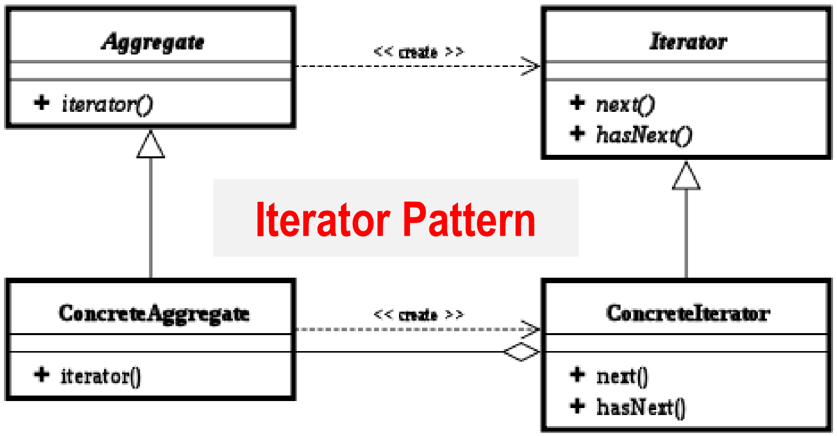
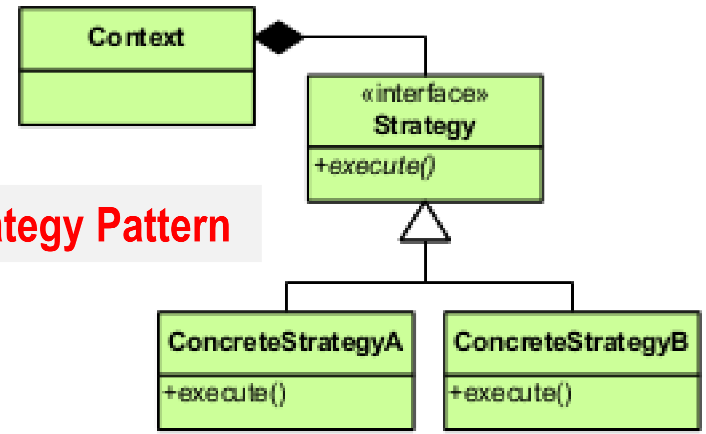
Behavioral Patterns

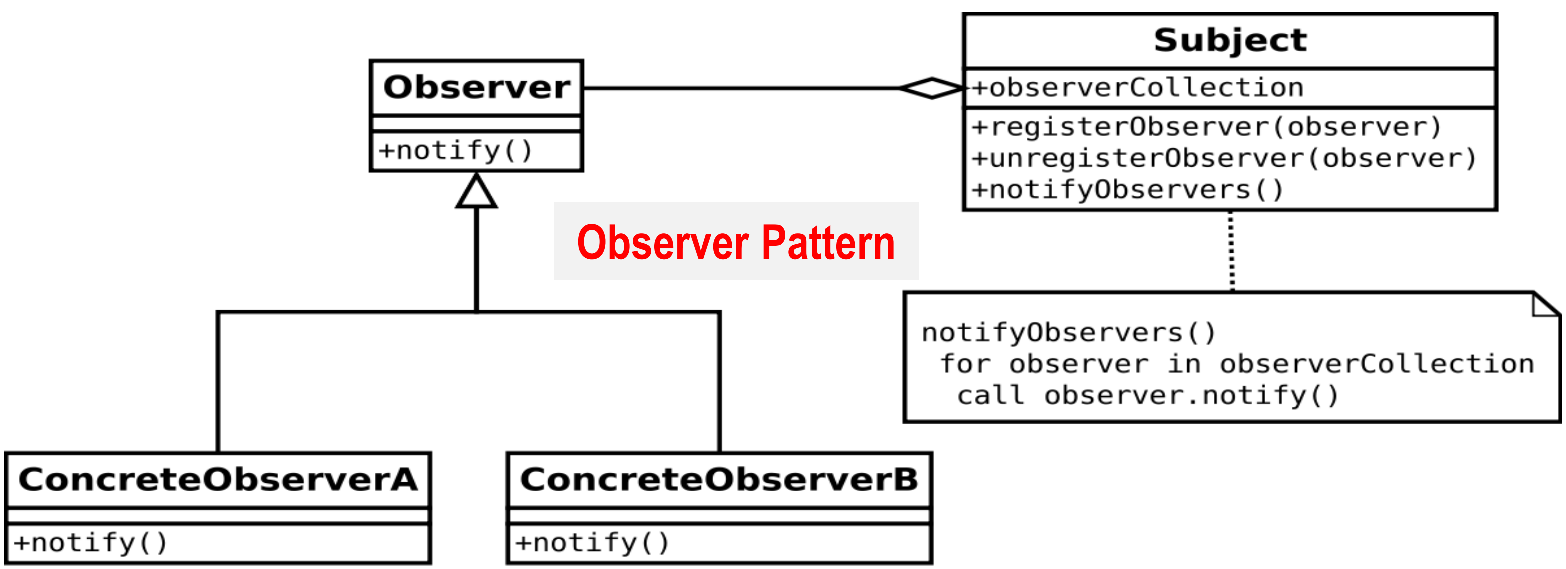
Design Patterns: Elements of Reusable OO Software



Iterator Pattern



Strategy Pattern



Observer Pattern

Characteristics and Benefits

of Design Patterns

- Characteristics of Design Patterns:
  - describes a recurring software structure or idiom
  - is abstract from any particular programming language
  - identifies classes and their roles in the solution to a problem

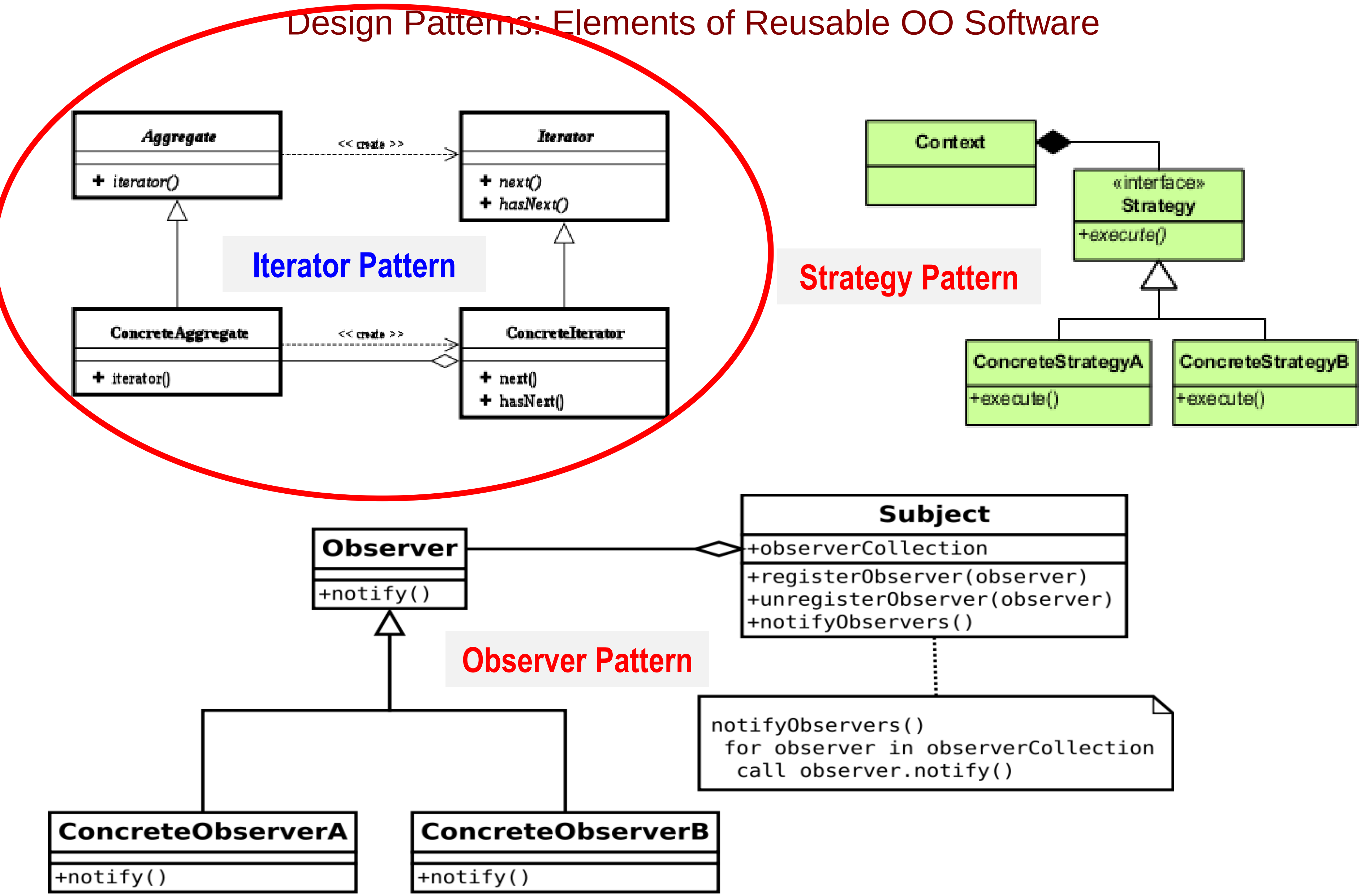
- **Benefits of understanding and using design patterns are:**
  - *Allows to build a common vocabulary in discussing software design.*
  - Allow us to abstract a problem and talk about that abstraction in isolation from its implementation.
  - Allows us to capture expertise
  - Improve on documentation. If we know the pattern of the design solution, we don't need as much to document the solution.

Categories of Design Patterns

- **Creational Patterns** (abstracting the object-instantiation process)
  - *Factory Method* | **Abstract Factory** | *Singleton*
  - Builder | Prototype
- **Structural Patterns** (how objects/classes can be combined)
  - **Adapter** | Bridge | **Composite**
  - **Decorator** | **Facade** | Flyweight
  - **Proxy**
- **Behavioral Patterns** (communication between objects)
  - Command | Interpreter | ***Iterator***
  - Mediator | **Observer** | State
  - **Strategy** | Chain of Responsibility | Visitor
  - Template Method

Behavioral Patterns

Design Patterns: Elements of Reusable OO Software



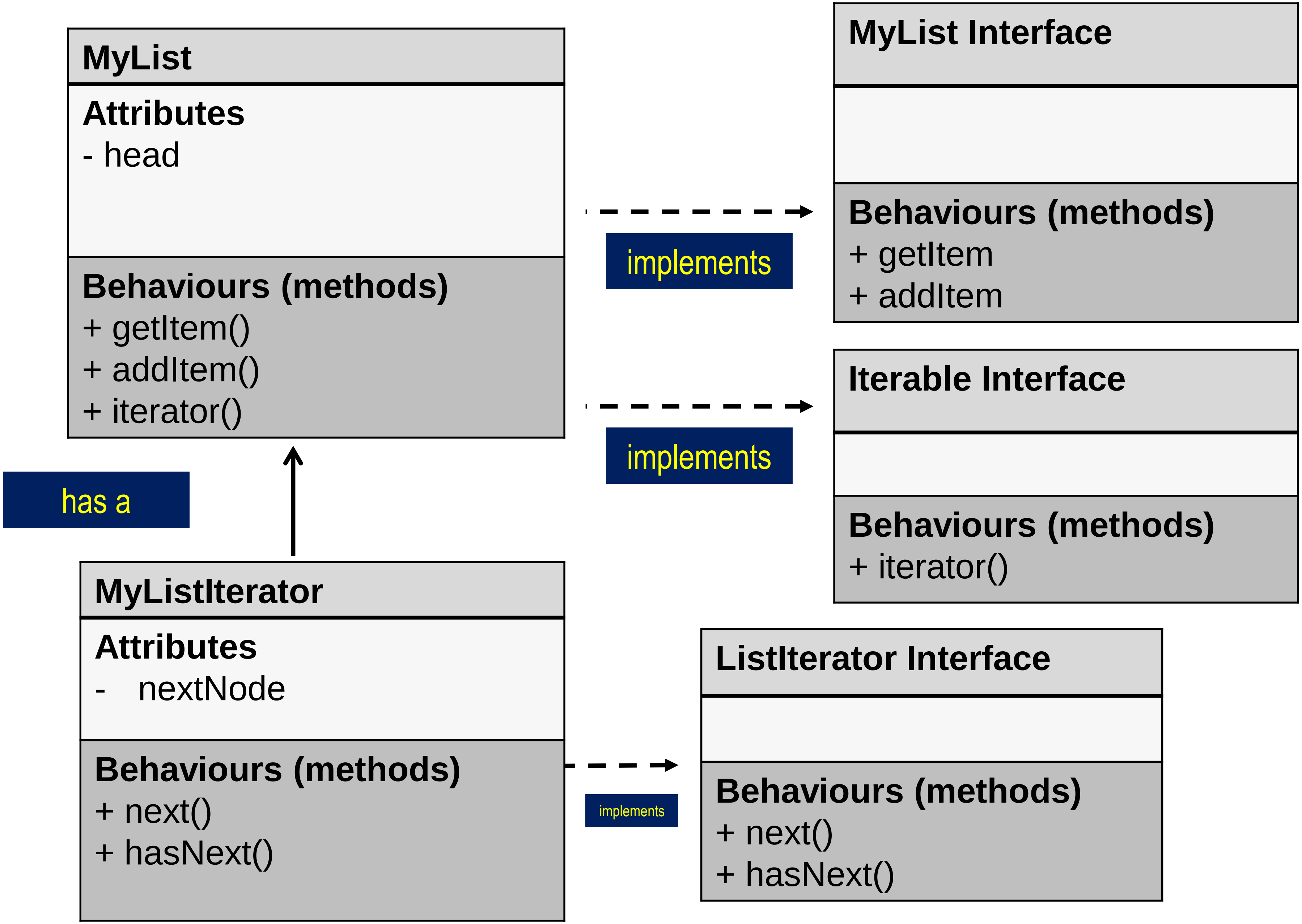
Iterator Pattern:

Elements of Reusable OO Software

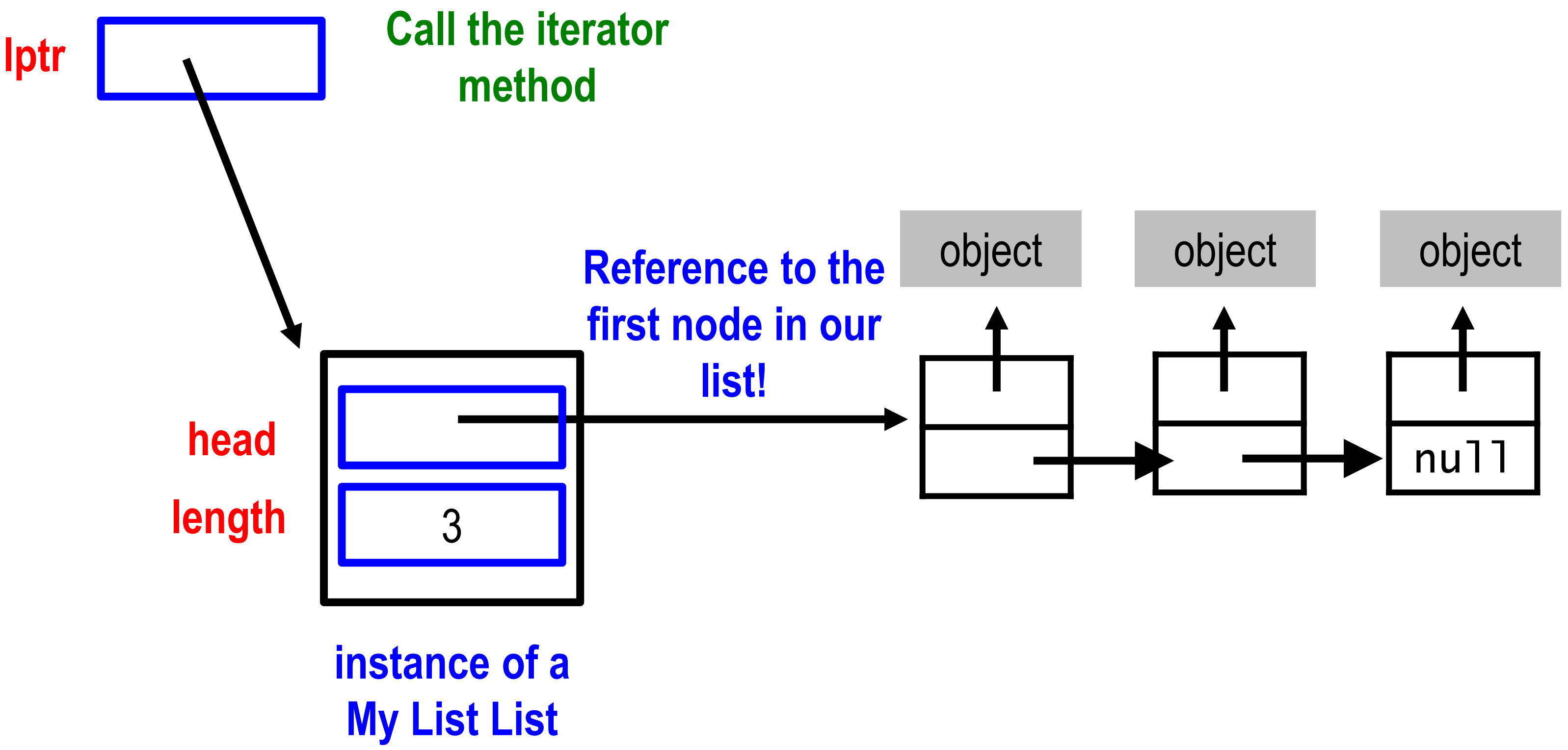
- **Intent**: Provide a way to access the elements of an aggregate object (i.e. a Collection) sequentially without exposing its underlying representation.

- Motivation and ***Applicability***: How to access or iterate over all members of a Collection (at the client level), without needing to know the specifics of the Collection or using specialized traversals for each data structure that underlies the Collection.
  - The focus of this pattern is to take responsibility for access and traversal out of the objects we are iterating over and put it into an iterator object.
  - The iterator class defines an interface for accessing the list's elements, and the iterator object is responsible for keeping track of the current element in the traversal and how to get to the next one.
  - To access an aggregate objects contents without exposing the objects internal representations (violating an objects data encapsulation).
  - To provide a uniform interface for supporting *polymorphic iteration*.

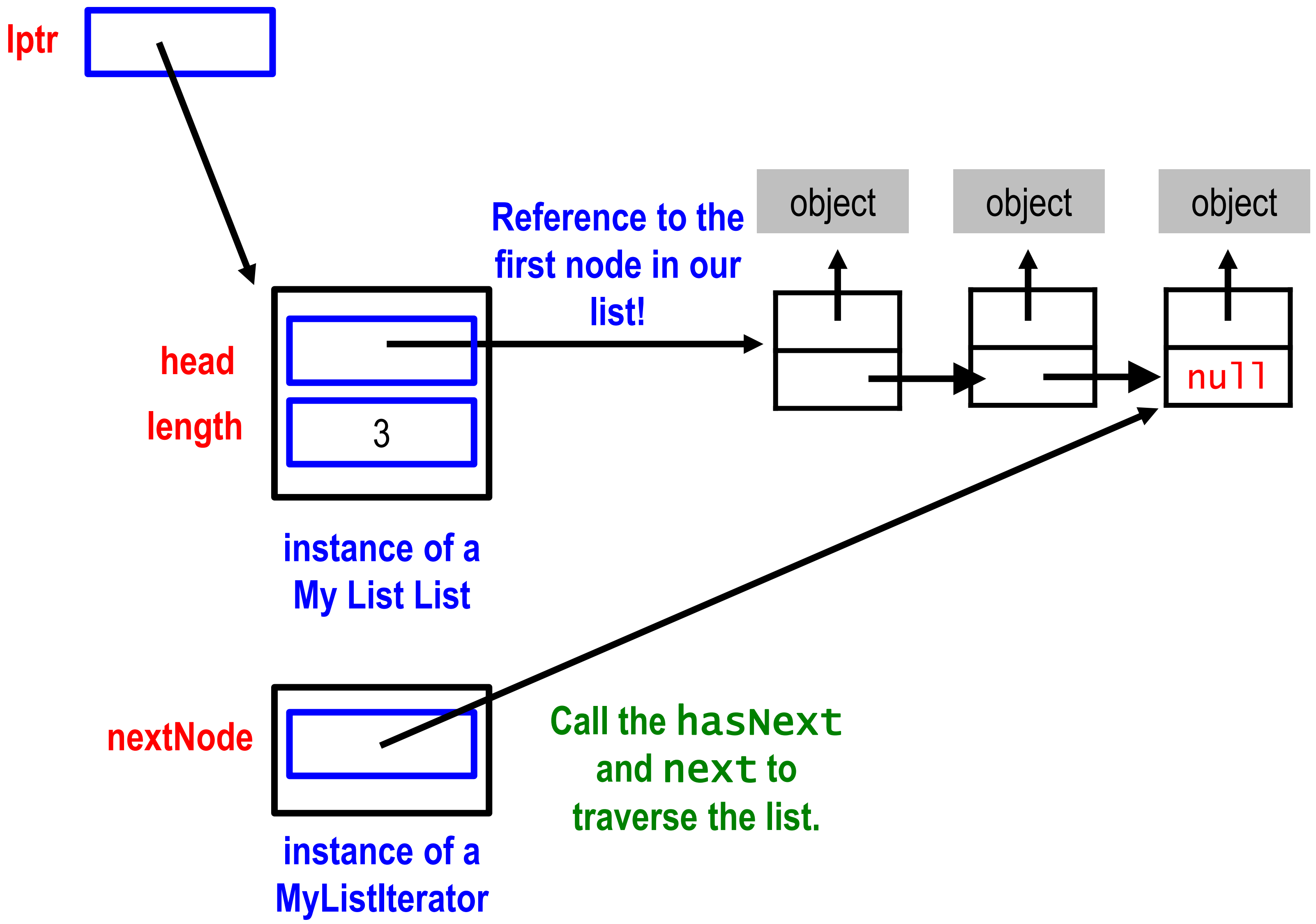
Recall out Iterator Example



MyList Class



MyList Class



Iterator Pattern:

Elements of Reusable OO Software

- **Consequences:** This pattern has *three* important stated consequences:
  - It separates the traversal from the Collection.
  - It supports variations in the traversal of a Collection, example: *preorder*, *postorder*, *inorder*. Depending on which tree traversal we are interested in, we create a new instance of the iterator that facilitates the traversal we want.
  - Multiple traversals can be active at the same time.

