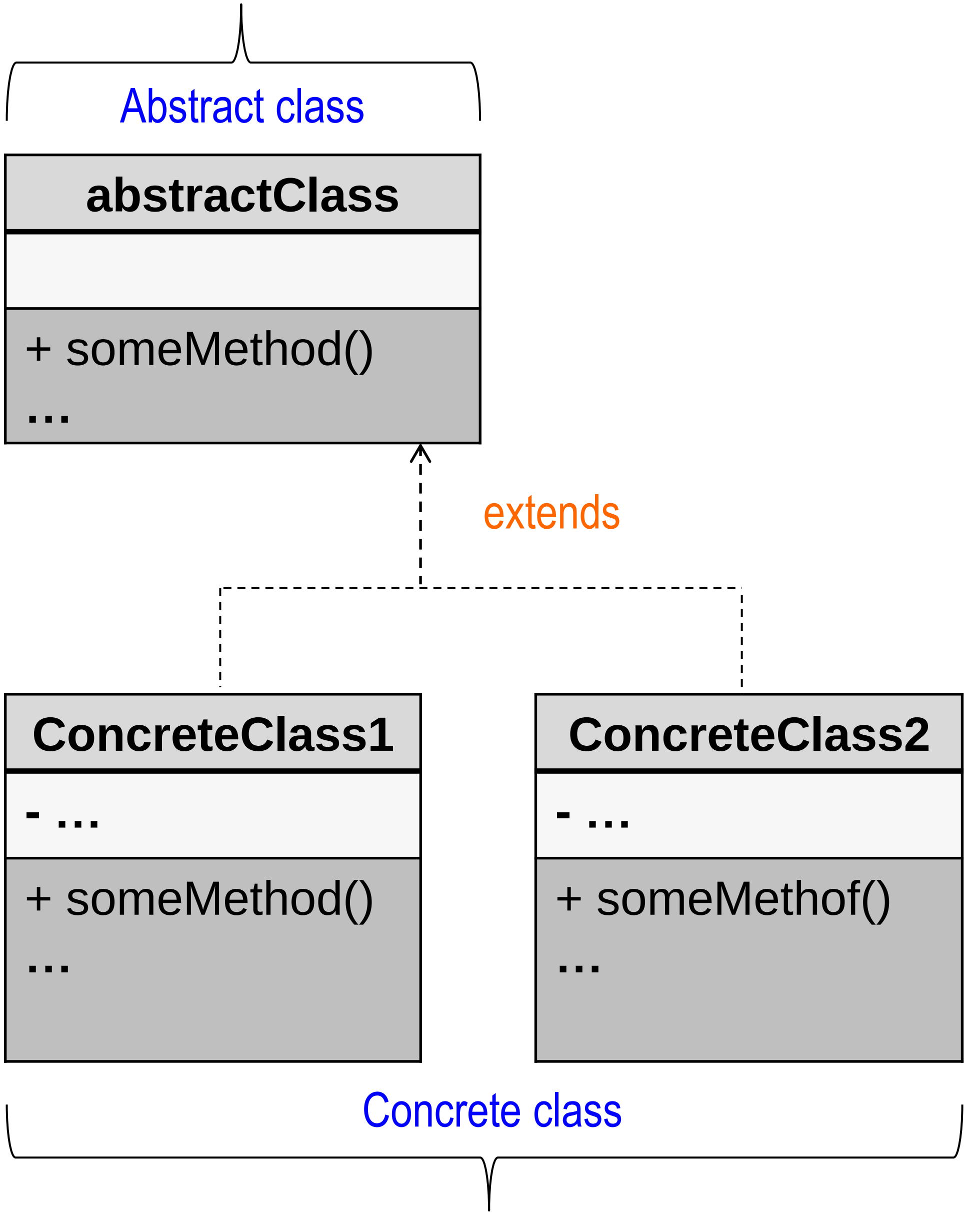
Null Object Pattern

- Motivation and … coding branches
  - When yo… Polymorphism

The only way to avoid conditional checks, including exception handling, `getStudent()` cannot return `null`!

```
public class StudentClassDemo {
   public static void main(String[] args) {
      Student student1 MyStudents.getStudent("U33838");
      Student student2 MyStudents.getStudent("U48744");
      Student student3 MyStudents.getStudent("X48790");
      Student student1 MyStudents.getStudent("X68944");
      if (student1 != null)
          System.out.println(student1.getGPA());
      System.out.println(student2.getGPA());
      System.out.println(student3.getGPA());
      System.out.println(student4.getGPA());
   }
} // class
```

