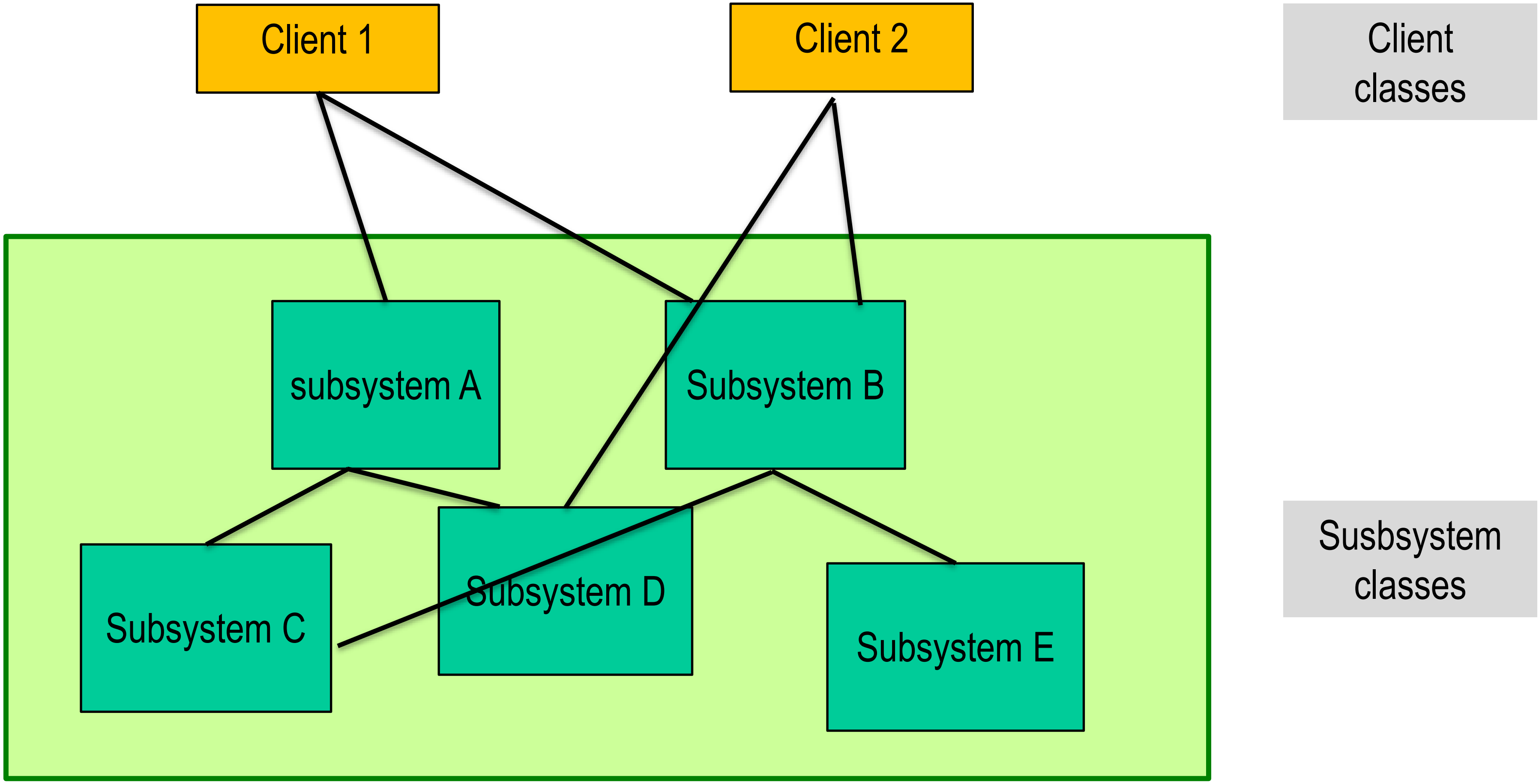
Facade Pattern

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

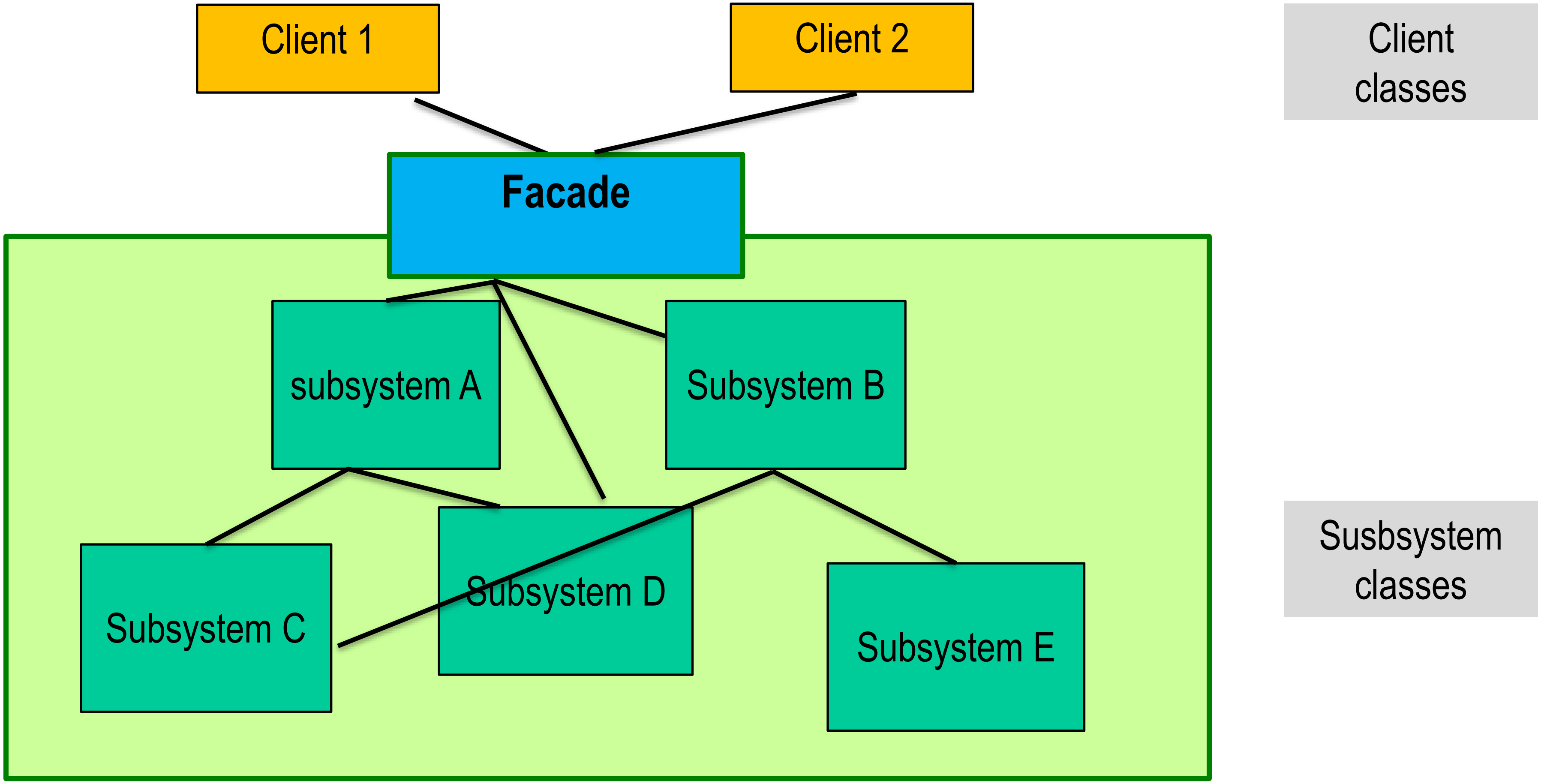
Subsystem D

Subsystem E

Susbsystem classes

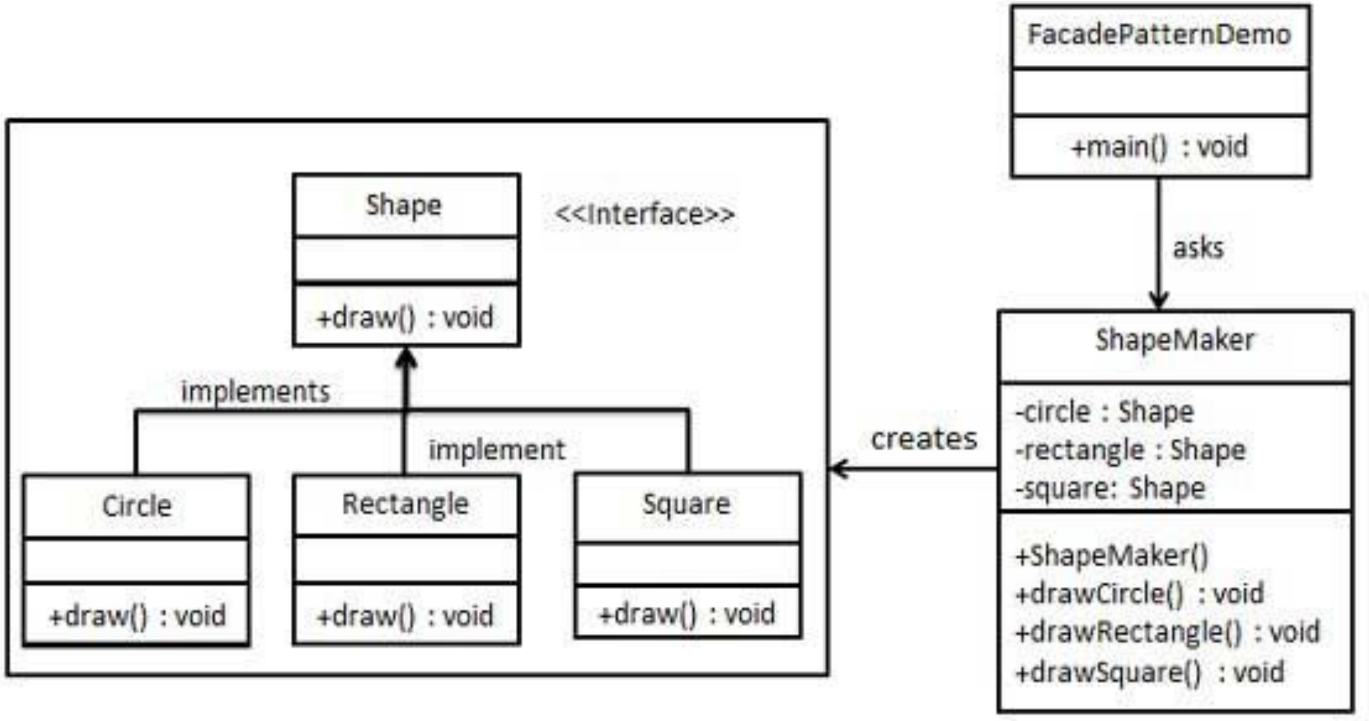
Facade Pattern

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.



Facade Pattern

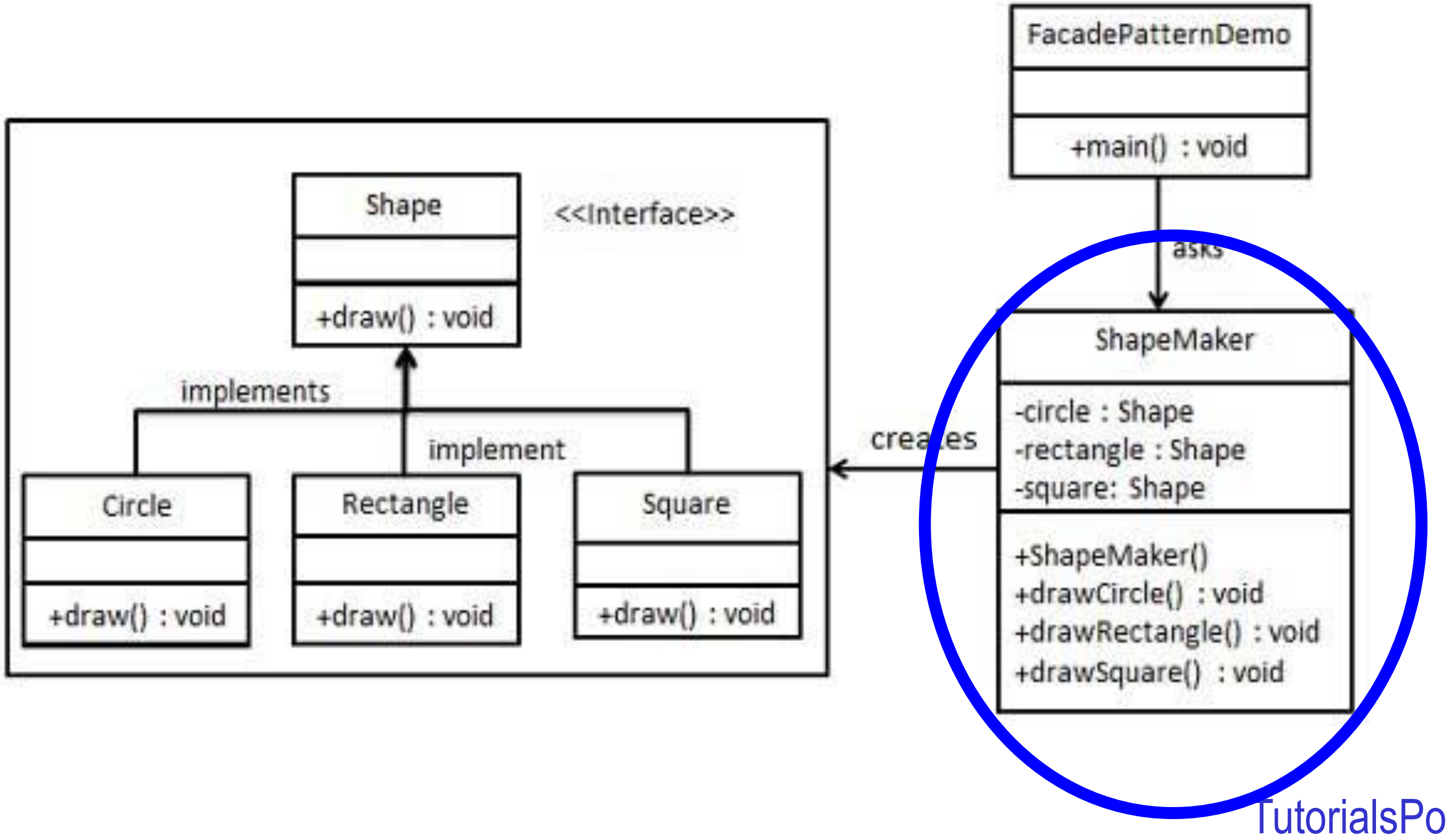
Intent*:* Provide a *unified interface* to a set of interfaces in a subsystem. Facade defines a *higher-level interface* that makes the subsystem easier to use.



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 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

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

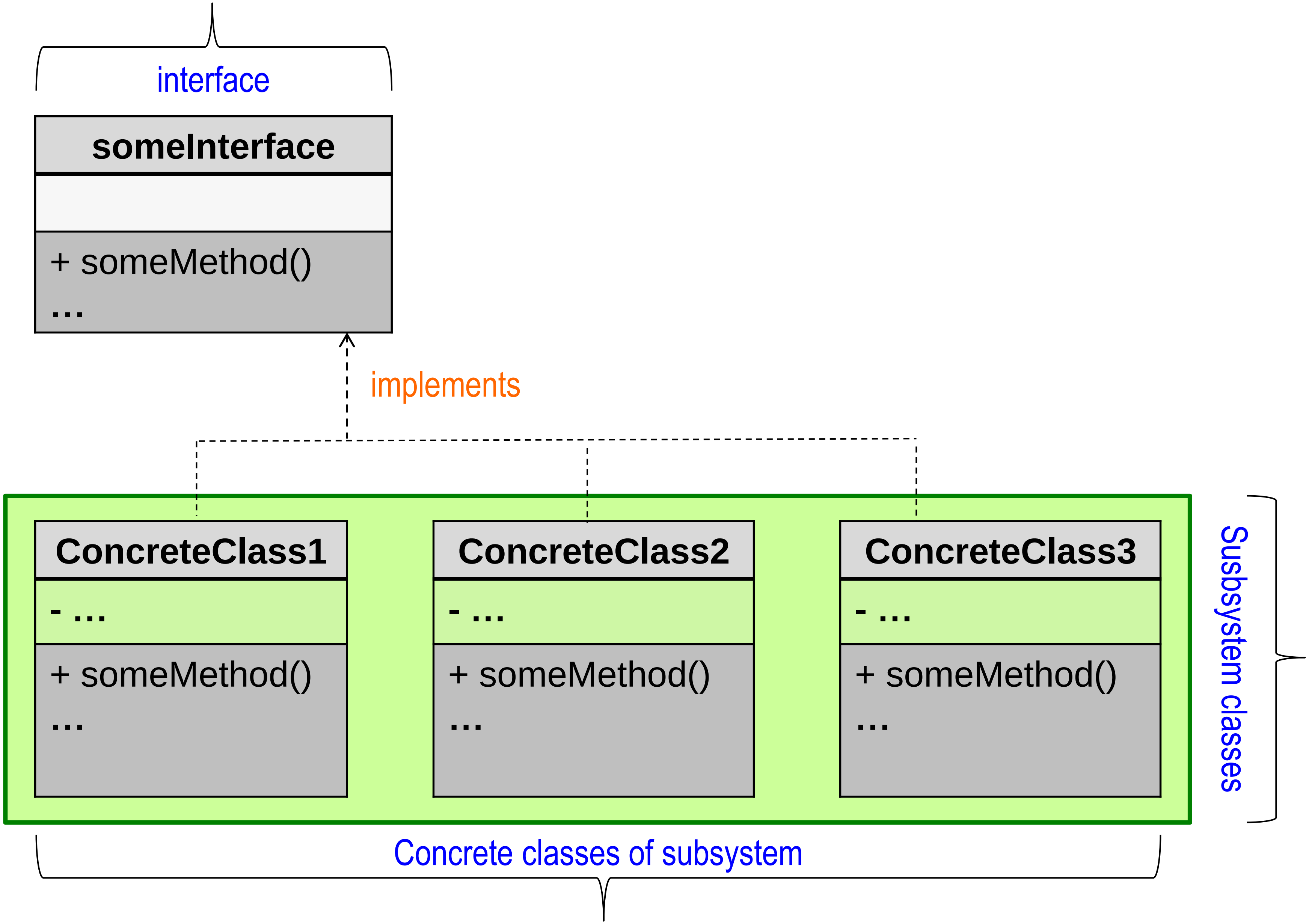
    know abo

  - 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()
…

implements

ConcreteClass1

- …

+ someMethod()
…

ConcreteClass2

- …

+ someMethod()
…

ConcreteClass3

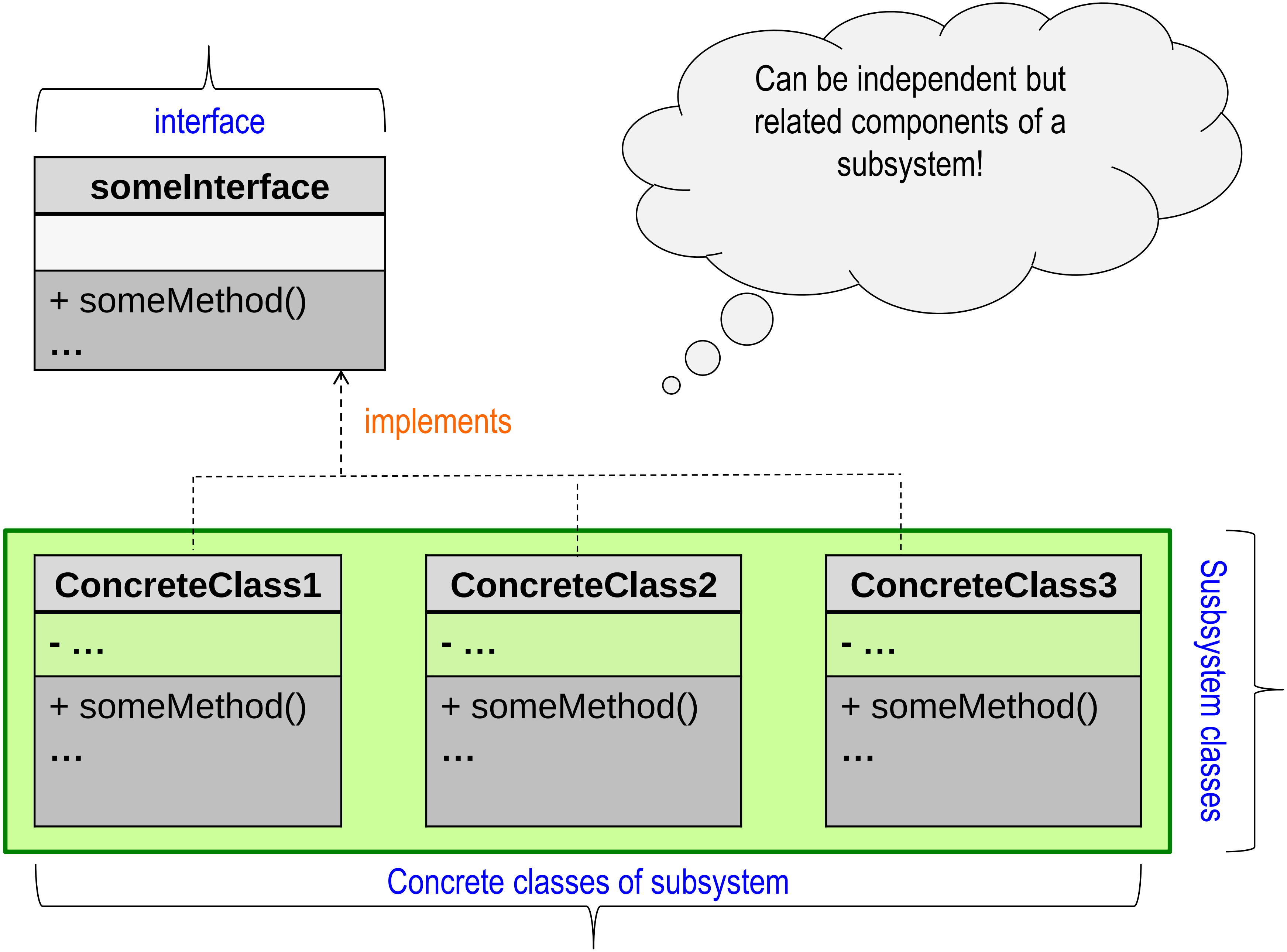
- …

+ someMethod()
…

Susbsystem classes

Concrete classes of subsystem

Facade Pattern



interface

someInterface

+ someMethod()

…

implements

Can be independent but related components of a subsystem!

ConcreteClass1

- …

+ someMethod()

…

ConcreteClass2

- …

+ someMethod()

…

ConcreteClass3

- …

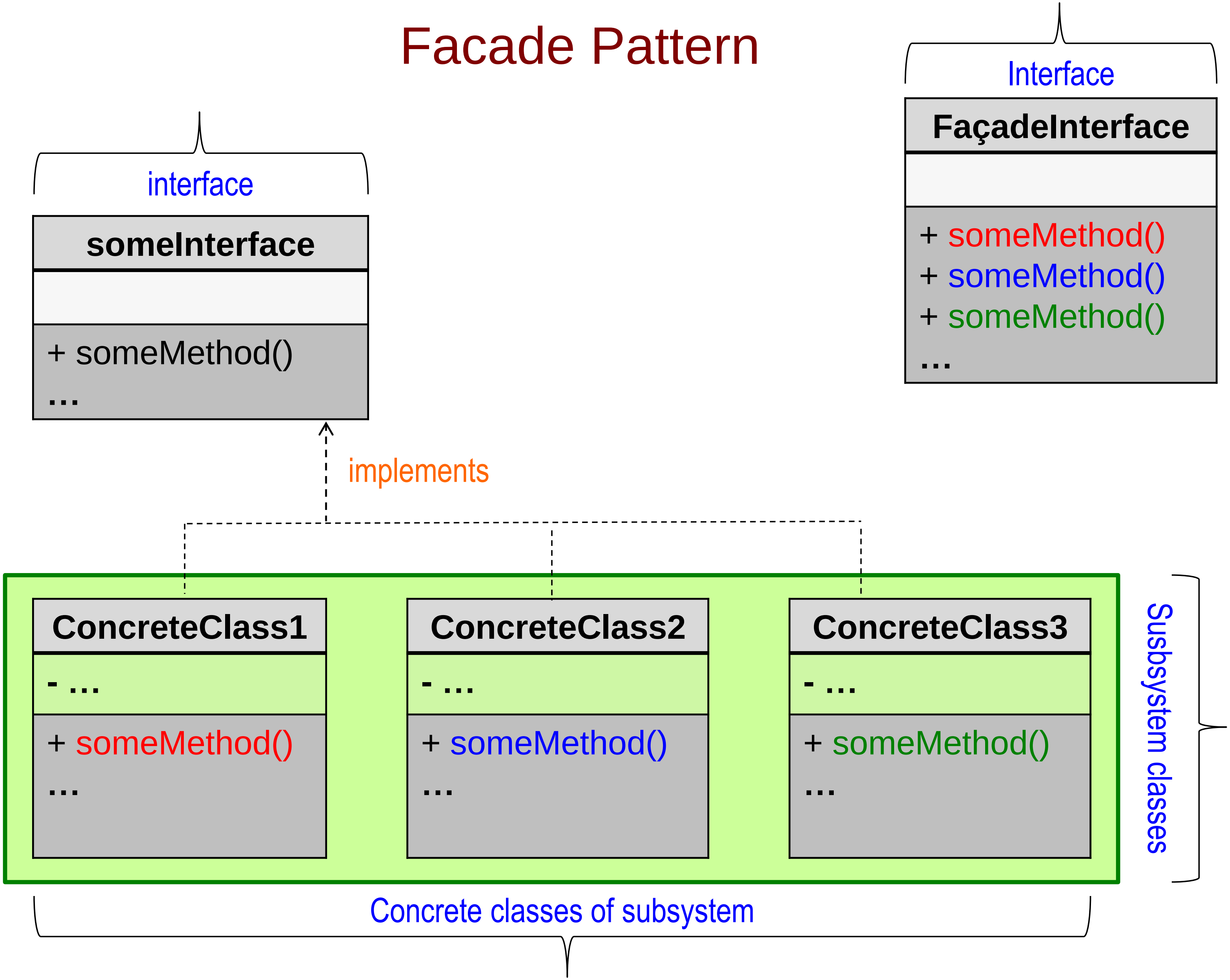
+ someMethod()