Iterator Pattern:

a summary

- Problem: How can we access or iterate over all members of a Collection (at the client level), without needing to know the specifics of the Collection or using specialized traversals for each data structure that underlies the Collection. A client should be able to access all elements of a collection without needing to introduce undesirable dependences.

- Solution:
  - Provide a standard iterator object supplied by all data structures.
  - The implementation performs traversals and has knowledge about the data structure.
  - Results are communicated to clients via a standard interface.

- *Advantages/Disadvantages:*
  - Allows for implementation independence.
  - Allows for multiple traversals of the same collection.
  - Iteration order is fixed by the implementation, not the client.

Iterator Pattern:

a summary

- Problem: How can we access or iterate over all members of a Collection (at the client level), without needing to know the specifics of the Collection or using specialized traversals for each data structure that underlies the Collection. A client should be able to access all elements of a collection without needing to introduce undesirable dependences.

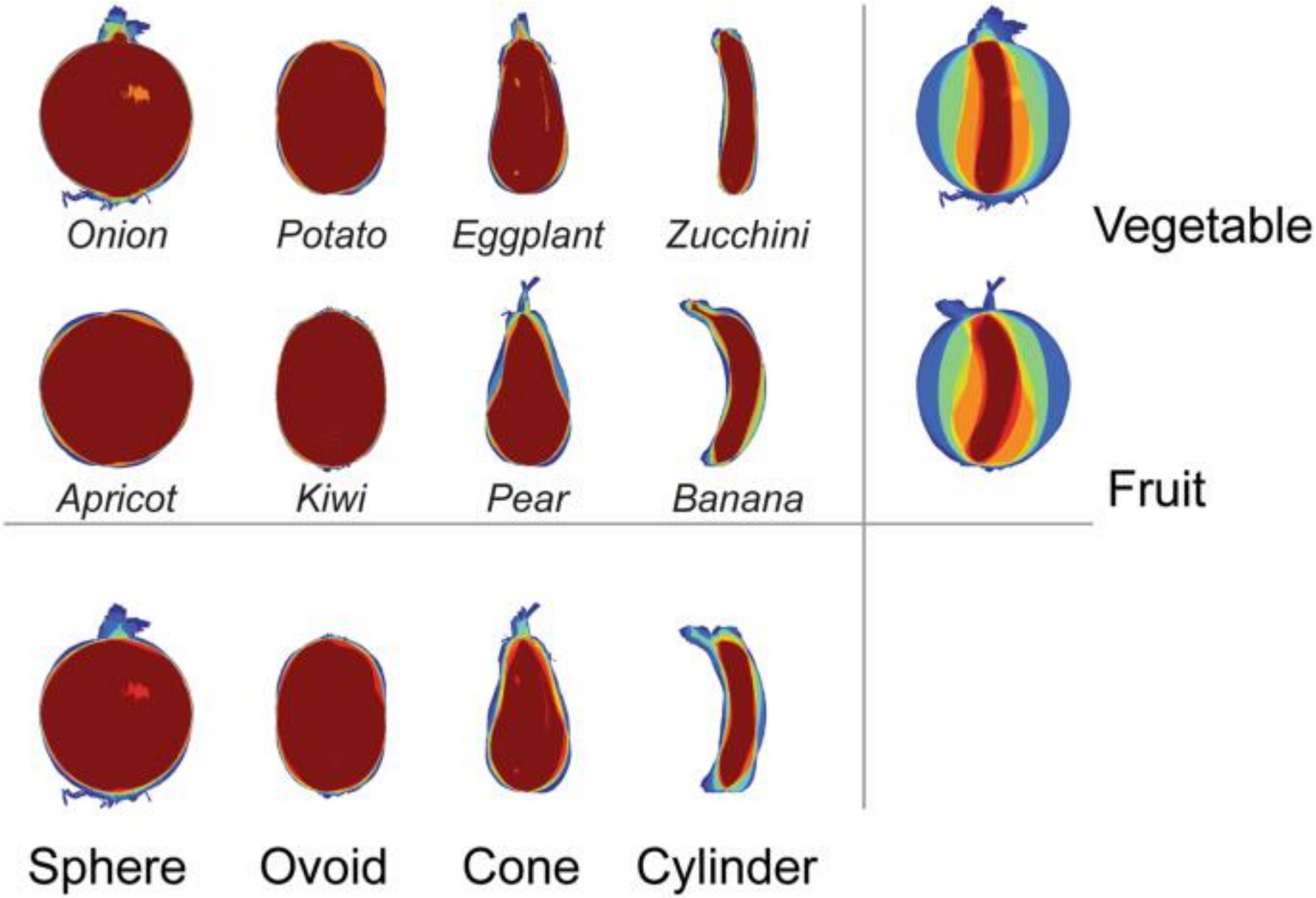
- Solution:
  - Provide a standard iterator object supplied by all data structures.
  - The implementation performs traversals and has knowledge about the data structure.
  - Results are communicated to clients via a standard interface.

- *Advantages/Disadvantages*:
  - Allows for implementation independence.
  - Allows for multiple traversals of the same collection.
  - Iteration order is fixed by the implementation, not the client.

Design Principles:
class vs. type

Edible

Drawable



Class vs. Type

a side note

- An object's class defines *how* the methods of an object are implemented, and it defines the internal state of an object.
- An object's *type* refers to an *interface* – the set of requests to which an object can respond.

This is **not** referring to a Java Interface. The behaviors of the class themselves represent an interface. Java Interfaces are a language specific implementation of how to enforces a class's behavior.

Class vs. Type

Given a student hierarchy, object `f` can be an instance of:

- `Freshman`
- `Undergraduate`
- `Student … Comparable, etc.`

- An object's class define[s] … implemented, and it de[fines] …
- An object's type refers … which an object can respond.
- An object can have many types, i.e.
  - polymorphic behavior.
- Objects of different classes can have the same type, i.e.
  - multiple classes implementing the same behavior or interface.

Class vs. Type

a side note

- An object's class defines *how* the methods of an object are implemented, and it defines the internal state of an object.
- An object's type refers to an interface – the set of requests to which an object can respond.
- An object can have many types, i.e.
  - polymorphic behavior
- Objects of different classes can have the same type, i.e.
  - multiple classes implementing the same behavior or interface.

Objects of Shape and Animal can be *drawable*, *comparable*, etc.

First Principle of Good Design as stated in:

Elements of Reusable Object Oriented Software

- Program to an Interface and not an Implementation:
  - Do not declare variables to be an instance of particular concrete classes. Instead commit only to an interface as defined by an Abstract Class *or a Java Interface*.

1. Clients remain unaware of the specific types of objects they use, as long as the objects adhere to the interface that the clients expect.
2. Clients *remain unaware of the classes that implement these objects*. Clients are only aware of the type (abstract class or interface) that defines the object type interface.

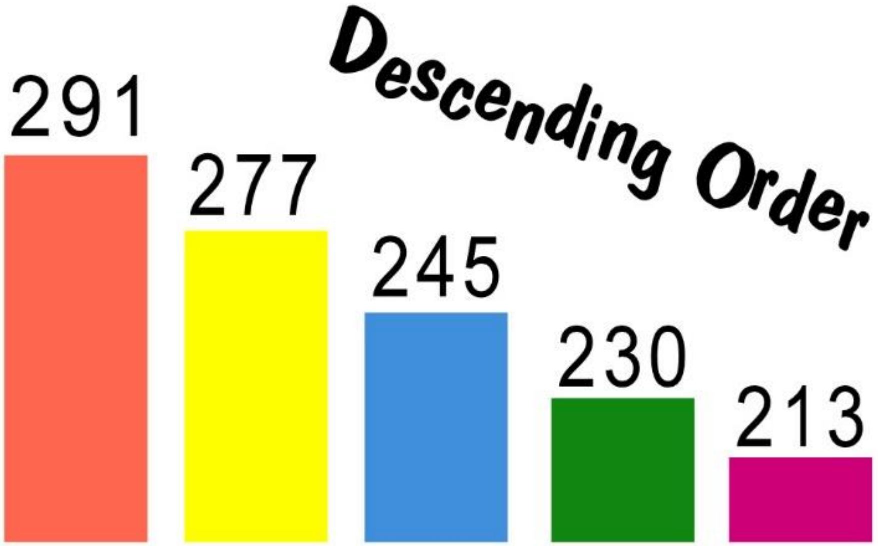
Second Principle of Good Design as stated in:

Elements of Reusable Object Oriented Software

- Favor object composition over class inheritance:
  - ***"has a"*** over "**is a**"
  - You shouldn't have to create new objects to achieve reuse.
  - You should be able to get all the desired functionality by assembling existing components through object composition.

- To accomplish this *varied* class *behavior* are turned into objects. Example: Iterator pattern, strategy pattern.

Recall Comparators



CollectionS Class

```
public class testClass {
    public static void main( String [] args ) {
        List<String> fruits = new ArrayList<String>();

        Collections.addAll(fruits,"Banana", "Mango"
                           , "Apples","Oranges","Kiwi");

        for ( String s : fruits )       // element based loop
            System.out.println( s );

        Collections.sort( fruits );
        for ( String s : fruits )       // element based loop
            System.out.println( s );
    }
}
```

```
Apples
Banana
Kiwi
Mango
Oranges
```

The natural order is established by the `CompareTo` method.

Comparator Interface

```
public class lengthComparator implements Comparator<String>
{
    public int compare(String s1, String s2){

        return( s1.length() - s2.length() );


    }
} // class
```

Note that this class does not contain any state. It only specifies a behavior!

Even though we can create an instance of this class, we only do so to invoke the specific behavior of this method.

Comparator Interface

```
public class lengthComparator implements Comparator<String>
{
    public int compare(String s1, String s2){

        return(s1.length() - s2.length());


    }
} // class
```

```
public class reverselengthComparator implements
Comparator<String>
{
    public int compare(String s1, String s2){

        return(s2.length() - s1.length();


    }
} // class
```

CollectionS Class

```
public class testClass {
    public sta
        List<S

        Collec
```

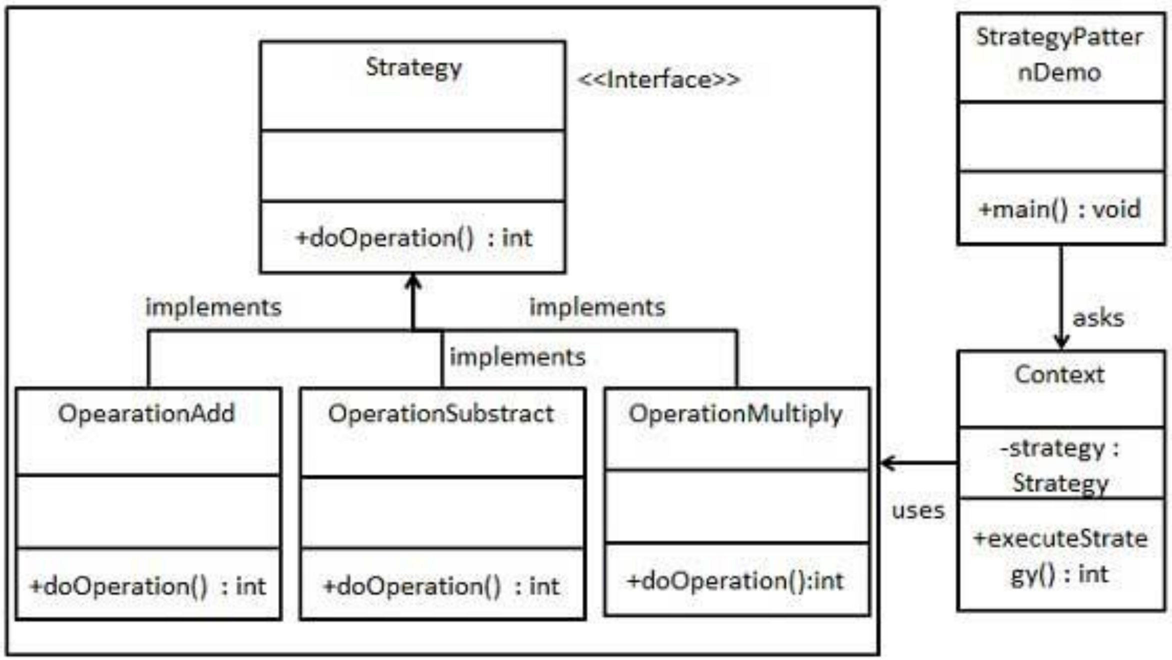
Sort method does not need to know about the class `reverselengthComparator`, it is of type **`Comparator`**.

```
        for ( String s : fruits )        // element based loop
            System.out.println( s );

        Collections.sort(fruits, new reverselengthComparator());
        for ( String s : fruits )        // element based loop
            System.out.println( s );
    }
}
```

```
Oranges
Apples
Banana
Mango
Kiwi
```

Strategy Pattern:
Reuse through object composition

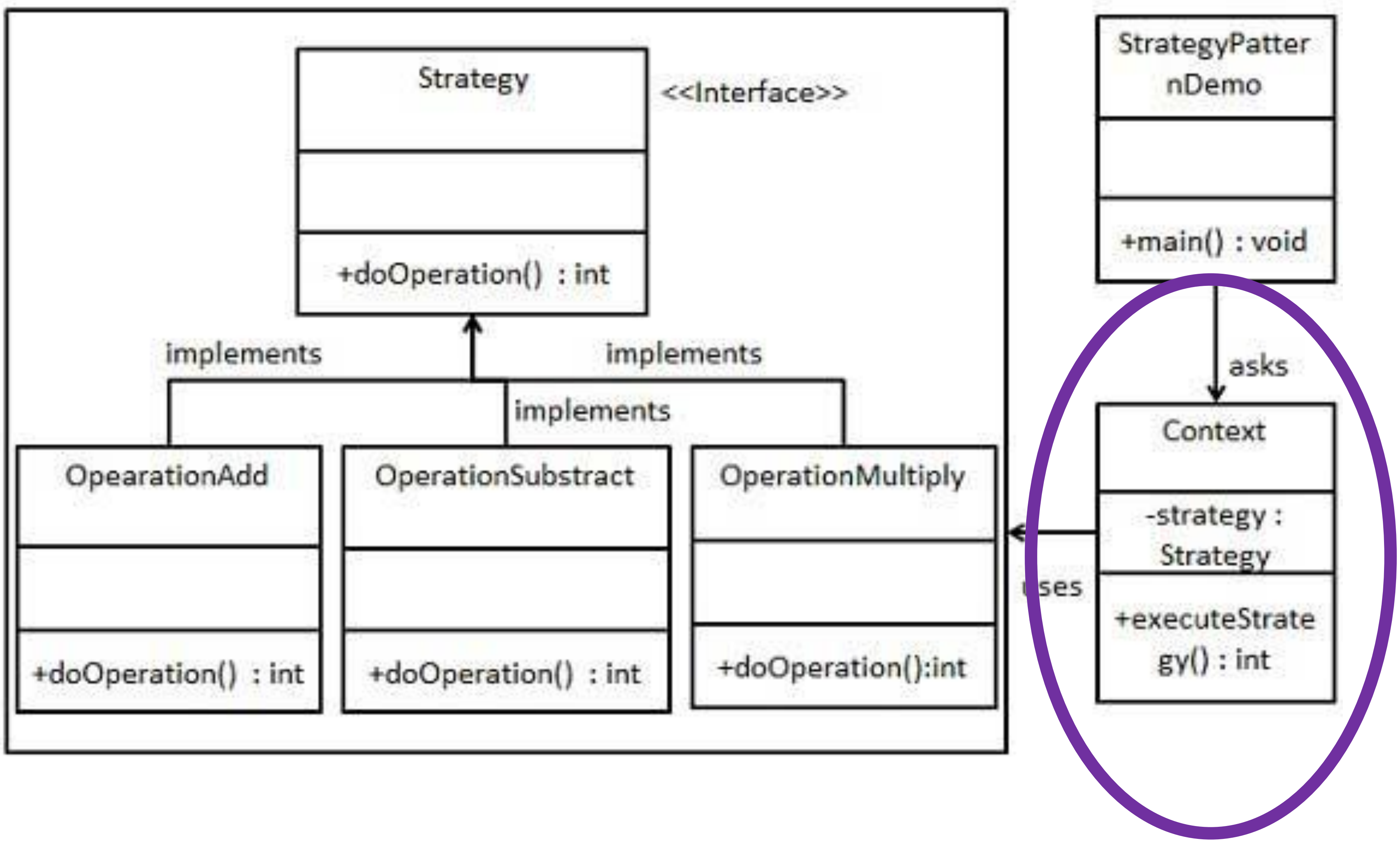
Intent: Define a **family of algorithms**, encapsulate each one, and make them interchangeable.



[Tutorials Point](#)

Strategy Pattern:

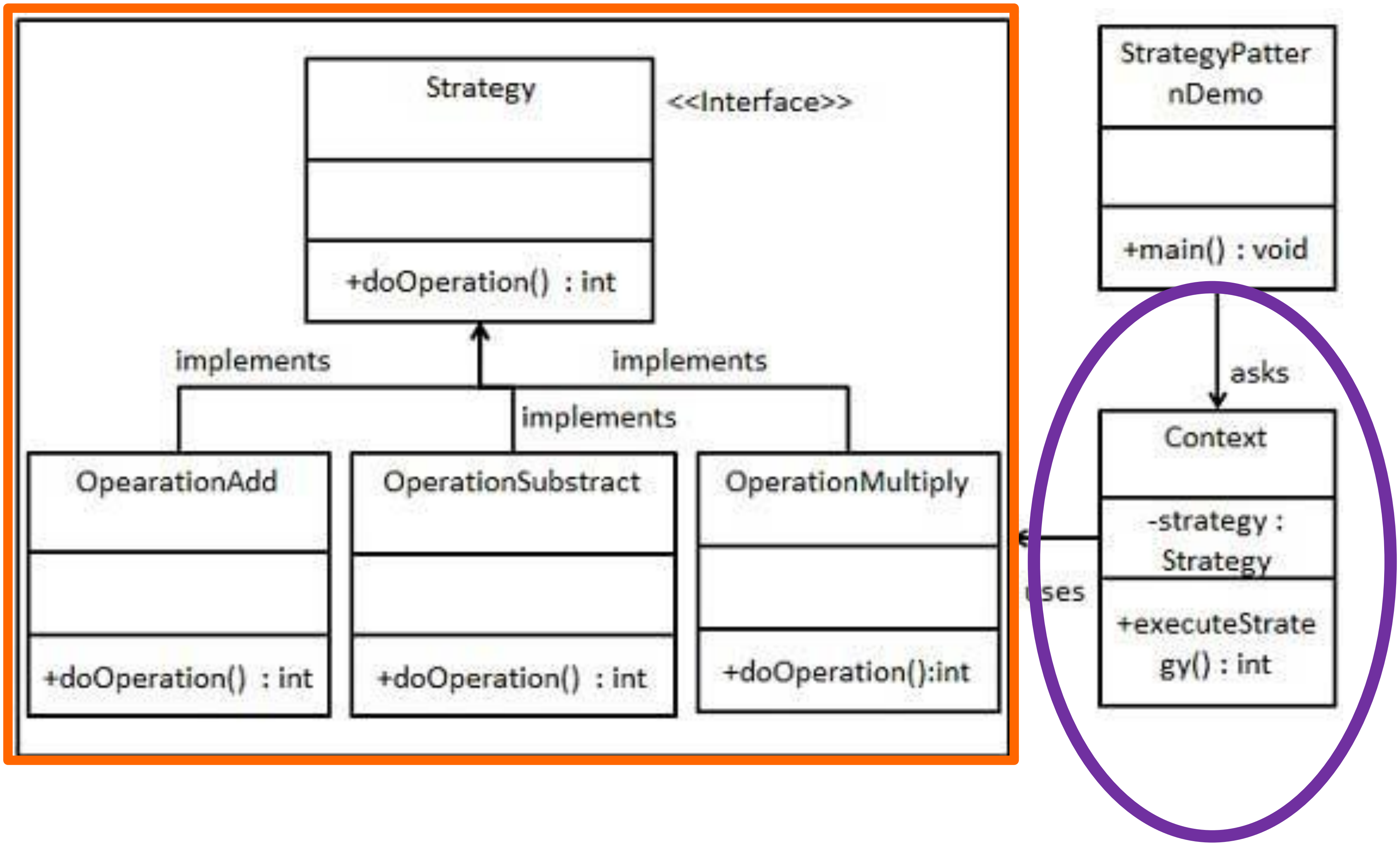
Reuse through object composition



Tutorials Point

Strategy Pattern:

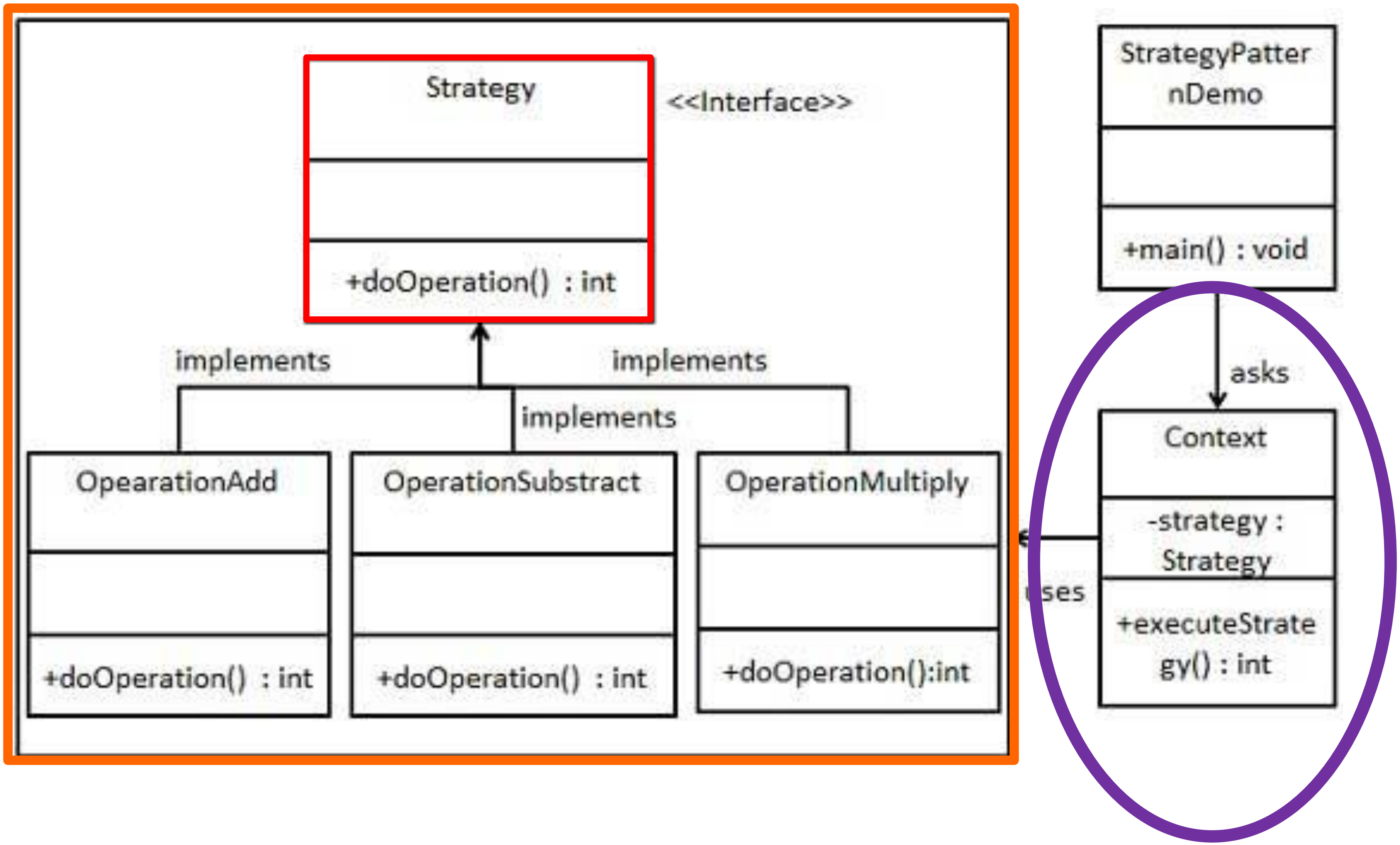
Reuse through object composition



[Tutorials Point](#)

Strategy Pattern:

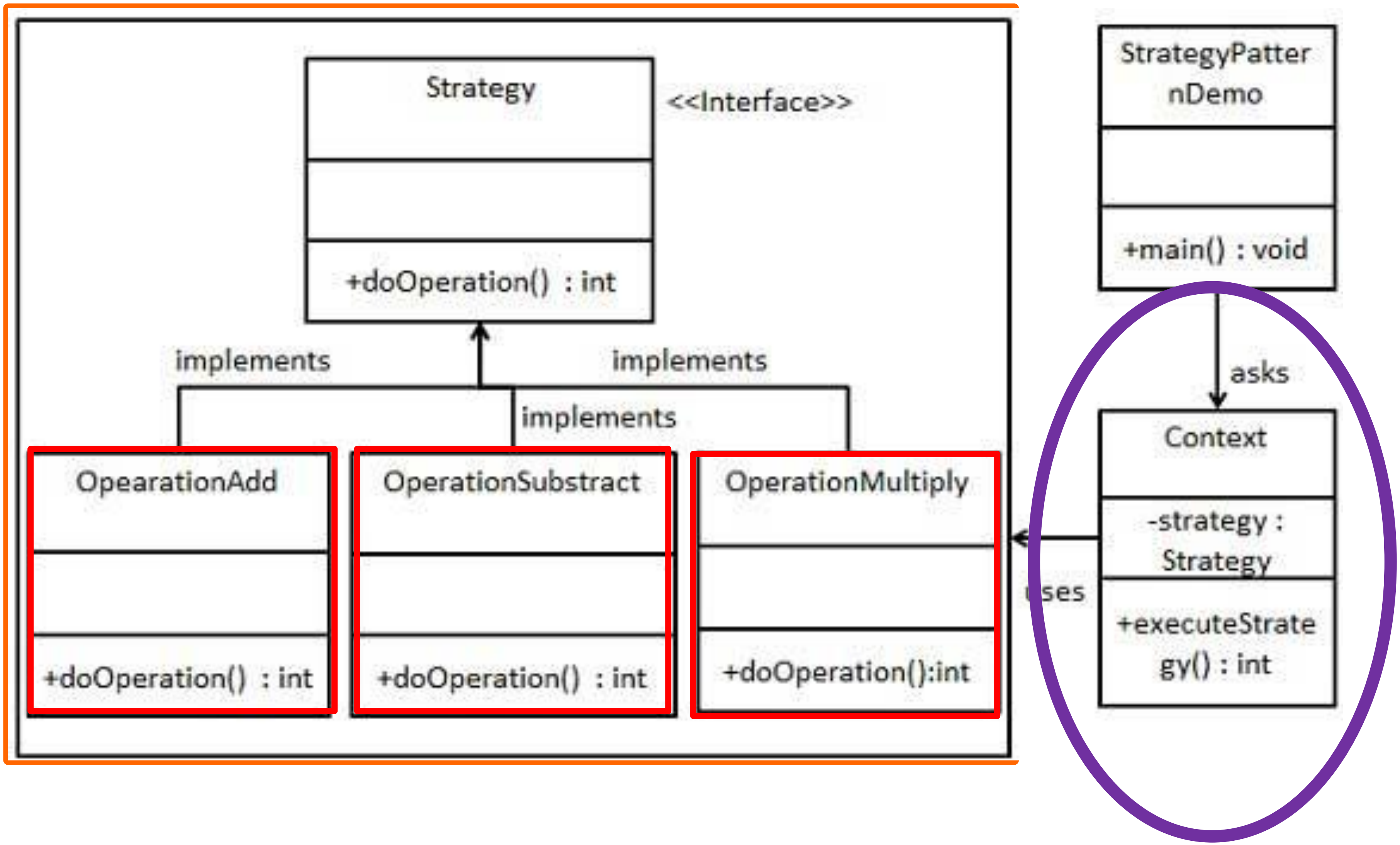
Reuse through object composition



[Tutorials Point]()

Strategy Pattern:

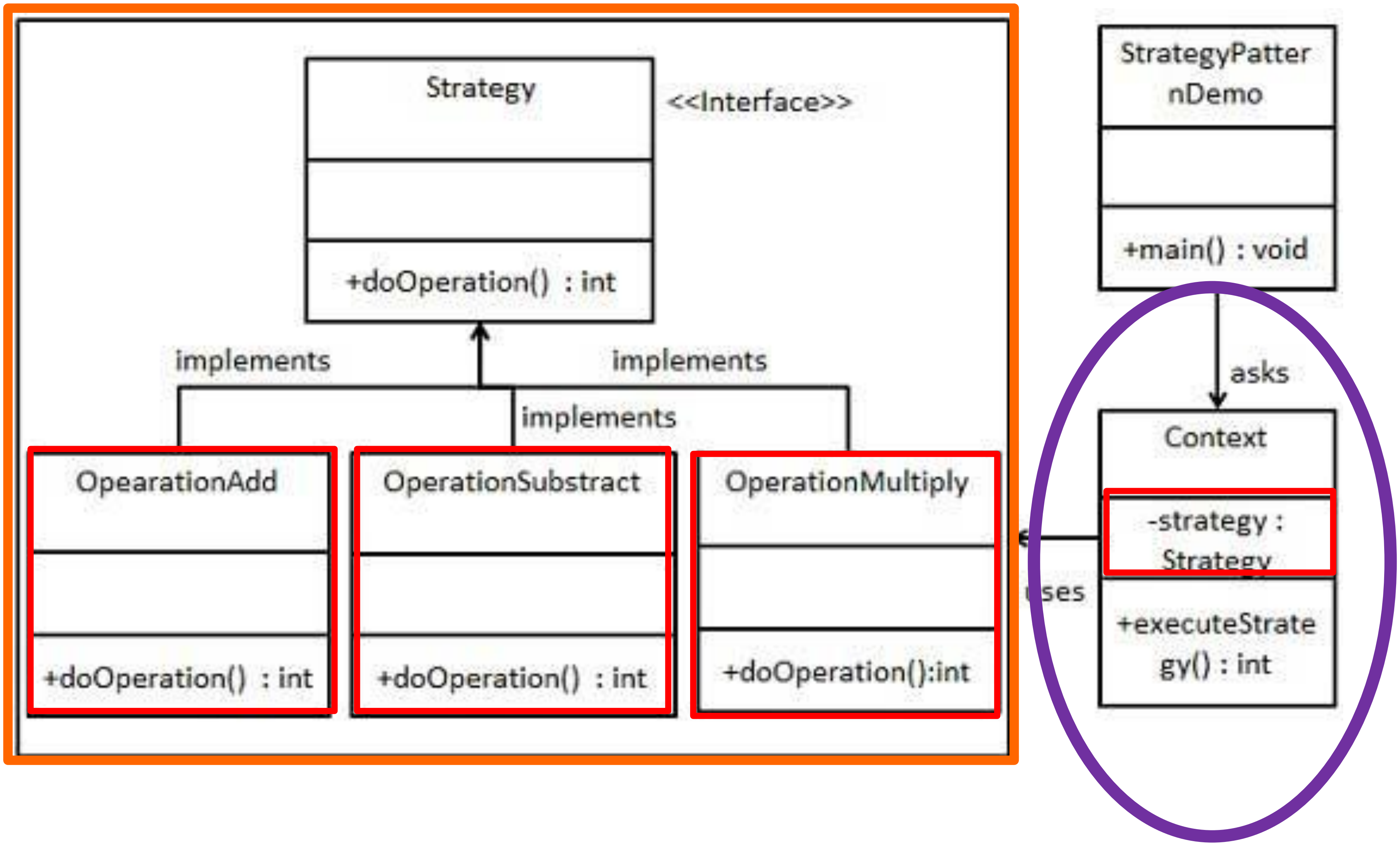
Reuse through object composition



Tutorials Point

Strategy Pattern:

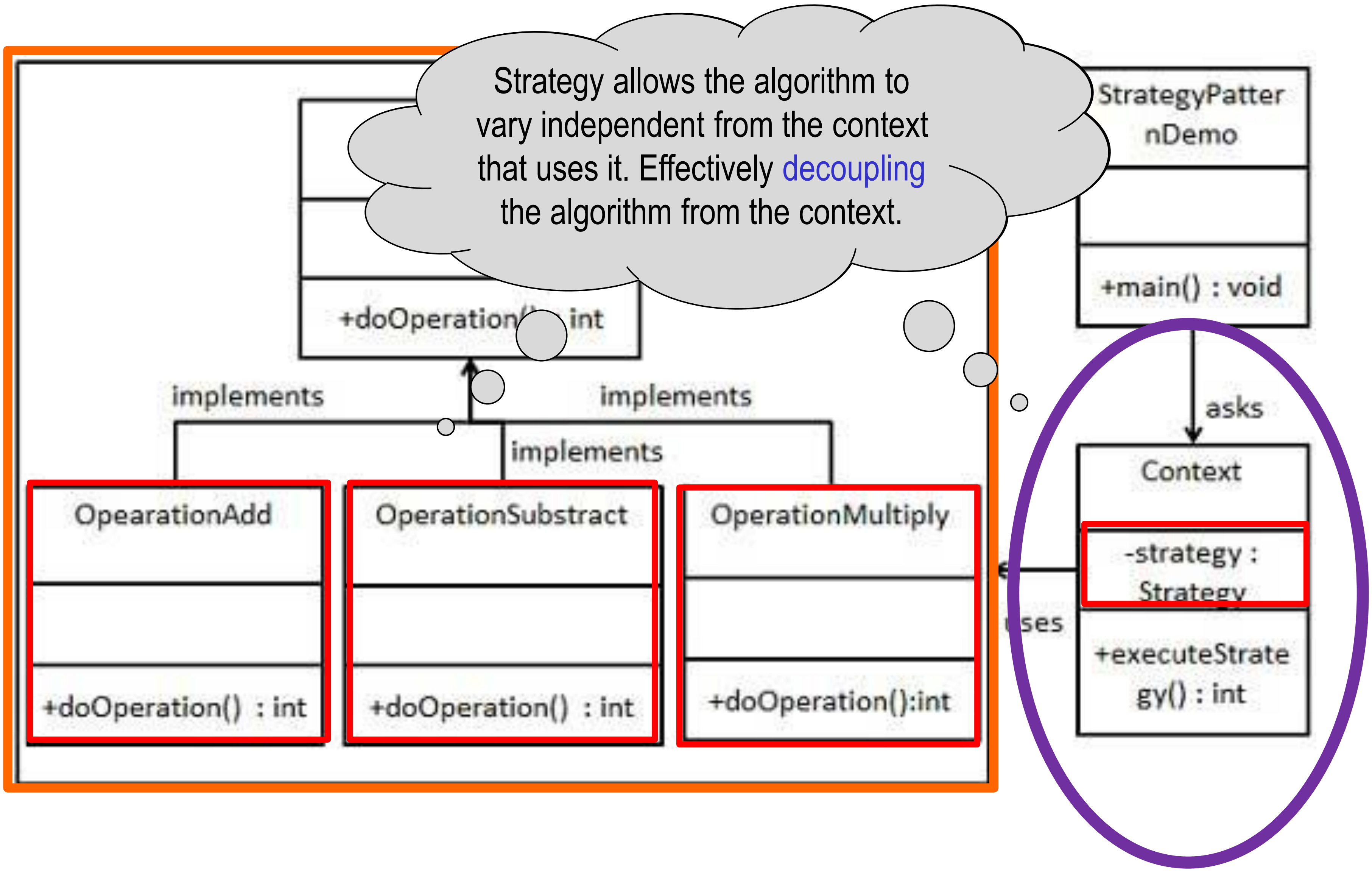
Reuse through object composition



Tutorials Point

Strategy Pattern:

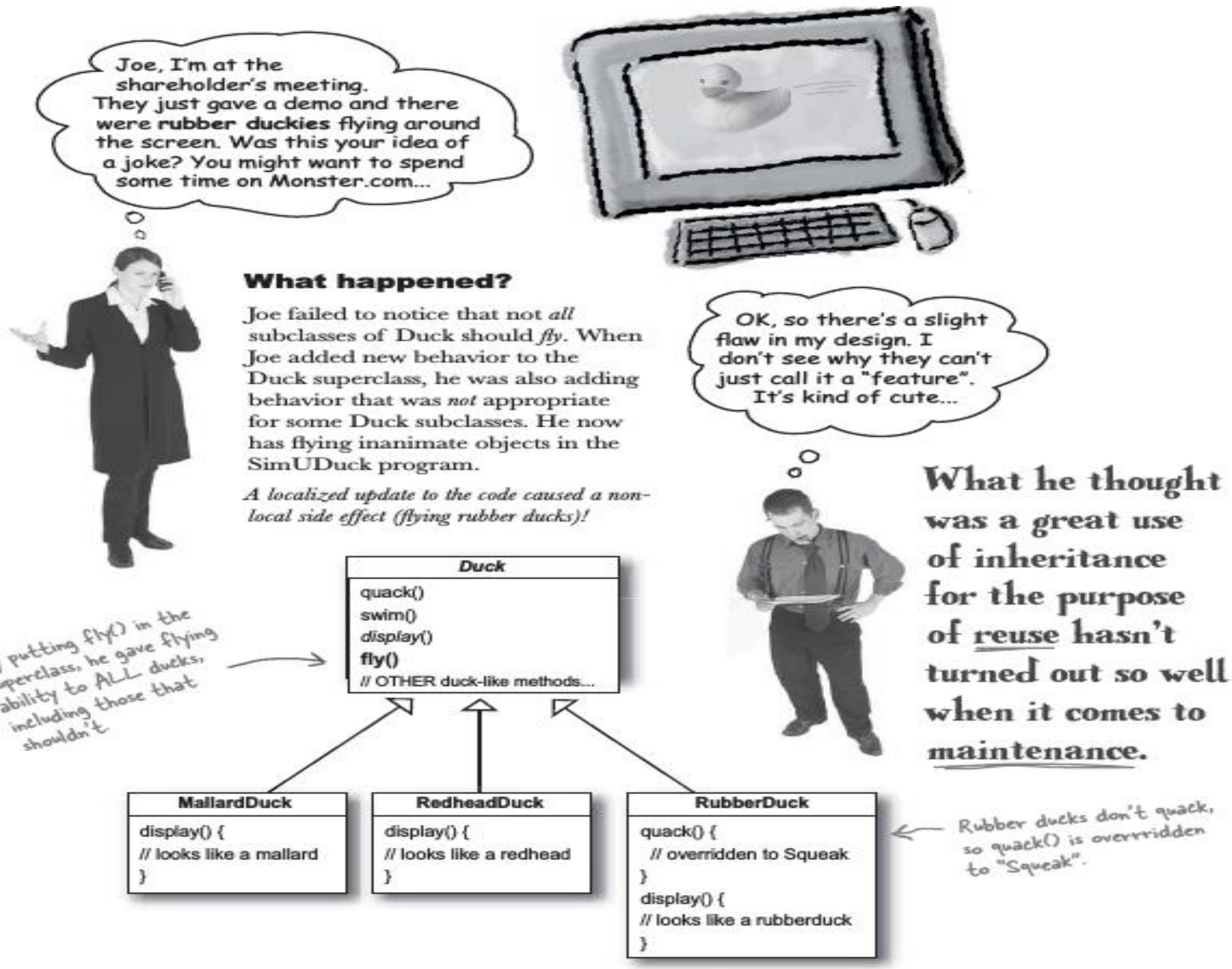
Reuse through object composition



Tutorials Point

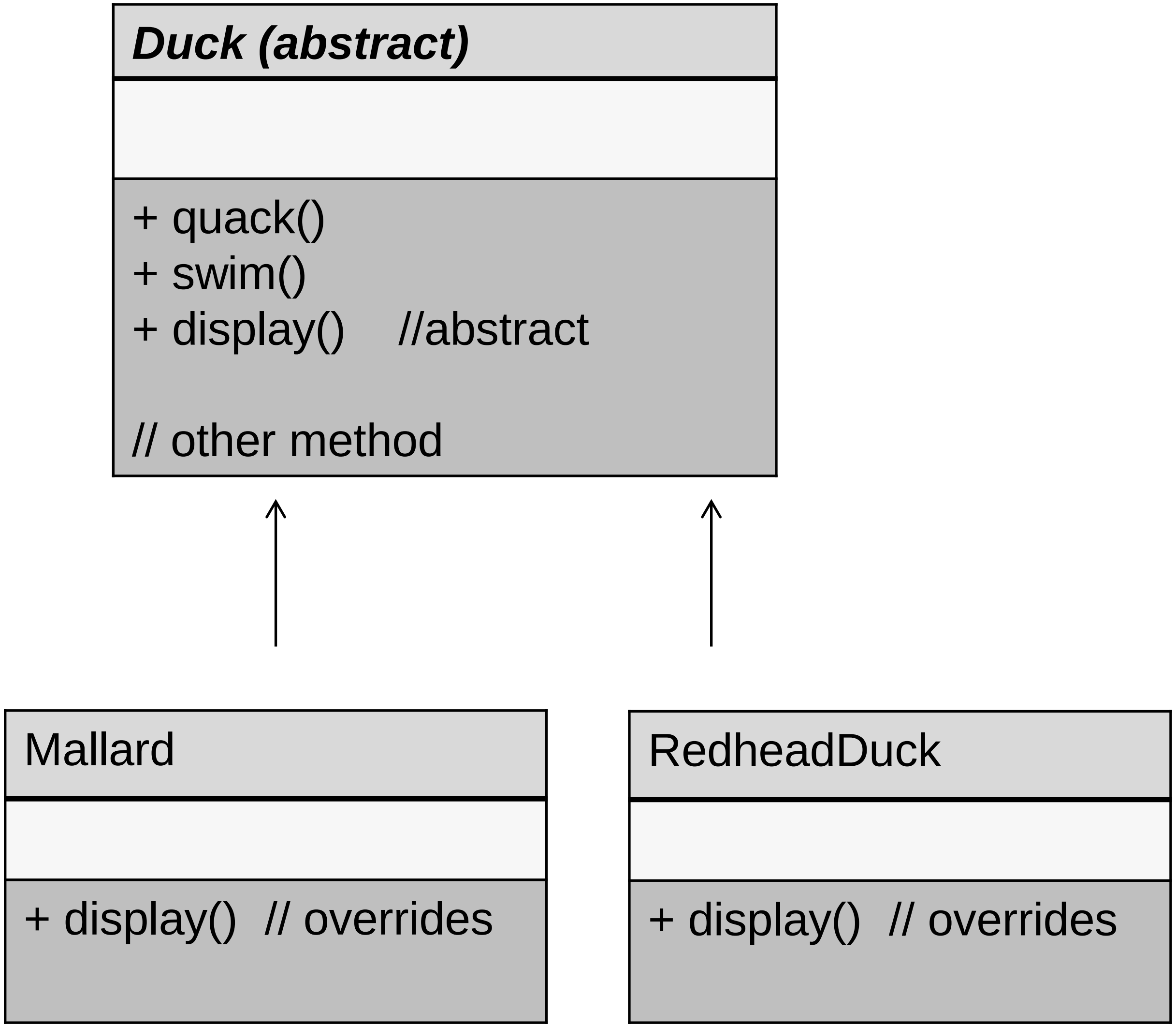
Strategy Pattern:

Example from: Head First Design Patterns; Sierra, Freeman, Robson, … (O'Reilly)



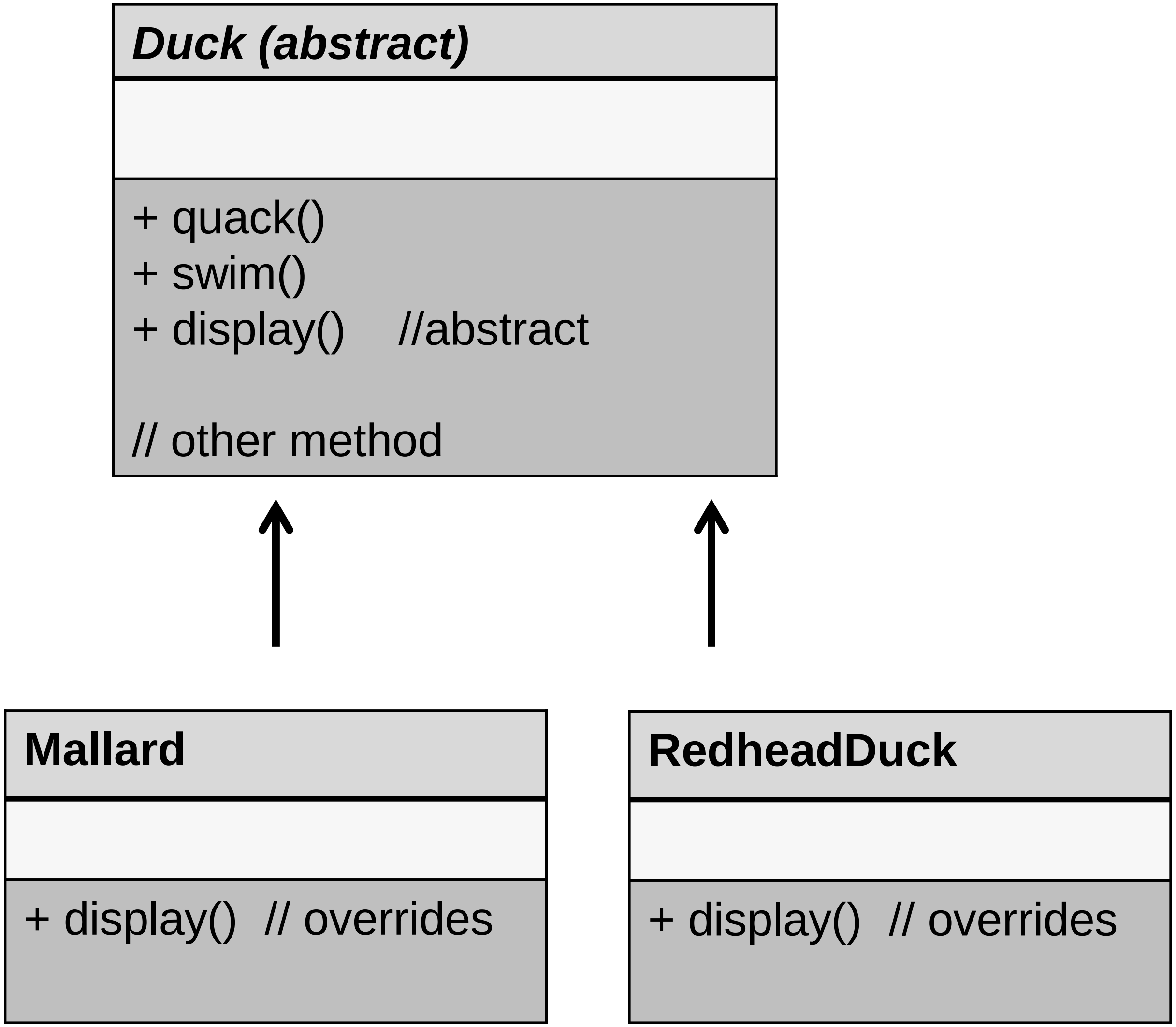
Inheritance:

drawbacks of



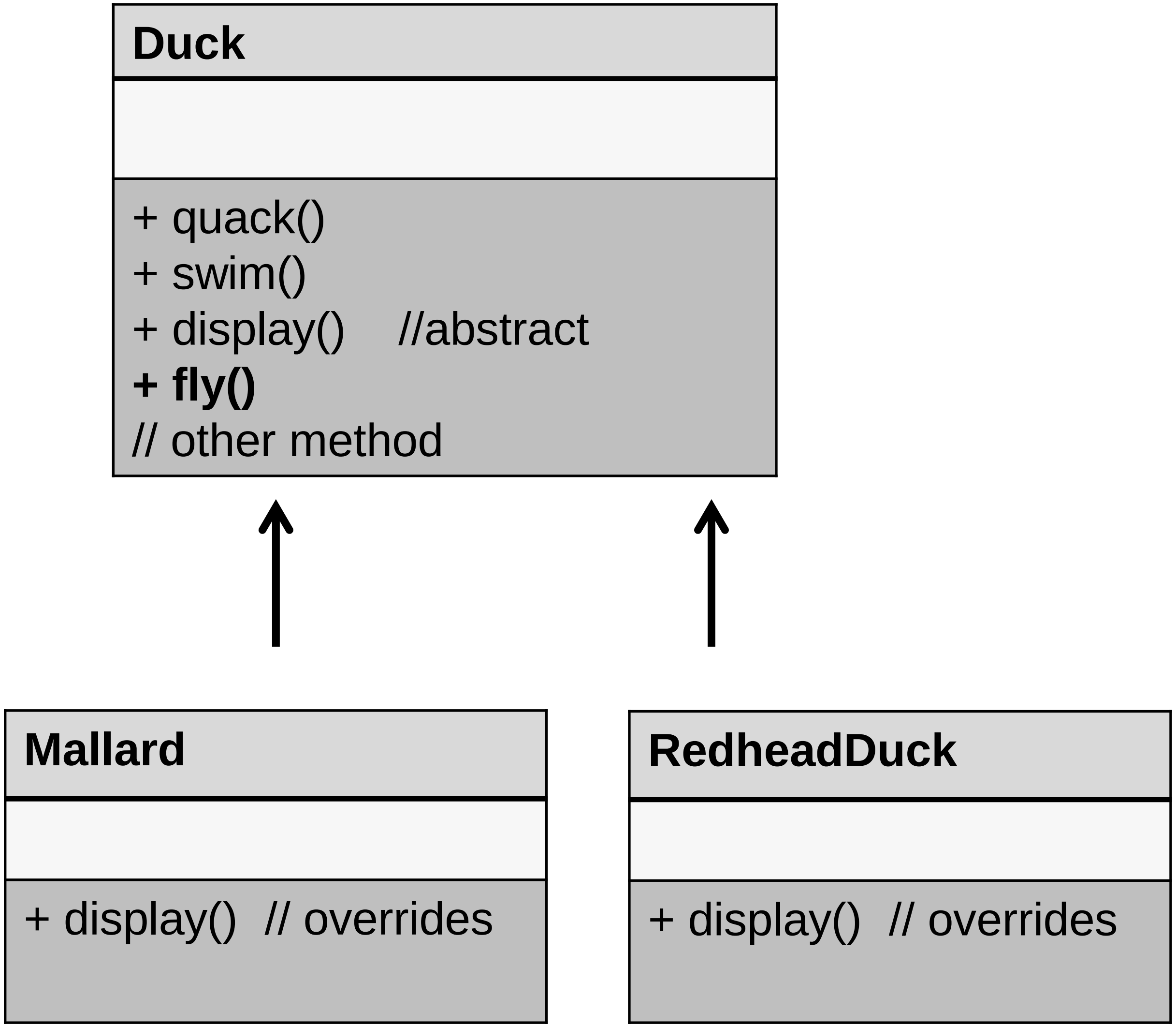
Inheritance:

drawbacks of

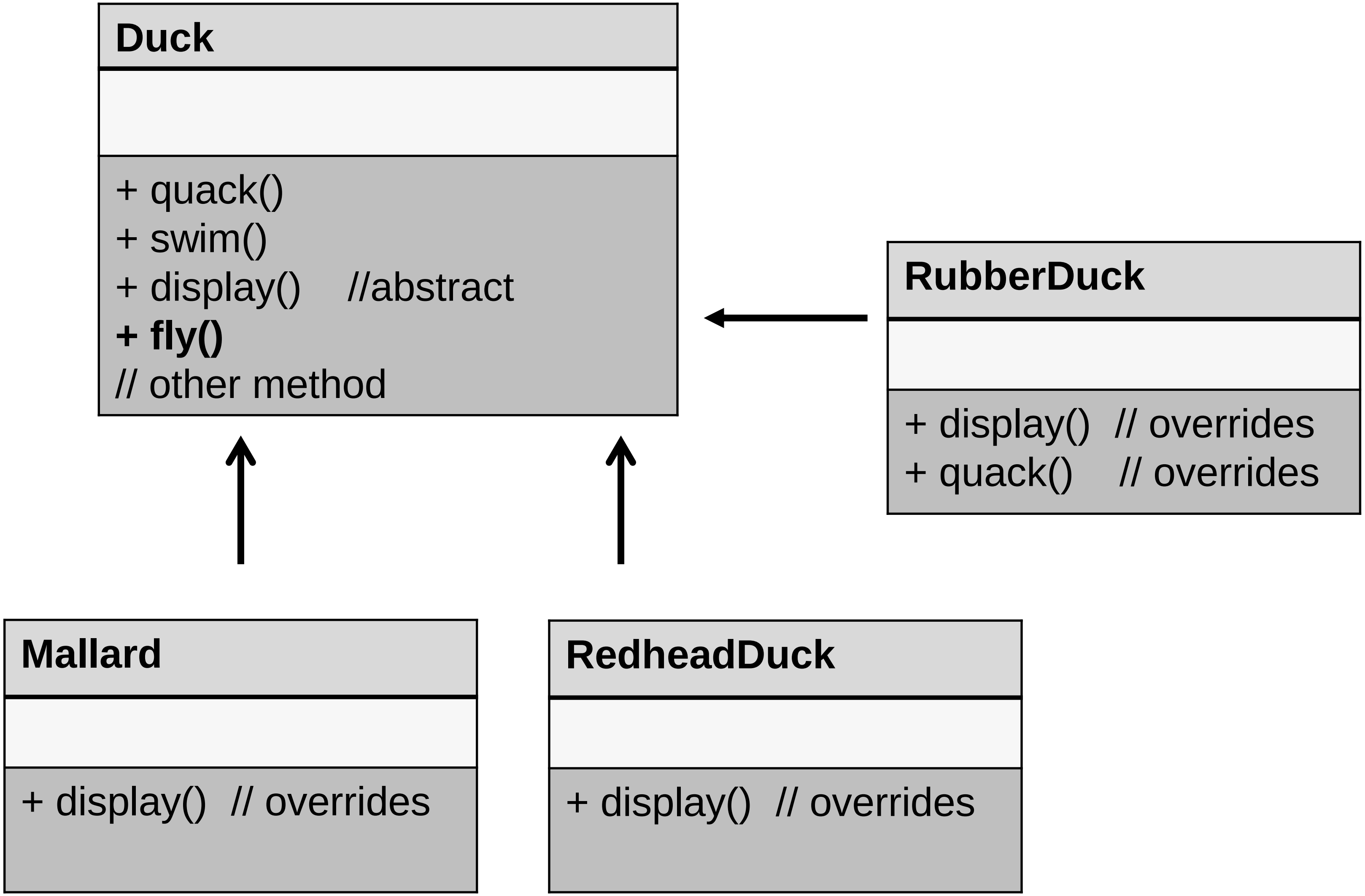


Inheritance:

drawbacks of

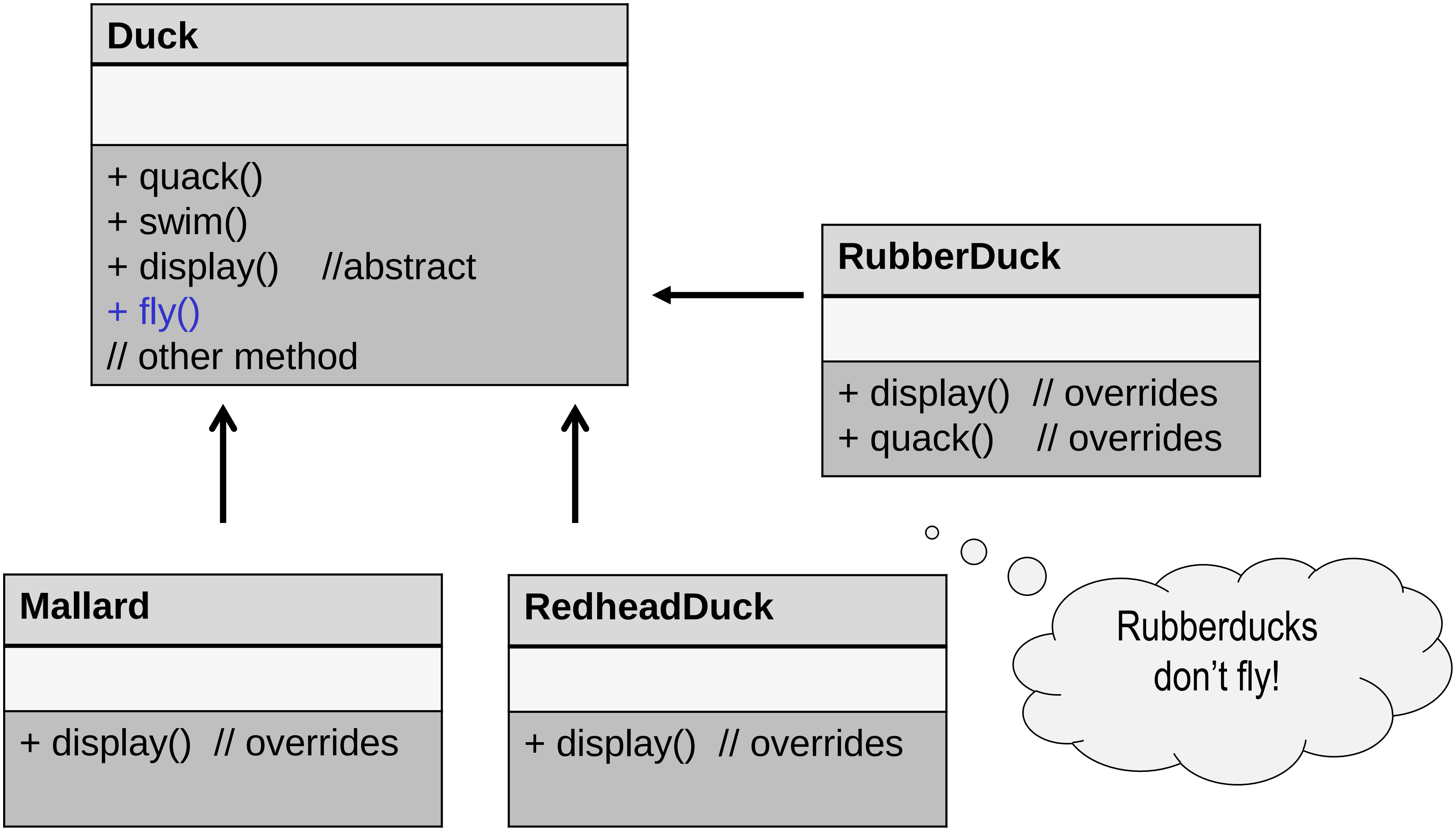


Inheritance:
drawbacks of



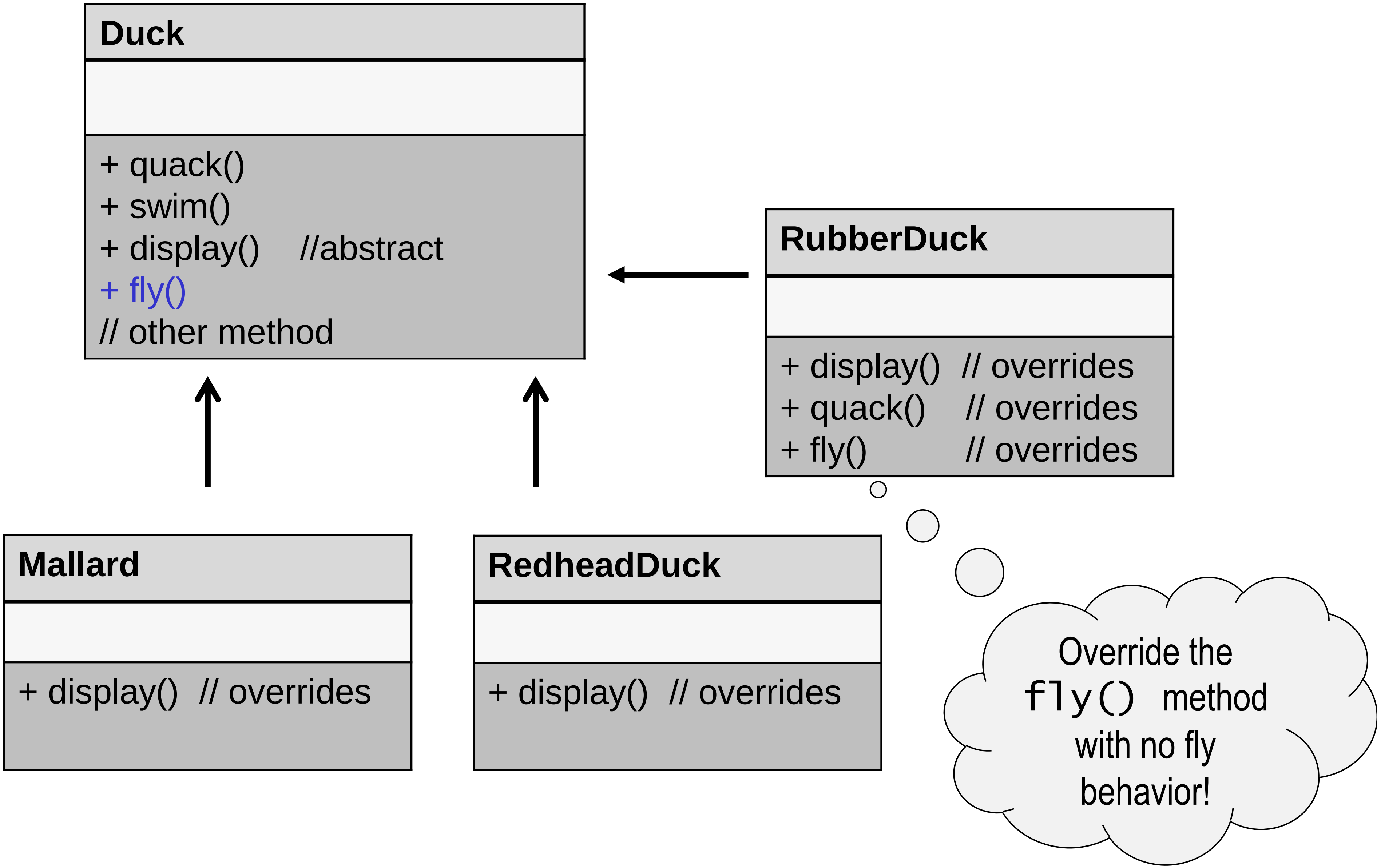
Inheritance:

drawbacks of



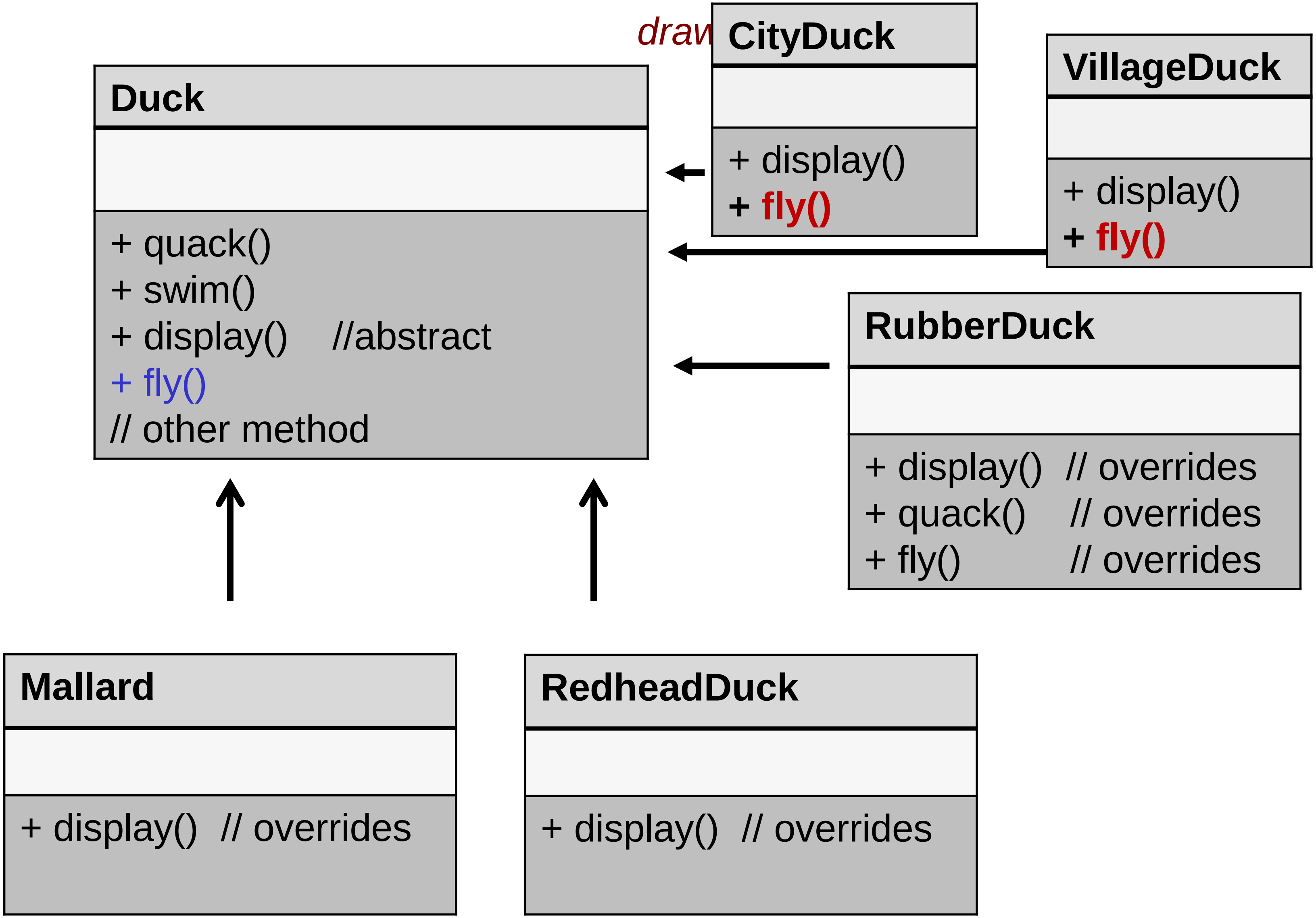
Inheritance:

drawbacks of

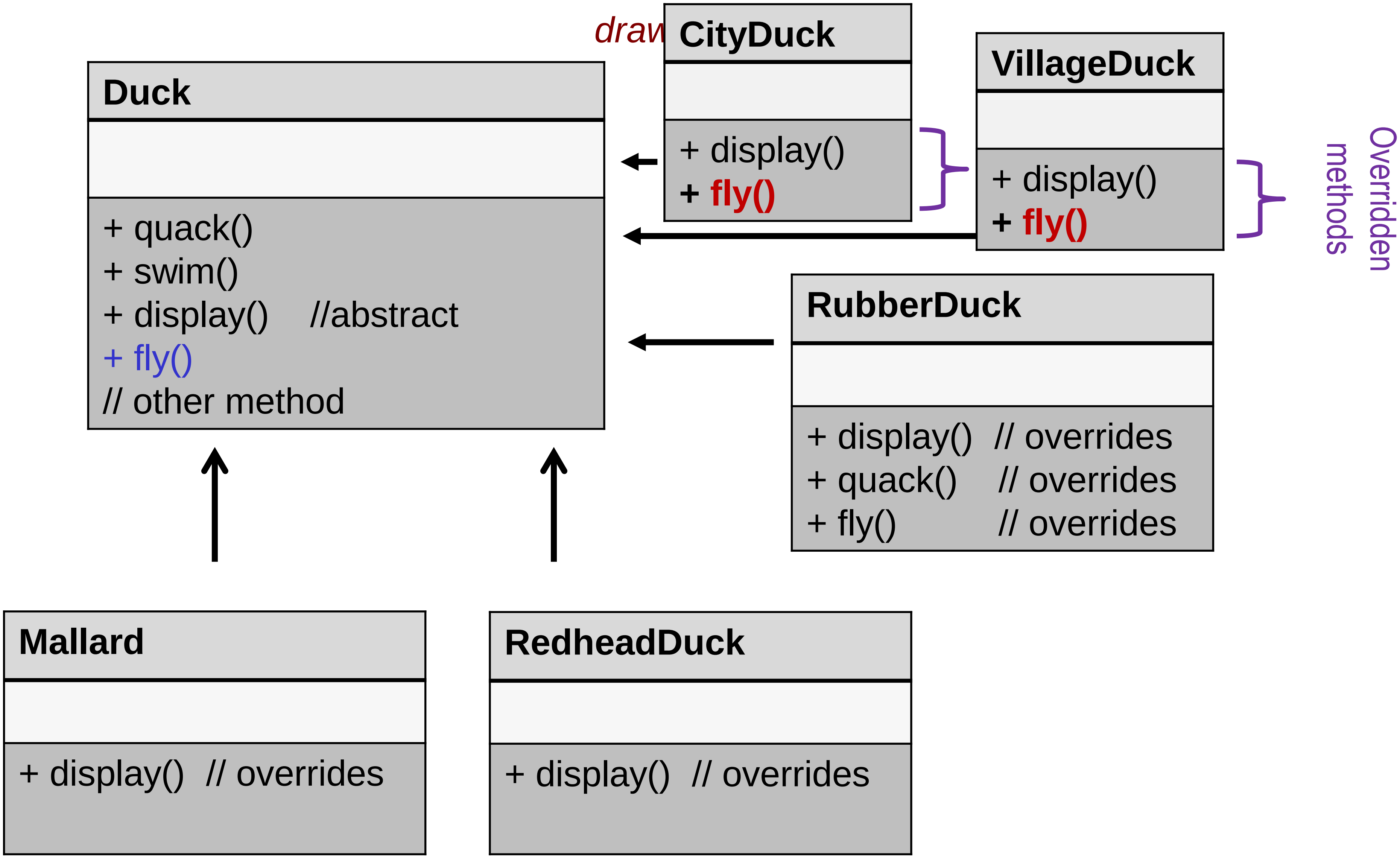


Inheritance:

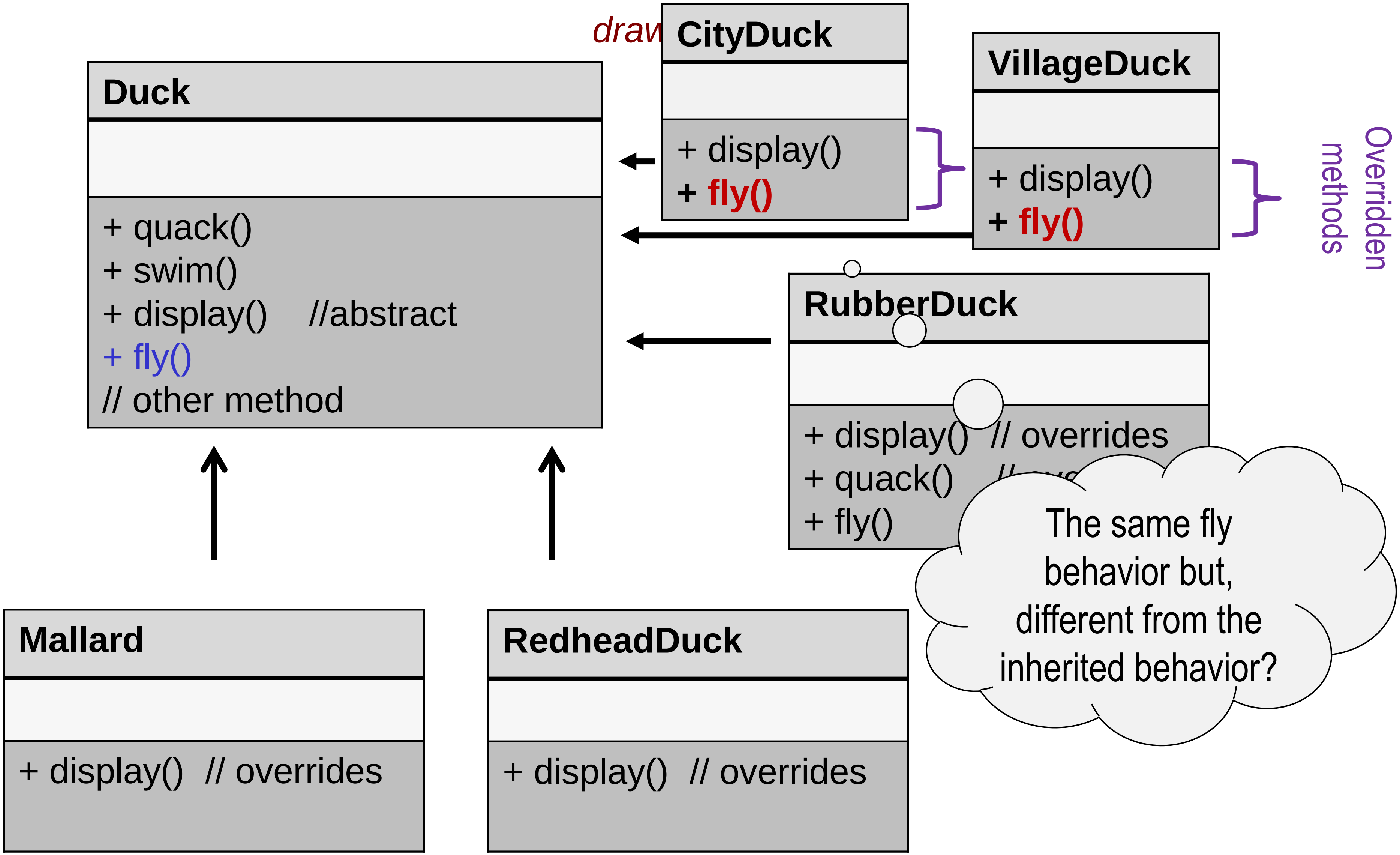
draw



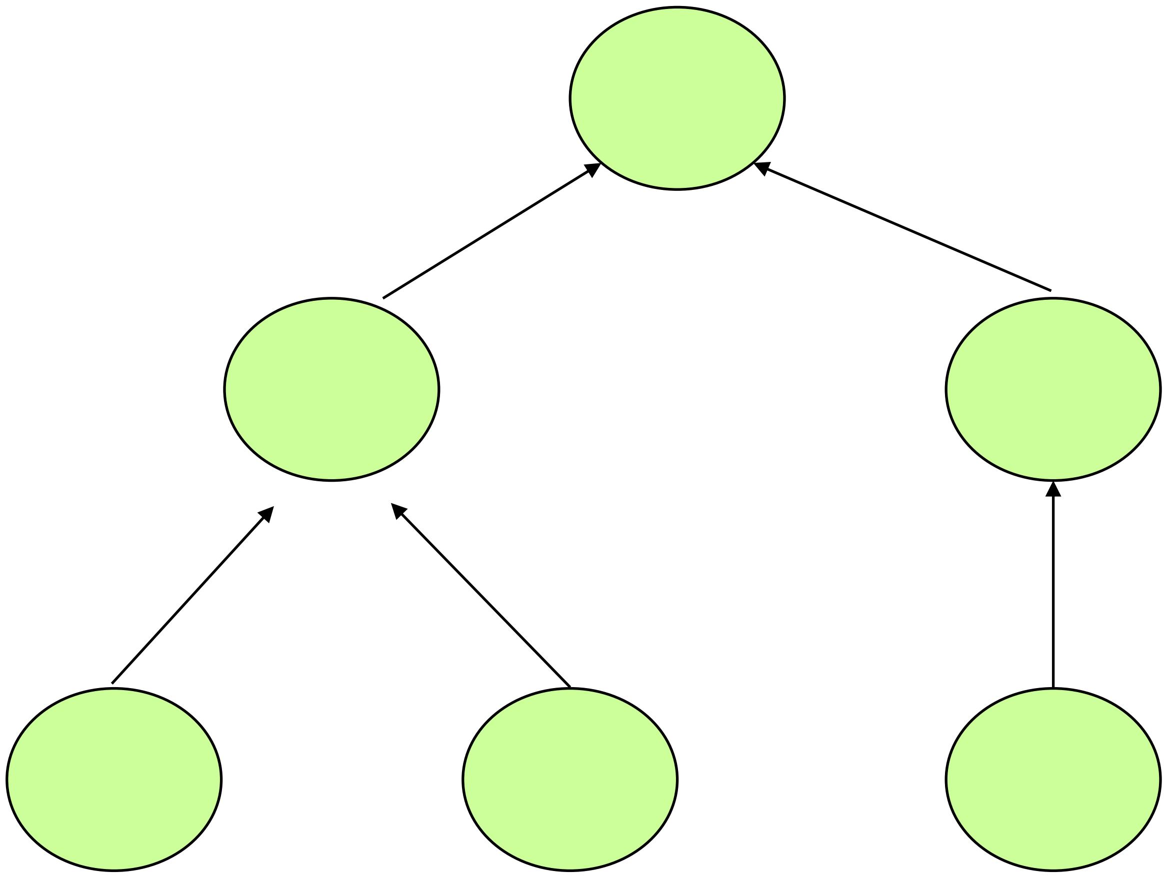
Inheritance:



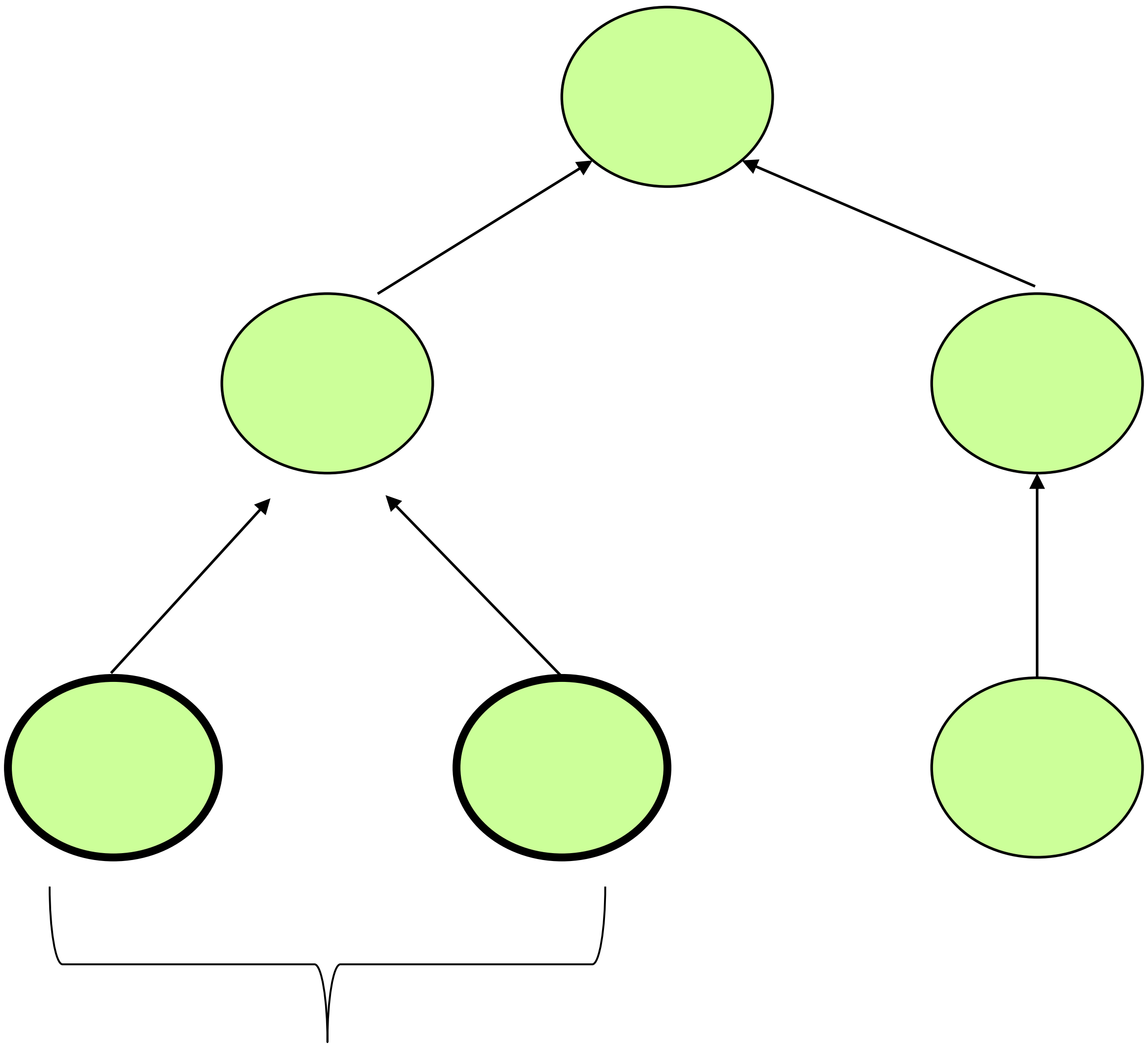
Inheritance:



Problem with (Single) Inheritance

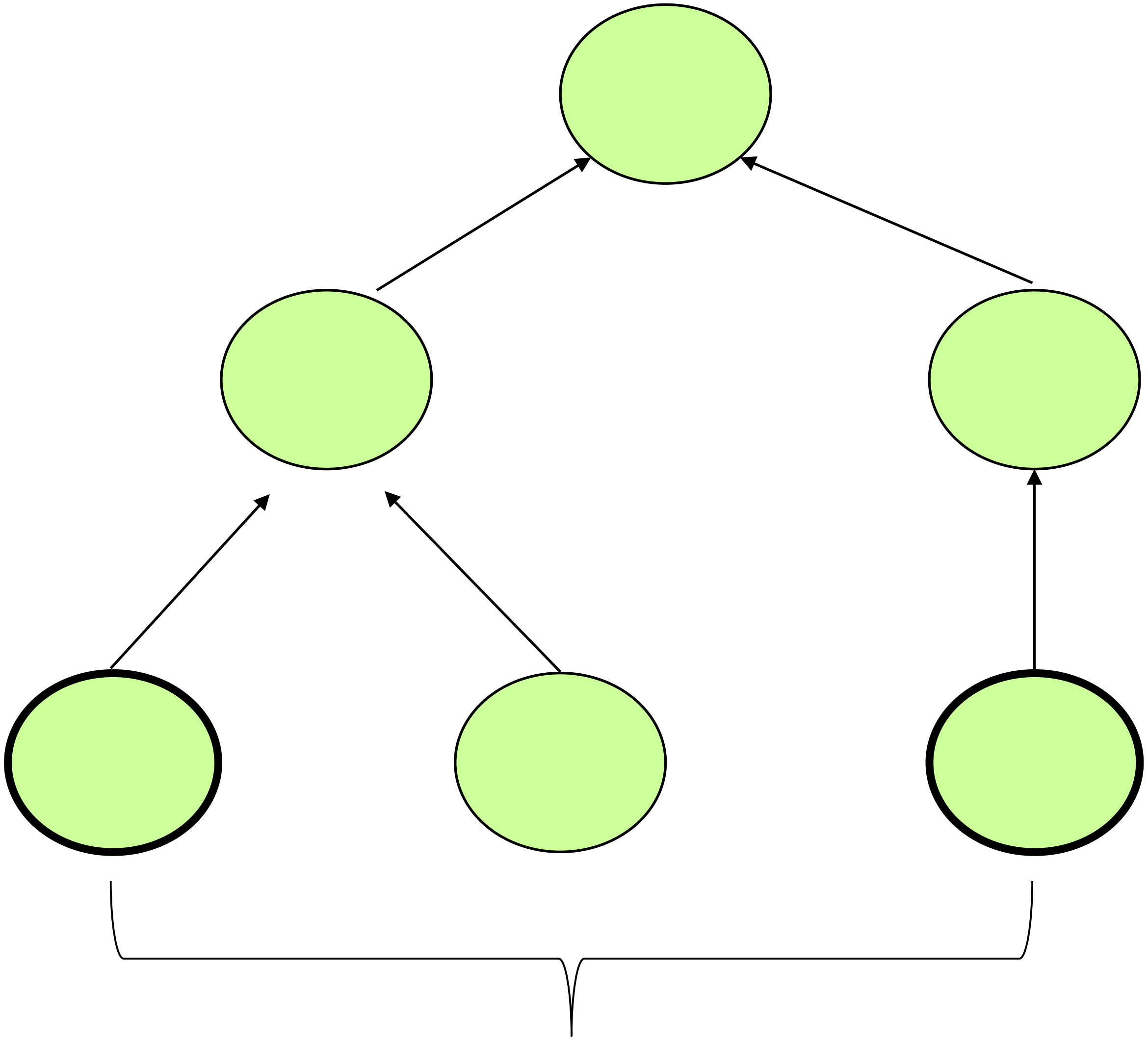


Problem with (Single) Inheritance



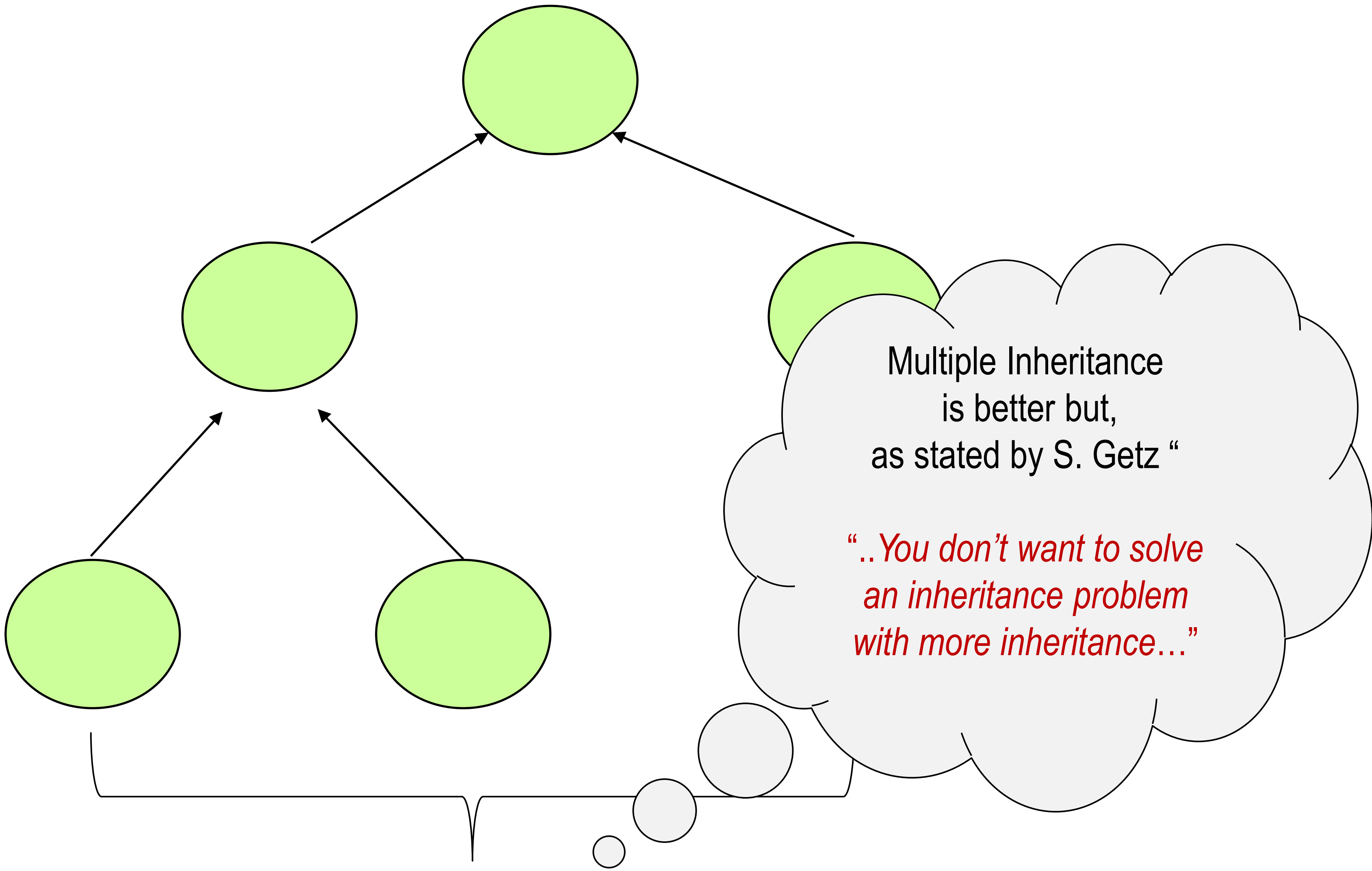
Sharing behavior *horizontally* is problematic in Single Inheritance