Null Object Pattern



Abstract class

abstractClass

+ someMethod()
…

extends

ConcreteClass1

- …

+ someMethod()
…

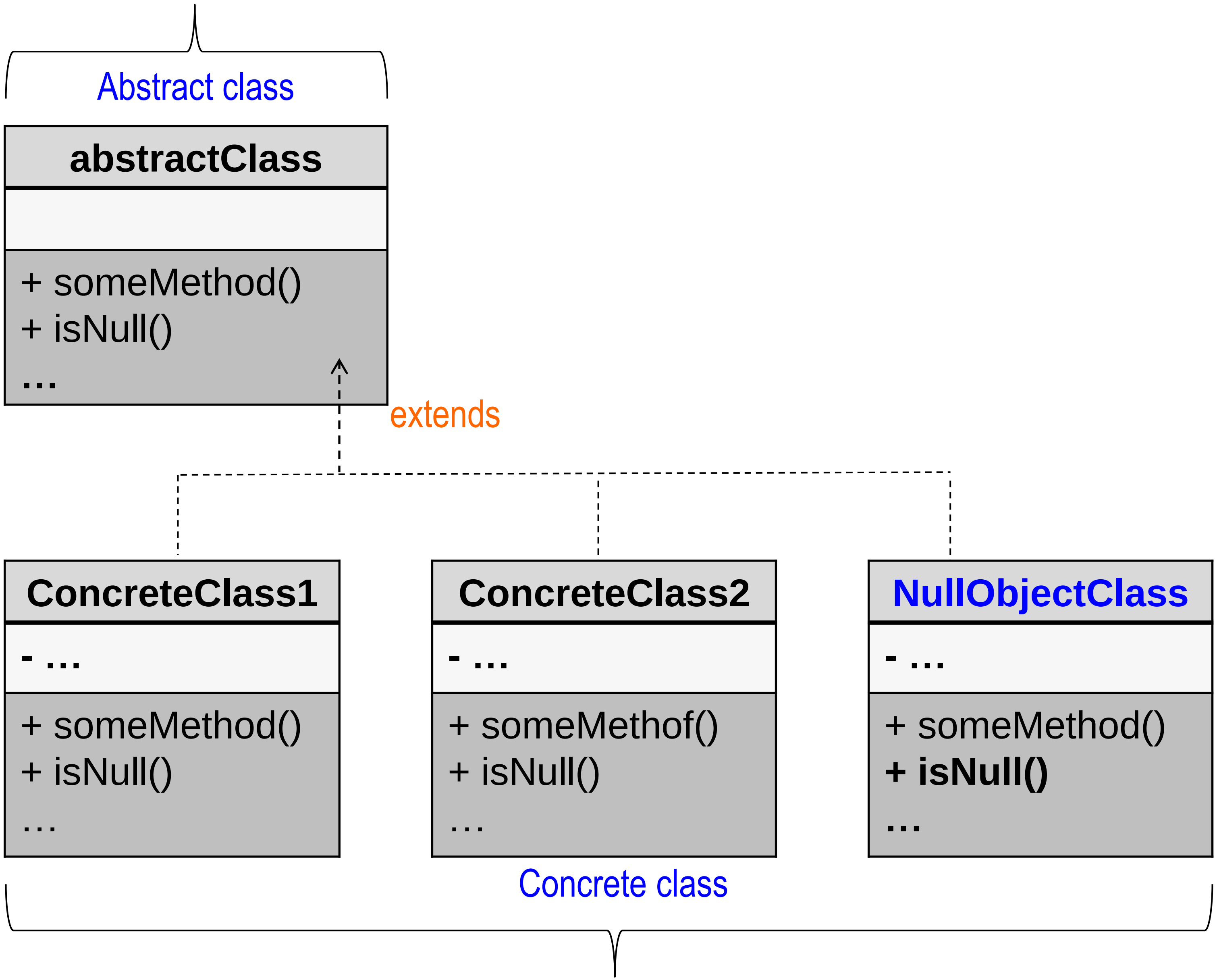
ConcreteClass2

- …

+ someMethof()
…

Concrete class

Null Object Pattern



Abstract class

abstractClass

+ someMethod()
+ isNull()
…

extends

ConcreteClass1

- …

+ someMethod()
+ isNull()
…

ConcreteClass2

- …

+ someMethof()
+ isNull()
…

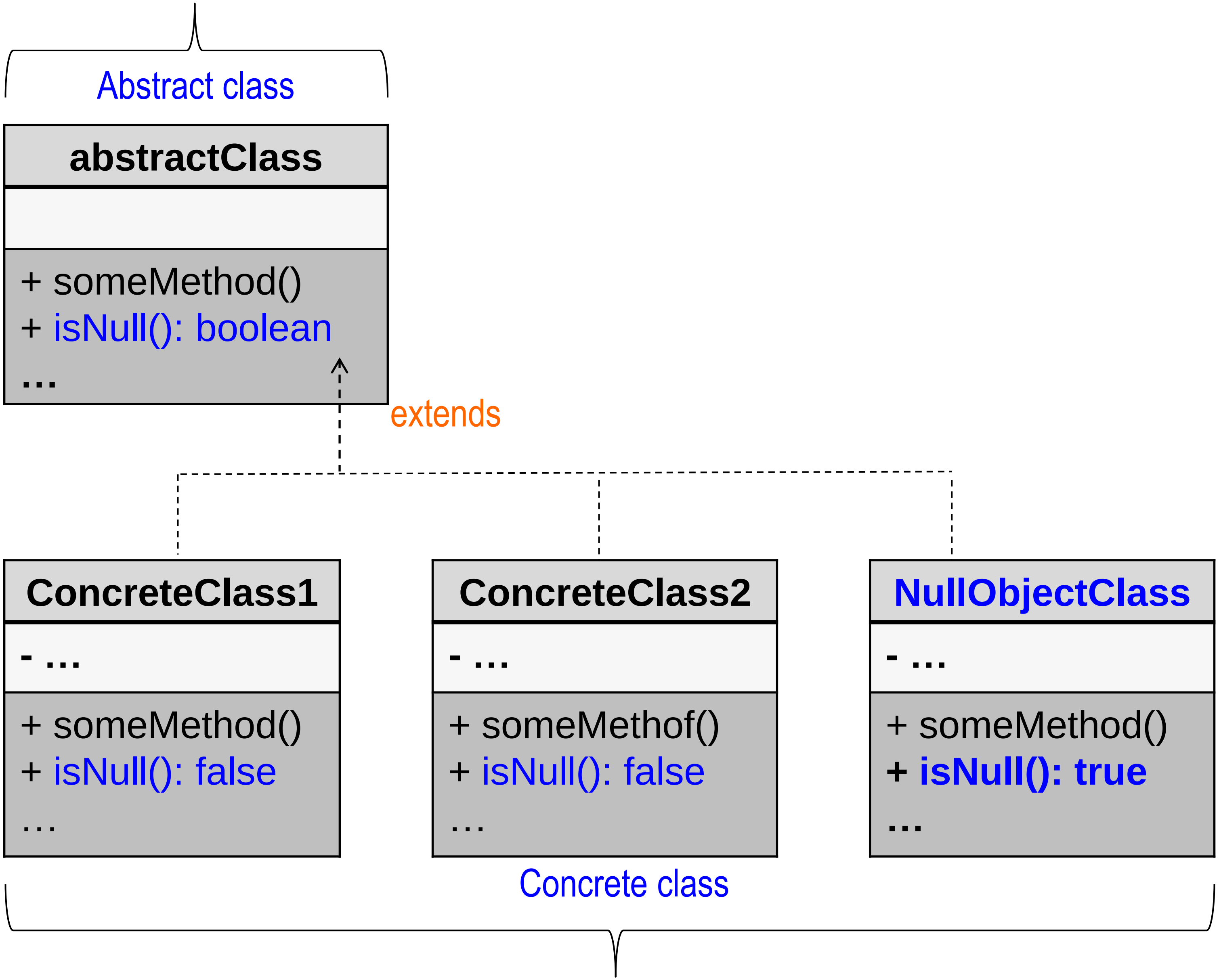
NullObjectClass

- …

+ someMethod()
+ isNull()
…

Concrete class

Null Object Pattern



Abstract class

abstractClass

+ someMethod()
+ isNull(): boolean
…