…

Susbsystem classes

Concrete classes of subsystem

Facade Pattern



Interface

FaçadeInterface

+ someMethod()
+ someMethod()
+ someMethod()
…

interface

someInterface

+ someMethod()
…

implements

ConcreteClass1

- …

+ someMethod()
…

ConcreteClass2

- …

+ someMethod()
…

ConcreteClass3

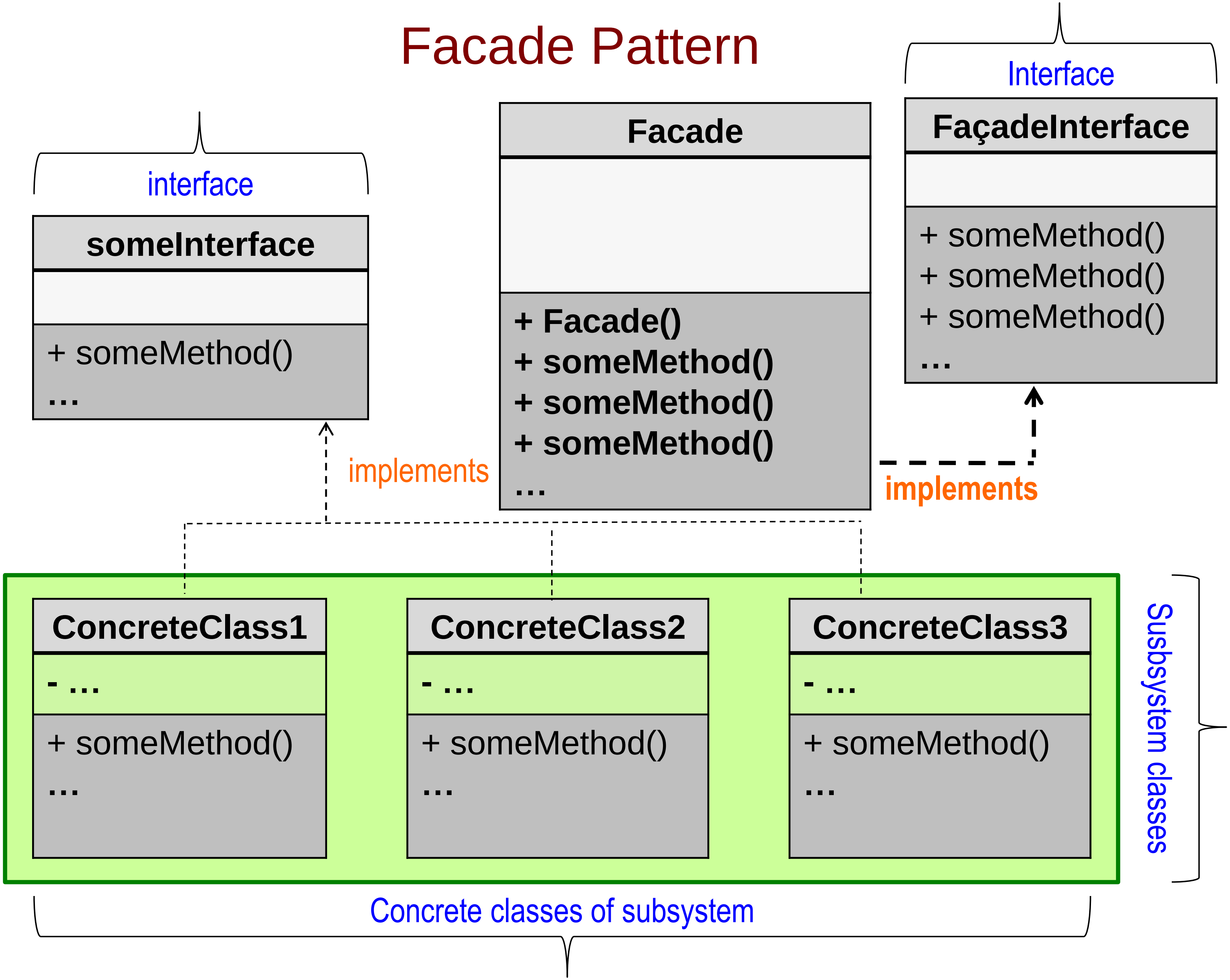
- …

+ someMethod()
…

Susbsystem classes

Concrete classes of subsystem

Facade Pattern



interface

someInterface

+ someMethod()
…

Facade

+ Facade()
+ someMethod()
+ someMethod()
+ someMethod()
…

Interface

FaçadeInterface

+ someMethod()
+ someMethod()
+ someMethod()
…

implements

implements

ConcreteClass1

- …

+ someMethod()
…

ConcreteClass2

- …

+ someMethod()
…

ConcreteClass3

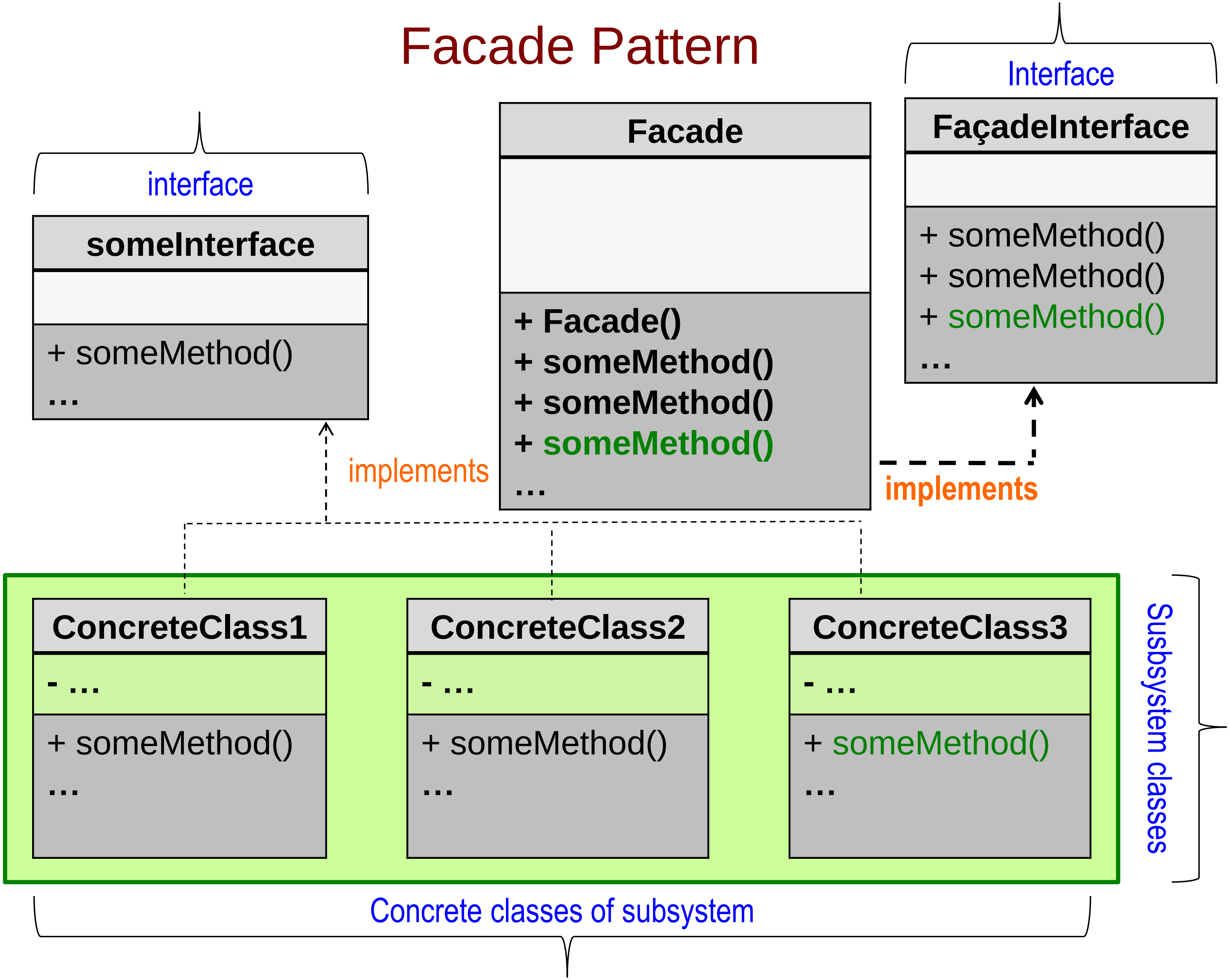
- …

+ someMethod()
…

Susbsystem classes

Concrete classes of subsystem

Facade Pattern



interface

someInterface

+ someMethod()
…

Facade

+ Facade()
+ someMethod()
+ someMethod()
+ someMethod()
…

Interface

FaçadeInterface

+ someMethod()
+ someMethod()
+ someMethod()
…

implements

implements

ConcreteClass1

- …

+ someMethod()
…

ConcreteClass2

- …

+ someMethod()
…

ConcreteClass3

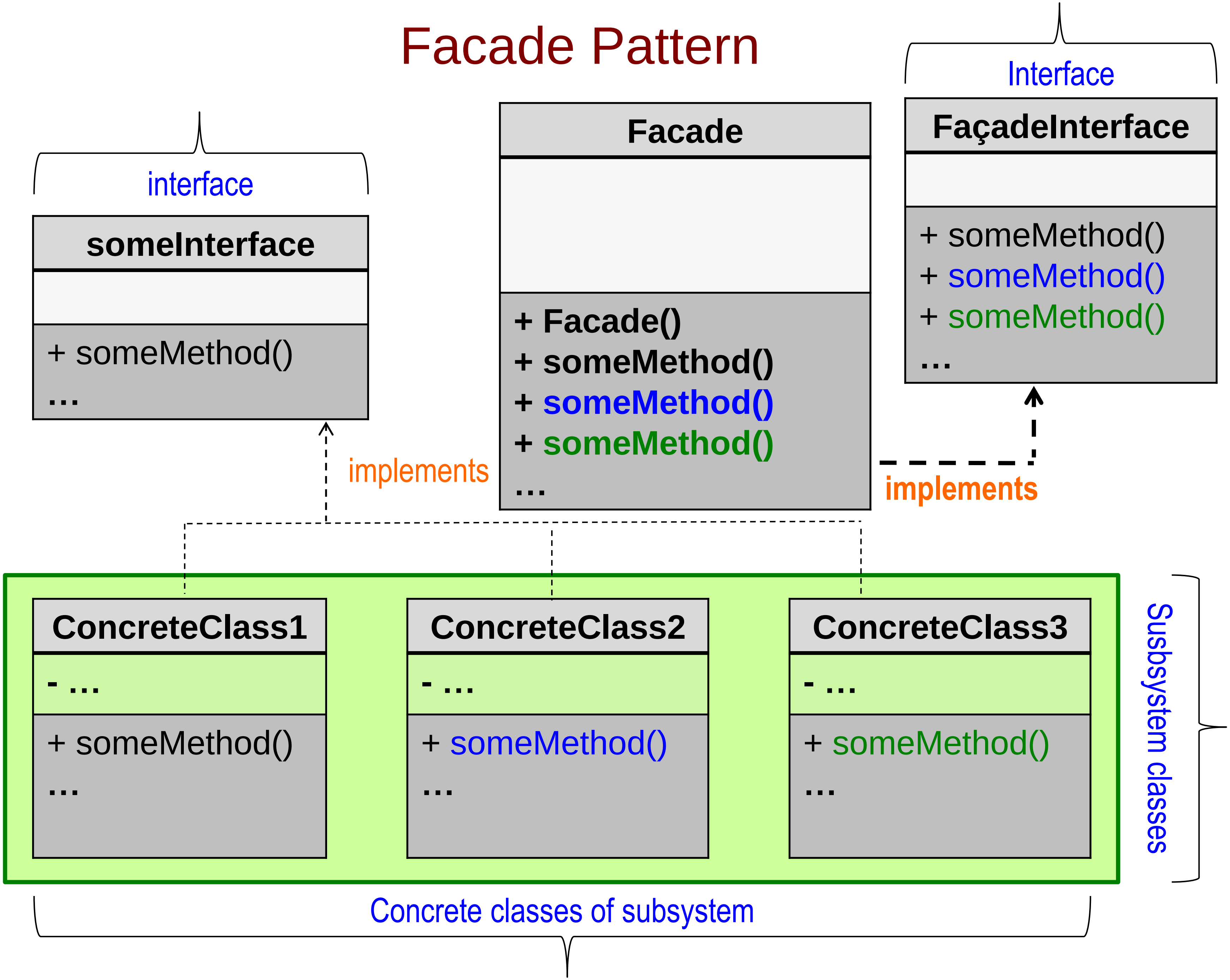
- …

+ someMethod()
…

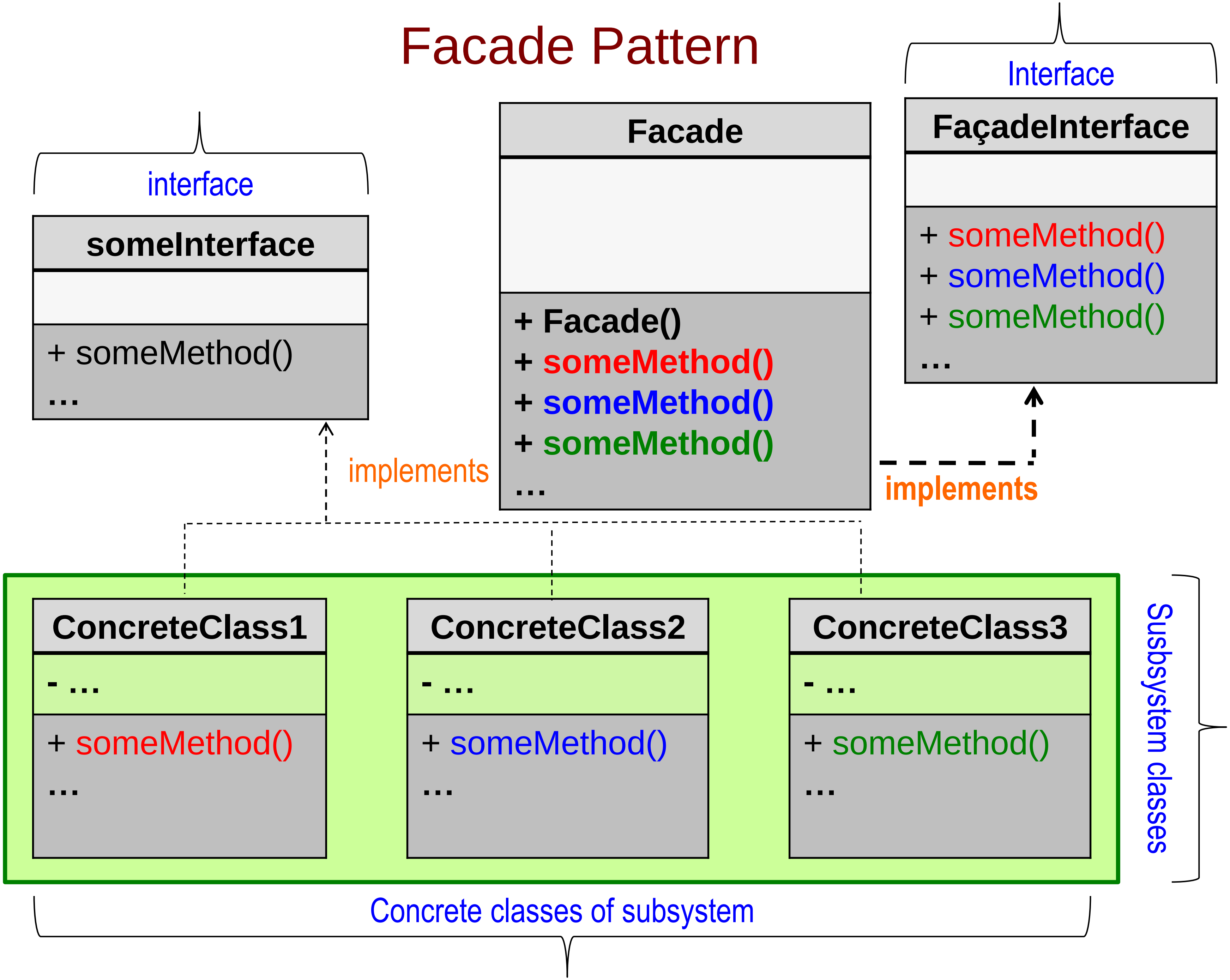
Susbsystem classes

Concrete classes of subsystem

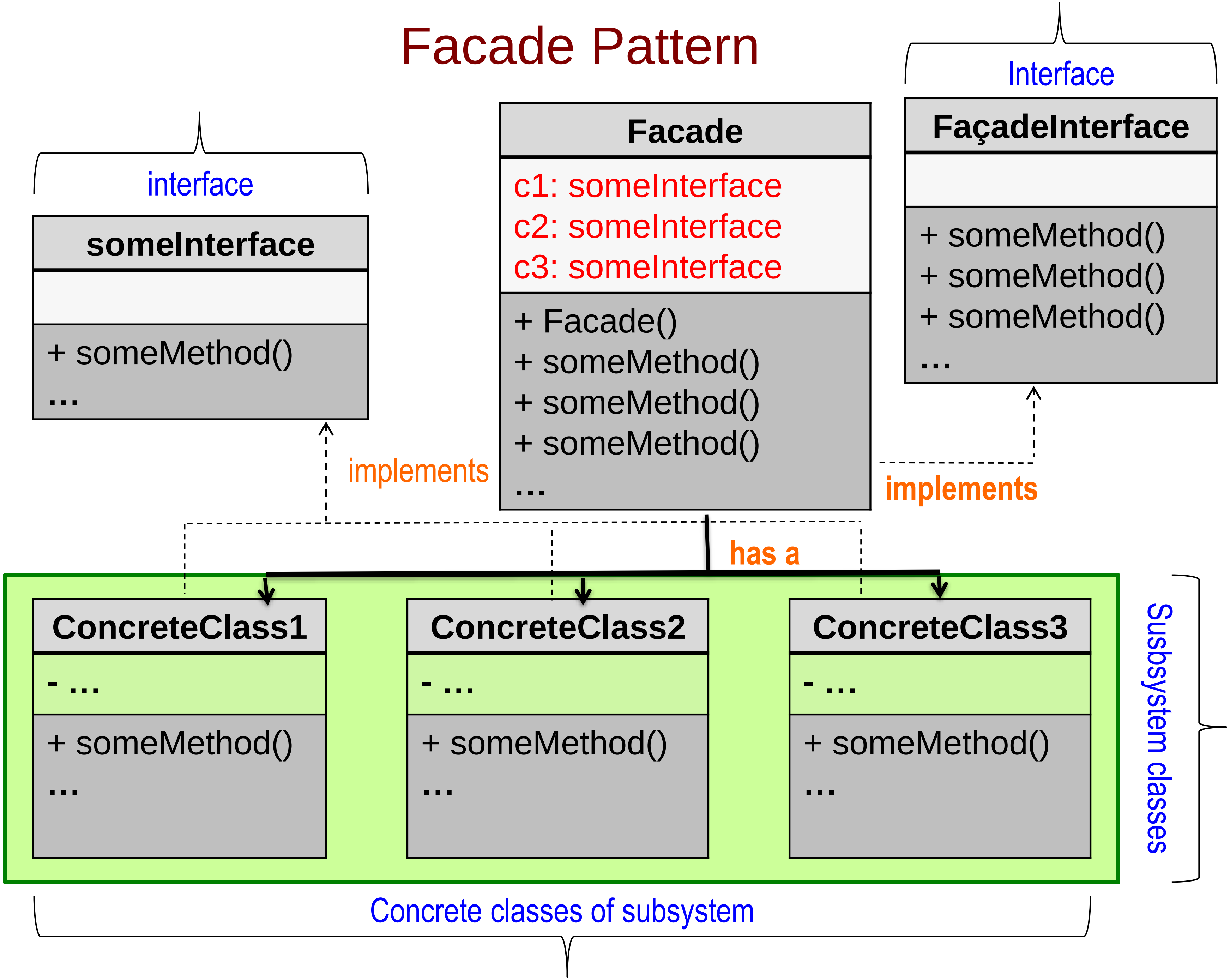
Facade Pattern



Facade Pattern



Facade Pattern



interface

someInterface

+ someMethod()
…

Facade

c1: someInterface
c2: someInterface
c3: someInterface

+ Facade()
+ someMethod()
+ someMethod()
+ someMethod()
…

Interface

FaçadeInterface

+ someMethod()
+ someMethod()
+ someMethod()
…

implements

implements

has a

ConcreteClass1

- …

+ someMethod()
…

ConcreteClass2

- …

+ someMethod()
…

ConcreteClass3

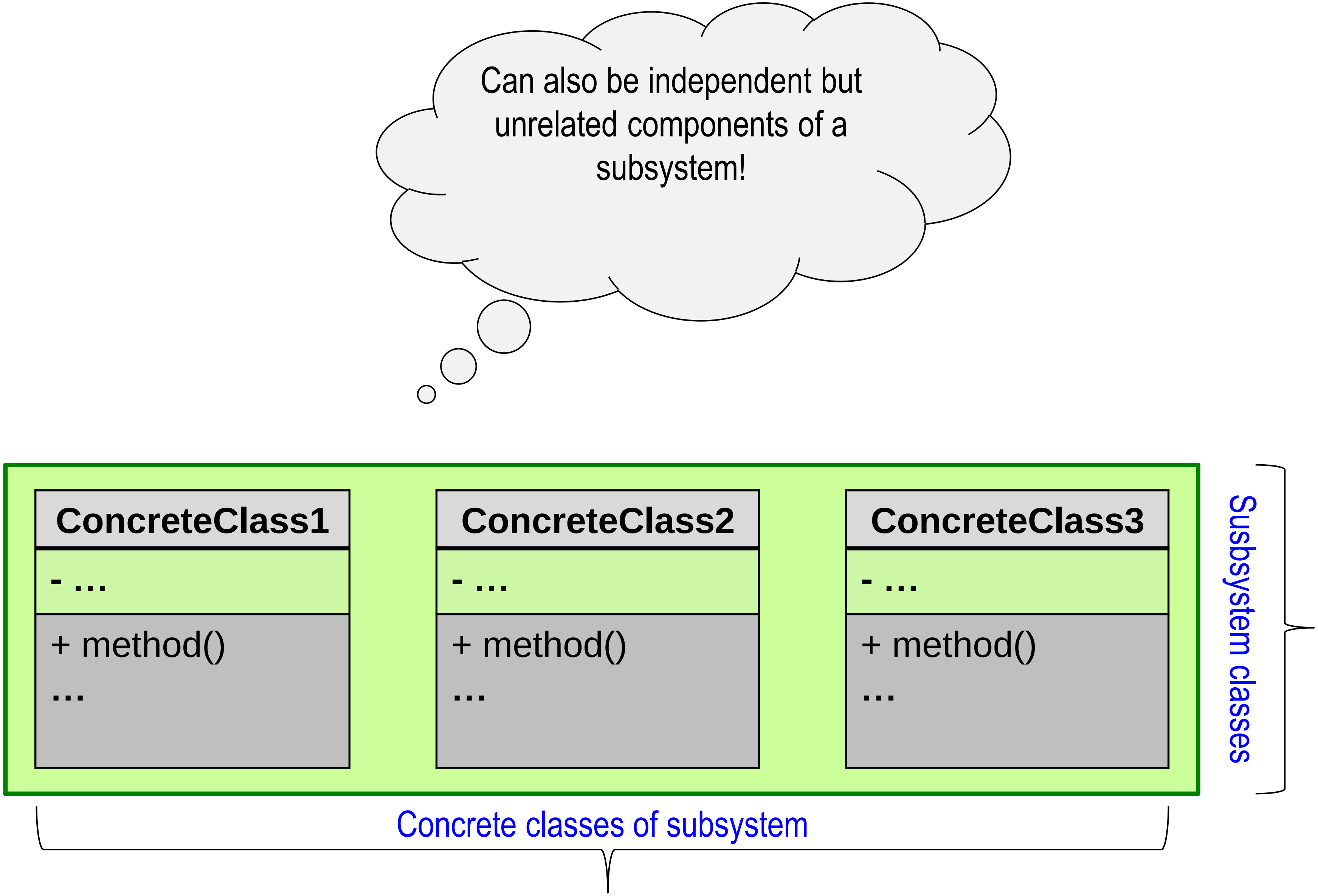
- …

+ someMethod()
…

Susbsystem classes

Concrete classes of subsystem

Facade Pattern



Can also be independent but unrelated components of a subsystem!

ConcreteClass1

- ...

+ method()

...

ConcreteClass2

- ...

+ method()

...

ConcreteClass3

- ...

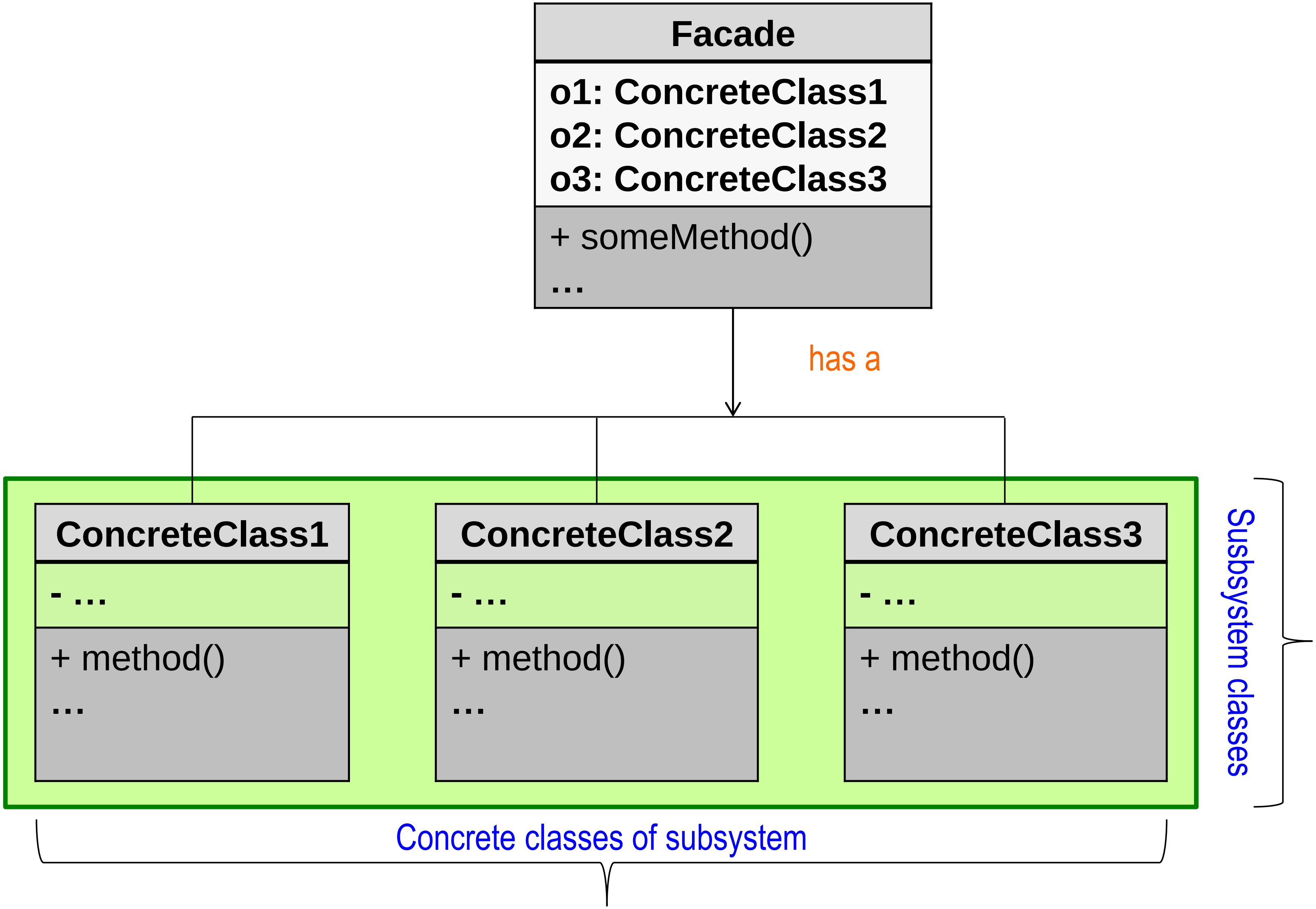
+ method()

...

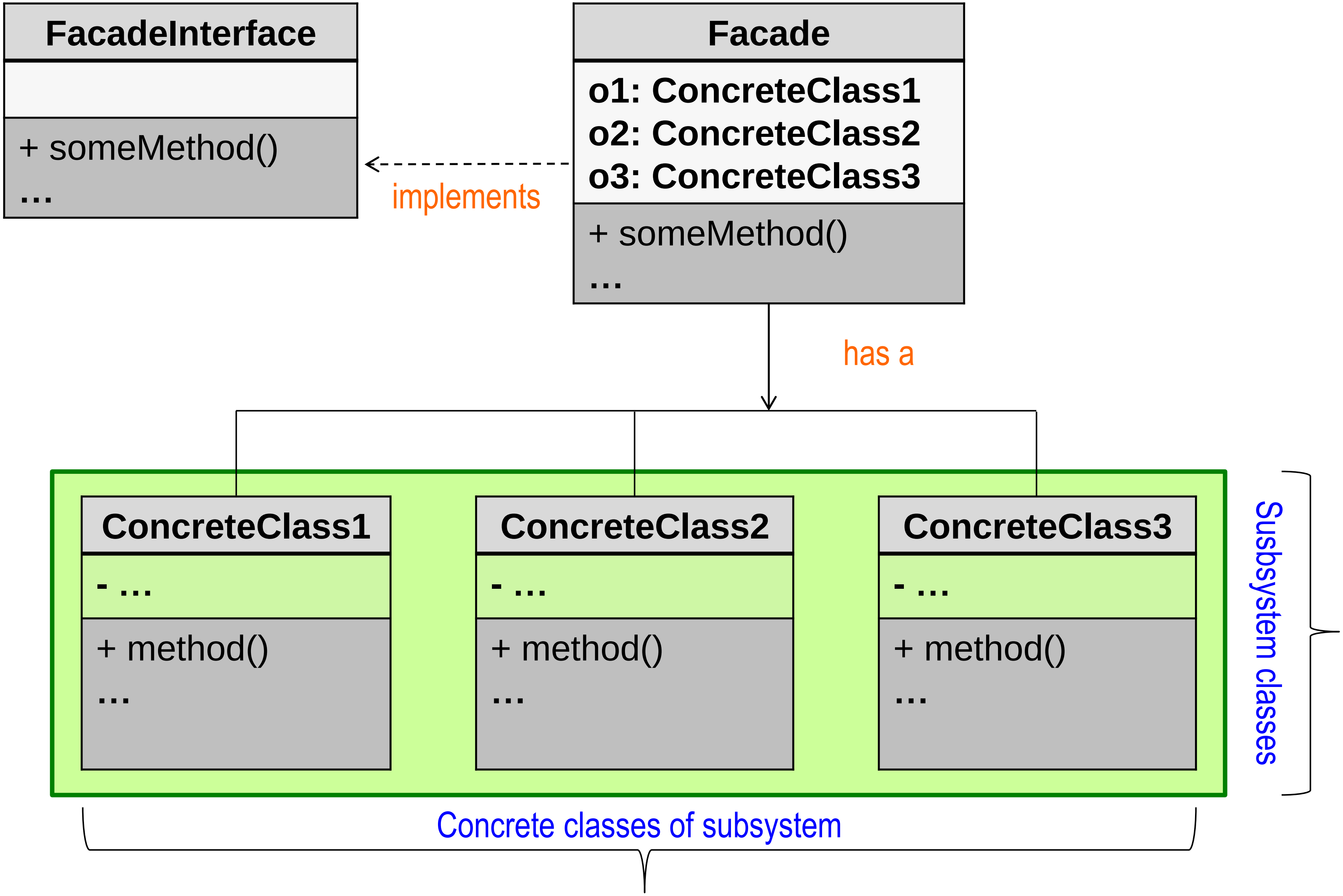
Susbsystem classes

Concrete classes of subsystem

Facade Pattern

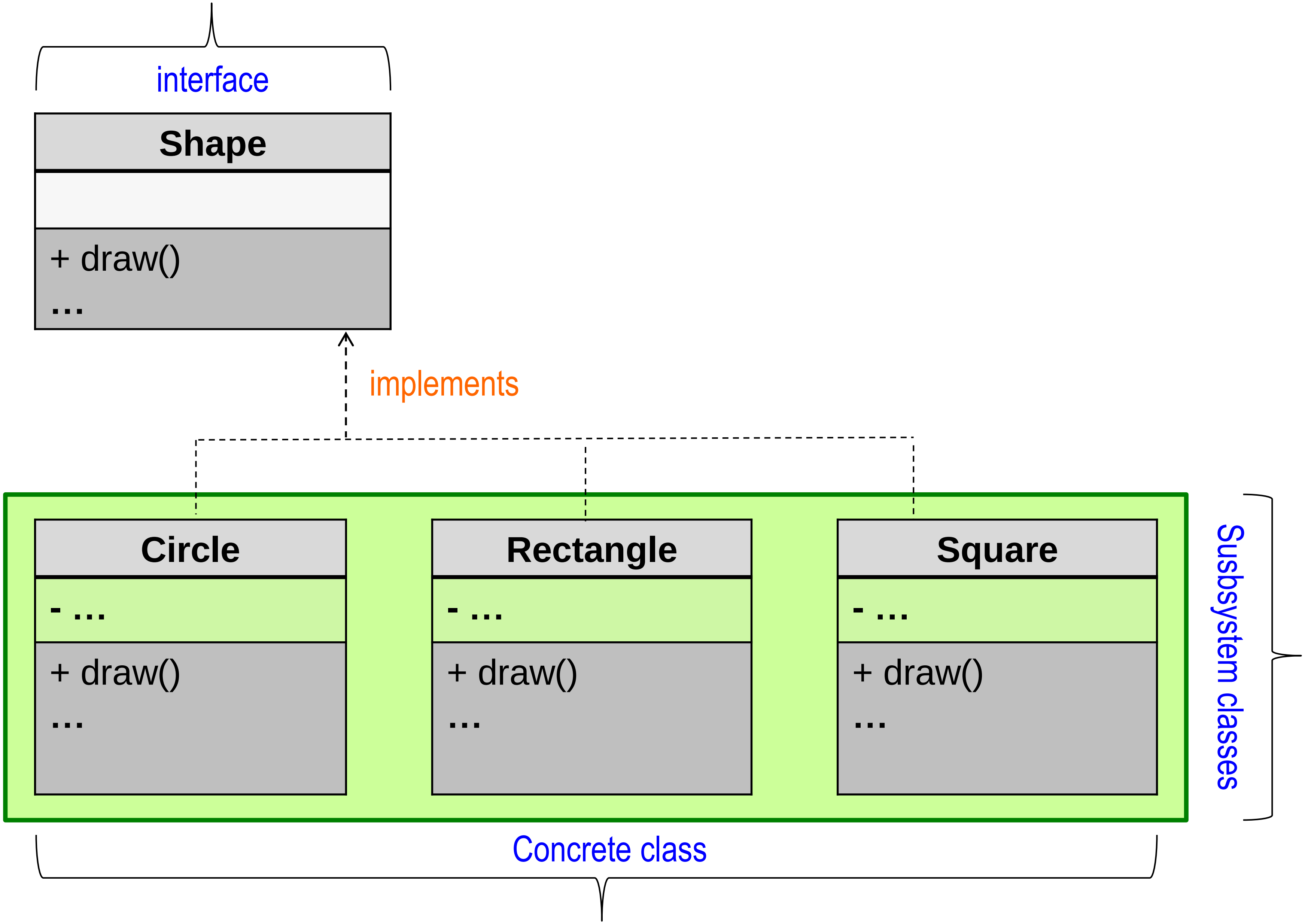


Facade Pattern



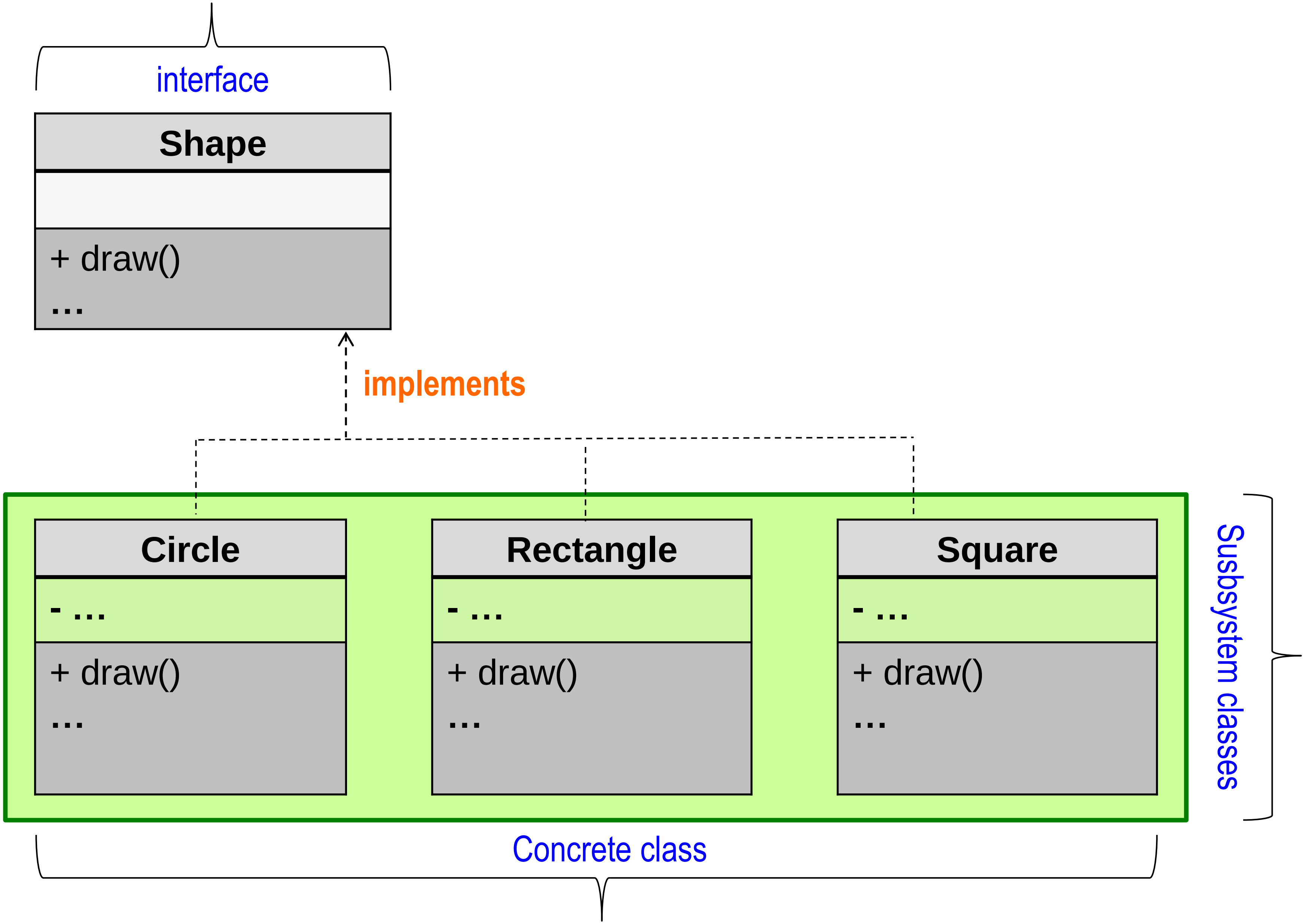
Facade Pattern:

ShapeMaker example



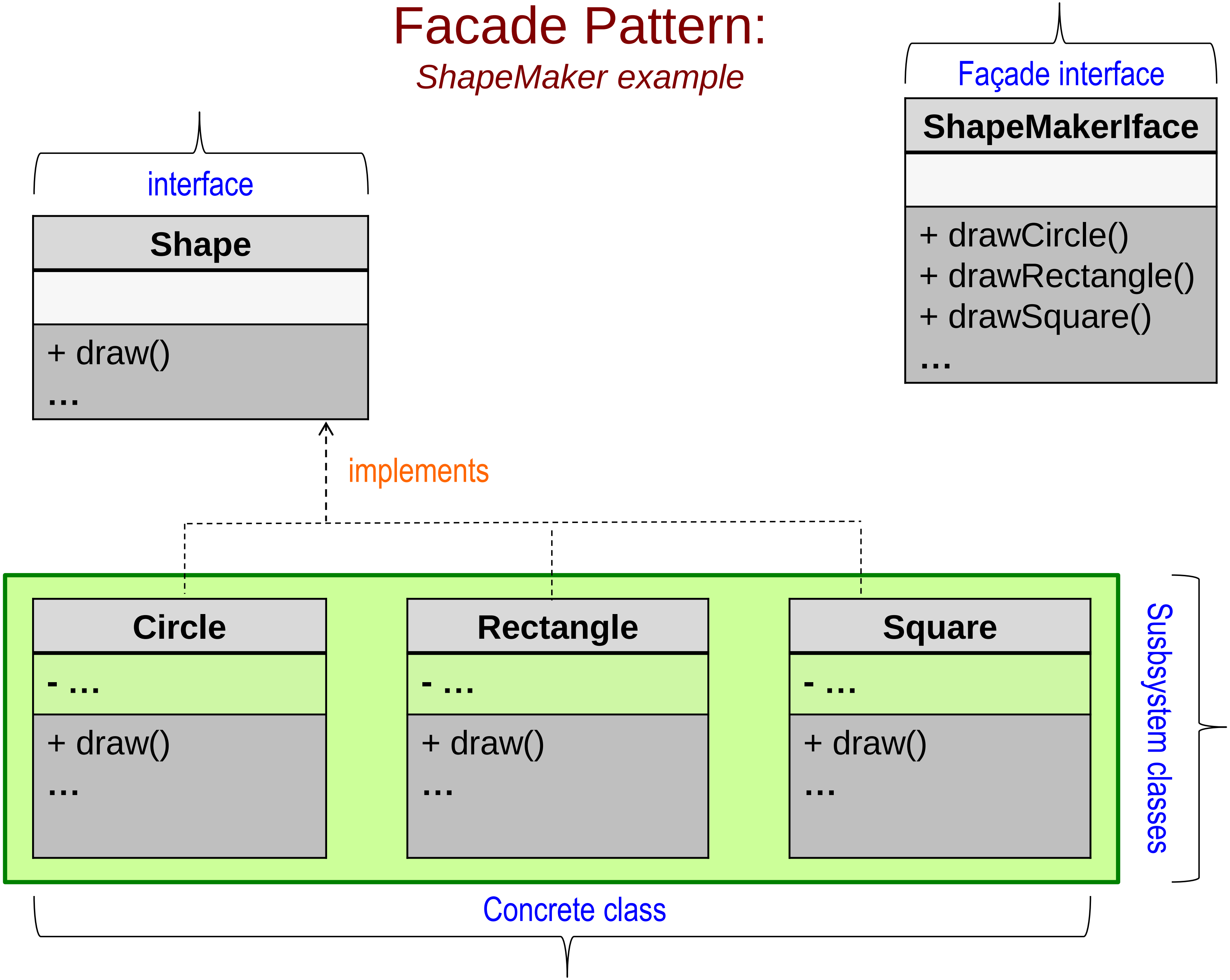
Facade Pattern:

ShapeMaker example



Facade Pattern:

ShapeMaker example



interface

Shape

+ draw()

…

implements

Façade interface

ShapeMakerIface

+ drawCircle()

+ drawRectangle()

+ drawSquare()

…

Circle

- …

+ draw()

…

Rectangle

- …

+ draw()

…

Square

- …

+ draw()

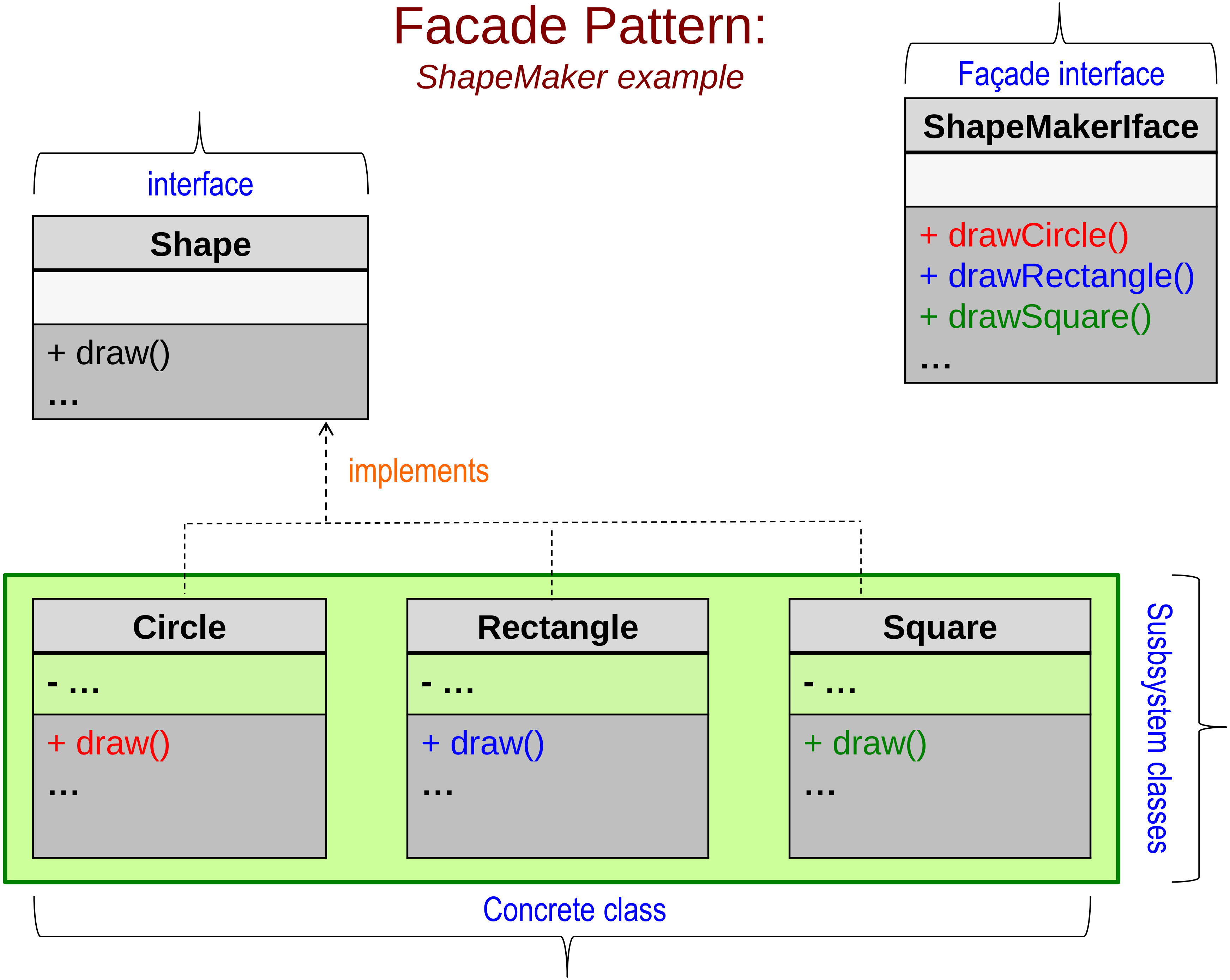
…

Susbsystem classes

Concrete class

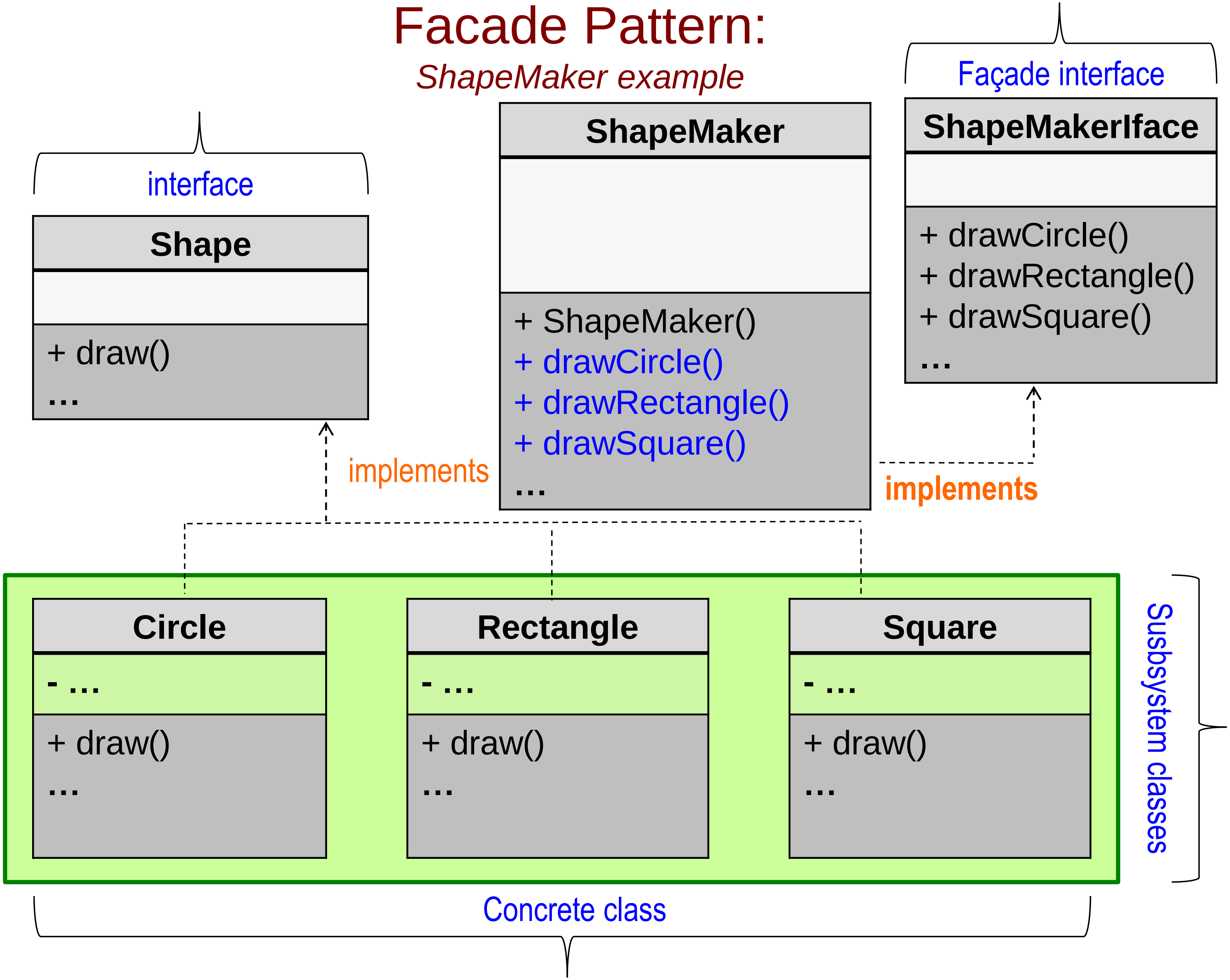
Facade Pattern:

ShapeMaker example



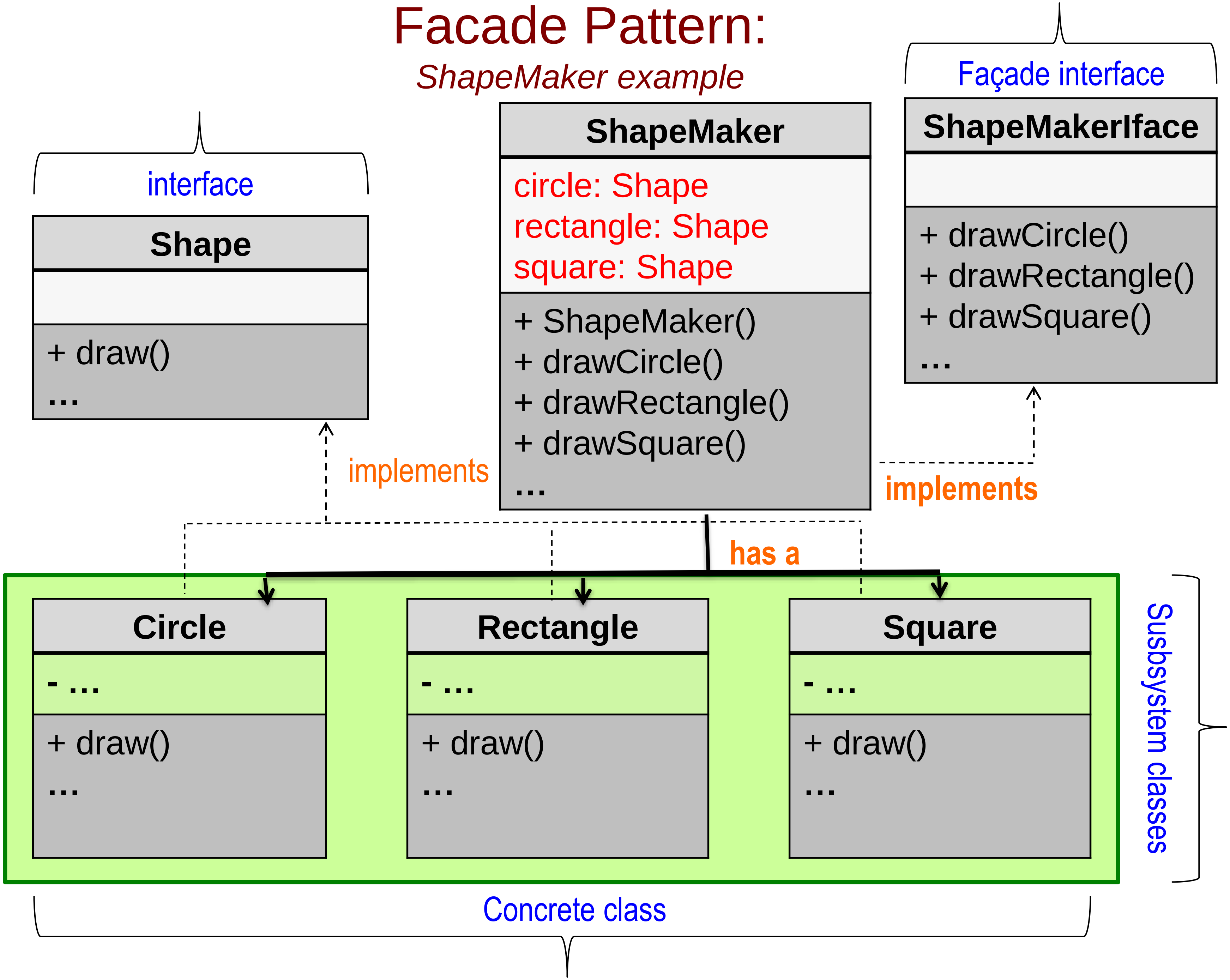
Facade Pattern:

ShapeMaker example



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

- …

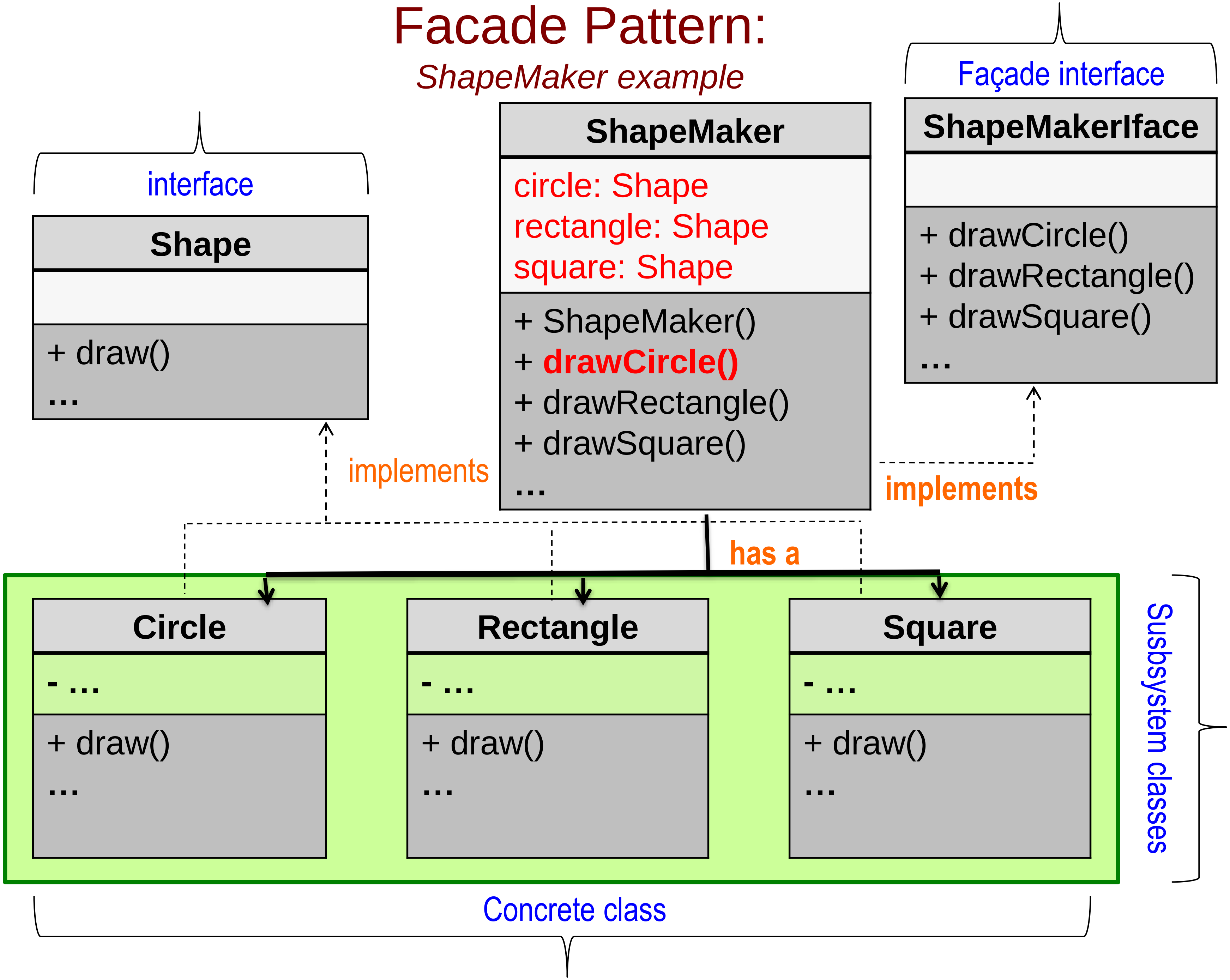
+ draw()
…

Susbsystem classes

Concrete class

Facade Pattern:

ShapeMaker example



interface

Shape

+ draw()
…

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

- …

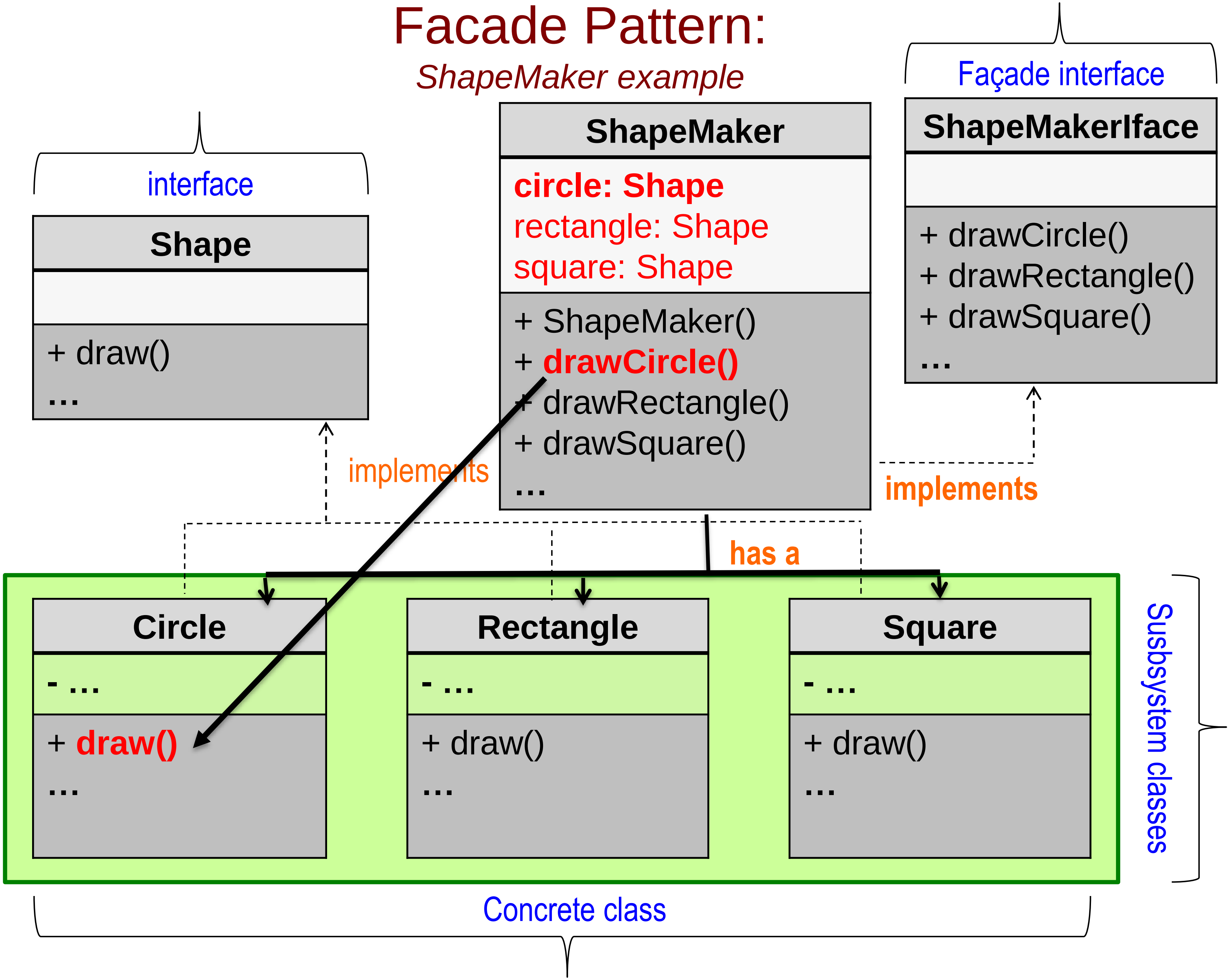
+ draw()
…

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()

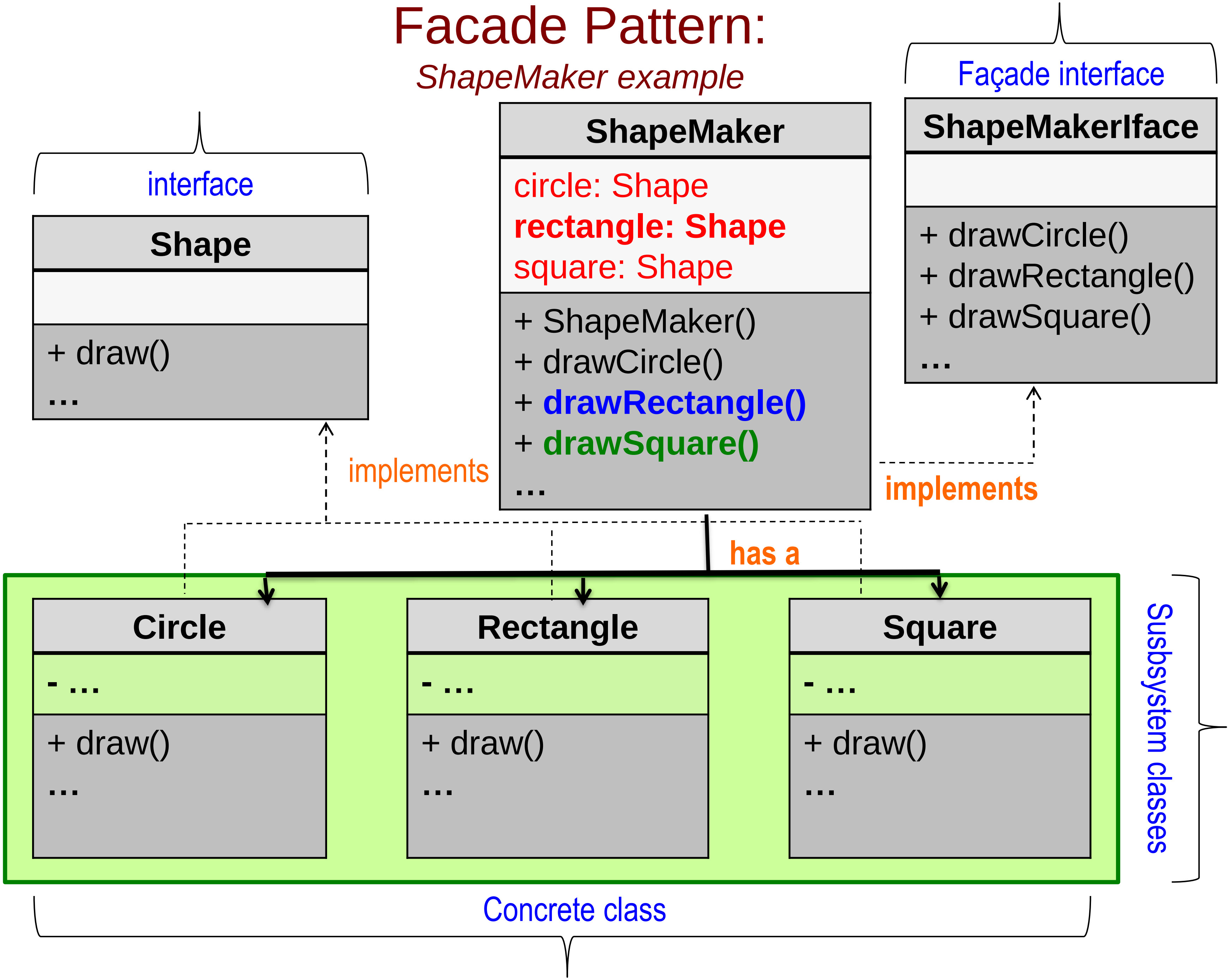
…

Susbsystem classes

Concrete class

Facade Pattern:

ShapeMaker example



interface

Shape

+ draw()
…

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

- …

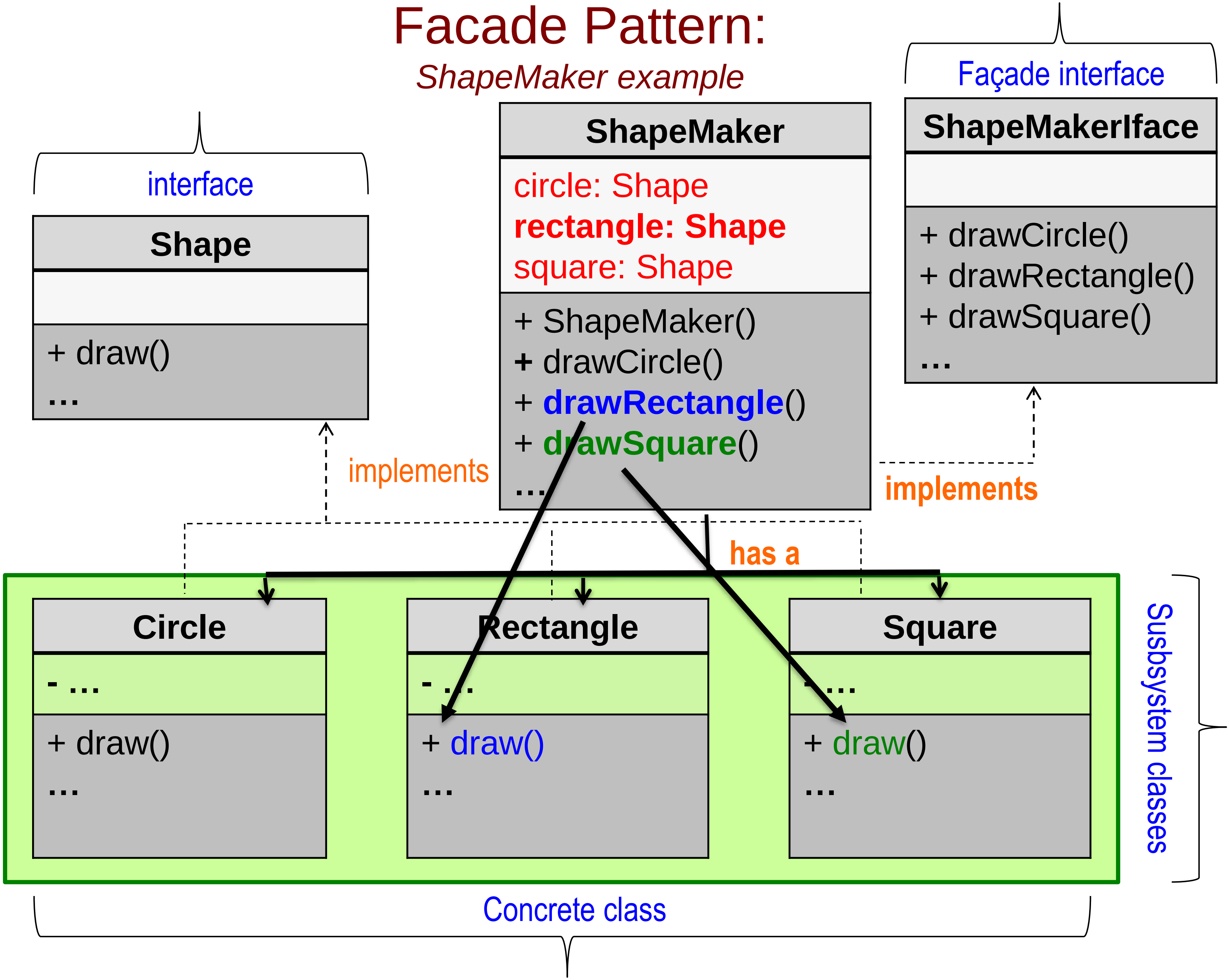
+ draw()
…

Susbsystem classes

Concrete class

Facade Pattern:

ShapeMaker example



interface

Shape

+ draw()