Problem with (Single) Inheritance



Sharing behavior *horizontally* is
problematic in Single Inheritance

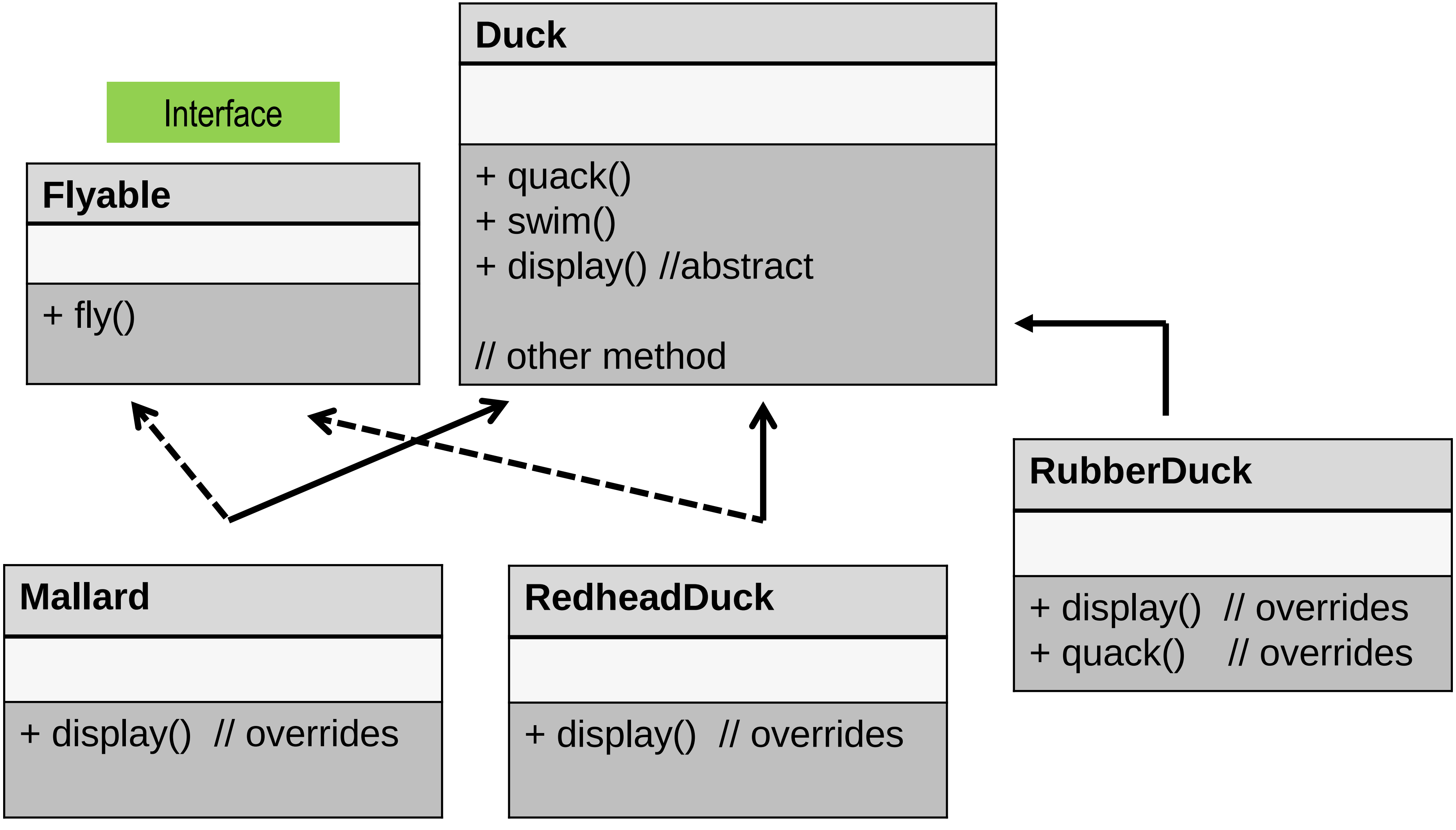
Problem with (Single) Inheritance



Sharing behavior *horizontally* is
problematic in Single Inheritance

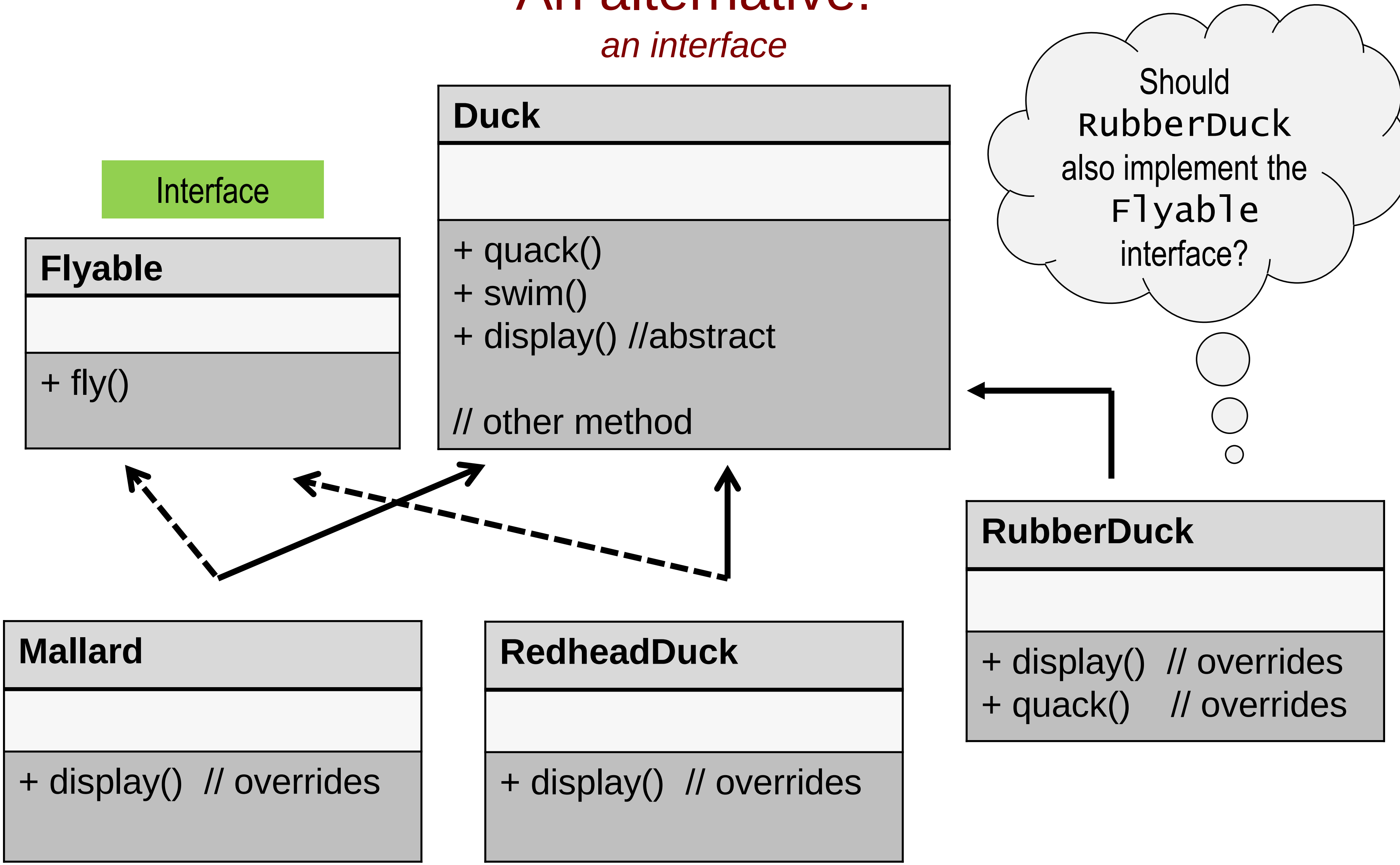
An alternative:

an interface

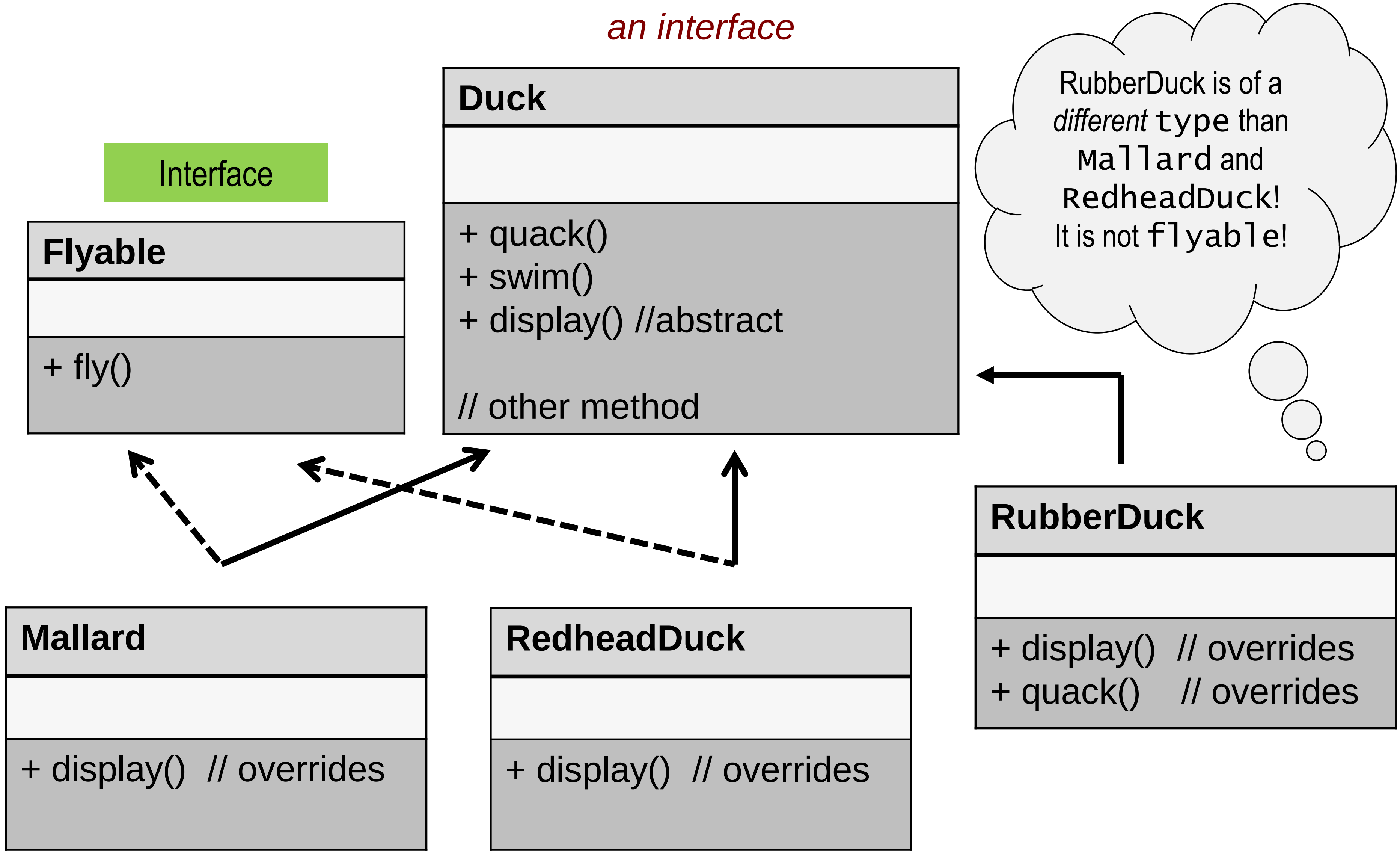


An alternative:

an interface

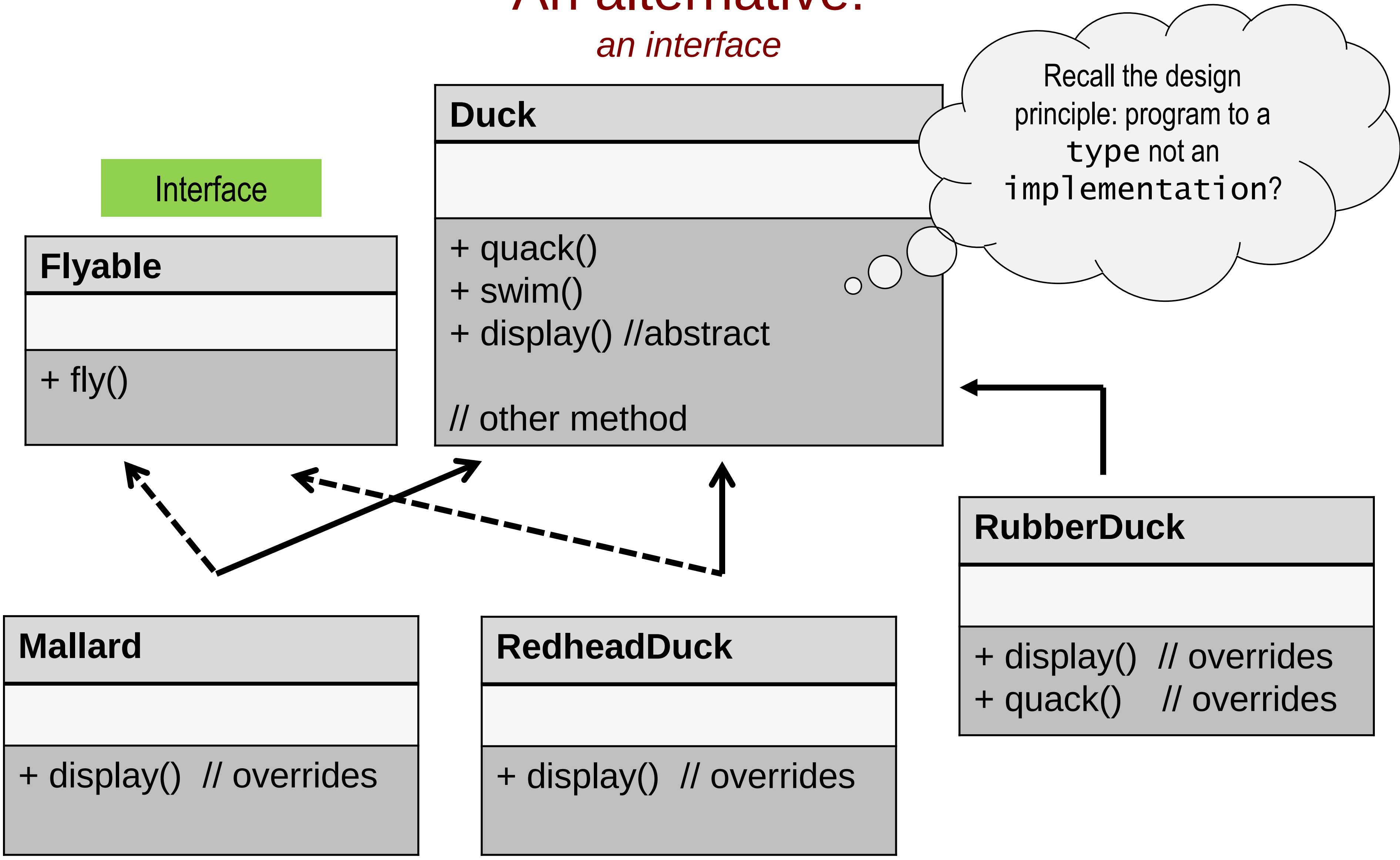


An alternative:
an interface



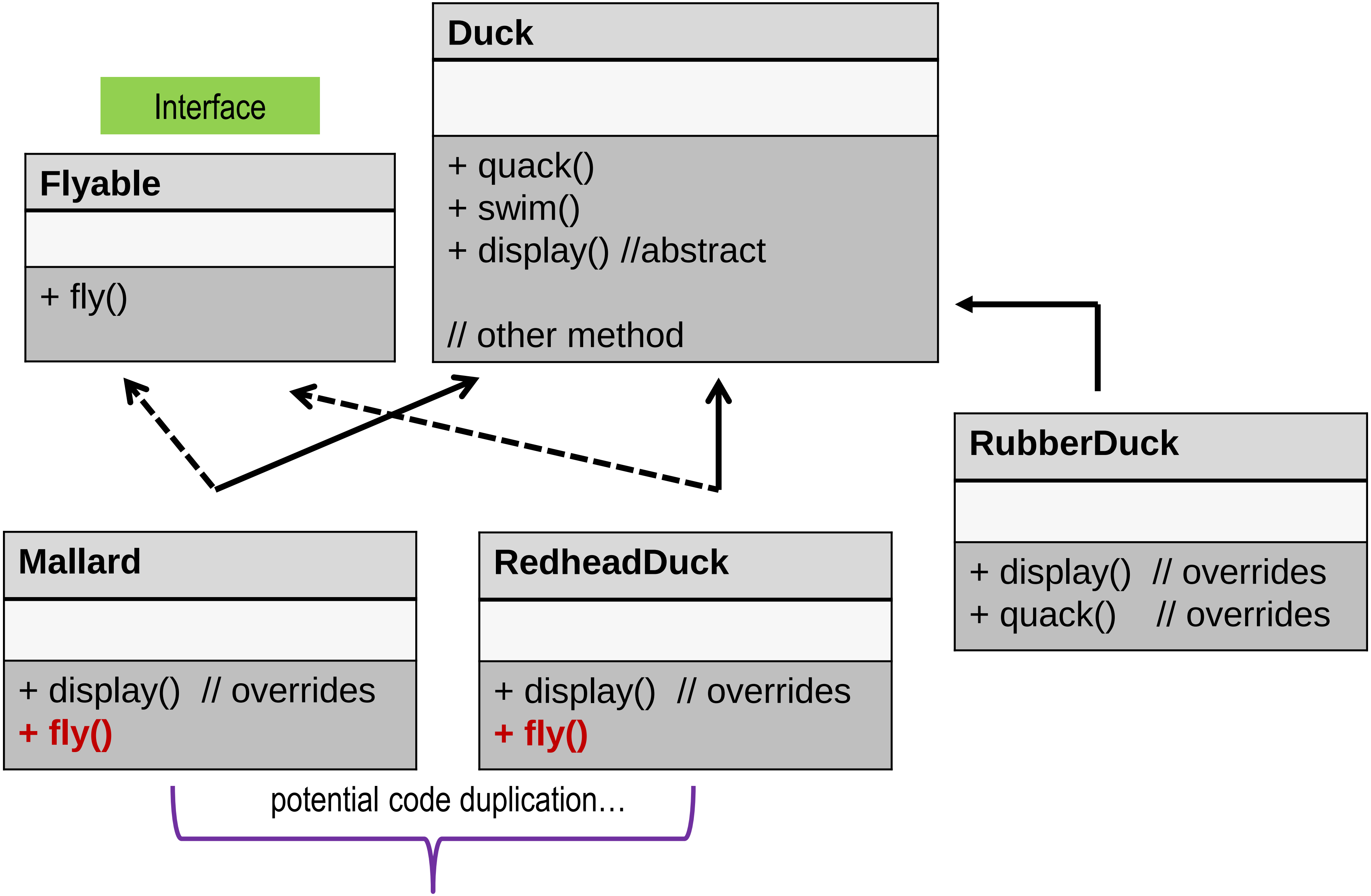
An alternative:

an interface



An alternative:

an interface



Core Design Principle

- Separate what changes from what stays the same. This is a core design principle. Recall *Abstraction by Parameterization*. The use of variables allow us to write logically structured code that operates on different variables.

- We can do the same thing with behaviors. Identify the behaviors of the objects that vary and separate them… pull them out.

- Encapsulate each behavior in a different class. Turn the behavior or the algorithm into an object.

- In our example, the behaviors that can vary are:
  - how ducks fly, and
  - how ducks quack.

Core Design Principle

- Separate what changes from what stays the same. This is a core design principle. Recall *Abstraction by Parameterization*. The use of variables allow us to write logically structured code that operates on different variables.