extends

ConcreteClass1

- …

+ someMethod()
+ isNull(): false
…

ConcreteClass2

- …

+ someMethof()
+ isNull(): false
…

NullObjectClass

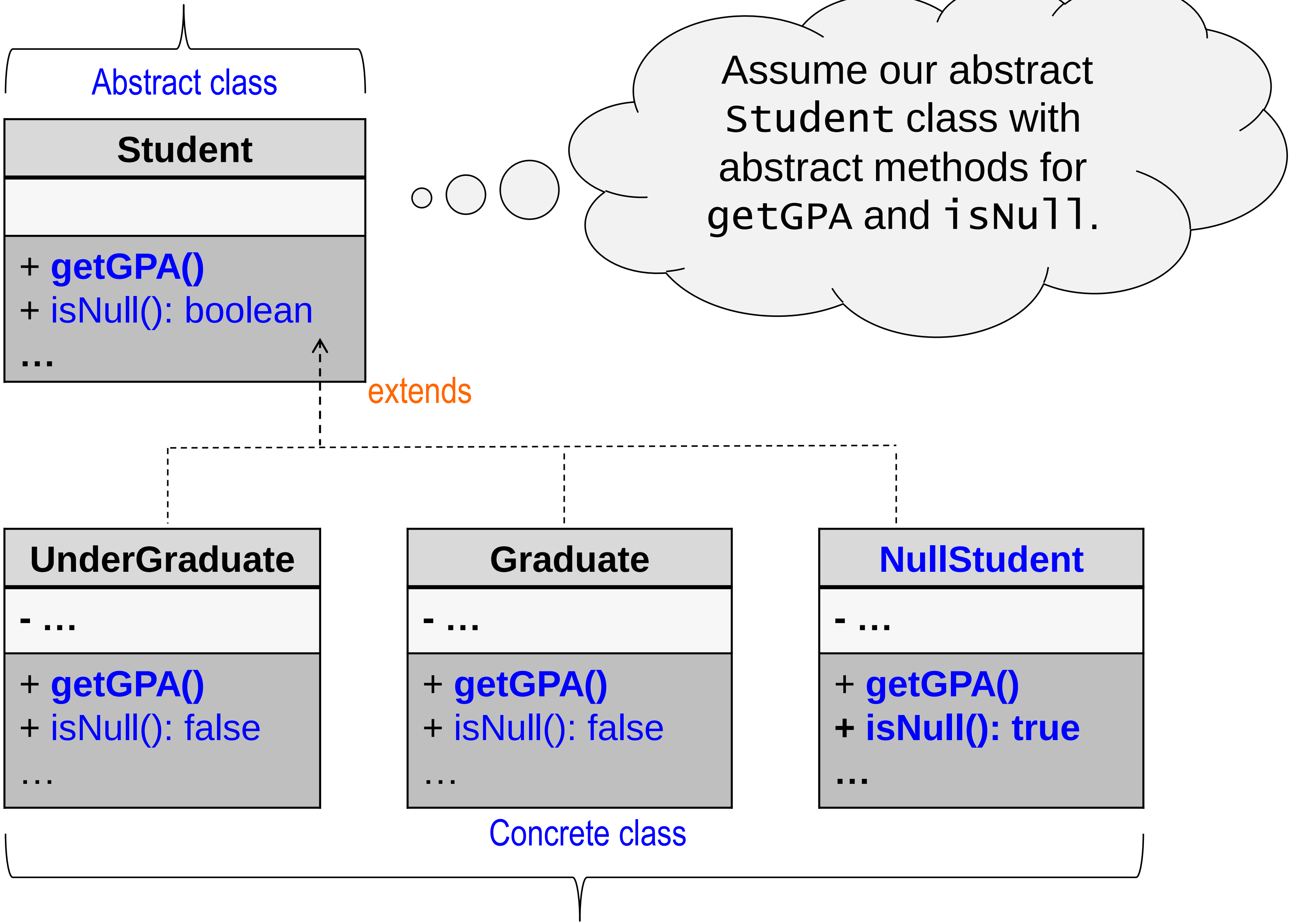
- …

+ someMethod()
+ isNull(): true
…

Concrete class

Null Object Pattern

Student example



Abstract class

Student

+ **getGPA()**
+ isNull(): boolean
…

extends

UnderGraduate

- …

+ **getGPA()**
+ isNull(): false
…

Graduate

- …

+ **getGPA()**
+ isNull(): false
…

NullStudent

- …

+ **getGPA()**
+ isNull(): true
…

Concrete class

Assume our abstract `Student` class with abstract methods for `getGPA` and `isNull`.

Implementation

```
public class NullStudent extends Student {

   public String getGPA() {
      return "Student not found“;
   }

   public boolean isNull() {
      return(true);
   }

} // class
```

Implementation

```
public class NullStudent extends Student {

   public String getGPA() {
      return "Student not found“;
   }

   public boolean isNull() {
      return(true);
   }

} // class
```

Implementation

```
public class NullStudent extends Student {

   public String getGPA() {
      return "Student not found“;
   }

   public boolean isNull() {
      return(true);
   }

} // class
```