…

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

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

```java
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 no

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

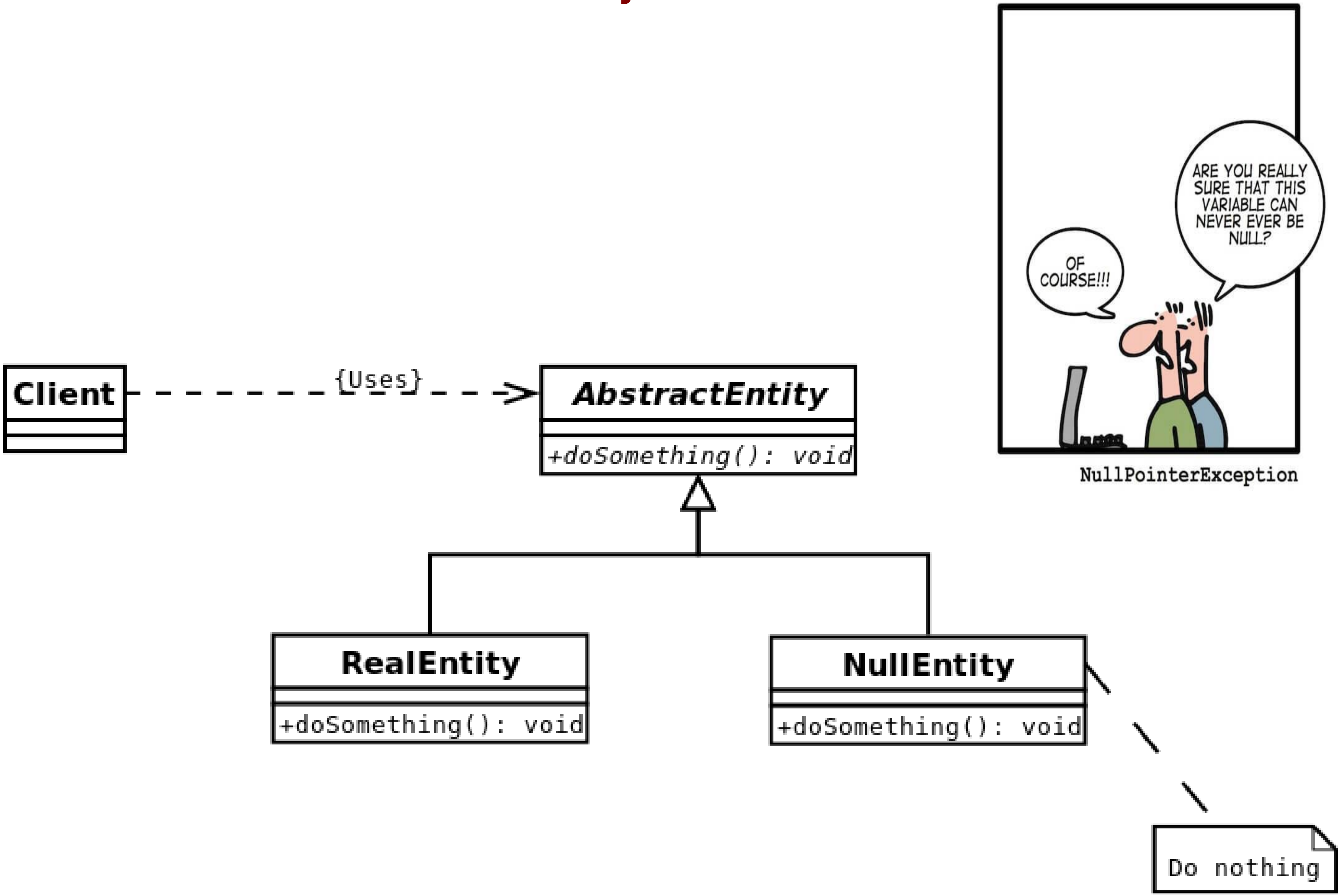
- There are overlapping similarities between many of the structural patterns because *they rely on the same set of*
language
namely: s
patterns a

- *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

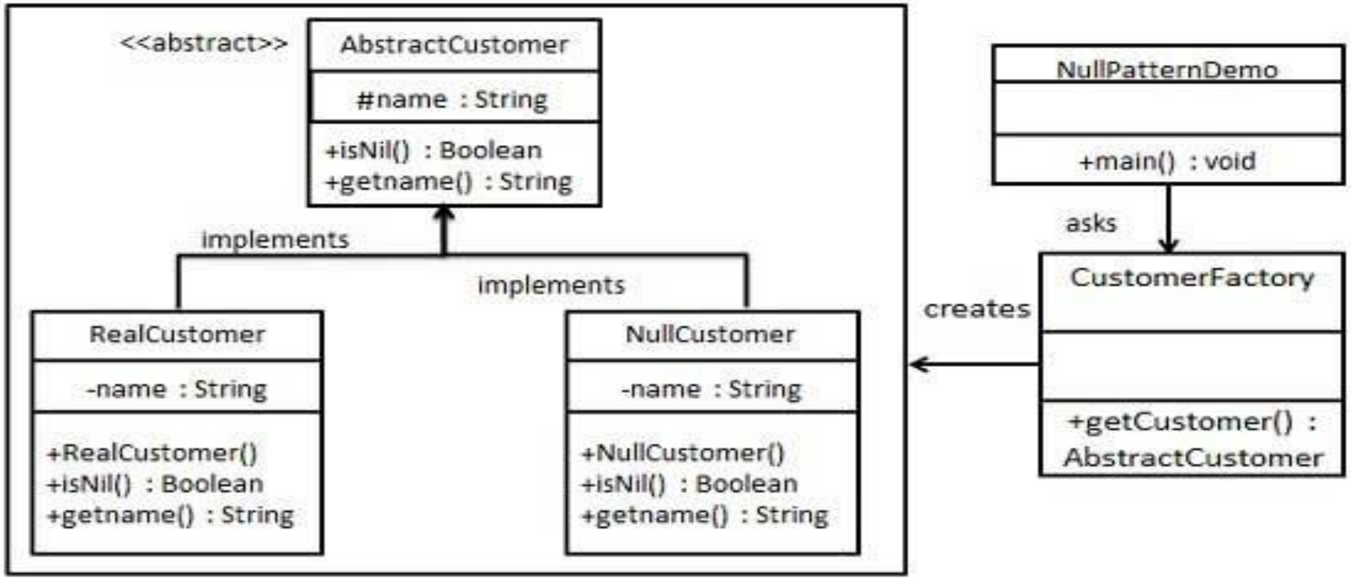
SIMPLY EXPLAINED



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 yo
    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: Re 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
```

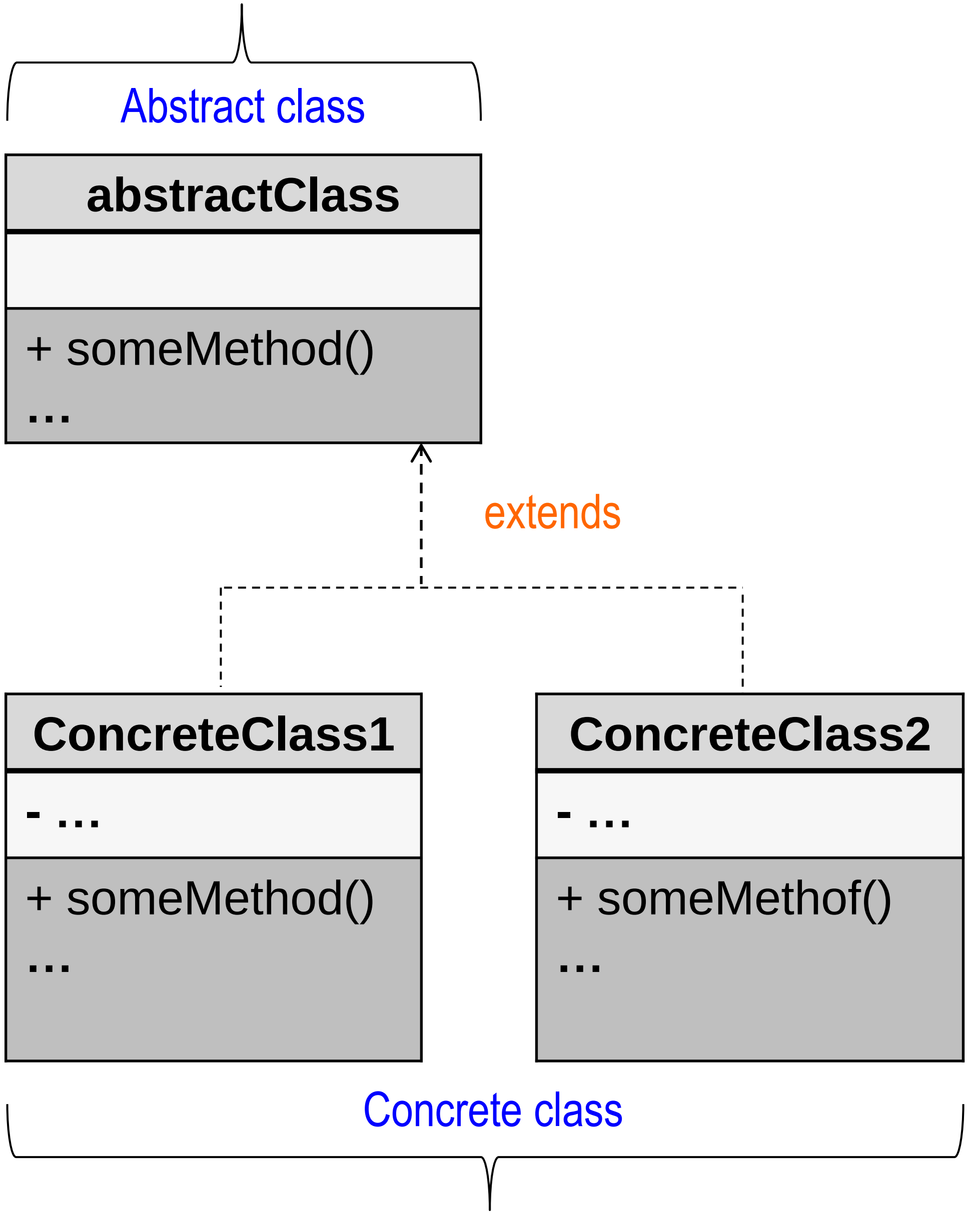
Null Object Pattern

- Motivation and
  coding
  branches
  - When y
    Polymorph

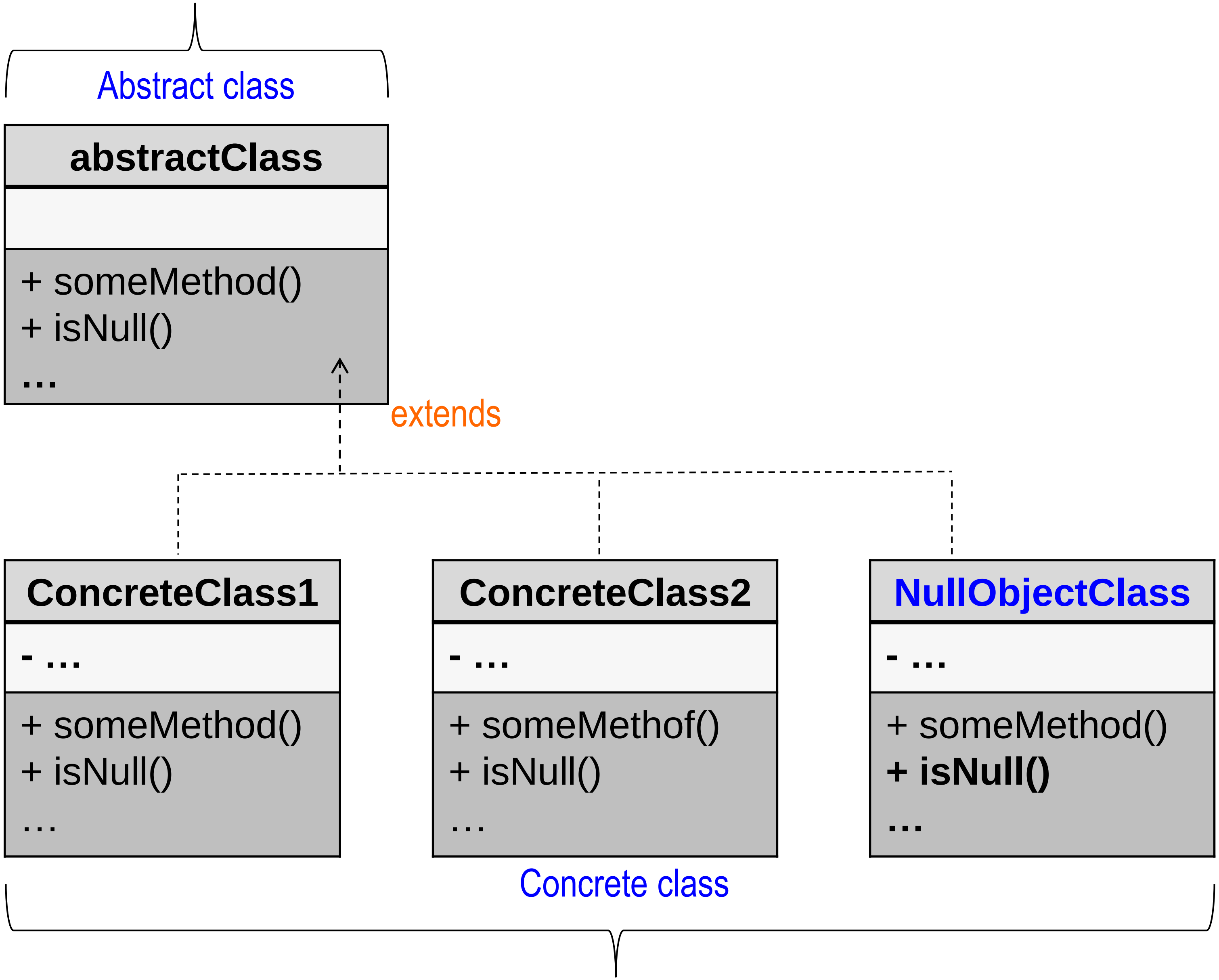
The only way to avoid conditional checks, including exception handling, `getStudent()` cannot return `null`!

```
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



Null Object Pattern



Abstract class

abstractClass

+ someMethod()
+ isNull()
…

extends

ConcreteClass1

- …

+ someMethod()
+ isNull()
…

ConcreteClass2

- …

+ someMethof()
+ isNull()
…

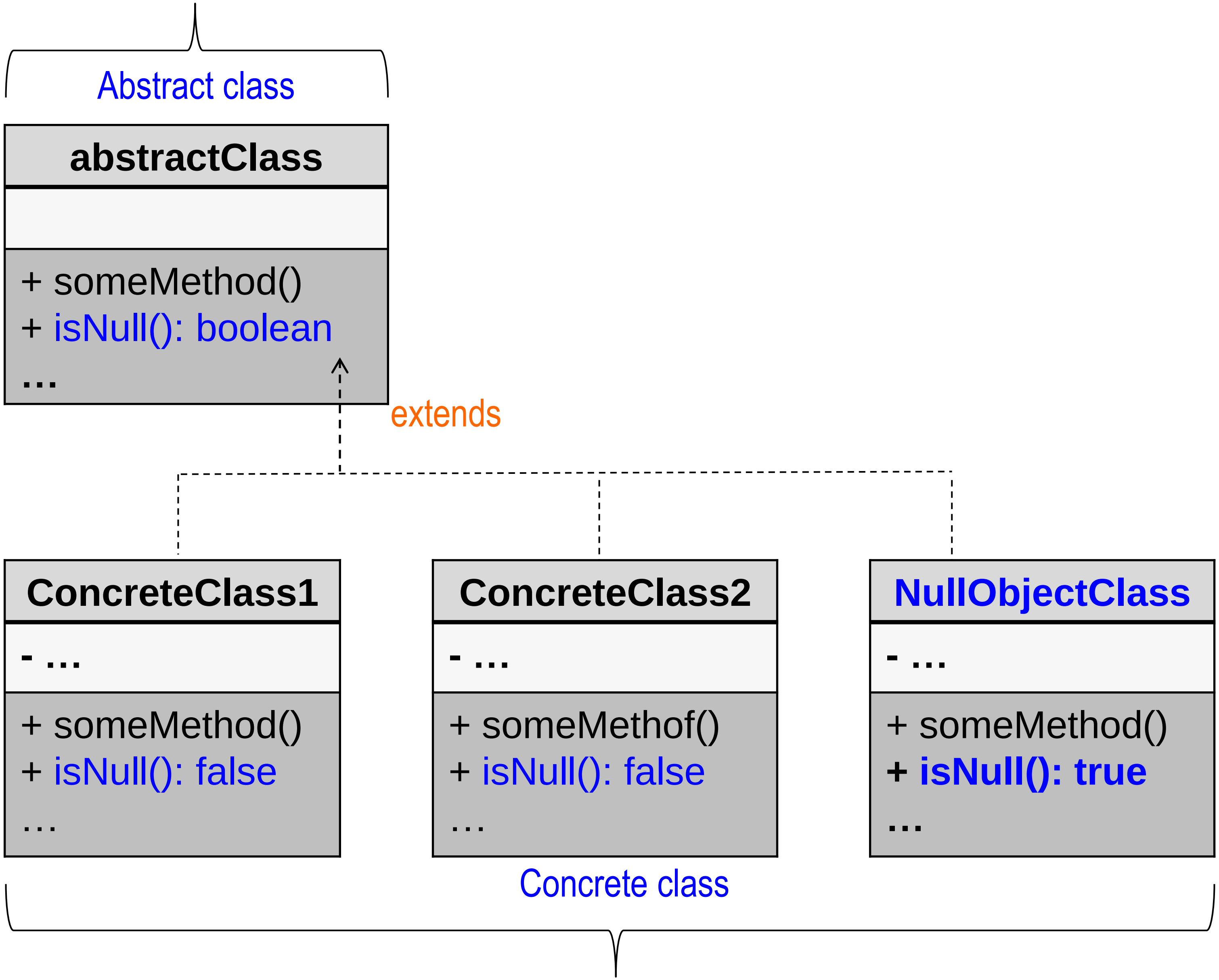
NullObjectClass

- …

+ someMethod()
+ isNull()
…

Concrete class

Null Object Pattern



Abstract class

abstractClass

+ someMethod()
+ isNull(): boolean
…

extends

ConcreteClass1

- …

+ someMethod()
+ isNull(): false
…

ConcreteClass2

- …

+ someMethof()
+ isNull(): false
…

NullObjectClass

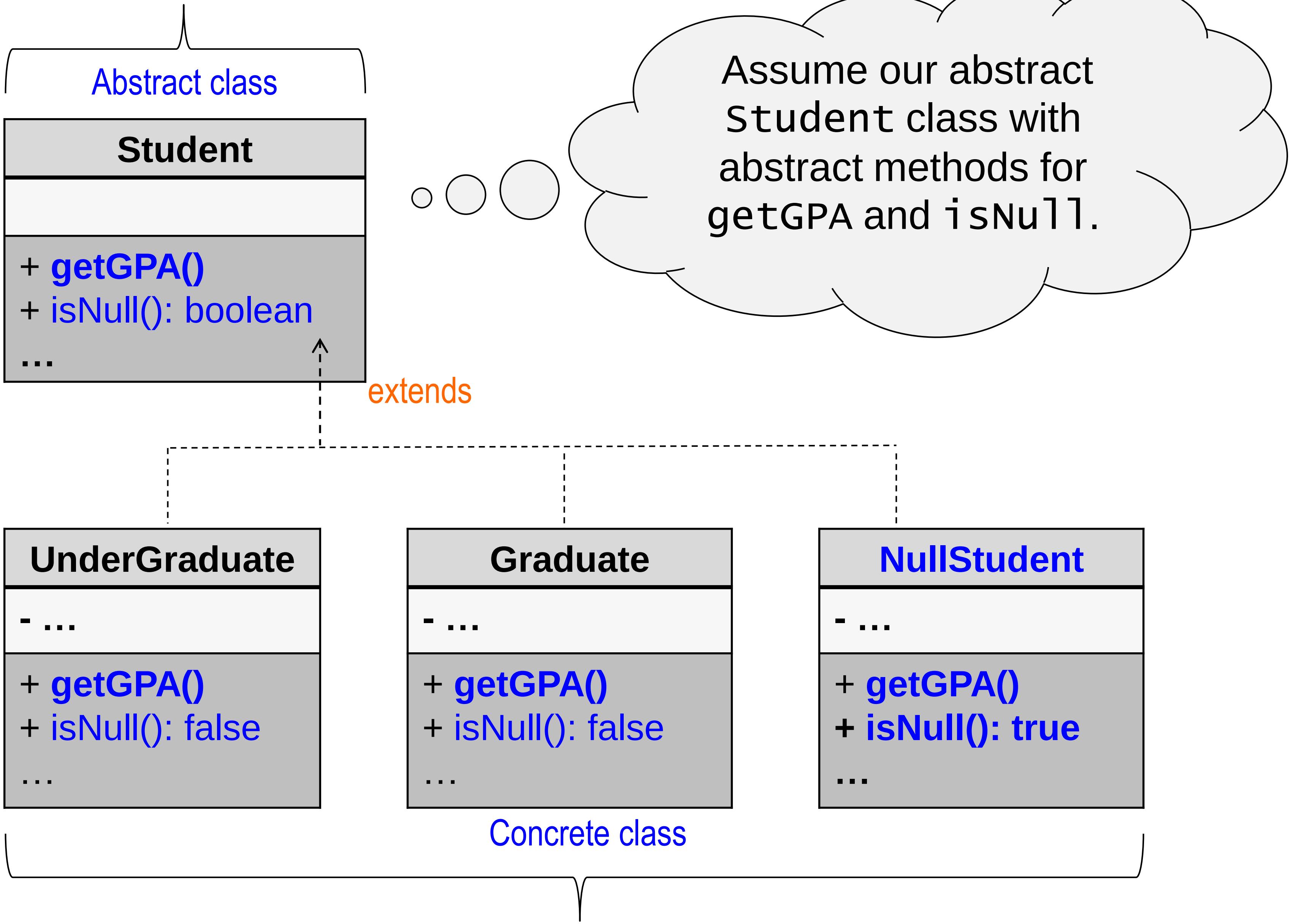
- …

+ someMethod()
+ isNull(): true
…

Concrete class

Null Object Pattern

Student example



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 necessitate creating a new Null Object class for every new Abstract class or interface.

Simplifies the need for conditional checks …

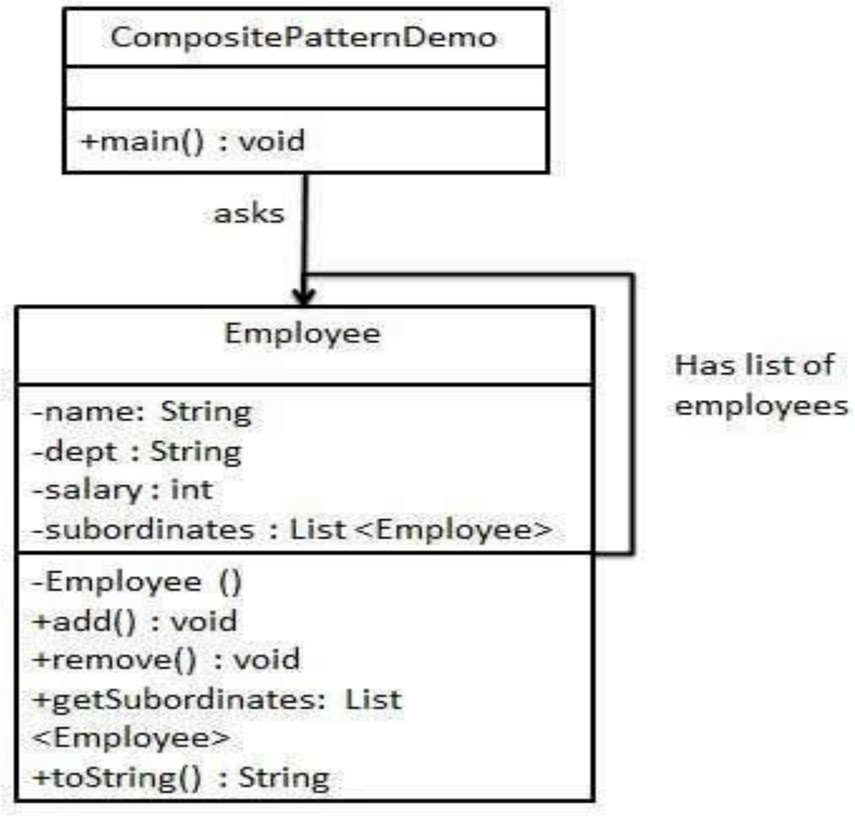
Null Object Pattern

- 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.

Composite Pattern

Intent*:* Composite pattern is ideal for when we want to treat a *group* of objects as a *single* object.



TutorialsPoint

Composite Pattern:

Elements of Reusable OO Software

- **Motivation** and Applicability: One reason for controlling access to an object is to defer the full cost of its creation and initialization until we actually n
  - Provide a
    control ac

When you want to treat a group of objects as a single object.

The composite class allows you to create an object which itself, is made of (i.e. *composed of*) object**s** of the same type.

Composite Pattern:

Elements of Reusable OO Software

- **Motivation** and Applicability: One reason for controlling access to an object is to defer the full cost of its creation and initialization until we actually n
  - Provide a control ac

Used when you want to uniformly treat *single* objects and a *composite of objects* the same way.

Composite Pattern:
Elements of Reusable OO Software

- **Motivation** and Applicability: One reason for controlling access to an object is to defer the full cost of its creation and initialization until we actually n
  - Provide a
    control ac

Used when you want to uniformly treat *single* objects and a *composite of objects* the same way.

This pattern composes objects into *tree structures* to represent **part-whole** hierarchy's and allows clients to treat individual objects and composition of objects uniformly.

Composite Pattern:

Elements of Reusable OO Software

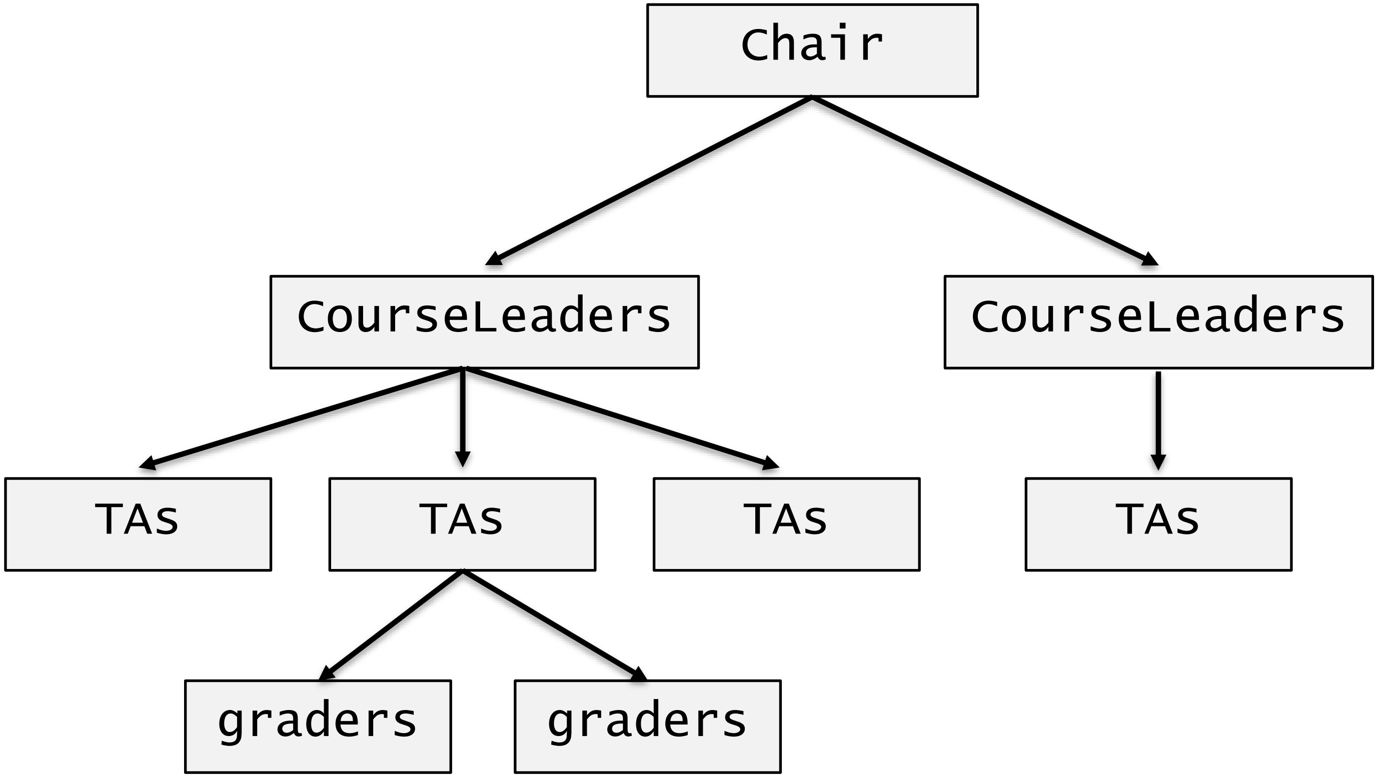
- **Motivation** and Applicability: One reason for c
  an object is to defer the full cost of its creatio
  we actually n
  - Provide a
    control ac

Used when you want to unif
objects and a *composite of c*
way.

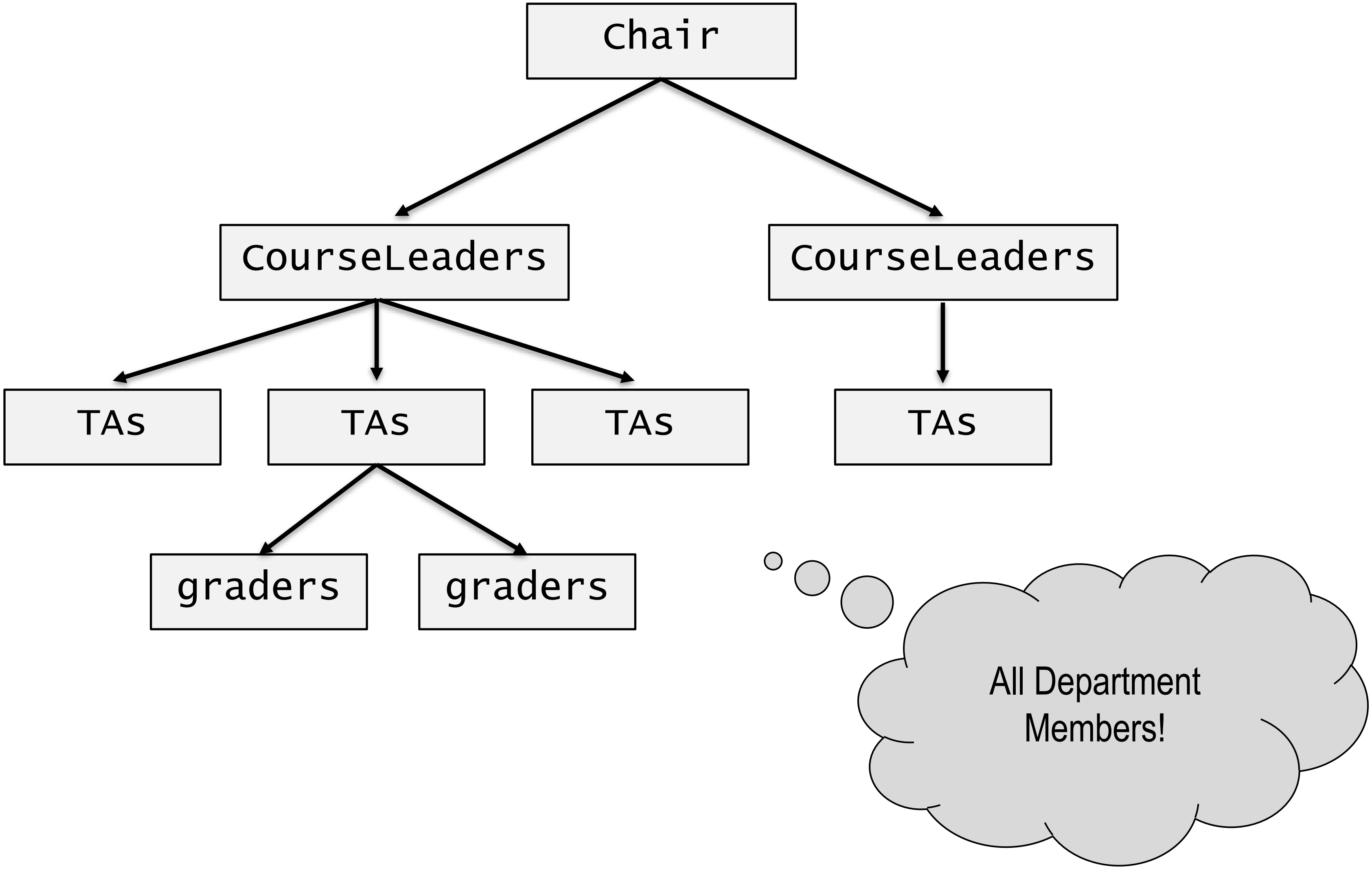
This pattern composes objec into *tree structures* to represent **part-whole** hierarchys and allows clients to treat individual objects and composition of objects uniformly.

Any object is a part of the whole and the whole is a collection of its' parts

CS Department Members



CS Department Members



An example…

```
public class DepartmentMember {
   private String name;
   private String role;
   private List<DepartmentMember> members;

   public DepartmentMember(String name, String role) {
      this.name = name;
      this.role= role;
      members = new ArrayList<DepartmentMember>();
   }
   public void add(DepartmentMember m) {
      members.add(m);
   }
   public void remove(DepartmentMember m) {
      members.remove(m);
   }
   public List<DepartmentMember> getMembers(){
     return members;
   }
   public String toString(){
      return ( name + " : " + role );
   }
   public static void outputRoles( DepartmentMember member ) {
      if ( member.members != null ) {
         System.out.print( member + "\n" );
         for ( DepartmentMember m : member.getMembers() )
            outputRoles(m);
      }
   }
}
```

An example…

```
public class DepartmentMember {
   private String name;
   private String role;
   private List<DepartmentMember> members;

   public DepartmentMember(String name, String role) {
      this.name = name;
      this.role= role;
      members = new ArrayList<DepartmentMember>();
   }
   public void add(DepartmentMember m) {
      members.add(m);
   }
   public void remove(DepartmentMember m) {
      members.remove(m);
   }
   public List<DepartmentMember> getMembers(){
     return members;
   }
   public String toString(){
      return ( name + " : " + role );
   }
   public static void outputRoles( DepartmentMember member ) {
      if ( member.members != null ) {
         System.out.print( member + "\n" );
         for ( DepartmentMember m : member.getMembers() )
            outputRoles(m);
      }
   }
}
```

An example…

```
public class DepartmentMember {
   private String name;
   private String role;
   private List<DepartmentMember> members;

   public DepartmentMember(String name, String role) {
      this.name = name;
      this.role= role;
      members = new ArrayList<DepartmentMember>();
   }
   public void add(DepartmentMember m) {
      members.add(m);
   }
   public void remove(DepartmentMember m) {
      members.remove(m);
   }
   public List<DepartmentMember> getMembers(){
     return members;
   }
   public String toString(){
      return ( name + " : " + role );
   }
   public static void outputRoles( DepartmentMember member ) {
      if ( member.members != null ) {
         System.out.print( member + "\n" );
         for ( DepartmentMember m : member.getMembers() )
            outputRoles(m);
      }
   }
}
```

An example…