- We can do the same thing with behaviors. Identify the behaviors of the objects that vary and separate them… pull them out.

- Encapsulate each behavior in a different class. Turn the behavior or the algorithm into an object.

- In our example, the behaviors that can vary are:
    - how ducks fly, and
    - how ducks quack.

Core Design Principle

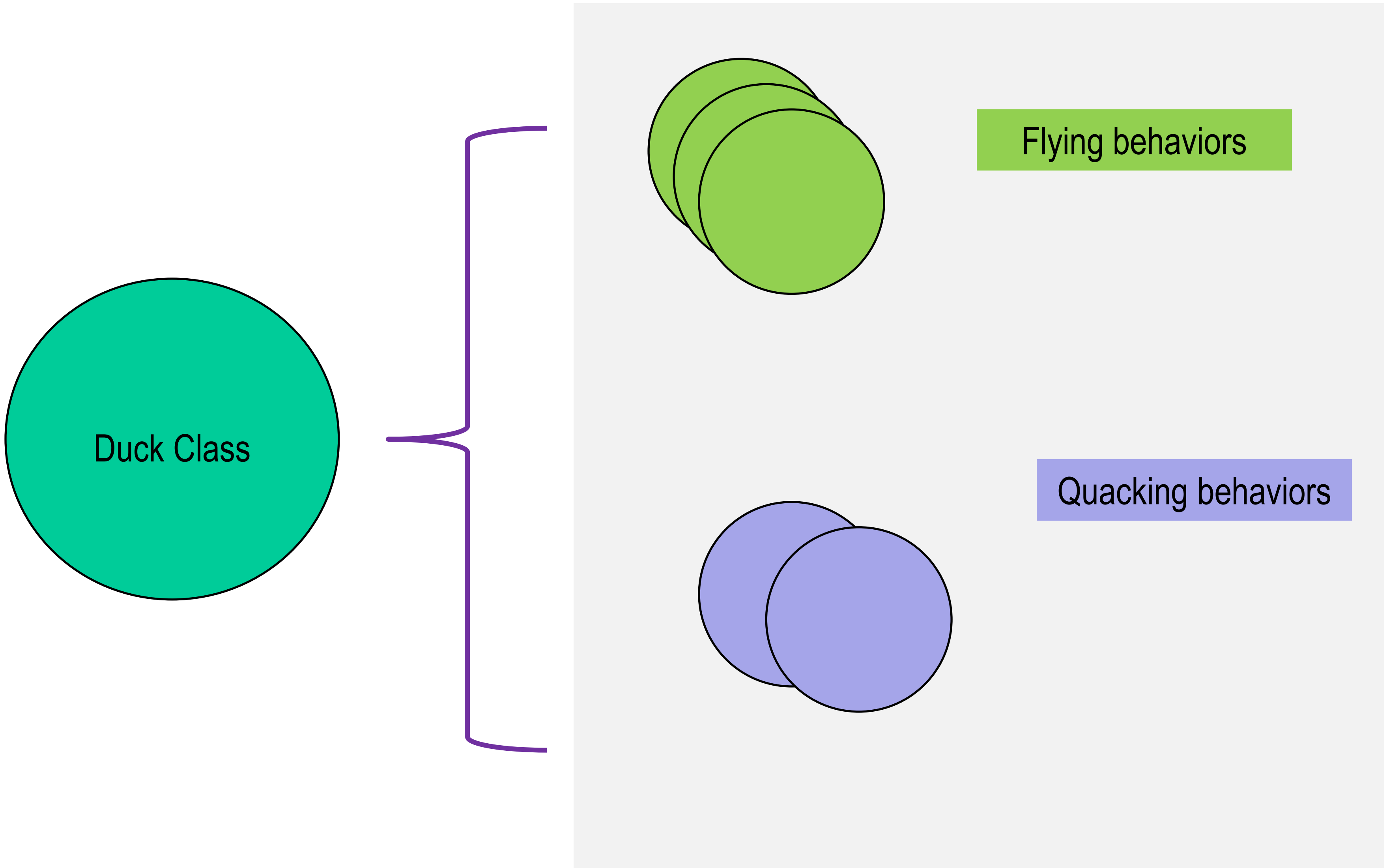
- Separate what changes from what stays the same. This is a core design principle. Recall *Abstraction by Parameterization*. The use of variables allow us to write logically structured code that operates on different variables.

- We can do the same thing with behaviors. Identify the behaviors of the objects that vary and separate them… pull them out.

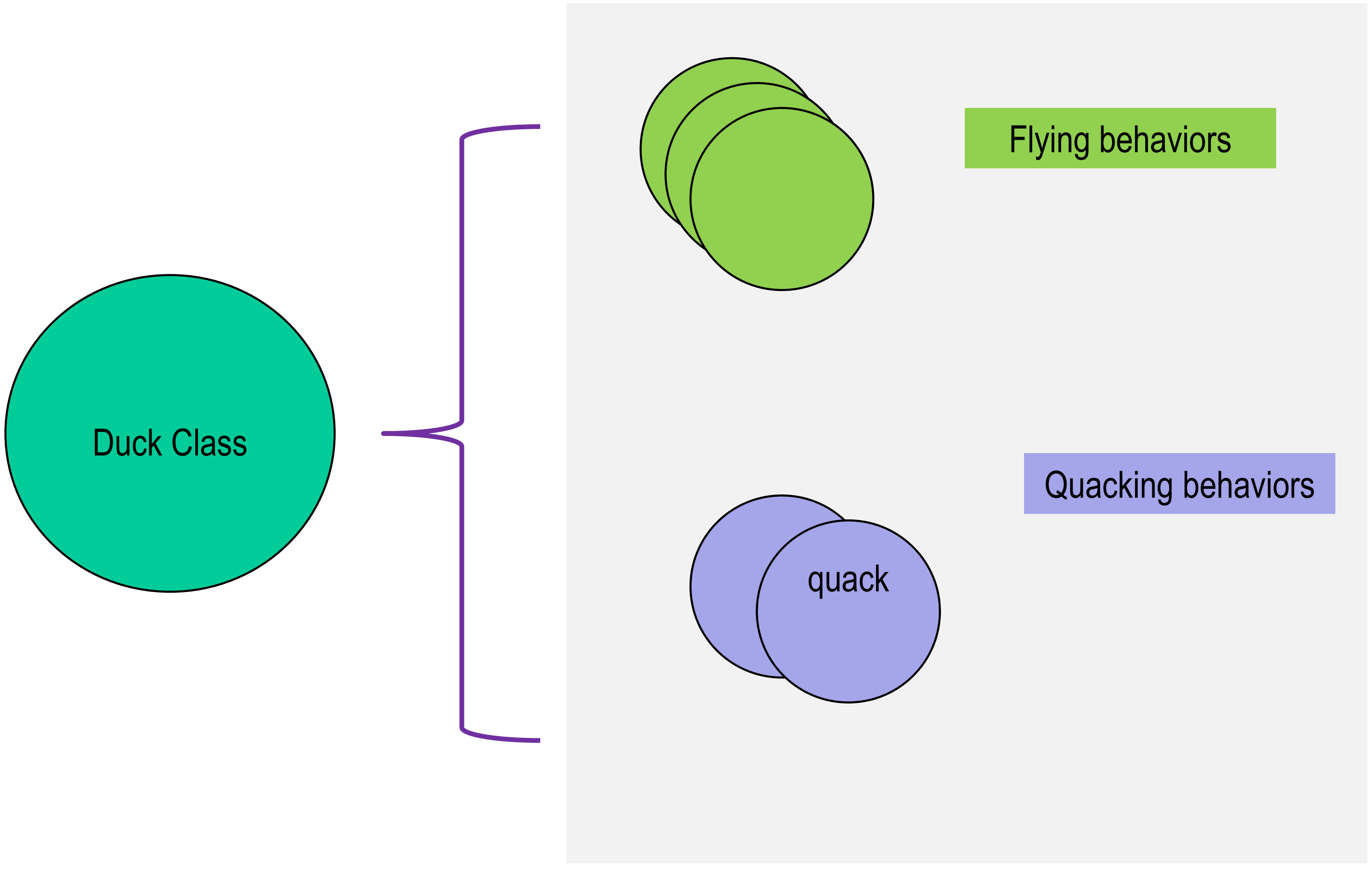
- Encapsulate each behavior in a different class. Turn the behavior or the algorithm into an object.

- In our example, the behaviors that can vary are:
  - how ducks fly, and
  - how ducks quack.