Implementation

```
public class MyStudents {
   private static final Student[] students =
      { new Student("U12345")
      , new Student("U78915")
      , new Student("X98716") ... };

   public static Student getStudent( String uid ) {
      Student student = new NullStudent();

      for (int i = 0; i < students.length; i++ ) {
         if ( names[i].equalsIgnoreCase(uid) ) {
            student = students[i];
            break;
         }
      }

      return( student );
   }

} // class
```

Implementation

```
public class MyStudents {
    private static final Student[] students =
        { new Student("U12345")
        , new Student("U78915")
        , new Student("X98716") ... };

    public static Student getStudent( String uid ) {
        Student student = new NullStudent();

        for (int i = 0; i < students.length; i++ ) {
            if ( names[i].equalsIgnoreCase(uid) ) {
                student = students[i];
                break;
            }
        }

        return( student );
    }

} // class
```

Implementation

```
public class MyStudents {
   private static final Student[] students =
      { new Student("U12345")
      , new Student("U78915")
      , new Student("X98716") ... };

   public static Student getStudent( String uid ) {
      Student student = new NullStudent();

      for (int i = 0; i < students.length; i++ ) {
         if ( names[i].equalsIgnoreCase(uid) ) {
            student = students[i];
            break;
         }
      }

      return( student );
   }

} // class
```

Implementation

```
public class MyStudents {
   private static final Student[] students =
      { new Student("U12345")
      , new Student("U78915")
      , new Student("X98716") ... };

   public static Student getStudent( String uid ) {
      Student student = new NullStudent();

      for (int i = 0; i < students.length; i++ ) {
         if ( names[i].equalsIgnoreCase(uid) ) {
            student = students[i];
            break;
         }
      }

      return( student );
   }

} // class
```

Implementation

```
public class MyStudents {
   private static final Student[] students =
      { new Student("U12345")
      , new Student("U78915")
      , new Student("X98716") ... };

   public static Student getStudent( String uid ) {
      Student student = new NullStudent();

      for (int i = 0; i < students.length; i++ ) {
         if ( names[i].equalsIgnoreCase(uid) ) {
            student = students[i];
            break;
         }
      }

      return( student );
   }

} // class
```

Implementation

```
public class MyStudents {
   private static final Student[] students =
      { new Student("U12345")
      , new Student("U78915")
      , new Student("X98716") ... };

   public static Student getStudent( String uid ) {
      Student student = new NullStudent();

      for (int i = 0; i < students.length; i++ ) {
         if ( names[i].equalsIgnoreCase(uid) ) {
            student = students[i];
            break;
         }
      }

      return( student );
   }

} // class
```

Implementation

```
public class MyStudents {
   private static final Student[] students =
      { new Student("U12345")
      , new Student("U78915")
      , new Student("X98716") ... };

   public static Student getStudent( String uid ) {
      Student student = new NullStudent();

      for (int i = 0; i < students.length; i++ ) {
         if ( names[i].equalsIgnoreCase(uid) ) {
            student = students[i];
            break;
         }
      }

      return( student );
   }

} // class
```

Implementation

```
public class StudentClassDemo {
   public static void main(String[] args) {
      Student student1 MyStudents.getStudent("U33838");
      Student student2 MyStudents.getStudent("U48744");
      Student student3 MyStudents.getStudent("X48790");
      Student student1 MyStudents.getStudent("X68944");

      System.out.println(student1.getGPA());
      System.out.println(student2.getGPA());
      System.out.println(student3.getGPA());
      System.out.println(student4.getGPA());
   }
} // class
```

Implementation

```
public class StudentClassDemo {
   public static void main(String[] args) {
      Student student1 MyStudents.getStudent("U33838");
      Student student2 MyStudents.getStudent("U48744");
      Student student3 MyStudents.getStudent("X48790");
      Student student1 MyStudents.getStudent("X68944");

      System.out.println(student1.getGPA());
      System.out.println(student2.getGPA());
      System.out.println(student3.getGPA());
      System.out.println(student4.getGPA());
   }
} // class
```

The `getStudent` method will never return `null`! Will always return a reference to Student or a `nullStudent` object!

Null Object Pattern

- **Consequences (Advantages**/Disadvantages):
  - Null objects can be used in place of real objects when the object expect
  - Simplif l checks
  - Can b *vior* of the anywa
  - Can n ecessitate creating a new Null Object class for every new Abstract class or interface.

Simplifies the need for conditional checks …

Null Object Pattern

- Consequences (Advantages/Disadvantages):
  - Null objects can be used in place of real objects when the object expect
  - Simpli l checks
  - Can b *vior* of the anywa
  - Can necessitate creating a new Null Object class for every new Abstract class or interface.

Simplifies the need for conditional checks, but still need to check if you have a null object if processing your objects.