```
public class DepartmentMember {
   private String name;
   private String role;
   private List<DepartmentMember> members;

   public DepartmentMember(String name, String role) {
      this.name = name;
      this.role= role;
      members = new ArrayList<DepartmentMember>();
   }
   public void add(DepartmentMember m) {
      members.add(m);
   }
   public void remove(DepartmentMember m) {
      members.remove(m);
   }
   public List<DepartmentMember> getMembers(){
     return members;
   }
   public String toString(){
      return ( name + " : " + role );
   }
   public static void outputRoles( DepartmentMember member ) {
      if ( member.members != null ) {
         System.out.print( member + "\n" );
         for ( DepartmentMember m : member.getMembers() )
            outputRoles(m);
      }
   }
}
```

An example…

```java
public class DepartmentMember {
   private String name;
   private String role;
   private List<DepartmentMember> members;

   public DepartmentMember(String name, String role) {
      this.name = name;
      this.role= role;
      members = new ArrayList<DepartmentMember>();
   }
   public void add(DepartmentMember m) {
      members.add(m);
   }
   public void remove(DepartmentMember m) {
      members.remove(m);
   }
   public List<DepartmentMember> getMembers(){
     return members;
   }
   public String toString(){
      return ( name + " : " + role );
   }
   public static void outputRoles( DepartmentMember member ) {
      if ( member.members != null ) {
         System.out.print( member + "\n" );
         for ( DepartmentMember m : member.getMembers() )
            outputRoles(m);
      }
   }
}
```

Implies that Department members are *mutable*!

An example…

```java
public class DepartmentMember {
   private String name;
   private String role;
   private List<DepartmentMember> members;

   public DepartmentMember(String name, String role) {
      this.name = name;
      this.role= role;
      members = new ArrayList<DepartmentMember>();
   }
   public void add(DepartmentMember m) {
      members.add(m);
   }
   public void remove(DepartmentMember m) {
      members.remove(m);
   }
   public List<DepartmentMember> getMembers(){
     return members;
   }
   public String toString(){
      return ( name + " : " + role );
   }
   public static void outputRoles( DepartmentMember member ) {
      if ( member.members != null ) {
         System.out.print( member + "\n" );
         for ( DepartmentMember m : member.getMembers() )
            outputRoles(m);
      }
   }
}
```

An example…

```
public class DepartmentMember {
   private String name;
   private String role;
   private List<DepartmentMember> members;

   public DepartmentMember(String name, String role) {
      this.name = name;
      this.role= role;
      members = new ArrayList<DepartmentMember>();
   }
   public void add(DepartmentMember m) {
      members.add(m);
   }
   public void remove(DepartmentMember m) {
      members.remove(m);
   }
   public List<DepartmentMember> getMembers(){
     return members;
   }
   public String toString(){
      return ( name + " : " + role );
   }
   public static void outputRoles( DepartmentMember member ) {
      if ( member.members != null ) {
         System.out.print( member + "\n" );
         for ( DepartmentMember m : member.getMembers() )
            outputRoles(m);
      }
   }
}
```

An example…

```
public class DepartmentMember {
   private String name;
   private String role;
   private List<DepartmentMember> members;

   public DepartmentMember(String name, String role) {
      this.name = name;
      this.role= role;
      members = new ArrayList<DepartmentMember>();
   }
   public void add(DepartmentMember m) {
      members.add(m);
   }
   public void remove(DepartmentMember m) {
      members.remove(m);
   }
   public List<DepartmentMember> getMembers(){
     return members;
   }
   public String toString(){
      return ( name + " : " + role );
   }
   public static void outputRoles( DepartmentMember member ) {
      if ( member.members != null ) {
         System.out.print( member + "\n" );
         for ( DepartmentMember m : member.getMembers() )
            outputRoles(m);
      }
   }
}
```

An example…

```
public static void main(String[] args) {
     DepartmentMember chair = new DepartmentMember("Abraham Matta","Chair");

     DepartmentMember softwareLead =
        new DepartmentMember("Christine Papadakis","Software Engineering");
     DepartmentMember foundationsLead =
        new DepartmentMember("David Sullivan","FoundationalCourses");

     DepartmentMember ta1 = new DepartmentMember("Richard","TA");
     DepartmentMember ta2 = new DepartmentMember("Vitor","TA");

     DepartmentMember grader1 = new DepartmentMember("Tania","grader");
     DepartmentMember grader2 = new DepartmentMember("Igor","grader");
     DepartmentMember grader3 = new DepartmentMember("Jack","grader");

     /* Add the Department Haads to the Chair */
     chair.add(softwareLead);
     chair.add(foundationsLead);

     /* Add the TAs to the Faculty leadss */
     softwareLead.add(ta1);
     softwareLead.add(ta2);

     /* Add the graders to the TAs */
     ta1.add(grader1);
     ta1.add(grader2);
     ta2.add(grader3);

     DepartmentMember.outputRoles(chair);
  }
```

An example…

```
public class DepartmentMember {
   private String name;
   private String role;
   private List<DepartmentMember> members;

   public DepartmentMember(String name, String role) {
      this.name = name;
      this.role= role;
      members = new ArrayList<DepartmentMember>();
   }
   public void add(DepartmentMember m) {
      members.add(m);
   }
   public void remove(DepartmentMember m) {
      members.remove(m);
   }
   public List<DepartmentMember> getMembers(){
     return members;
   }
   public String toString(){
      return ( name + " : " + role );
   }

   public void outputRole() {
      System.out.println( toString() );
      for ( DepartmentMember m : getMembers() )
         m.outputRole();
   }
}
```

An example…

```
public class DepartmentMember {
   private String name;
   private String role;
   private List<DepartmentMember> members;

   public DepartmentMember(String name, String role) {
      this.name = name;
      this.role= role;
      members = new ArrayList<DepartmentMember>();
   }
   public void add(DepartmentMember m) {
      members.add(m);
   }
   public void remove(DepartmentMember m) {
      members.remove(m);
   }
   public List<DepartmentMember> getMembers(){
     return members;
   }
   public String toString(){
      return ( name + " : " + role );
   }

   public void outputRole() {
      System.out.println( toString() );
      for ( DepartmentMember m : getMembers() )
         m.outputRole();
   }
}
```

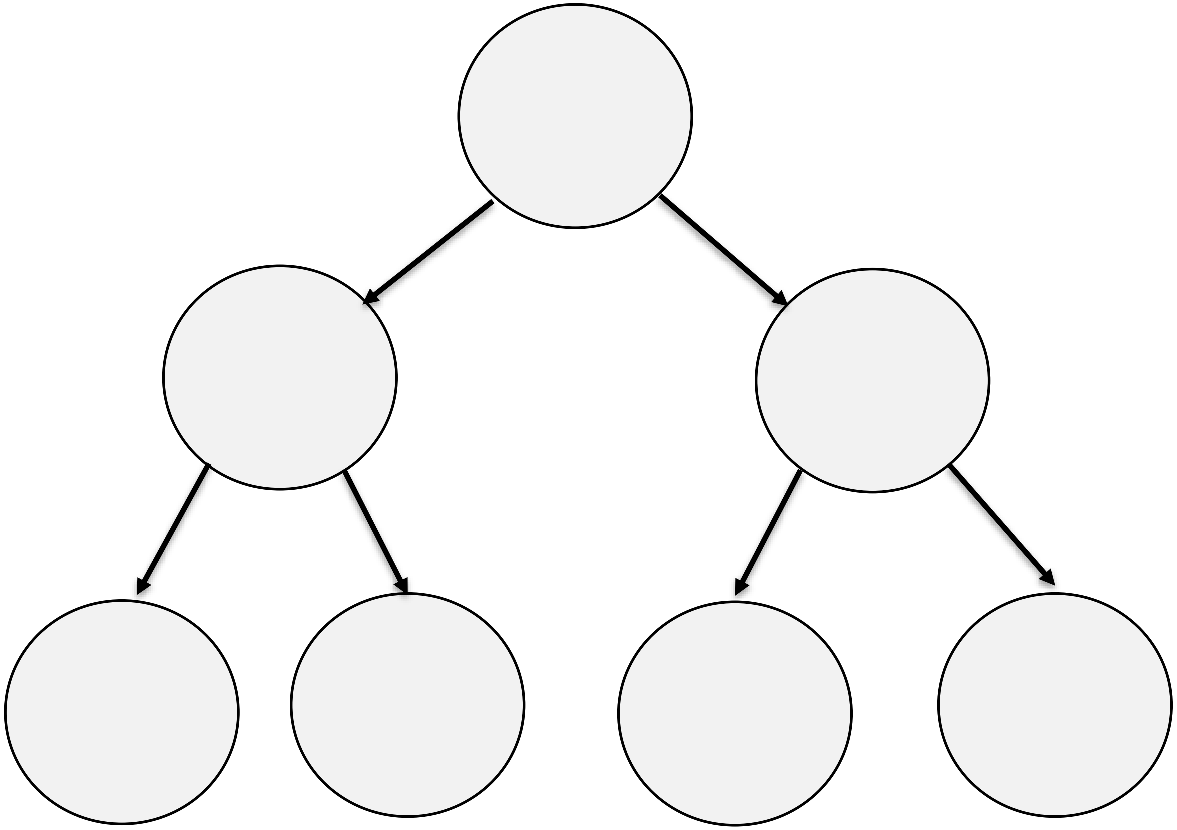
An example…

```
public static void main(String[] args) {
     DepartmentMember chair = new DepartmentMember("Abraham Matta","Chair");

     DepartmentMember softwareLead =
        new DepartmentMember("Christine Papadakis","Software Engineering");
     DepartmentMember foundationsLead =
        new DepartmentMember("David Sullivan","FoundationalCourses");

     DepartmentMember ta1 = new DepartmentMember("Richard","TA");
     DepartmentMember ta2 = new DepartmentMember("Vitor","TA");

     DepartmentMember grader1 = new DepartmentMember("Tania","grader");
     DepartmentMember grader2 = new DepartmentMember("Igor","grader");
     DepartmentMember grader3 = new DepartmentMember("Jack","grader");

     /* Add the Department Haads to the Chair */
     chair.add(softwareLead);
     chair.add(foundationsLead);

     /* Add the TAs to the Faculty leadss */
     softwareLead.add(ta1);
     softwareLead.add(ta2);

     /* Add the graders to the TAs */
     ta1.add(grader1);
     ta1.add(grader2);
     ta2.add(grader3);

     chair.outputRole();
  }
```

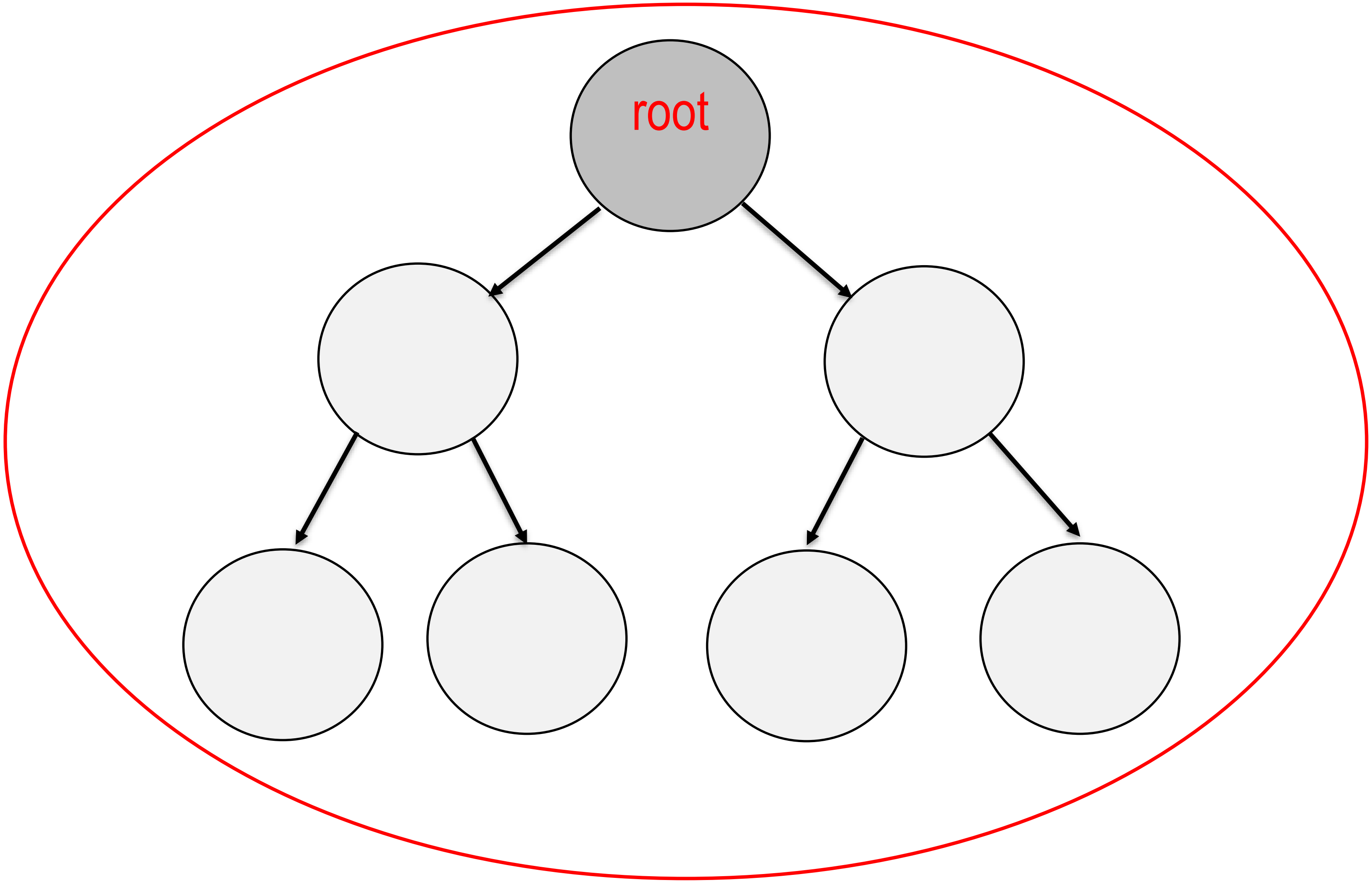
An example…

```
public class DepartmentMember {
   private String name;
   private String role;
   private List<DepartmentMember> members;

   public DepartmentMember(String name, String role) {
      this.name = name;
      this.role= role;
      members = new ArrayList<DepartmentMember>();
   }
   public void add(DepartmentMember m) {
      members.add(m);
   }
   public void remove(DepartmentMember m) {
      members.remove(m);
   }
   public List<DepartmentMember> getMembers(){
     return members;
   }
   public String toString(){
      return ( name + " : " + role );
   }

   public void outputRole() {
      System.out.println( toString() );
      for ( DepartmentMember m : getMembers() )
         m.outputRole();
   }
}
```

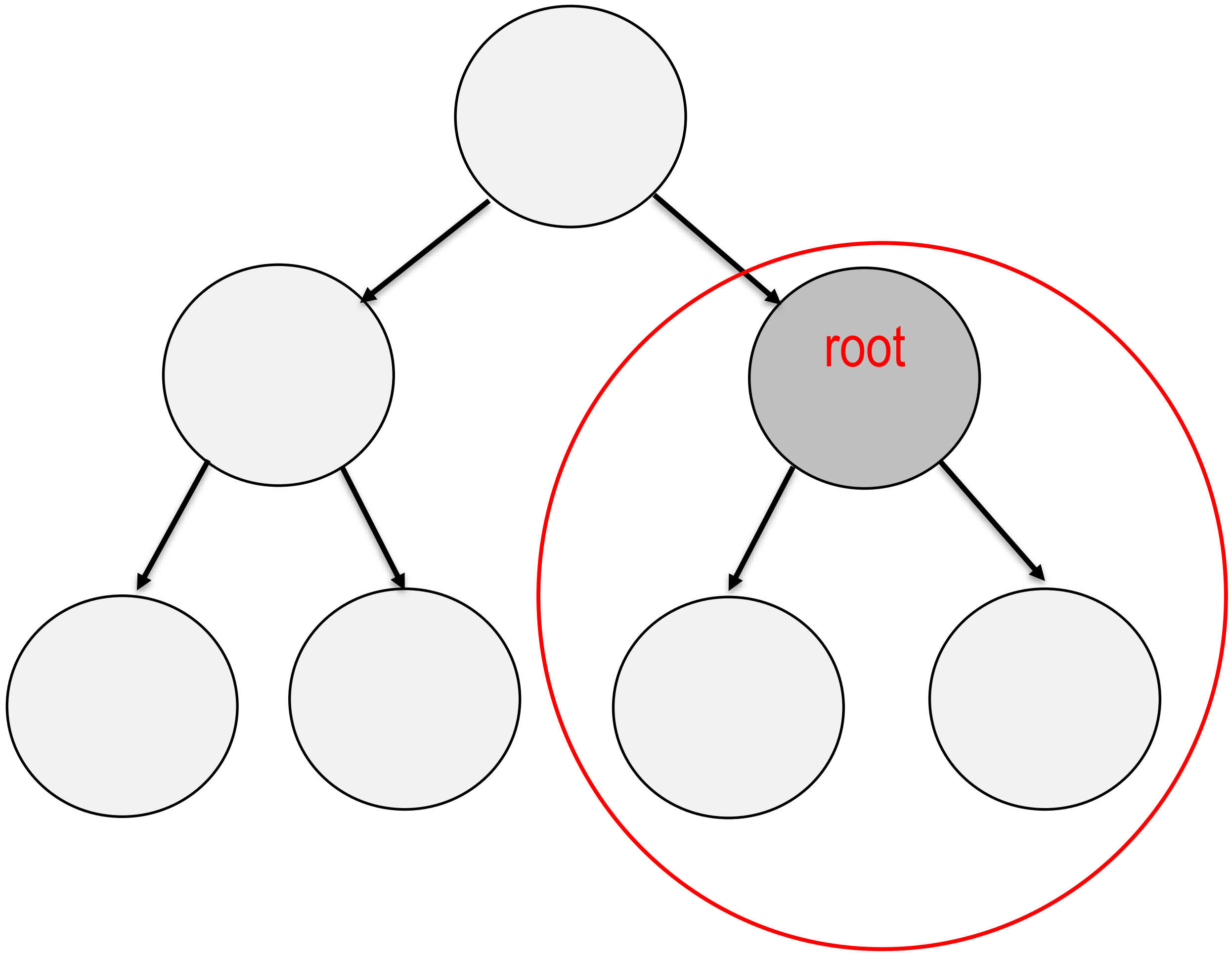
Trees as a Composite Object



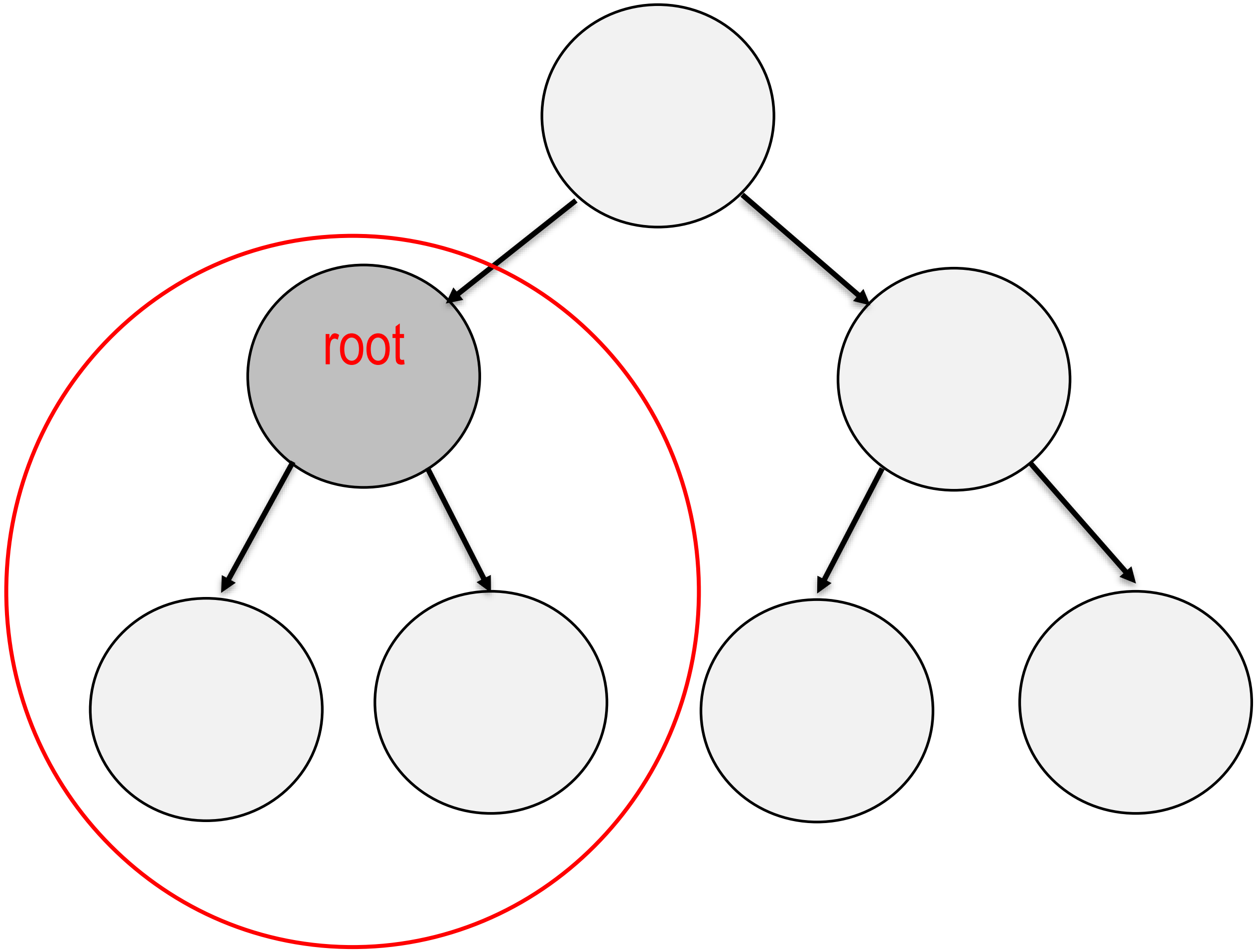
Trees as a Composite Object



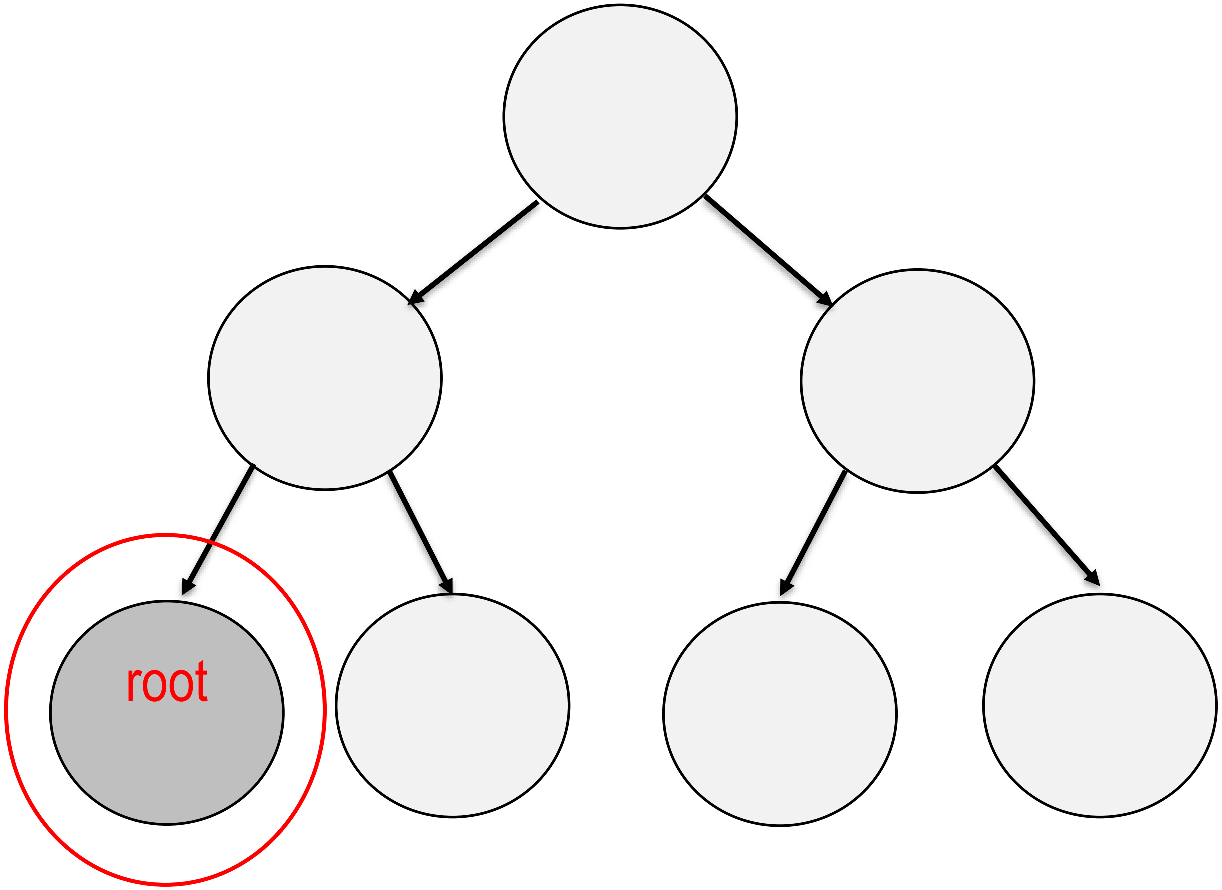
Trees as a Composite Object



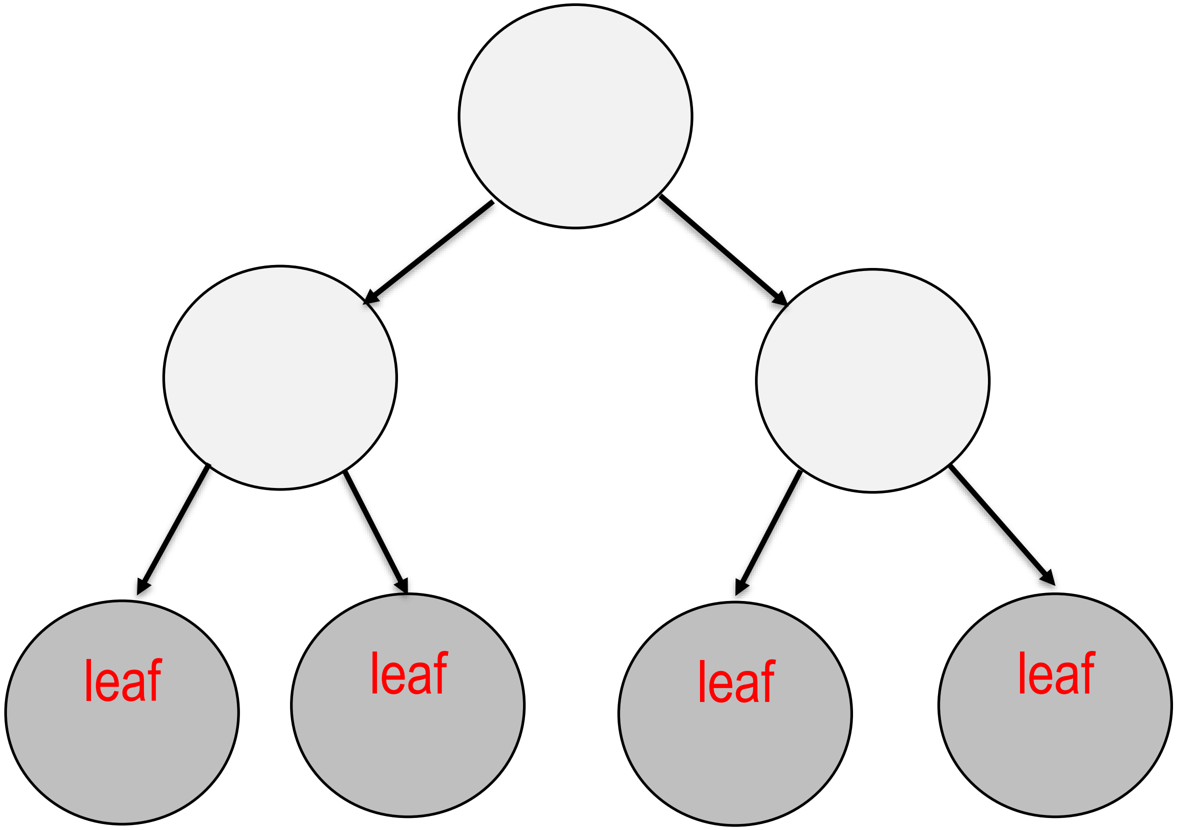
Trees as a Composite Object



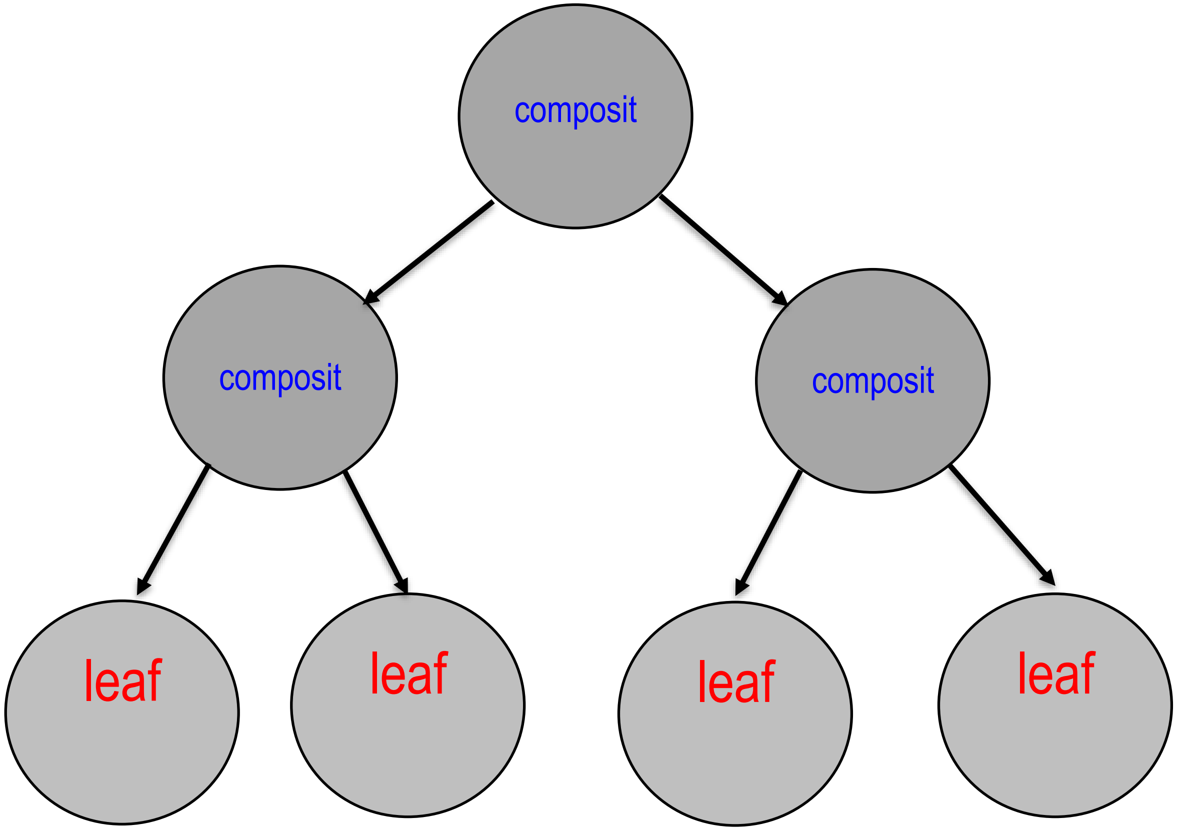
Trees as a Composite Object



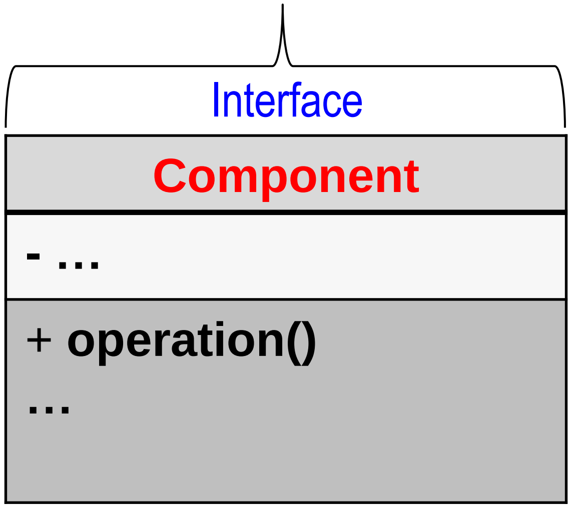
Trees as a Composite Object



Trees as a Composite Object



Composite Pattern



Interface

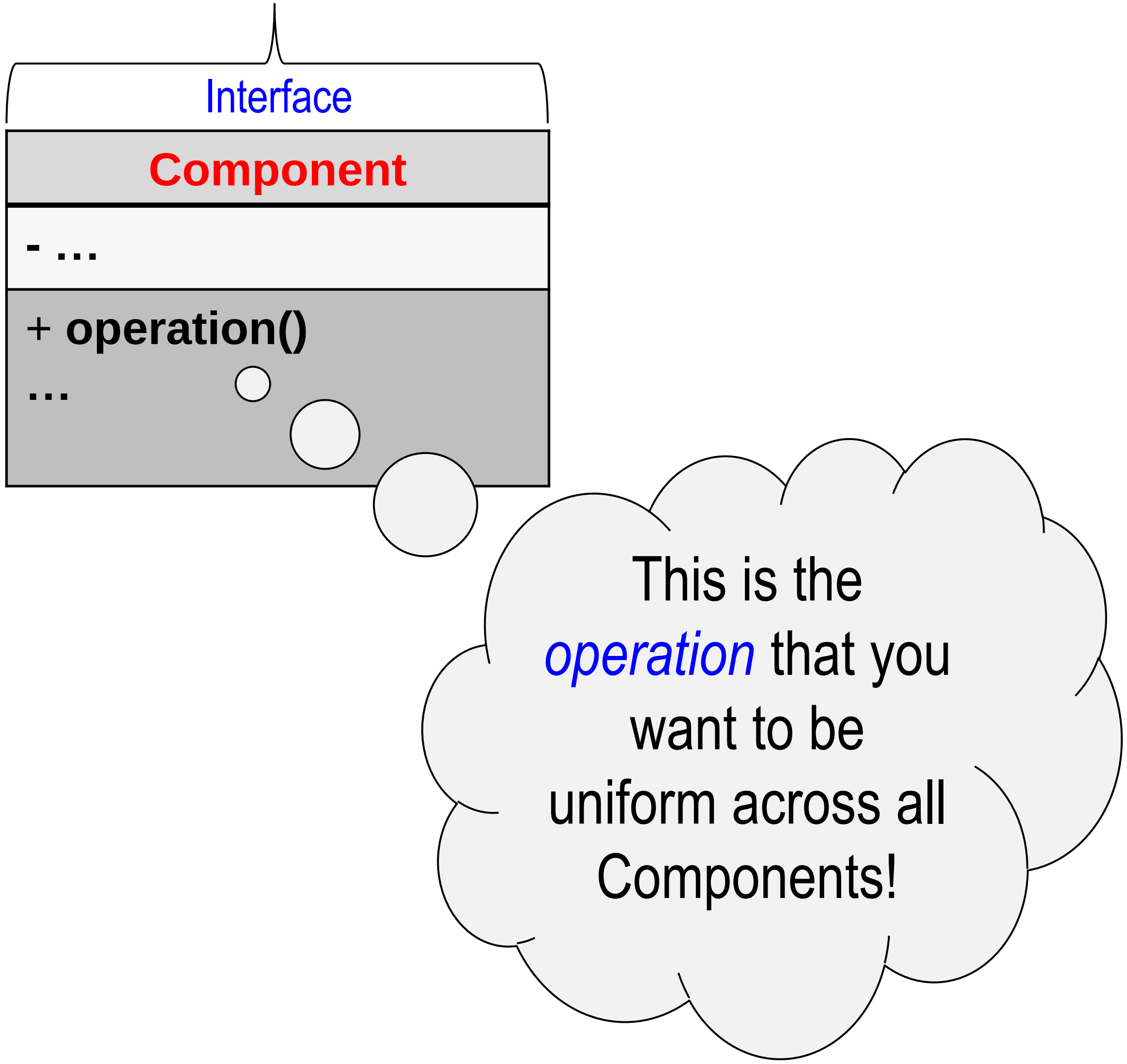
Component

- …

+ **operation()**

…

Composite Pattern



Interface

Component

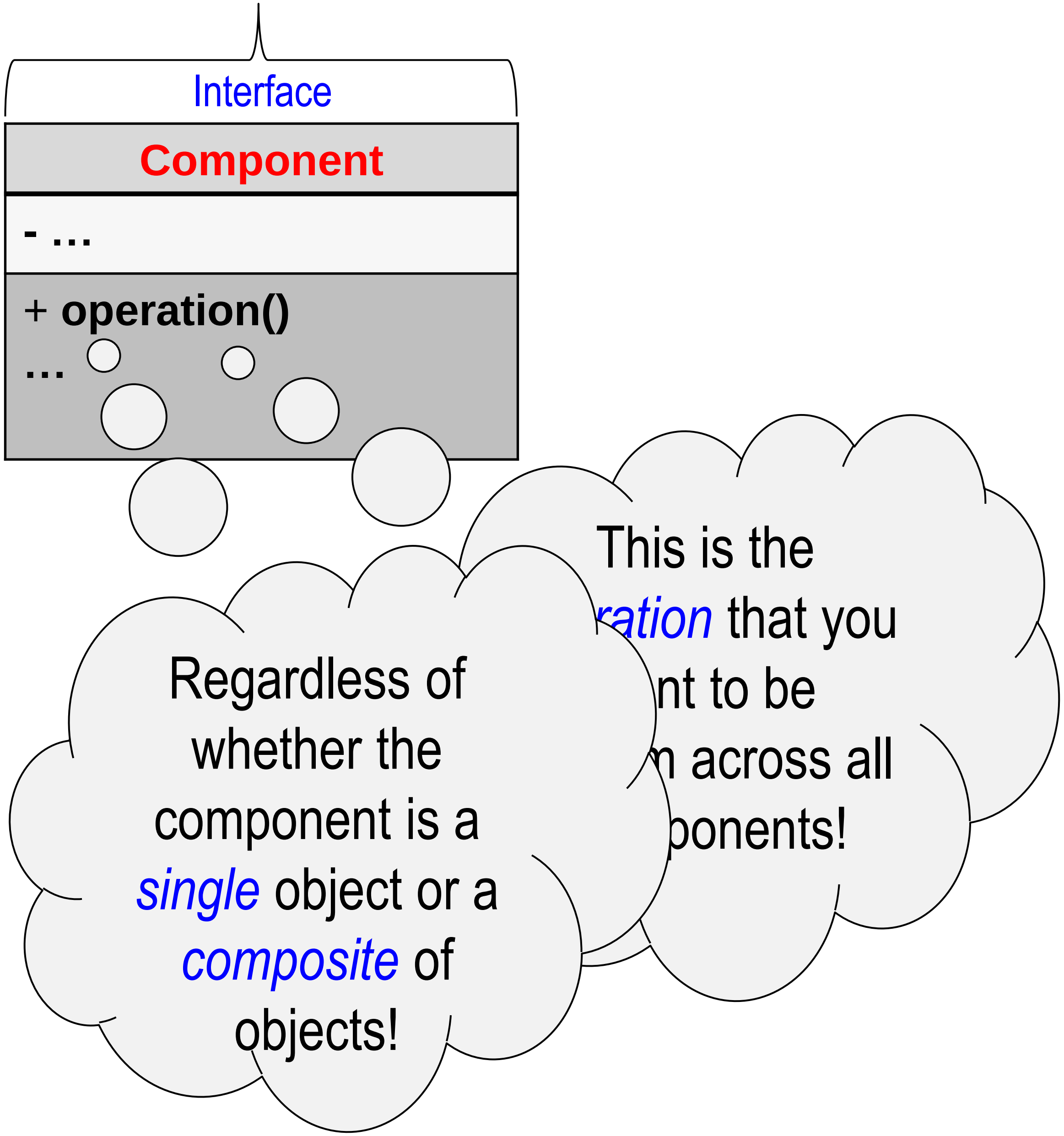
- …

+ **operation()**

…

This is the *operation* that you want to be uniform across all Components!

Composite Pattern



Interface

Component

- …

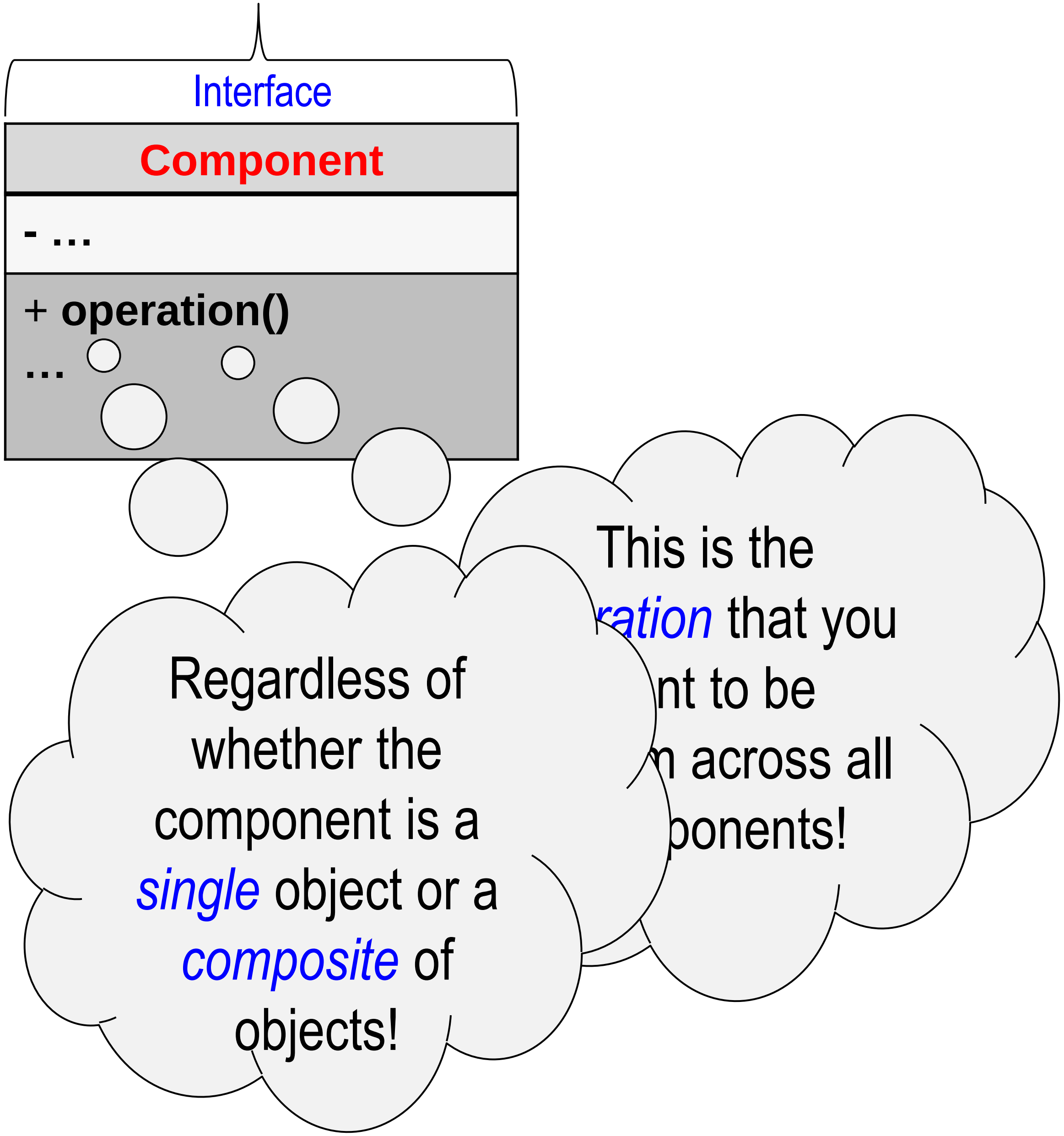
+ **operation()**

…

This is the *ration* that you nt to be n across all ponents!

Regardless of whether the component is a *single* object or a *composite* of objects!

Composite Pattern



Interface

Component

- …

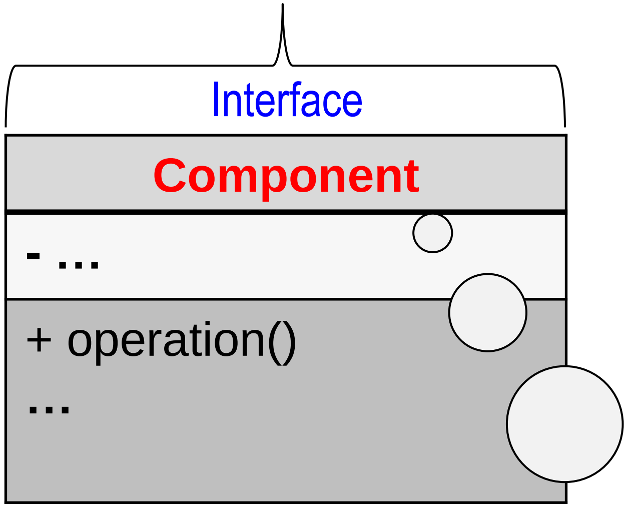
+ **operation()**

…

This is the
ration that you
nt to be
n across all
ponents!

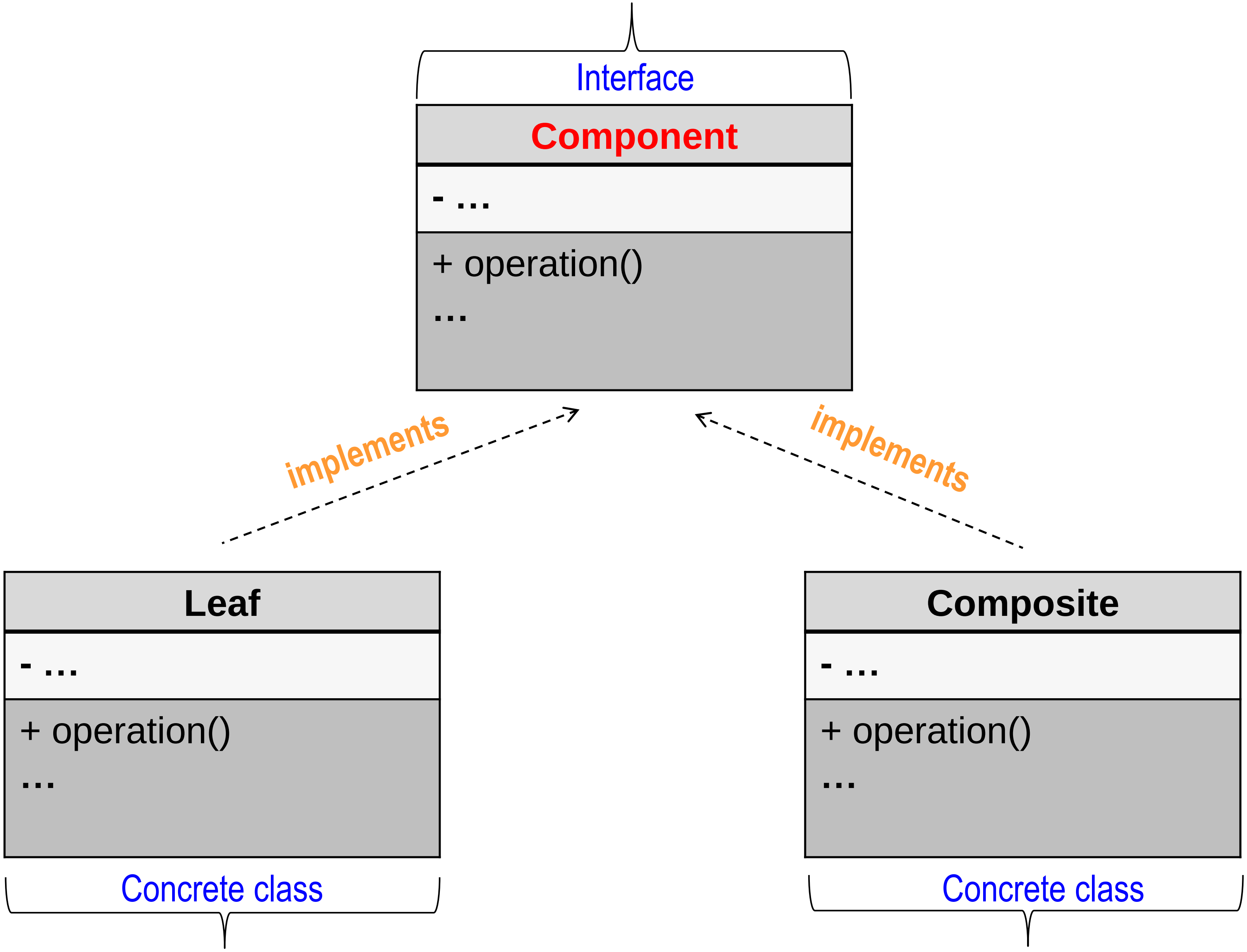
Regardless of
whether the
component is a
single object or a
composite of
objects!

Composite Pattern



The *component* interface will be used to refer uniformly to both individual objects and composites.

Composite Pattern



Interface

Component

- …

+ operation()

…

implements

implements

Leaf

- …

+ operation()

…

Concrete class

Composite

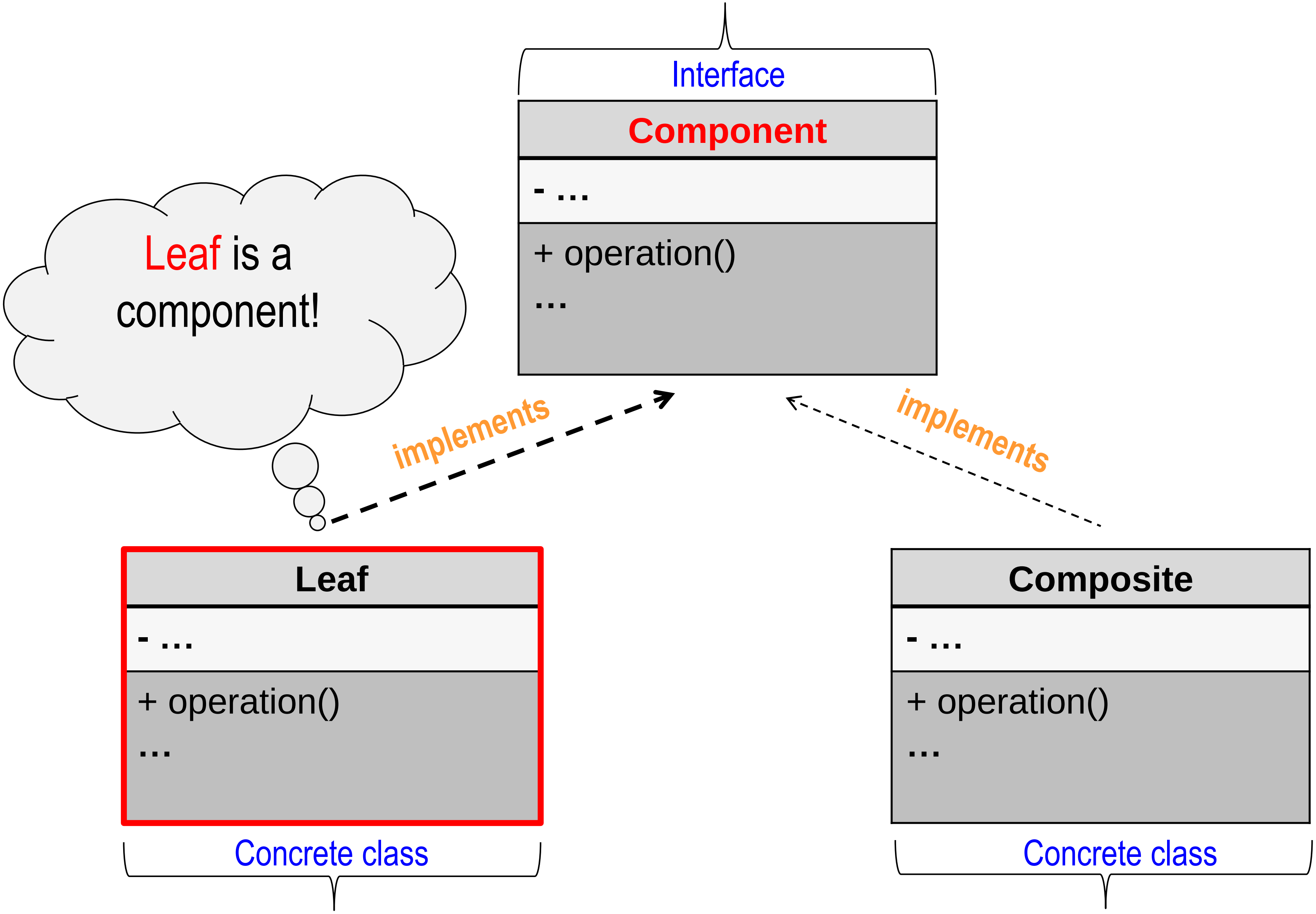
- …

+ operation()

…

Concrete class

Composite Pattern



Interface

Component

- ...

+ operation()

...

Leaf is a component!

implements

implements

Leaf

- ...

+ operation()

...

Concrete class

Composite

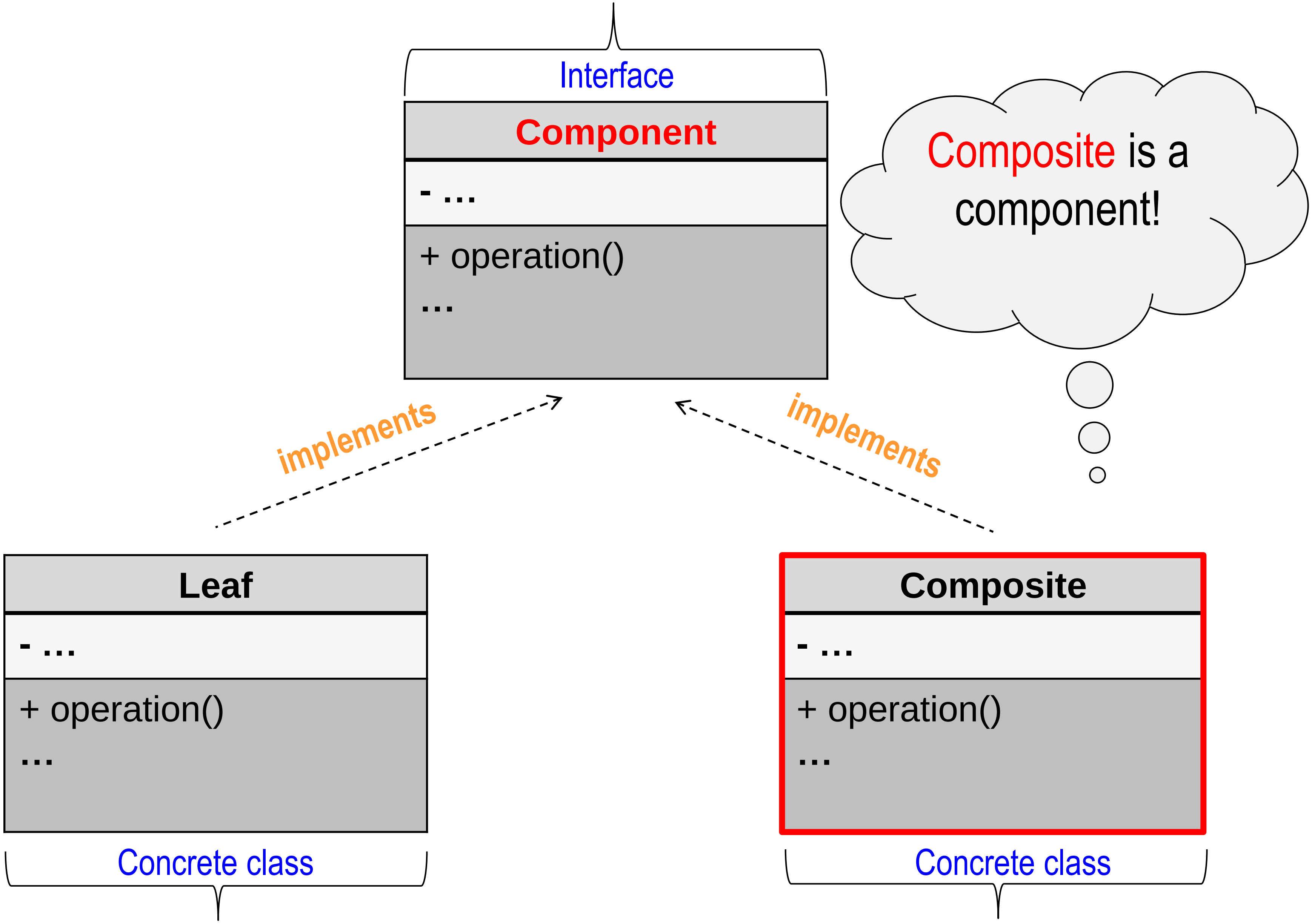
- ...

+ operation()

...

Concrete class

Composite Pattern



Interface

Component

- ...

+ operation()

...

implements

implements

Composite is a component!

Leaf

- ...

+ operation()

...

Concrete class

Composite

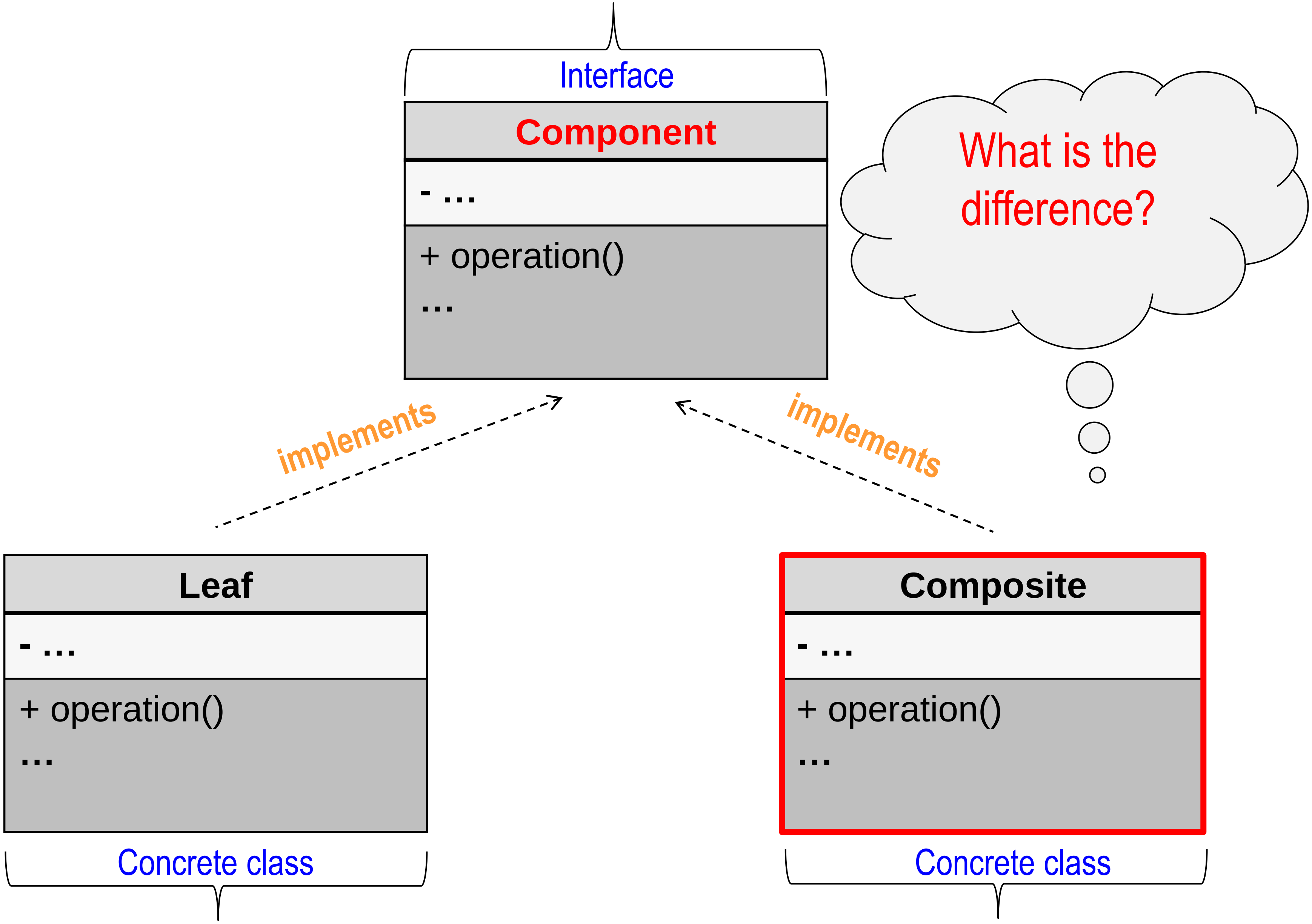
- ...

+ operation()

...

Concrete class

Composite Pattern



Interface

Component

- …

+ operation()

…

implements

implements

What is the difference?

Leaf

- …

+ operation()

…

Concrete class

Composite

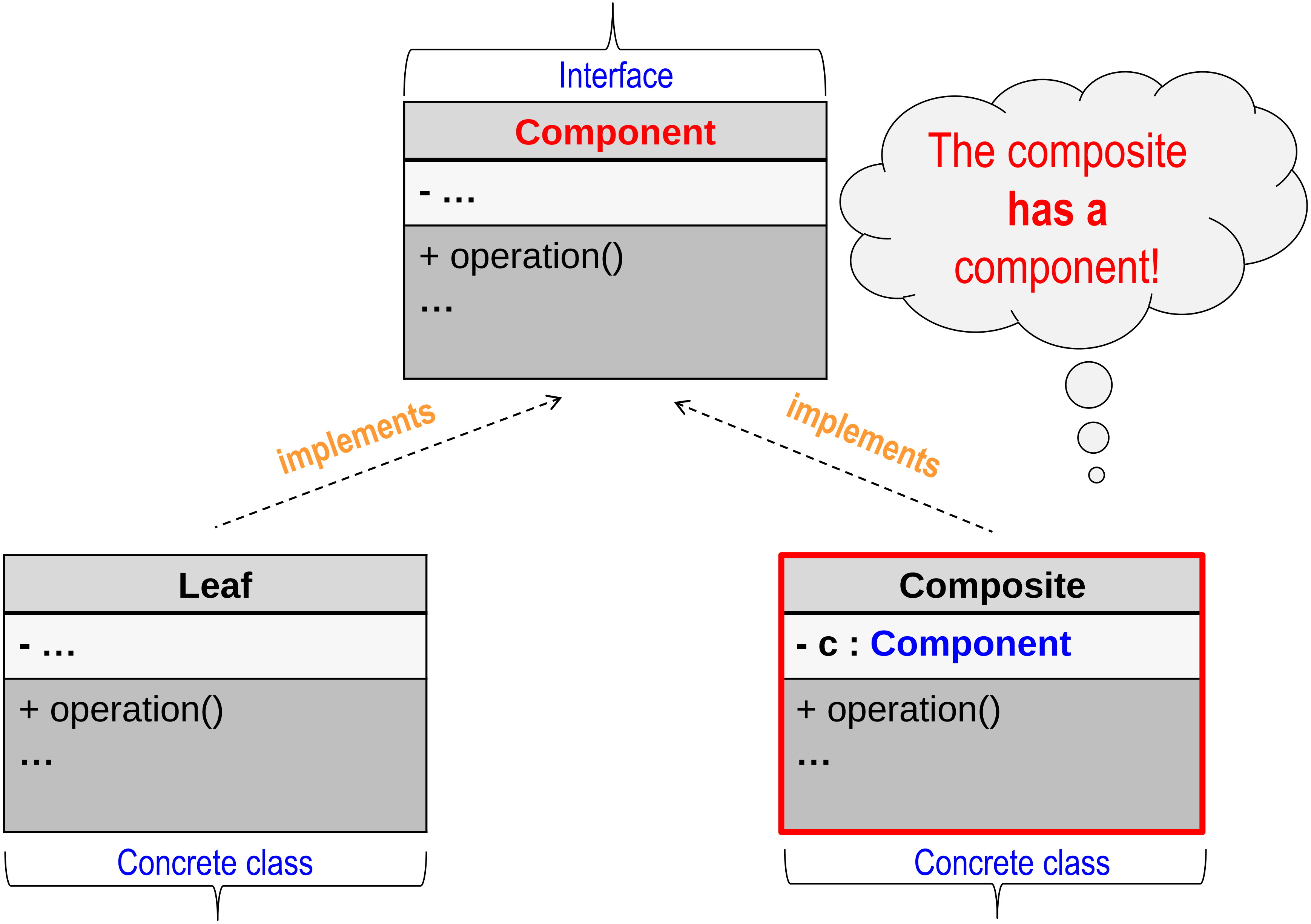
- …

+ operation()

…

Concrete class

Composite Pattern



Interface

Component

- …

+ operation()

…

implements

implements

The composite **has a** component!

Leaf

- …

+ operation()

…

Concrete class

Composite

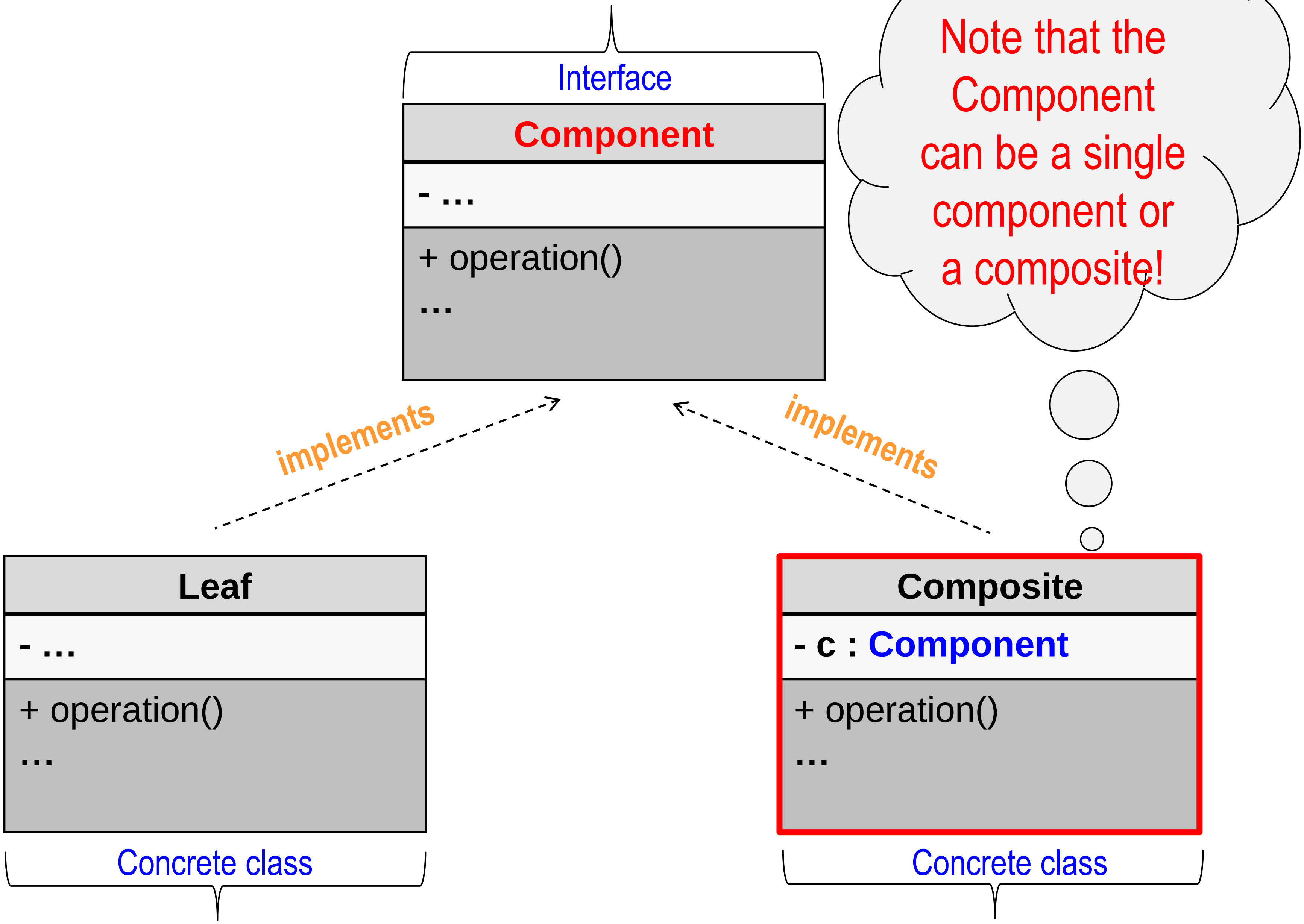
- c : **Component**

+ operation()

…

Concrete class

Composite Pattern



Interface

Component

- …

+ operation()

…

implements

implements

Note that the Component can be a single component or a composite!

Leaf

- …

+ operation()

…

Concrete class

Composite

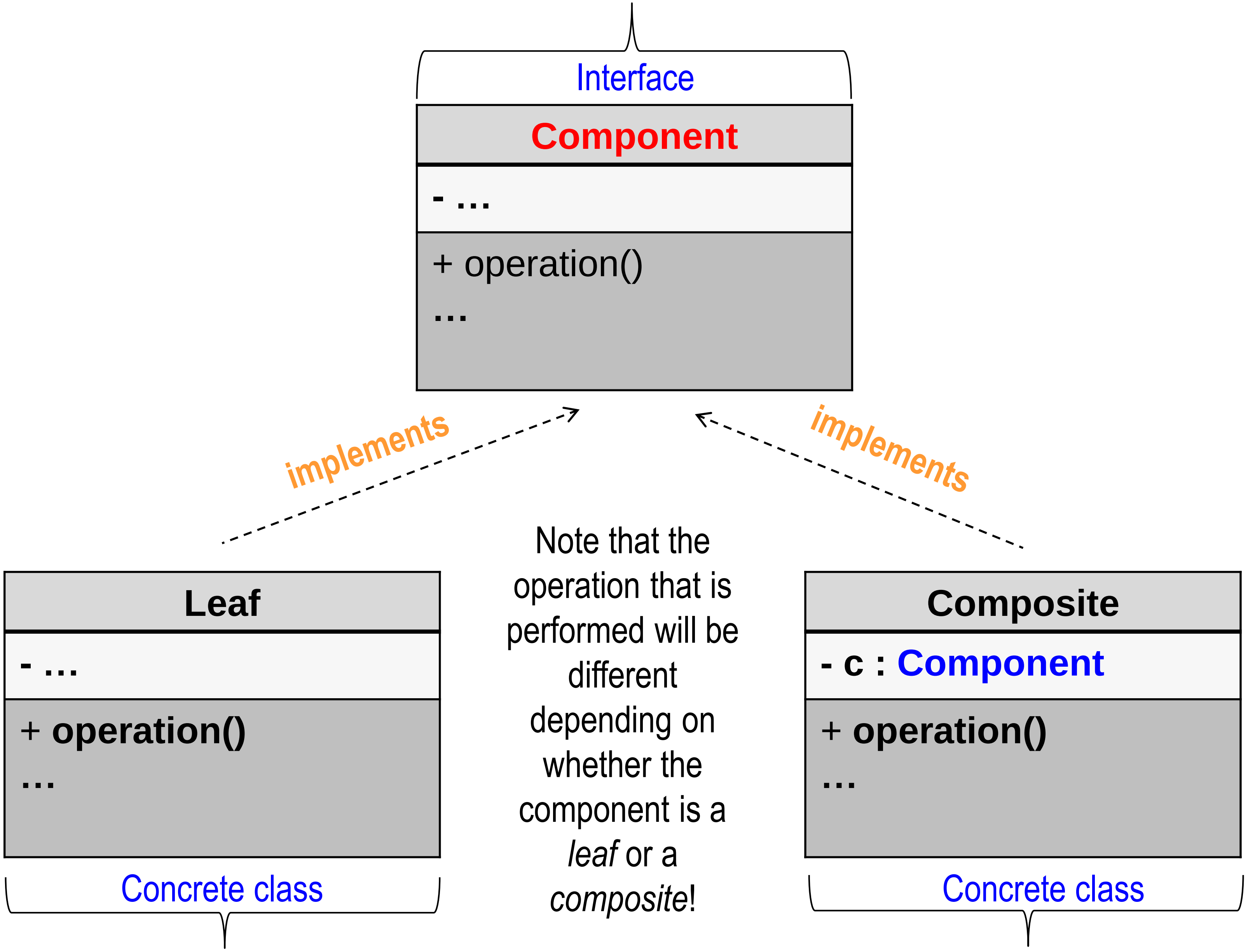
- c : **Component**

+ operation()

…

Concrete class

Composite Pattern



Interface

Component

- …

+ operation()

…

implements

implements

Leaf

- …

+ **operation()**

…

Concrete class

Composite

- c : Component

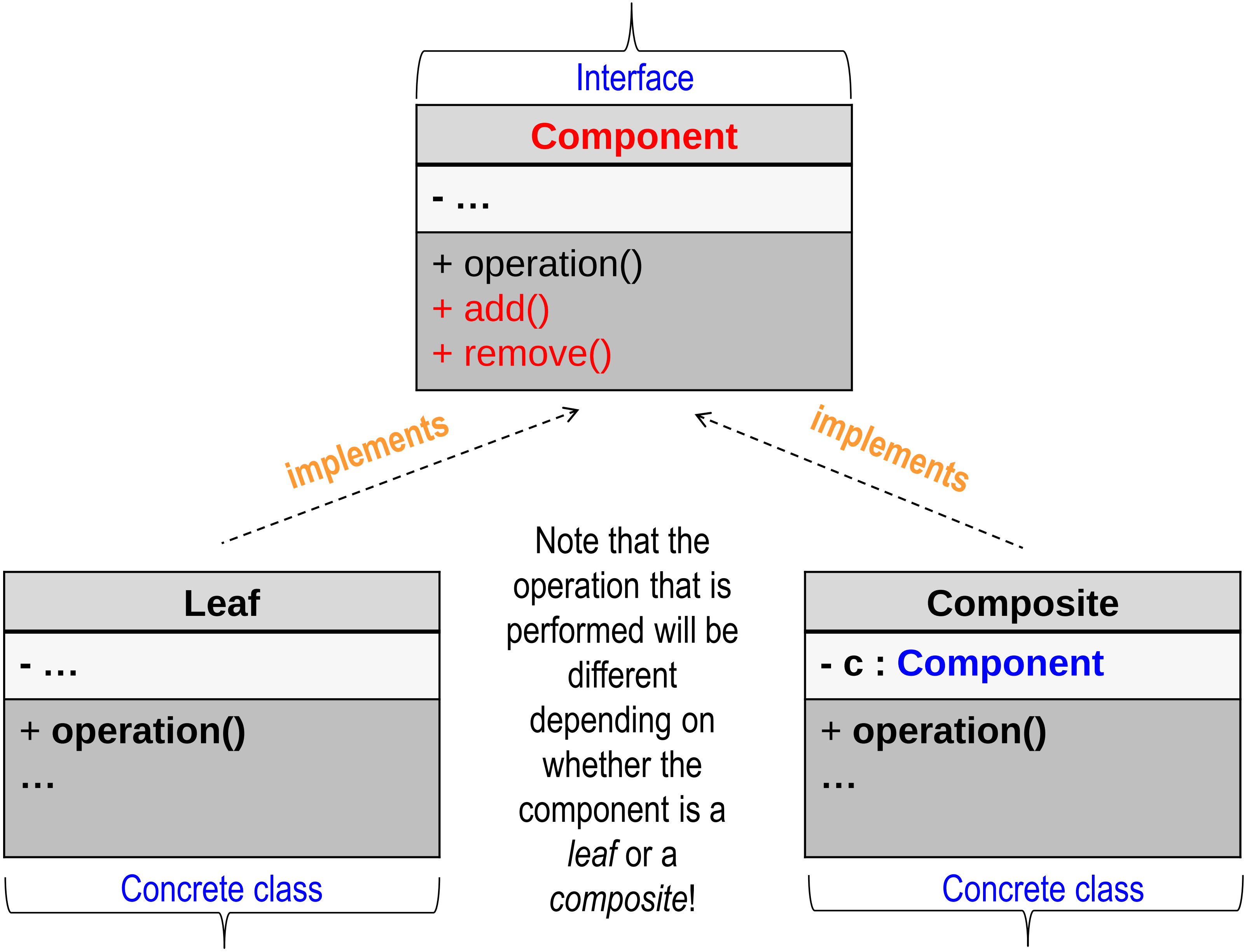
+ **operation()**

…

Concrete class

Note that the operation that is performed will be different depending on whether the component is a *leaf* or a *composite*!

Composite Pattern



Interface

Component

- …

+ operation()
+ add()
+ remove()

implements

implements

Leaf

- …

+ **operation()**

…

Concrete class

Note that the operation that is performed will be different depending on whether the component is a *leaf* or a *composite*!

Composite

- c : Component

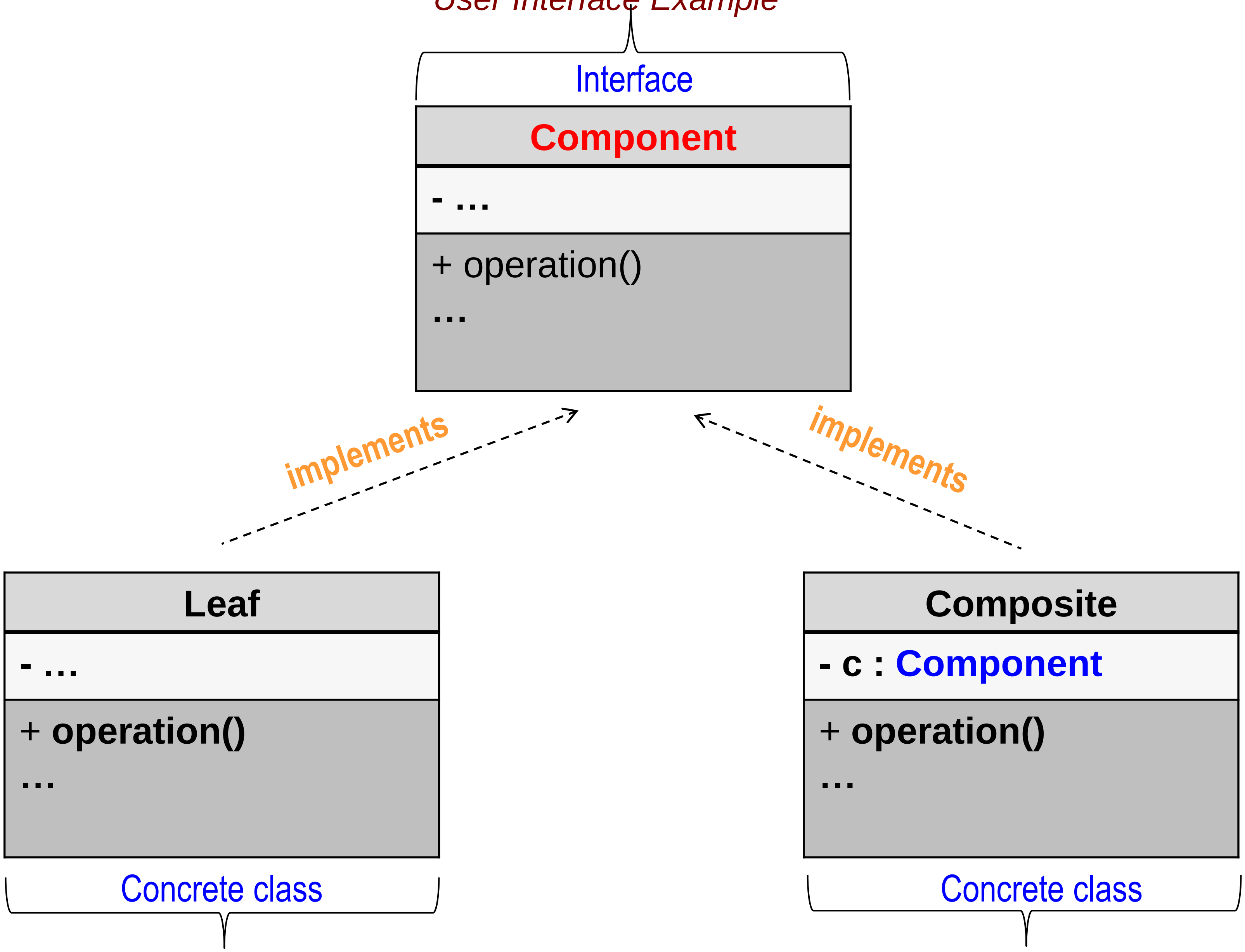
+ **operation()**

…

Concrete class

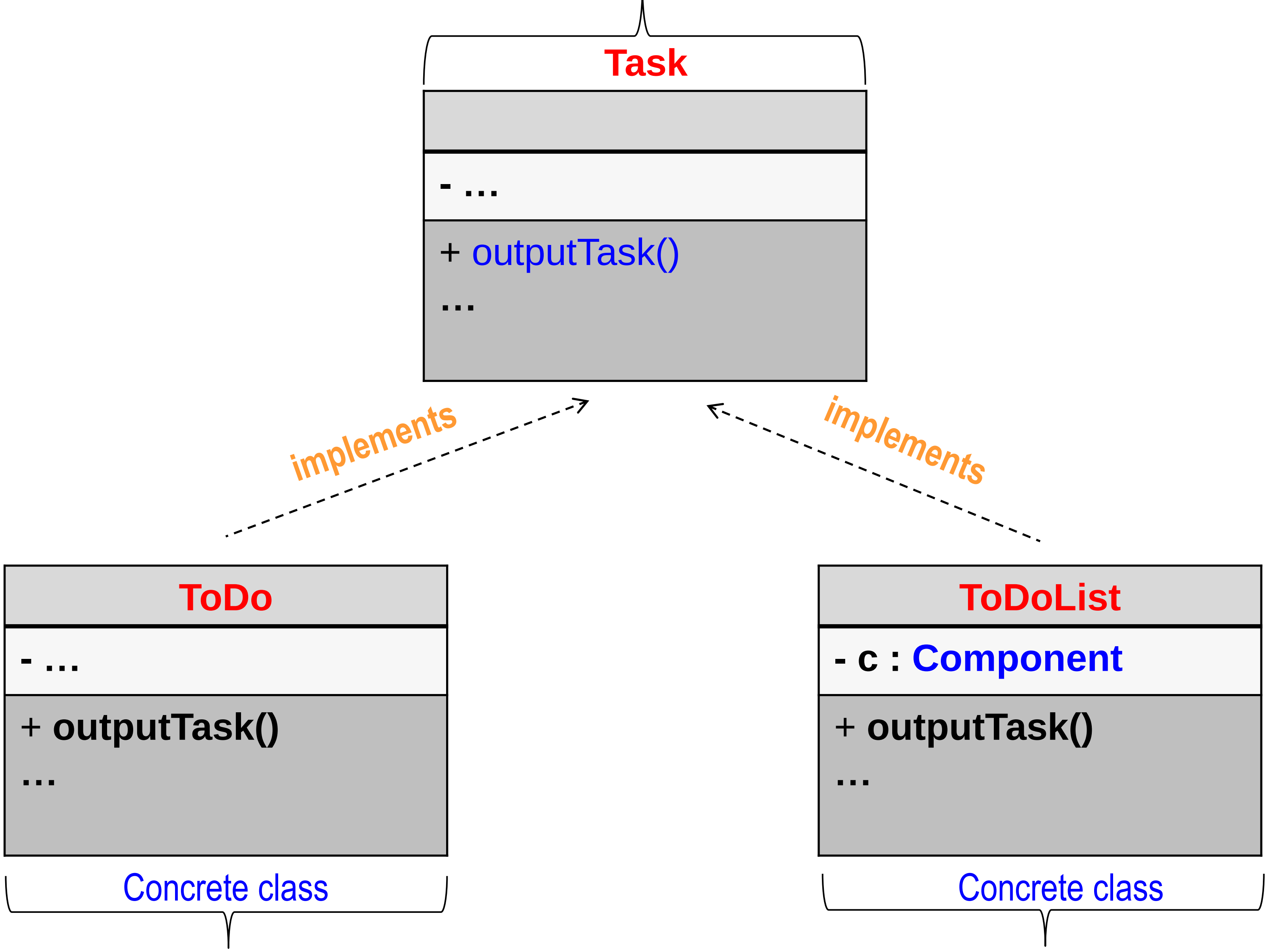
Composite Pattern:

User Interface Example



Composite Pattern:

User Interface Example



Task

- …

+ outputTask()

…

implements

implements

ToDo

- …

+ outputTask()

…

ToDoList

- c : Component

+ outputTask()

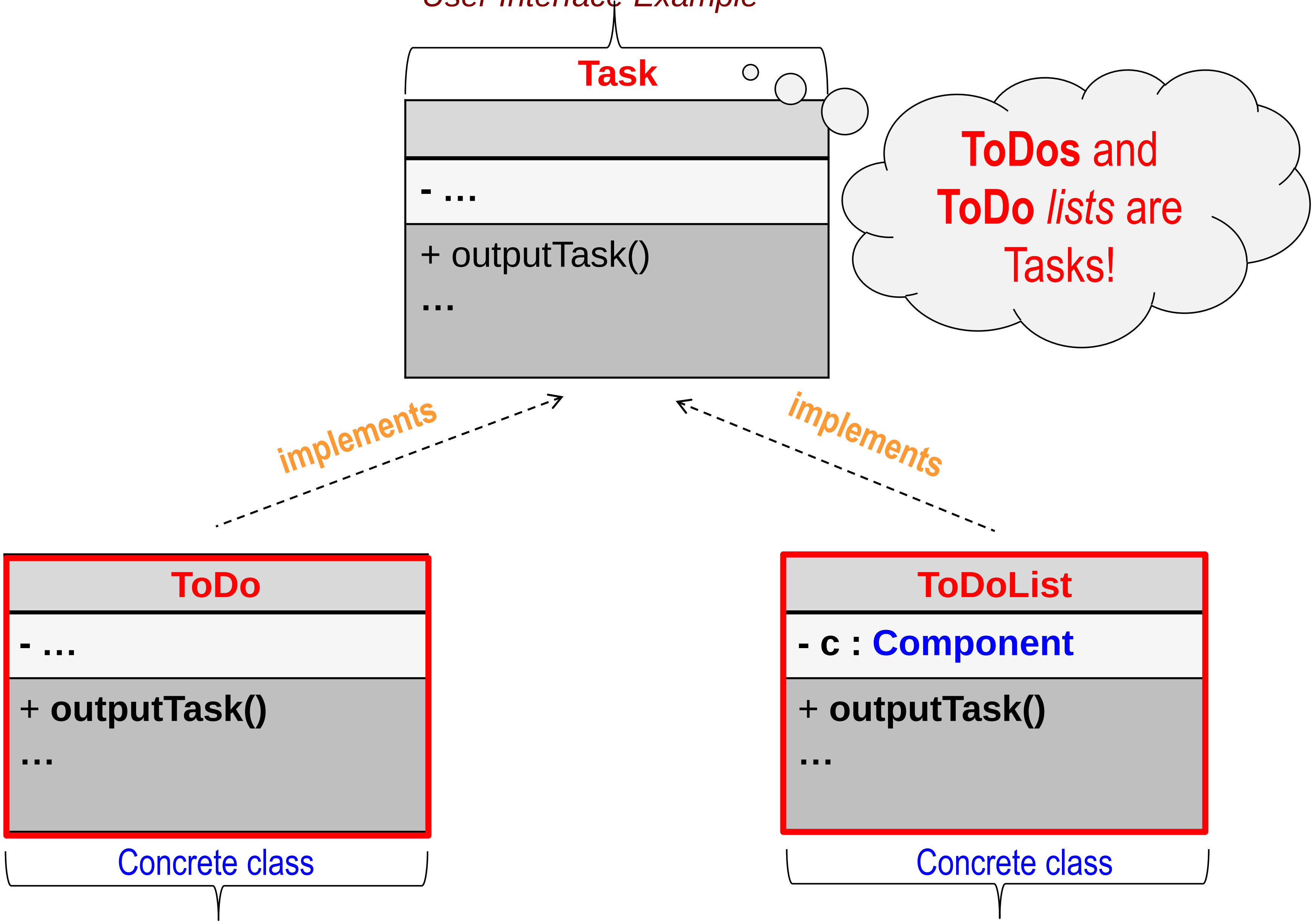
…

Concrete class

Concrete class

Composite Pattern:

User Interface Example



Task

- …

+ outputTask()

…

ToDos and **ToDo** *lists* are Tasks!

implements

implements

ToDo

- …

+ outputTask()

…

Concrete class

ToDoList

- c : Component

+ outputTask()

…

Concrete class

Example:

task of tasks;

```
public Interface Task {
   String getTask();
}
```

```
public lass ToDo implements Task {

   private string toDo;

   public ToDo( String toDo ) {
     this.toDo = toDo;
   }

   public String getTask() {
      return(toDo);
   }
}
```

Example:

task of tasks;

```
public Interface Task {
   String getTask();
}
```

```
public lass ToDo implements Task {

   private string toDo;

   public ToDo( String toDo ) {
     this.toDo = toDo;
   }

   public String getTask() {
      return(toDo);
   }
}
```

Example:

task of tasks;

```
public Interface Task {
   String getTask();
}
```

```
public lass ToDo implements Task {

   private string toDo;

   public ToDo( String toDo ) {
     this.toDo = toDo;
   }

   public String getTask() {
      return(toDo);
   }
}
```

Example:

task of tasks;

```
public Interface Task {
   String getTask();
}
```

```
public lass ToDo implements Task {

   private string toDo;

   public ToDo( String toDo ) {
     this.toDo = toDo;
   }

   public String getTask() {
      return(toDo);
   }
}
```

Example:

task of tasks

```
class ToDoList implements Task {
   private string title;
   private List<Task> toDos;

   public Task( String title, List<Task> todos ) {
     this.title = title;
     this.todos = todos;
   }

   public String getTask() {
     String s = "[" + title;
     for (TodoList td : toDos ) {
        s += td.getTask();

     s += "]";
     return( s );
   }
}
```

Example:

task of tasks

```
class ToDoList implements Task {
   private string title;
   private List<Task> toDos;

   public Task( String title, List<Task> todos ) {
     this.title = title;
     this.todos = todos;
   }

   public String getTask() {
     String s = "[" + title;
     for (TodoList td : toDos ) {
        s += td.getTask();

     s += "]";
     return( s );
   }
}
```

Example:

task of tasks

```
class ToDoList implements Task {
   private string title;
   private List<Task> toDos;

   public Task( String title, List<Task> todos ) {
     this.title = title;
     this.todos = todos;
   }

   public String getTask() {
     String s = "[" + title;
     for (TodoList td : toDos ) {
        s += td.getTask();

     s += "]";
     return( s );
   }
}
```

Example:

task of tasks

```
class ToDoList implements Task {
   private string title;
   private List<Task> toDos;

   public Task( String title, List<Task> todos ) {
     this.title = title;
     this.todos = todos;
   }

   public String getTask() {
     String s = "[" + title;
     for (TodoList td : toDos ) {
        s += td.getTask();

     s += "]";
     return( s );
   }
}
```

Example:

task of tasks

```
class ToDoList implements Task {
   private string title;
   private List<Task> toDos;

   public Task( String title, List<Task> todos ) {
     this.title = title;
     this.todos = todos;
   }

   public String getTask() {
     String s = "[" + title;
     for (TodoList td : toDos ) {
        s += td.getTask();

     s += "]";
     return( s );
   }
}
```

Example:

task of tasks

```
public class TaskManager {

   public static void main( String[] s ) {
       List<Task> tasks = new ArrayList<Task>();

       // Cleaning the floor
       Task vacuum = new ToDo( "vacuum" );
       Task sweep = new ToDo( "sweep" );
       tasks.add(vacuum); tasks.add(sweep);
       Task cleanFloor = new TodoList( "clean the floor", tasks );
       // Cleaning the furniture
       Task dust = new ToDo( "dust" );
       Task polish = new Todo( "polish" );
       tasks = new ArrayList<Task>();
       tasks.add(dust); tasks.add(polish);
       Task furniture = new ToDoLost( "clean the furniture", tasks );
       // Cleaning the house
       tasks = new ArrayList<Task>();
       tasks.add(cleanFloor); tasks.add(furniture);
       Task cleanHouse = new ToDoList( "clean the house", tasks );
       cleanHouse.getTask();
   }
}
```

Example:

task of tasks

```
public class TaskManager {

   public static void main( String[] s ) {
       List<Task> tasks = new ArrayList<Task>();

       // Cleaning the floor
       Task vacuum = new ToDo( "vacuum" );
       Task sweep = new ToDo( "sweep" );
       tasks.add(vacuum); tasks.add(sweep);
       Task cleanFloor = new TodoList( "clean the floor", tasks );
       // Cleaning the furniture
       Task dust = new ToDo( "dust" );
       Task polish = new Todo( "polish" );
       tasks = new ArrayList<Task>();
       tasks.add(dust); tasks.add(polish);
       Task furniture = new ToDoLost( "clean the furniture", tasks );
       // Cleaning the house
       tasks = new ArrayList<Task>();
       tasks.add(cleanFloor); tasks.add(furniture);
       Task cleanHouse = new ToDoList( "clean the house", tasks );
       cleanHouse.getTask();
   }
}
```

Example:

task of tasks

```
public class TaskManager {

   public static void main( String[] s ) {
       List<Task> tasks = new ArrayList<Task>();

       // Cleaning the floor
       Task vacuum = new ToDo( "vacuum" );
       Task sweep = new ToDo( "sweep" );
       tasks.add(vacuum); tasks.add(sweep);
       Task cleanFloor = new TodoList( "clean the floor", tasks );
       // Cleaning the furniture
       Task dust = new ToDo( "dust" );
       Task polish = new Todo( "polish" );
       tasks = new ArrayList<Task>();
       tasks.add(dust); tasks.add(polish);
       Task furniture = new ToDoLost( "clean the furniture", tasks );
       // Cleaning the house
       tasks = new ArrayList<Task>();
       tasks.add(cleanFloor); tasks.add(furniture);
       Task cleanHouse = new ToDoList( "clean the house", tasks );
       cleanHouse.getTask();
   }
}
```

Example:

task of tasks

```
public class TaskManager {

   public static void main( String[] s ) {
       List<Task> tasks = new ArrayList<Task>();

       // Cleaning the floor
       Task vacuum = new ToDo( "vacuum" );
       Task sweep = new ToDo( "sweep" );
       tasks.add(vacuum); tasks.add(sweep);
       Task cleanFloor = new TodoList( "clean the floor", tasks );
       // Cleaning the furniture
       Task dust = new ToDo( "dust" );
       Task polish = new Todo( "polish" );
       tasks = new ArrayList<Task>();
       tasks.add(dust); tasks.add(polish);
       Task furniture = new ToDoLost( "clean the furniture", tasks );
       // Cleaning the house
       tasks = new ArrayList<Task>();
       tasks.add(cleanFloor); tasks.add(furniture);
       Task cleanHouse = new ToDoList( "clean the house", tasks );
       cleanHouse.getTask();
   }
}
```

Example:

task of tasks

```
public class TaskManager {

   public static void main( String[] s ) {
       List<Task> tasks = new ArrayList<Task>();

       // Cleaning the floor
       Task vacuum = new ToDo( "vacuum" );
       Task sweep = new ToDo( "sweep" );
       tasks.add(vacuum); tasks.add(sweep);
       Task cleanFloor = new TodoList( "clean the floor", tasks );
       // Cleaning the furniture
       Task dust = new ToDo( "dust" );
       Task polish = new Todo( "polish" );
       tasks = new ArrayList<Task>();
       tasks.add(dust); tasks.add(polish);
       Task furniture = new ToDoLost( "clean the furniture", tasks );
       // Cleaning the house
       tasks = new ArrayList<Task>();
       tasks.add(cleanFloor); tasks.add(furniture);
       Task cleanHouse = new ToDoList( "clean the house", tasks );
       cleanHouse.getTask();
   }
}
```

Example:

task of tasks

```
public class TaskManager {

   public static void main( String[] s ) {
       List<Task> tasks = new ArrayList<Task>();

       // Cleaning the floor
       Task vacuum = new ToDo( "vacuum" );
       Task sweep = new ToDo( "sweep" );
       tasks.add(vacuum); tasks.add(sweep);
       Task cleanFloor = new TodoList( "clean the floor", tasks );
       // Cleaning the furniture
       Task dust = new ToDo( "dust" );
       Task polish = new Todo( "polish" );
       tasks = new ArrayList<Task>();
       tasks.add(dust); tasks.add(polish);
       Task furniture = new ToDoLost( "clean the furniture", tasks );
       // Cleaning the house
       tasks = new ArrayList<Task>();
       tasks.add(cleanFloor); tasks.add(furniture);
       Task cleanHouse = new ToDoList( "clean the house", tasks );
       cleanHouse.getTask();
   }
}
```

Example:

task of tasks

```
public class TaskManager {

   public static void main( String[] s ) {
       List<Task> tasks = new ArrayList<Task>();

       // Cleaning the floor
       Task vacuum = new ToDo( "vacuum" );
       Task sweep = new ToDo( "sweep" );
       tasks.add(vacuum); tasks.add(sweep);
       Task cleanFloor = new TodoList( "clean the floor", tasks );
       // Cleaning the furniture
       Task dust = new ToDo( "dust" );
       Task polish = new Todo( "polish" );
       tasks = new ArrayList<Task>();
       tasks.add(dust); tasks.add(polish);
       Task furniture = new ToDoLost( "clean the furniture", tasks );
       // Cleaning the house
       tasks = new ArrayList<Task>();
       tasks.add(cleanFloor); tasks.add(furniture);
       Task cleanHouse = new ToDoList( "clean the house", tasks );
       cleanHouse.getTask();
   }
}
```

Example:

task of tasks

```
public class TaskManager {

   public static void main( String[] s ) {
       List<Task> tasks = new ArrayList<Task>();

       // Cleaning the floor
       Task vacuum = new ToDo( "vacuum" );
       Task sweep = new ToDo( "sweep" );
       tasks.add(vacuum); tasks.add(sweep);
       Task cleanFloor = new TodoList( "clean the floor", tasks );
       // Cleaning the furniture
       Task dust = new ToDo( "dust" );
       Task polish = new Todo( "polish" );
       tasks = new ArrayList<Task>();
       tasks.add(dust); tasks.add(polish);
       Task furniture = new ToDoLost( "clean the furniture", tasks );
       // Cleaning the house
       tasks = new ArrayList<Task>();
       tasks.add(cleanFloor); tasks.add(furniture);
       Task cleanHouse = new ToDoList( "clean the house", tasks );
       cleanHouse.getTask();
   }
}
```

Example:

task of tasks

```
public class TaskManager {

   public static void main( String[] s ) {
       List<Task> tasks = new ArrayList<Task>();

       // Cleaning the floor
       Task vacuum = new ToDo( "vacuum" );
       Task sweep = new ToDo( "sweep" );
       tasks.add(vacuum); tasks.add(sweep);
       Task cleanFloor = new TodoList( "clean the floor", tasks );
       // Cleaning the furniture
       Task dust = new ToDo( "dust" );
       Task polish = new Todo( "polish" );
       tasks = new ArrayList<Task>();
       tasks.add(dust); tasks.add(polish);
       Task furniture = new ToDoLost( "clean the furniture", tasks );
       // Cleaning the house
       tasks = new ArrayList<Task>();
       tasks.add(cleanFloor); tasks.add(furniture);
       Task cleanHouse = new ToDoList( "clean the house", tasks );
       cleanHouse.getTask();
   }
}
```

Example:

task of tasks

```
public class TaskManager {

   public static void main( String[] s ) {
       List<Task> tasks = new ArrayList<Task>();

       // Cleaning the floor
       Task vacuum = new ToDo( "vacuum" );
       Task sweep = new ToDo( "sweep" );
       tasks.add(vacuum); tasks.add(sweep);
       Task cleanFloor = new TodoList( "clean the floor", tasks );
       // Cleaning the furniture
       Task dust = new ToDo( "dust" );
       Task polish = new Todo( "polish" );
       tasks = new ArrayList<Task>();
       tasks.add(dust); tasks.add(polish);
       Task furniture = new ToDoLost( "clean the furniture", tasks );
       // Cleaning the house
       tasks = new ArrayList<Task>();
       tasks.add(cleanFloor); tasks.add(furniture);
       Task cleanHouse = new ToDoList( "clean the house", tasks );
       cleanHouse.getTask();
   }
}
```

Example:

task of tasks

```
public class TaskManager {

   public static void main( String[] s ) {
       List<Task> tasks = new ArrayList<Task>();

       // Cleaning the floor
       Task vacuum = new ToDo( "vacuum" );
       Task sweep = new ToDo( "sweep" );
       tasks.add(vacuum); tasks.add(sweep);
       Task cleanFloor = new TodoList( "clean the floor", tasks );
       // Cleaning the furniture
       Task dust = new ToDo( "dust" );
       Task polish = new Todo( "polish" );
       tasks = new ArrayList<Task>();
       tasks.add(dust); tasks.add(polish);
       Task furniture = new ToDoLost( "clean the furniture", tasks );
       // Cleaning the house
       tasks = new ArrayList<Task>();
       tasks.add(cleanFloor); tasks.add(furniture);
       Task cleanHouse = new ToDoList( "clean the house", tasks );
       cleanHouse.getTask();
   }
}
```

Example:

task of tasks

```
public class TaskManager {

   public static void main( String[] s ) {
       List<Task> tasks = new ArrayList<Task>();

       // Cleaning the floor
       Task vacuum = new ToDo( "vacuum" );
       Task sweep = new ToDo( "sweep" );
       tasks.add(vacuum); tasks.add(sweep);
       Task cleanFloor = new TodoList( "clean the floor", tasks );
       // Cleaning the furniture
       Task dust = new ToDo( "dust" );
       Task polish = new Todo( "polish" );
       tasks = new ArrayList<Task>();
       tasks.add(dust); tasks.add(polish);
       Task furniture = new ToDoLost( "clean the furniture", tasks );
       // Cleaning the house
       tasks = new ArrayList<Task>();
       tasks.add(cleanFloor); tasks.add(furniture);
       Task cleanHouse = new ToDoList( "clean the house", tasks );
       cleanHouse.getTask();
   }
}
```

Example:

task of tasks

```
public class TaskManager {

   public static void main( String[] s ) {
       List<Task> tasks = new ArrayList<Task>();

       // Cleaning the floor
       Task vacuum = new ToDo( "vacuum" );
       Task sweep = new ToDo( "sweep" );
       tasks.add(vacuum); tasks.add(sweep);
       Task cleanFloor = new TodoList( "clean the floor", tasks );
       // Cleaning the furniture
       Task dust = new ToDo( "dust" );
       Task polish = new Todo( "polish" );
       tasks = new ArrayList<Task>();
       tasks.add(dust); tasks.add(polish);
       Task furniture = new ToDoLost( "clean the furniture", tasks );
       // Cleaning the house
       tasks = new ArrayList<Task>();
       tasks.add(cleanFloor); tasks.add(furniture);
       Task cleanHouse = new ToDoList( "clean the house", tasks );
       cleanHouse.getTask();
   }
}
```

Elements of Java GUI

- The key elements of Java's graphical user interface are:
  - GUI components
  - Layout managers, and other *helper* classes
  - Event processing

- The GUI components are all the screen elements that a user manipulates with the mouse and keyboard, such text fields, buttons, check boxes, etc.

- The use of Layout managers is how Java governs how the components appear on the screen.

- Event processing is how user actions, like a mouse click, or the <enter> button are handled by the system.

Elements of Java GUI

- The key elements of Java’s graphical user interface are:
  - GUI components
  - Layout managers, and other *helper* classes
  - Event processing

- The GUI components are all the screen elements that a user manipulates with the mouse and keyboard, such text fields, buttons, check boxes, etc.

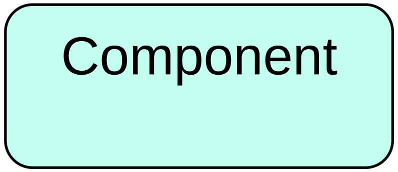
- The use of Layout managers is how Java governs how the components appear on the screen.

- Event processing is how user actions, like a mouse click, or the <enter> button are handled by the system.

Java GUI structure

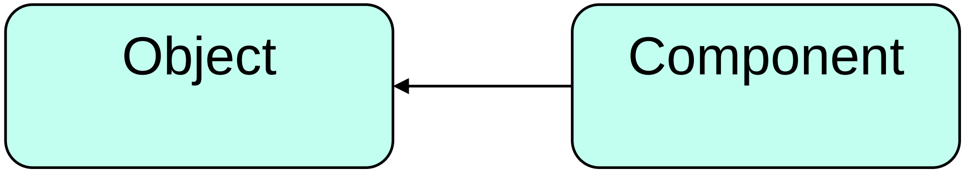
- Java’s GUI classes fall into three categories:
    - Container Classes
    - Helper Classes
    - Component Classes

Java GUI Class Hierarchy

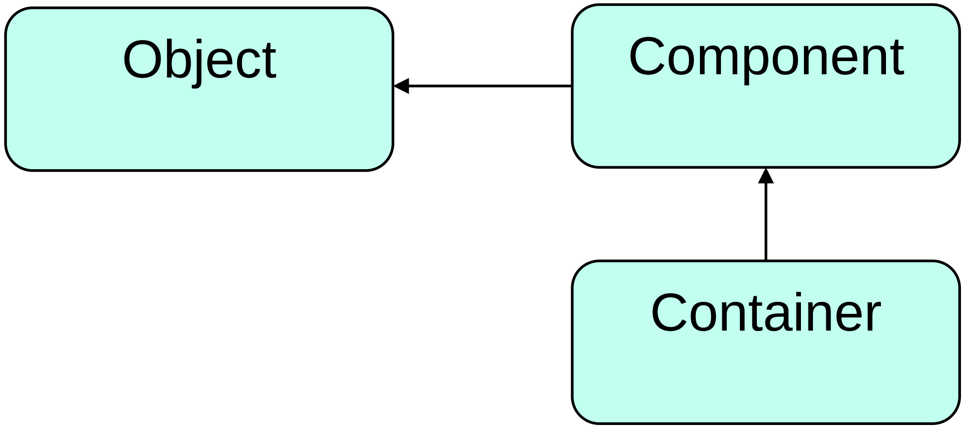


Component

Java GUI Class Hierarchy



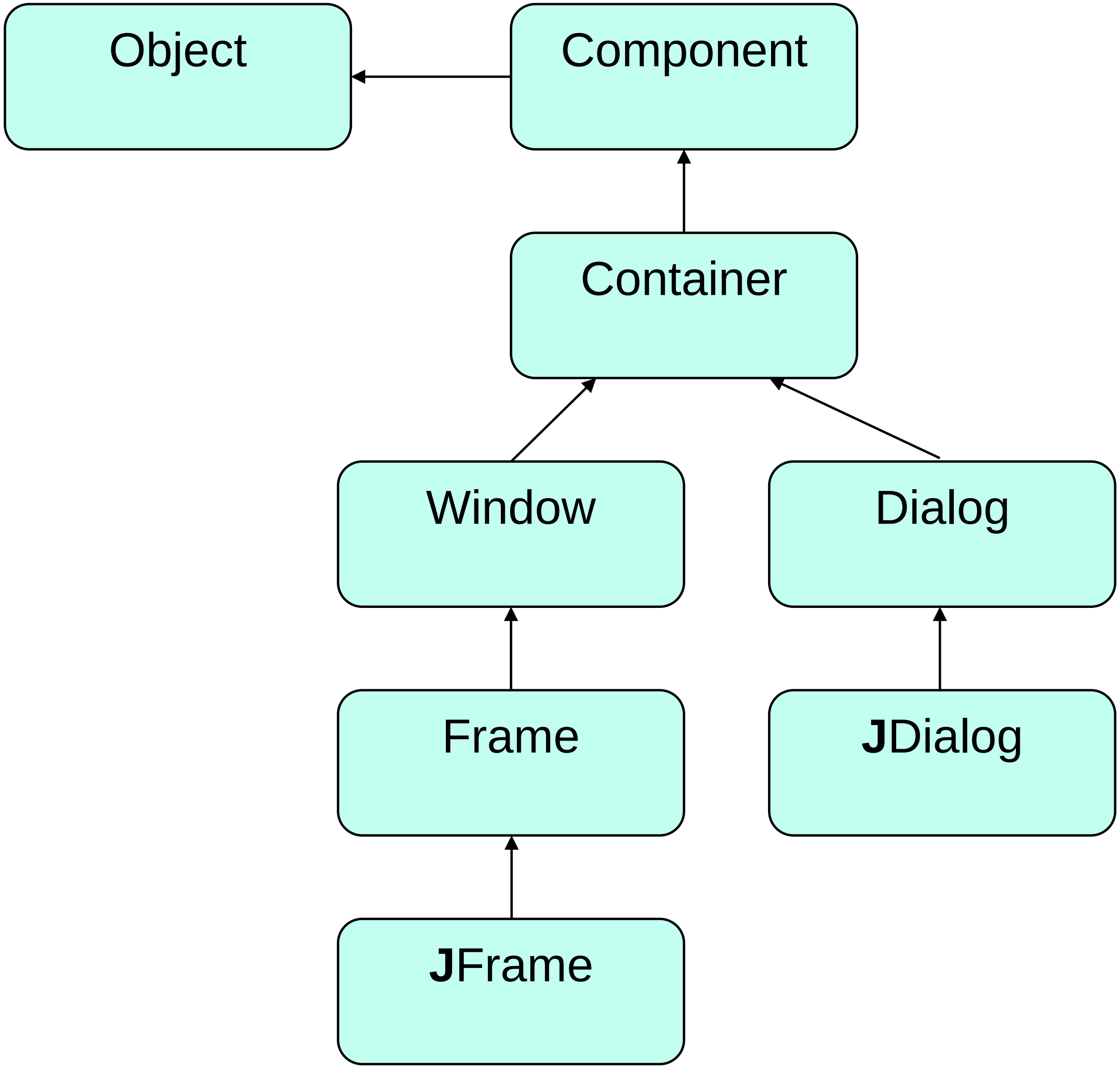
Java GUI Class Hierarchy


Object

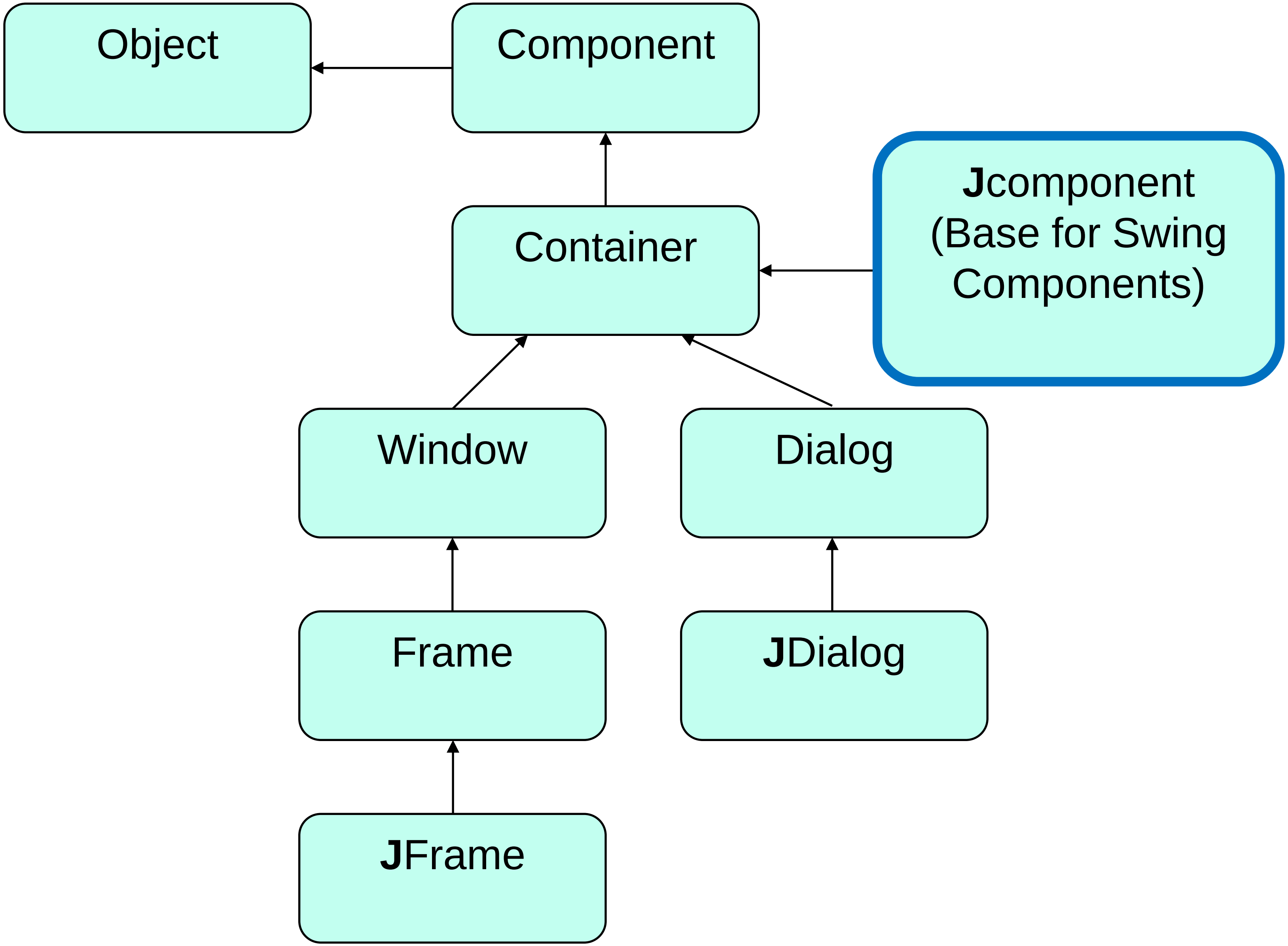
Component

Container

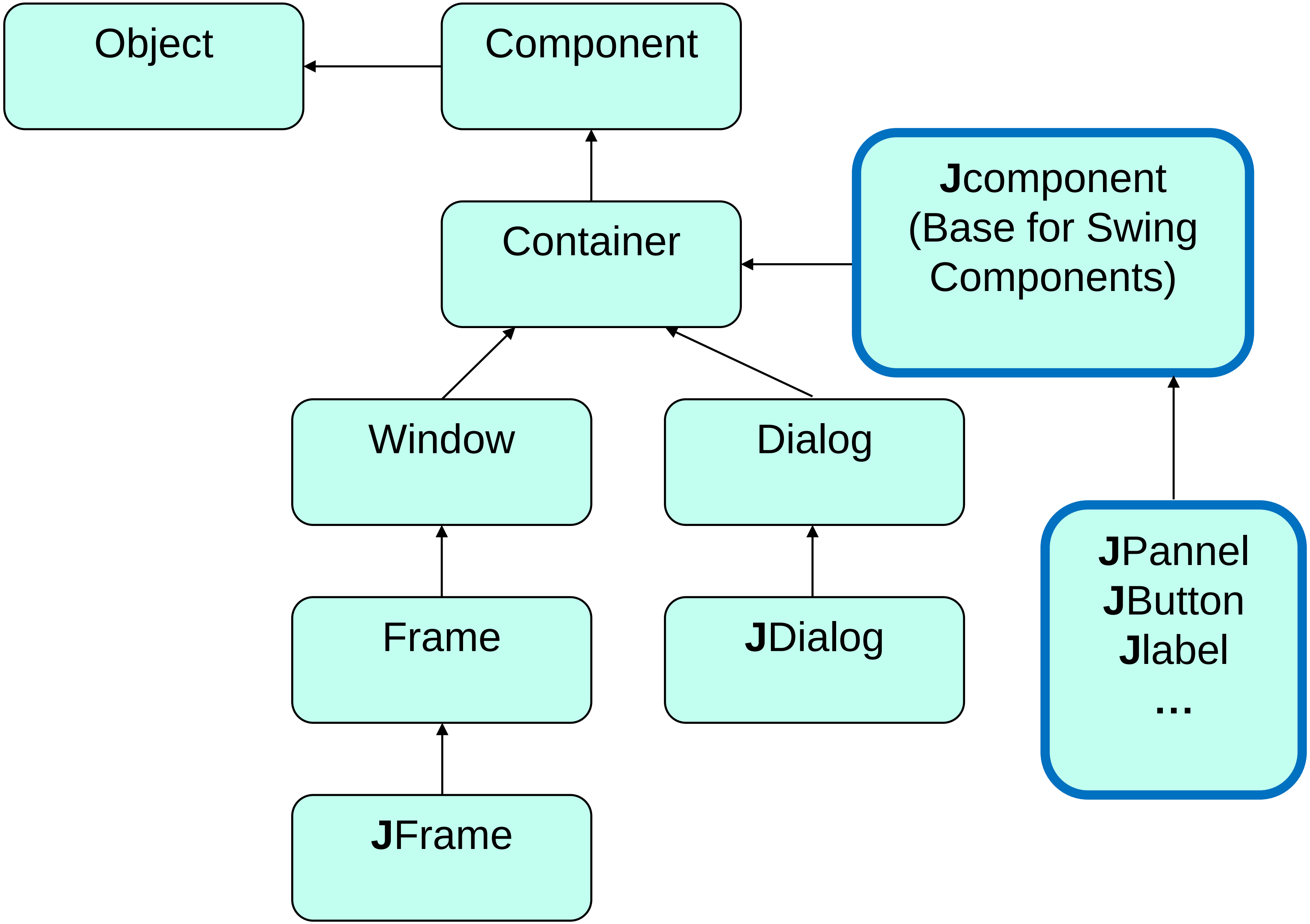
Java GUI Class Hierarchy



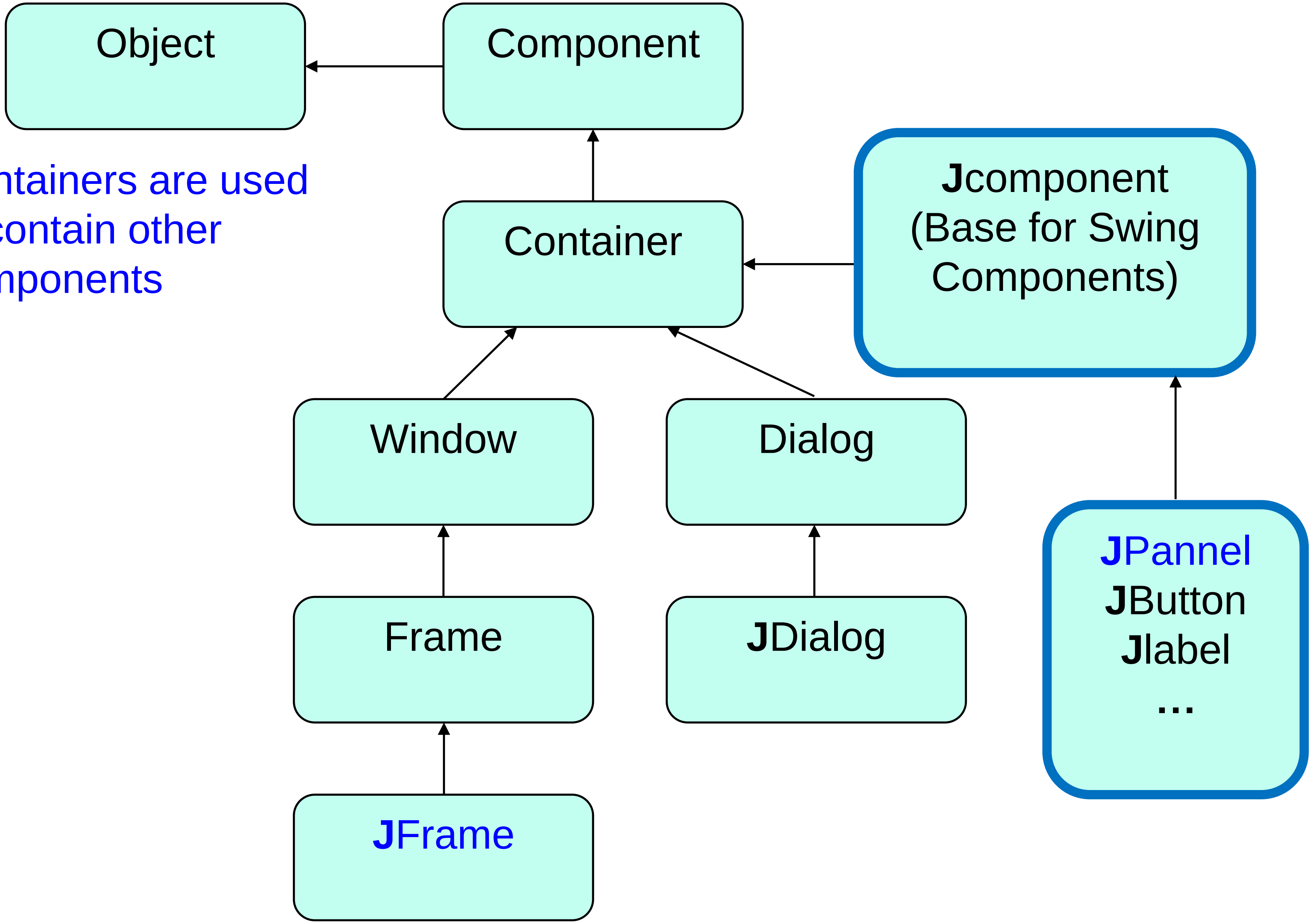
Java GUI Class Hierarchy



Java GUI Class Hierarchy



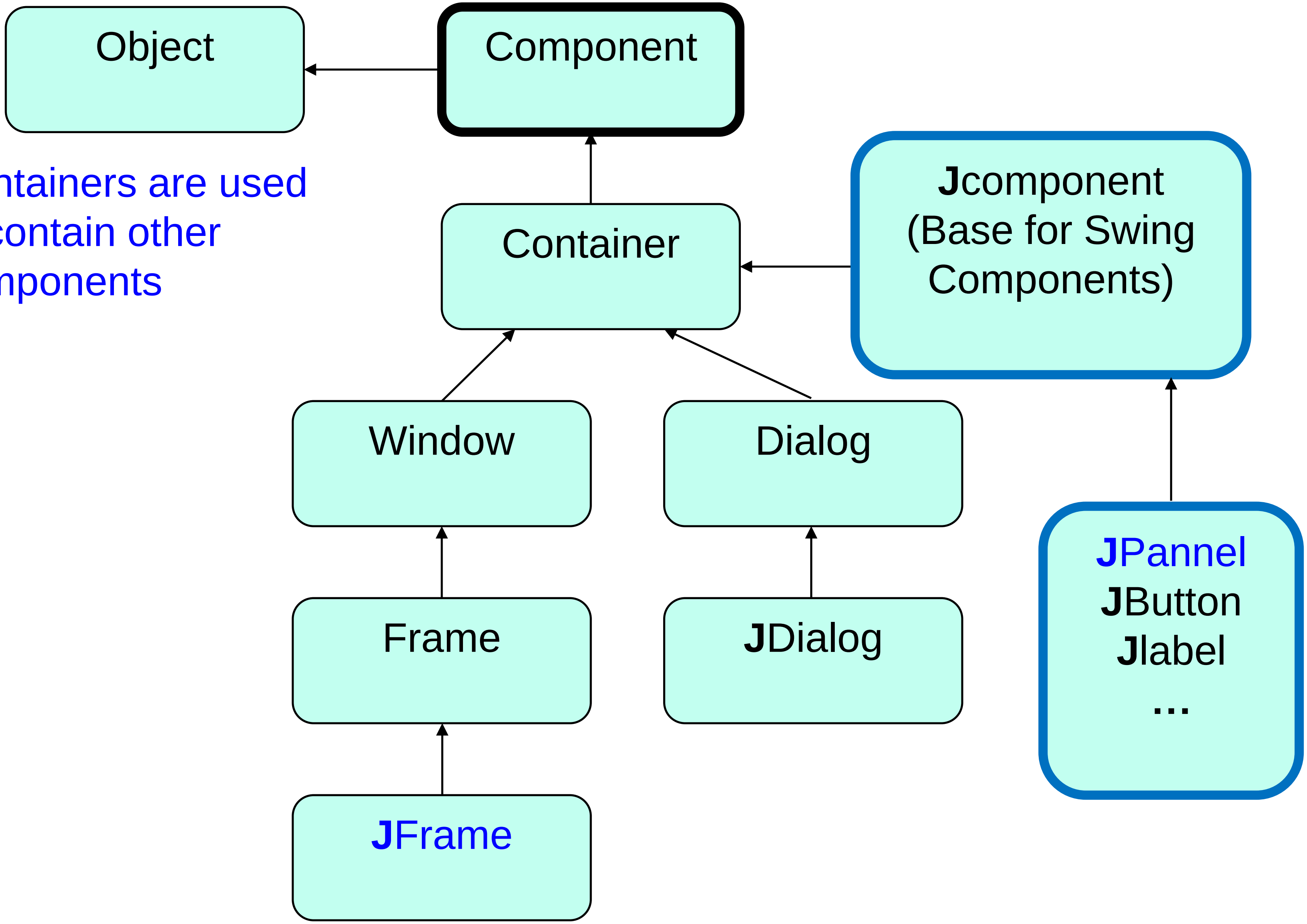
Java GUI Class Hierarchy



- Containers are used to contain other components

Java GUI Class Hierarchy

- Containers are used to contain other components



Java GUI Class Hierarchy

- Java uses the **Composite** Design Pattern to create GUI components that can also serve as containers to hold more GUI components.

The Composite Design Pattern allows a client object to treat both single components and collections of components identically.

- It accomplishes this by creating an abstraction that unifies both the single components and composed collections as abstract equivalents. Mathematically, the single components and composed collections are homomorphically.
- This equivalence of single and composite components is a recursive data structure.
- Since every node is abstractly equivalent, the entire, possibly infinitely large and complex data structure can be succinctly described in terms of just three distinct things:
  - the single components,
  - the composite components, and
  - their abstract representation.

Java GUI Class Hierarchy

- Java uses the Composite Design Pattern to create GUI components that can also serve as containers to hold more GUI components.

This Composite **Design Pattern** allows a client object to treat both single components and collections of components identically.

- It accomplishes this by creating an abstraction that unifies both the single components and composed collections as abstract equivalents. Mathematically, the single components and composed collections are **homomorphically.**
- This equivalence of single and composite components is a **recursive data structure**.
- Since every node is abstractly equivalent, the entire, possibly infinitely large and complex data structure can be succinctly described in terms of just three distinct things:
  - the single components,
  - the composite components, and
  - their abstract representation.

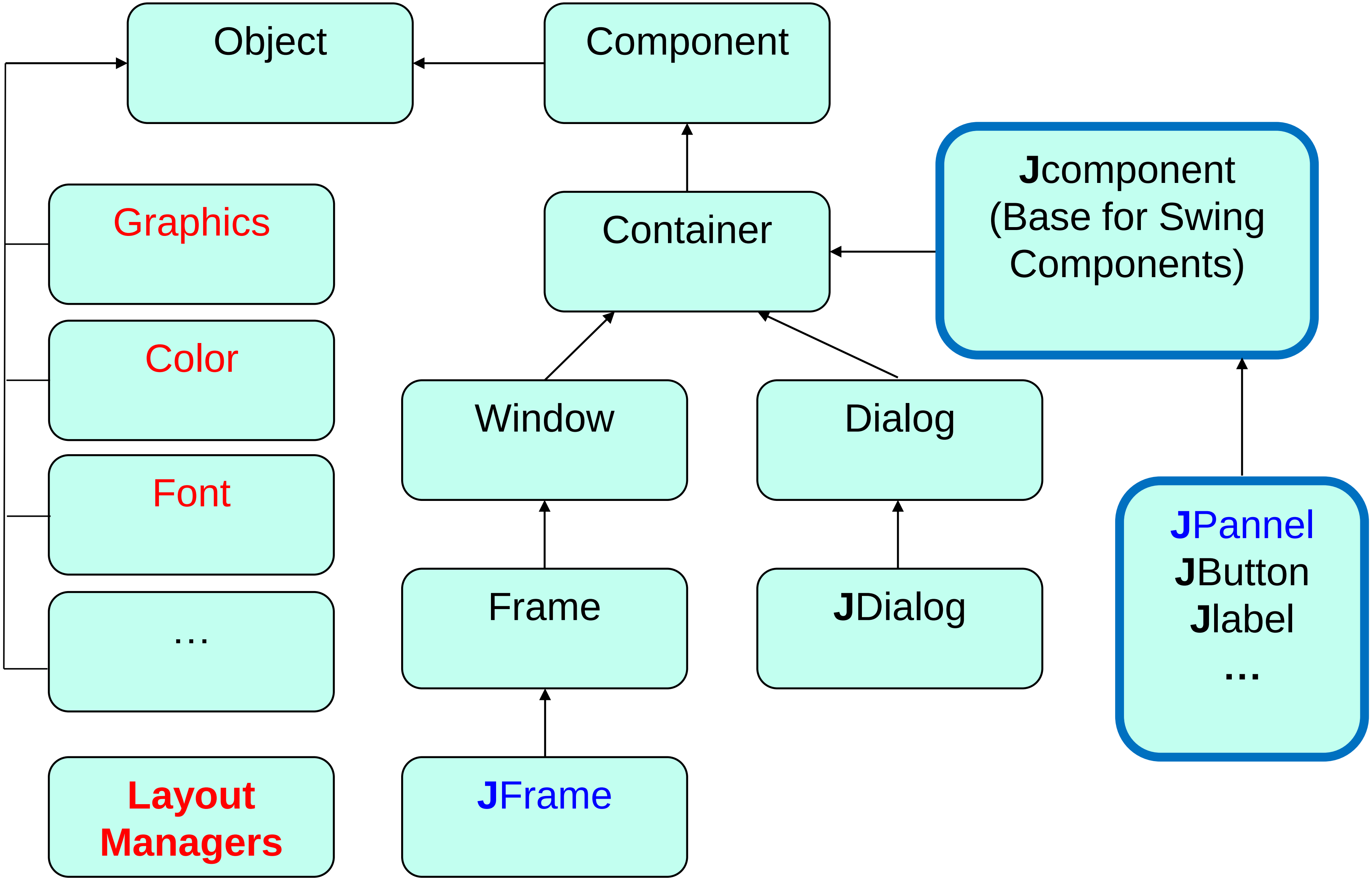
Java GUI Class Hierarchy

- Java uses the Composite Design Pattern to create GUI components that can also serve as containers to hold more GUI components.

This Composite **Design Pattern** allows a client object to treat both single components and collections of components identically.

- It accomplishes this by creating an abstraction that unifies both the single components and composed collections as abstract equivalents. Mathematically, the single components and composed collections are **homomorphically.**
- This equivalence of single and composite components is a **recursive data structure**.
- Since every node is abstractly equivalent, the entire, possibly infinitely large and complex data structure can be succinctly described in terms of just three distinct things:
  - the single components,
  - the composite components, and
  - their abstract representation.

Java GUI Class Hierarchy


Object
Component
Graphics
Container
Jcomponent
(Base for Swing
Components)
Color
Window
Dialog
Font
Frame
JDialog
JPannel
JButton
Jlabel
…
…
Layout
Managers
JFrame