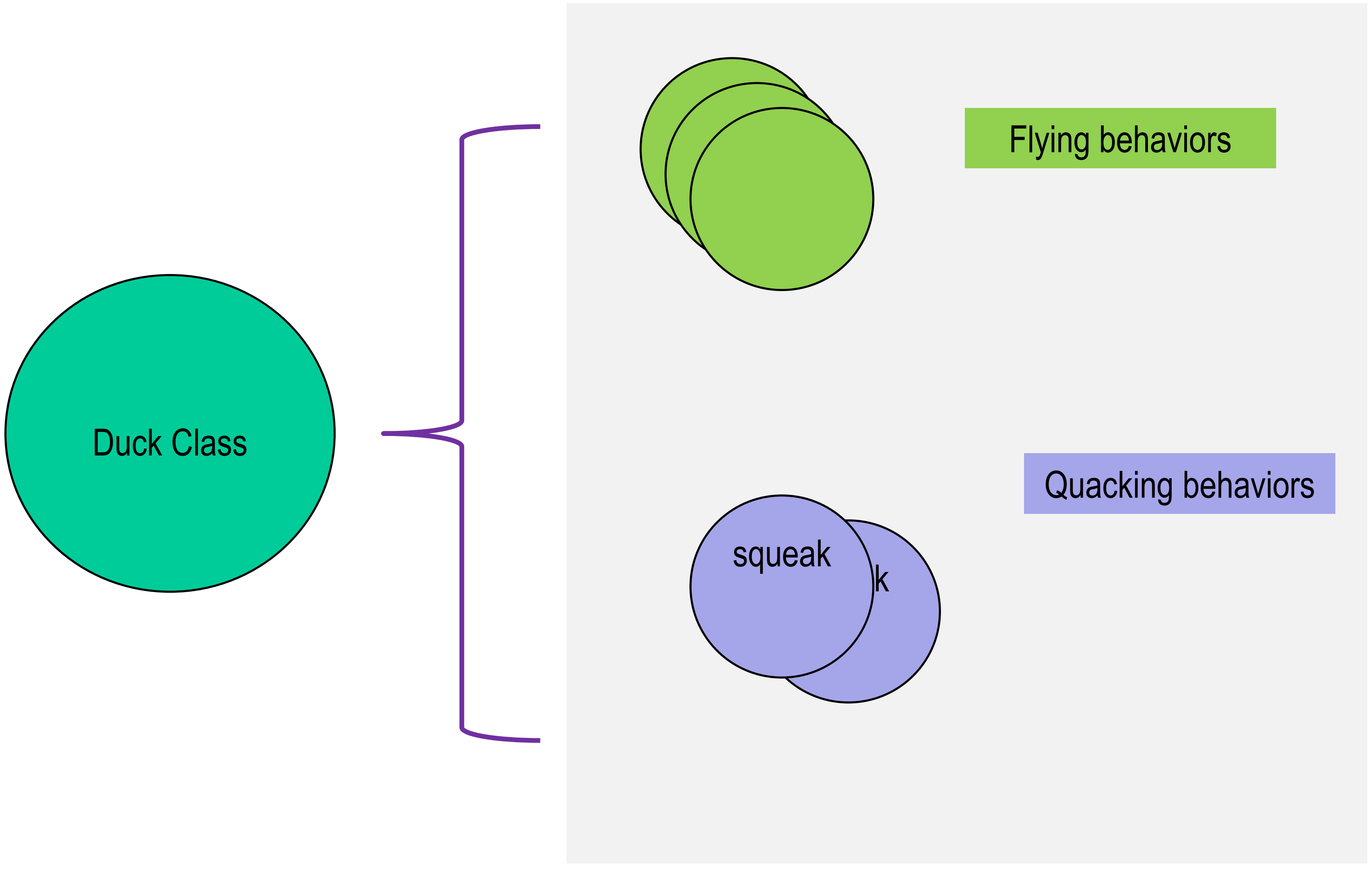
Duck class revisited



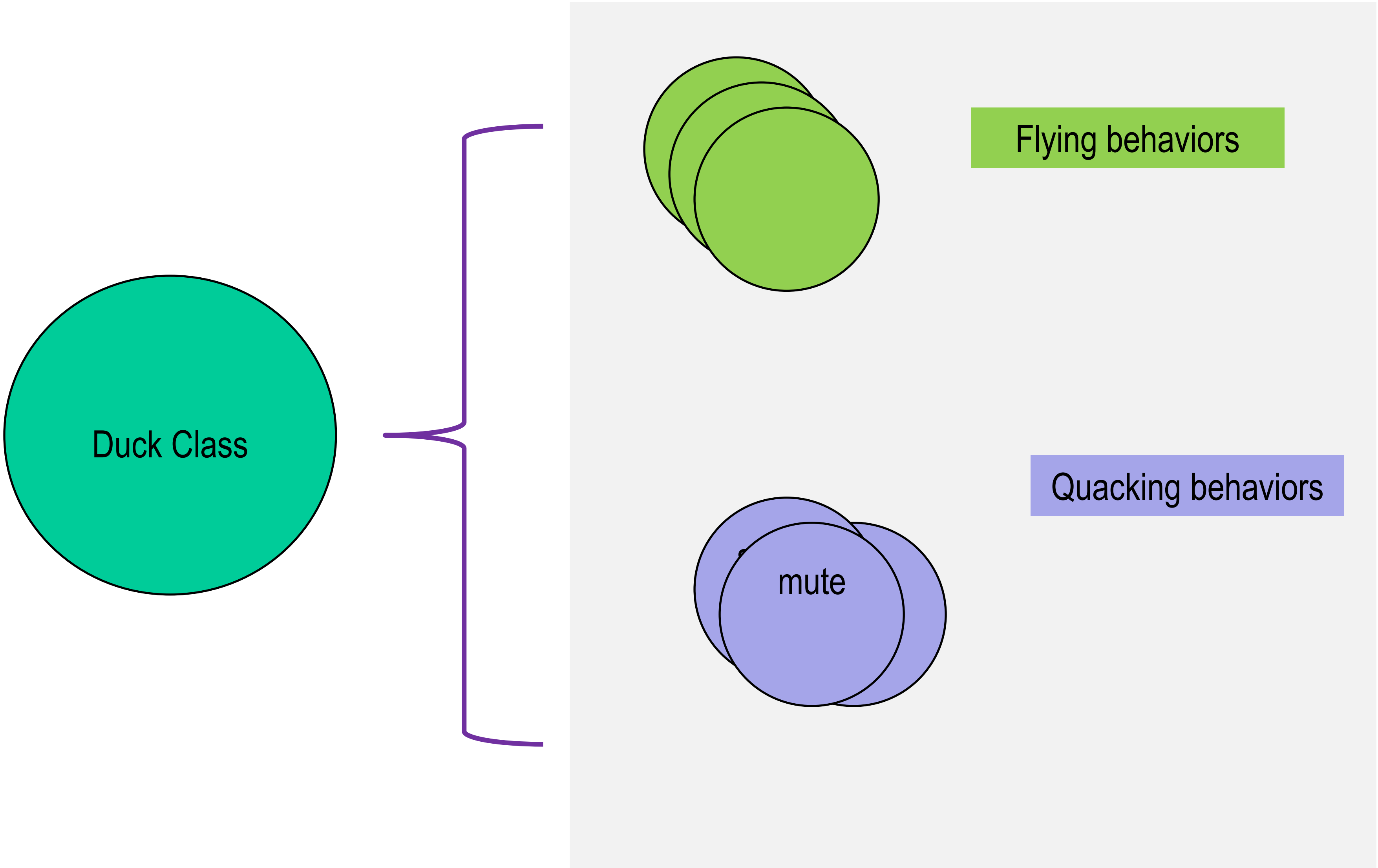
Duck class revisited



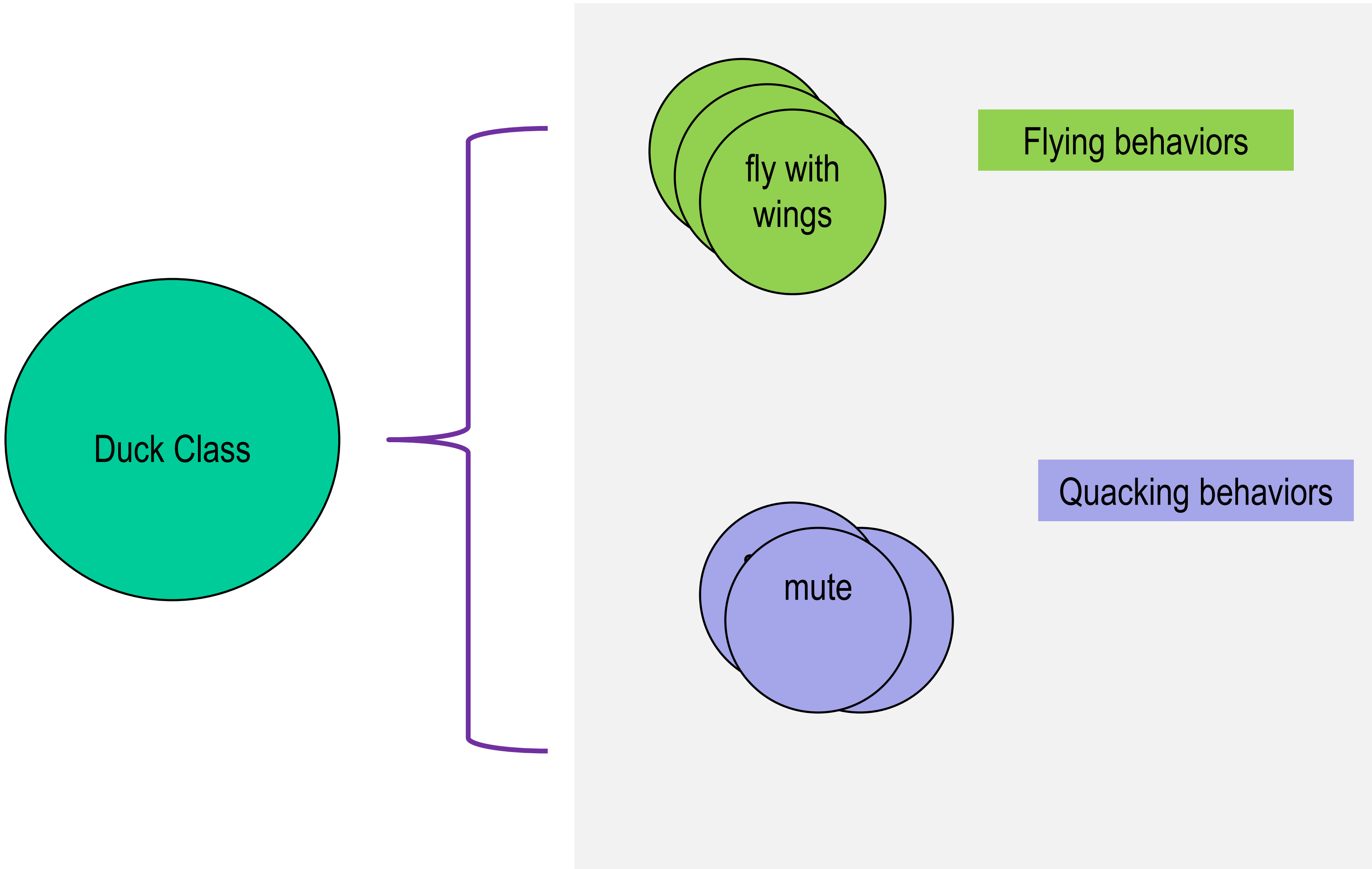
Duck class revisited



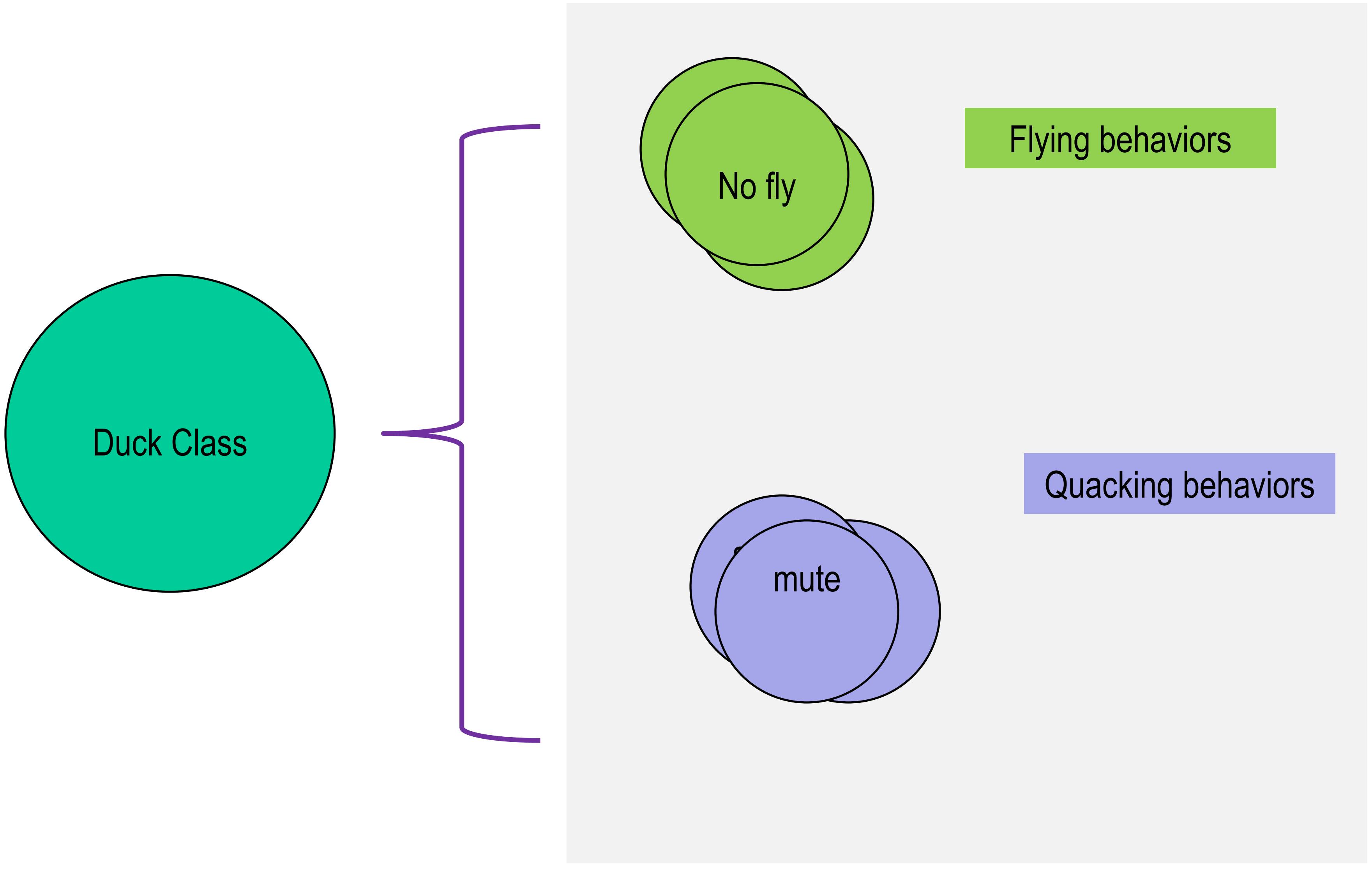
Duck class revisited



Duck class revisited

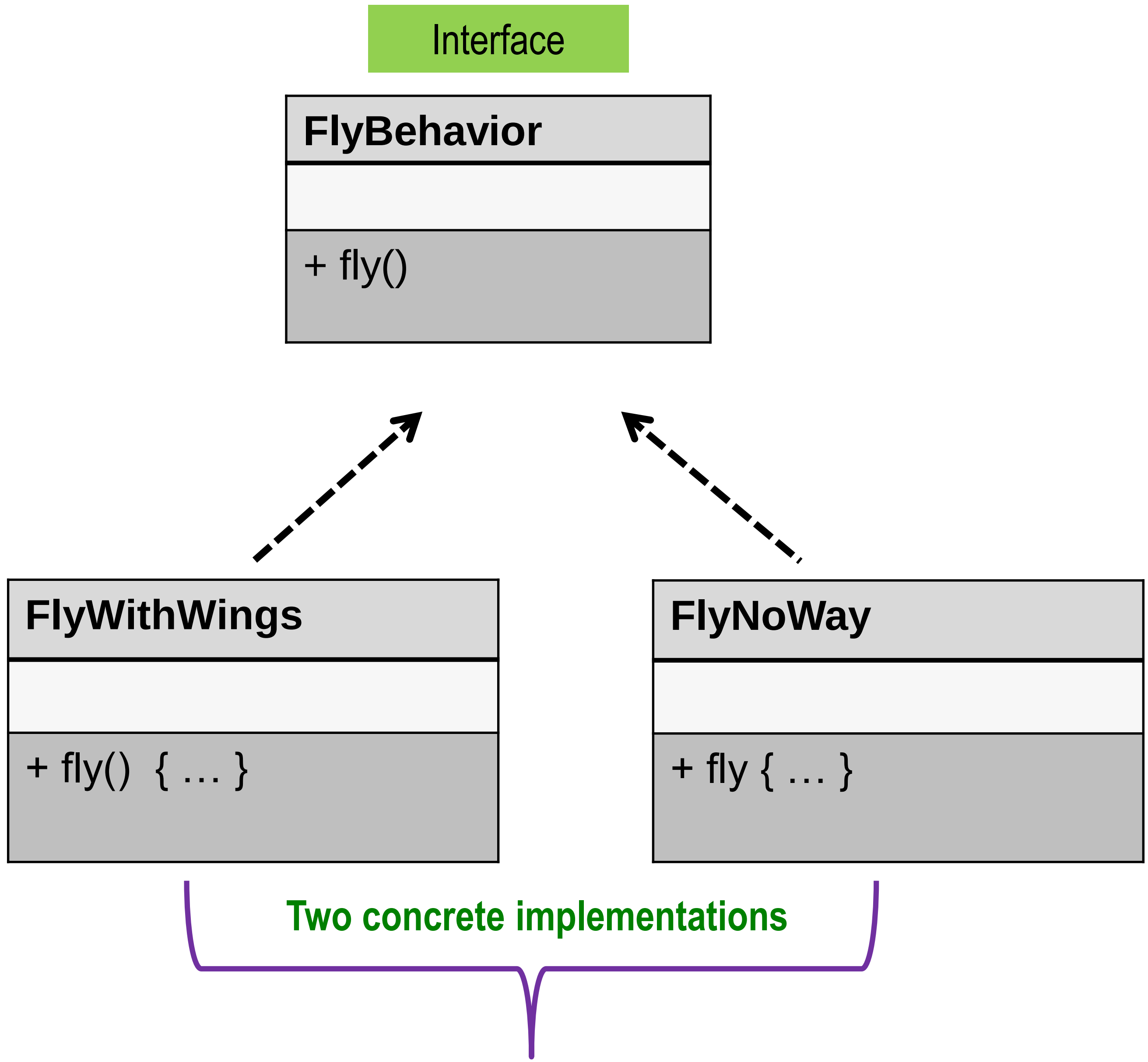


Duck class revisited



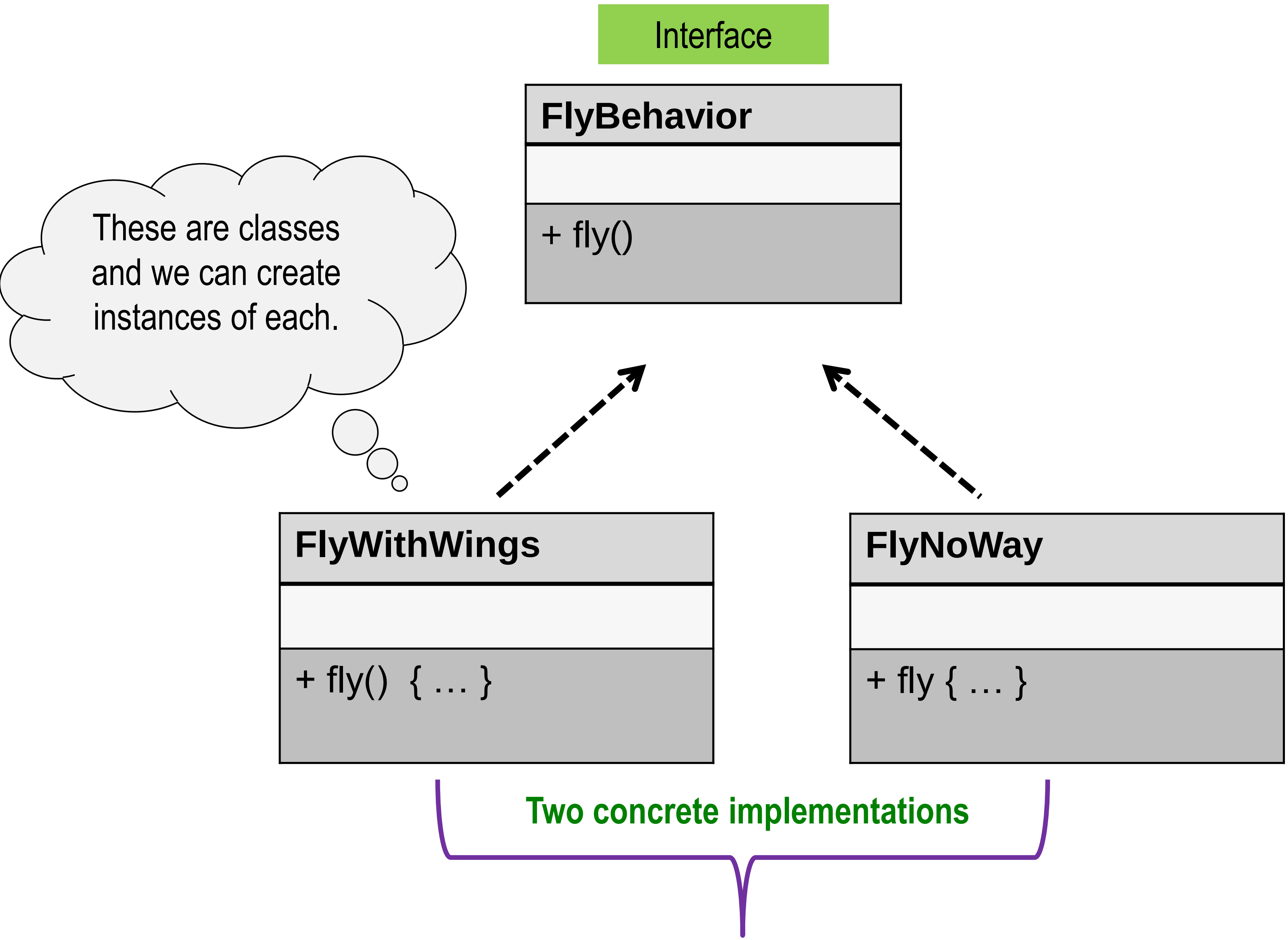
An alternative:

an interface



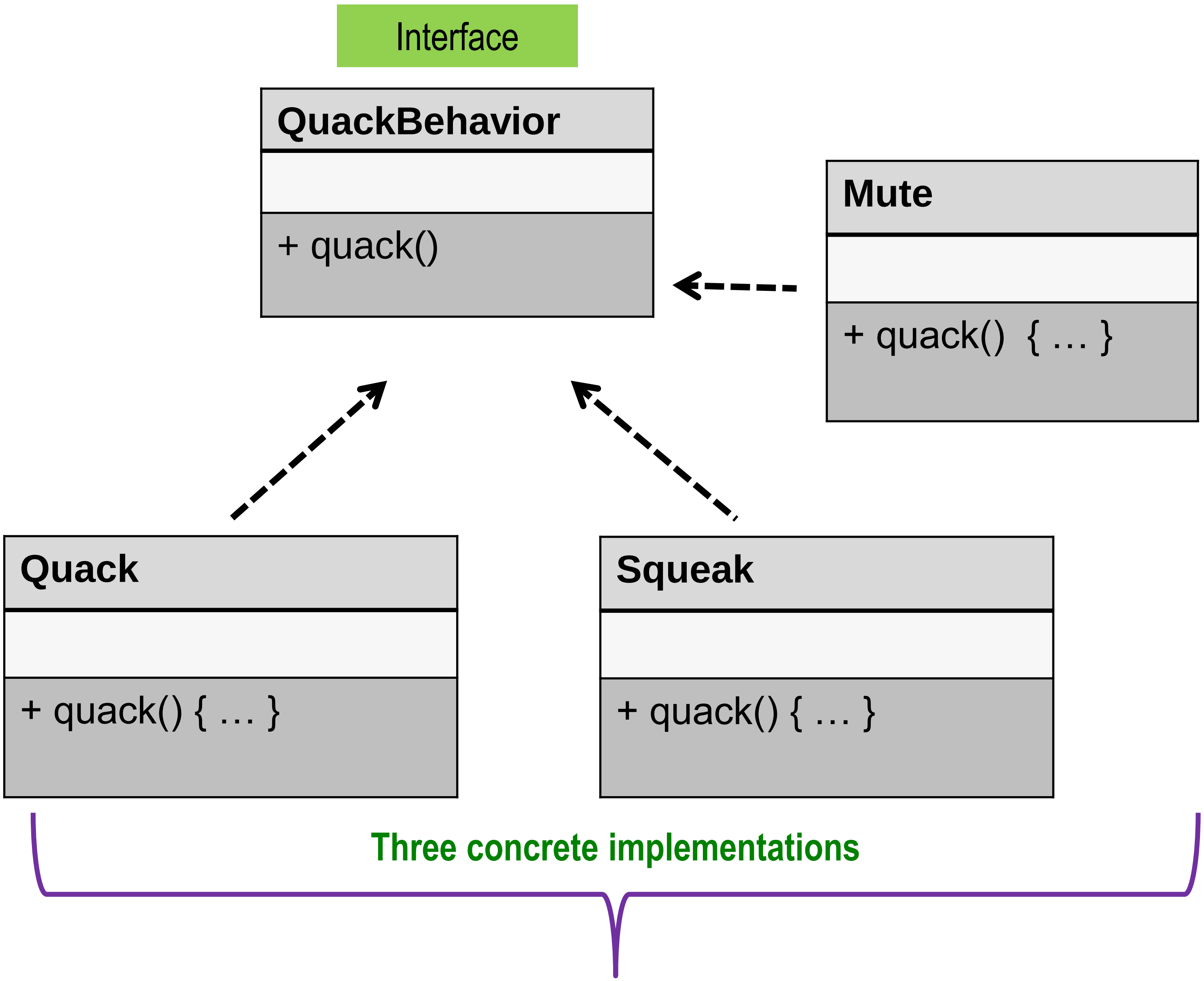
An alternative:

an interface



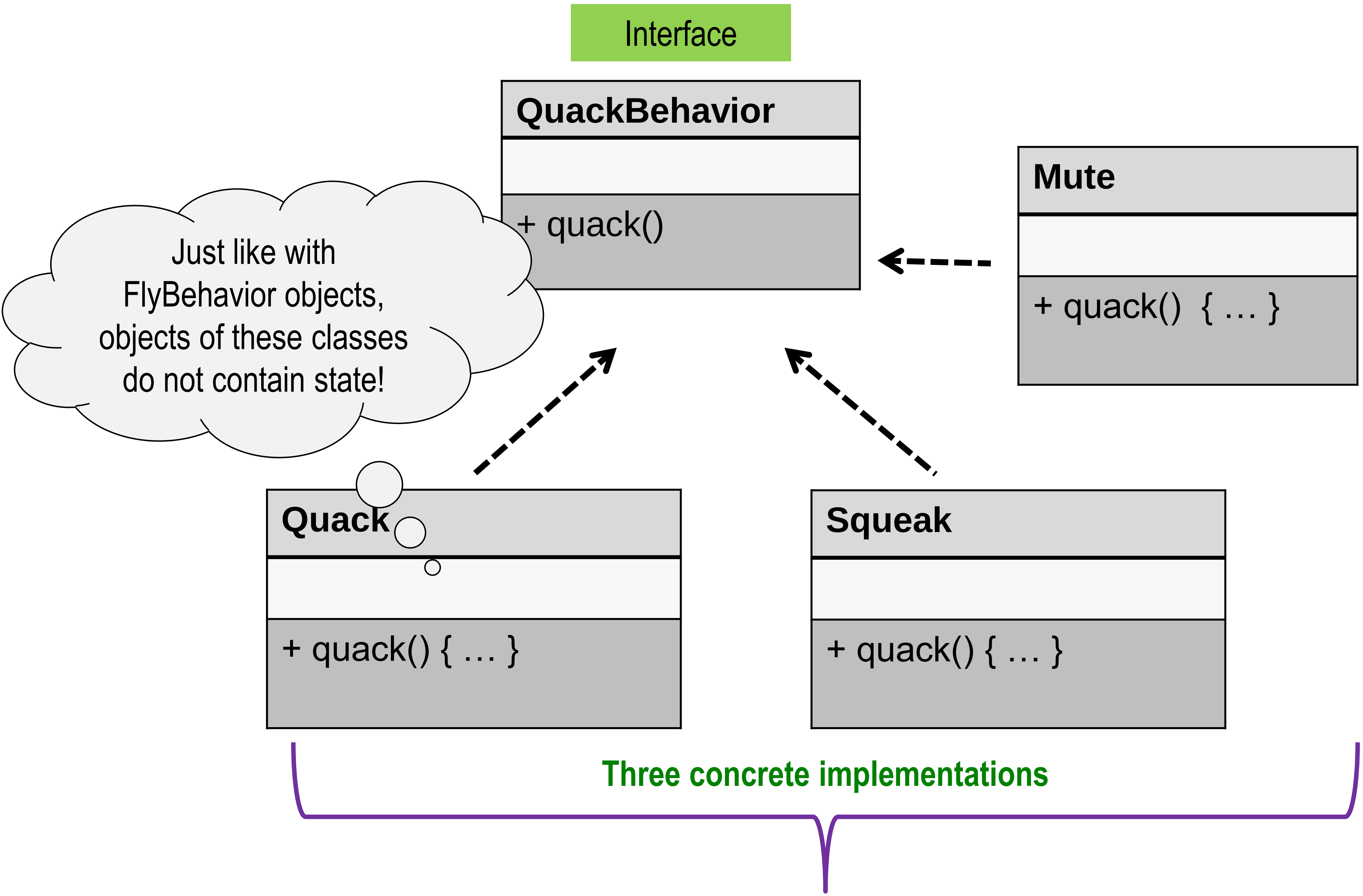
An alternative:

an interface



An alternative:

an interface



Duck class revisited…

Duck Abstract Class

+ **Quack()**
+ swim()
+ display() //abstract
+ **Fly()**

// other method

Duck class revisited…

Duck Abstract Class
- FlyBehavior flyBehavior **- QuackBehavior quackBehavior**

+ perform**Quack()**
+ swim()
+ display() //abstract
+ perform**Fly()**
 	
// other method

Duck class revisited…

Duck Abstract Class

- FlyBehavior flyBehavior
- QuackBehavior quackBehavior

+ perform**Quack()**
+ swim()
+ display() //abstract
+ perform**Fly()**

// other method

Ducks now ***have*** `flyBehavior` and `quackBehavior`!

Duck class revisited…

Duck Abstract Class

- FlyBehavior flyBehavior
- QuackBehavior quackBehavior

+ perform**Quack()**
+ swim()
+ display() //abstract
+ perform**Fly()**
// other method

```
public abstract class Duck {
   protected FlyBehavior flyBehavior;
   protected QuackBehavior quackBehavior;
   ...
   public void performQuack() {
      quackBehavior.quack();
   }
   public void performFly() {
       flyBehavior.fly();
   }
}
```

Duck class revisited…

Duck Abstract Class

- FlyBehavior flyBehavior
- QuackBehavior quackBehavior

+ perform**Quack()**
+ swim()
+ display() //abstract
+ perform**Fly()**
// other method

```
public abstract class Duck {
   protected FlyBehavior flyBehavior;
   protected QuackBehavior quackBehavior;
   ...
   public void performQuack() {
      quackBehavior.quack();
   }
   public void performFly() {
       flyBehavior.fly();
   }
}
```

Duck class revisited…

Duck Abstract Class
- FlyBehavior flyBehavior **- QuackBehavior quackBehavior**

+ perform**Quack()**
+ swim()
+ display() //abstract
+ perform**Fly()**
// other method

```
public abstract class Duck {
   protected FlyBehavior flyBehavior;
   protected QuackBehavior quackBehavior;
   ...
   public void performQuack() {
      quackBehavior.quack();
   }
   public void performFly() {
       flyBehavior.fly();
   }
}
```

Duck class revisited…

Duck Abstract Class

- FlyBehavior flyBehavior
- QuackBehavior quackBehavior

+ perform**Quack()**
+ swim()
+ display() //abstract
+ perform**Fly()**
// other method

```
public abstract class Duck {
   protected FlyBehavior flyBehavior;
   protected QuackBehavior quackBehavior;
   ...
   public void performQuack() {
      quackBehavior.quack();
   }
   public void performFly() {
       flyBehavior.fly();
   }
}
```

Duck class revisited…

Duck

- FlyBehavior flyBehavior
- QuackBehavior quackBehavior

+ perform**Quack()**
+ swim()
+ display() //abstract
+ perform**Fly()**
// other method

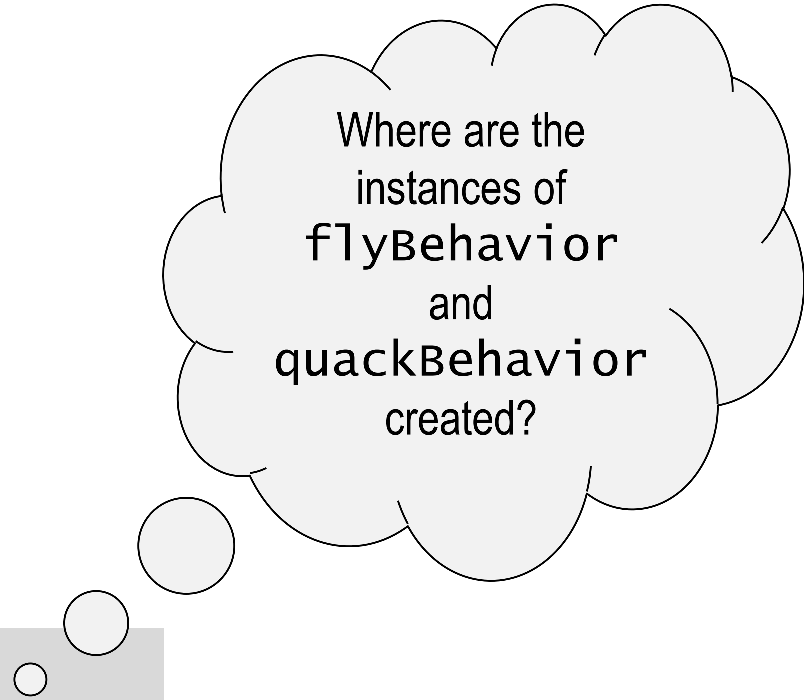
```
public abstract class Duck {
   protected FlyBehavior flyBehavior;
   protected QuackBehavior quackBehavior;
   ...
   public void performQuack() {
      quackBehavior.quack();
   }
   public void performFly() {
       flyBehavior.fly();
   }
}
```

Where are the instances of `flyBehavior` and `quackBehavior` created?

Duck class revisited…

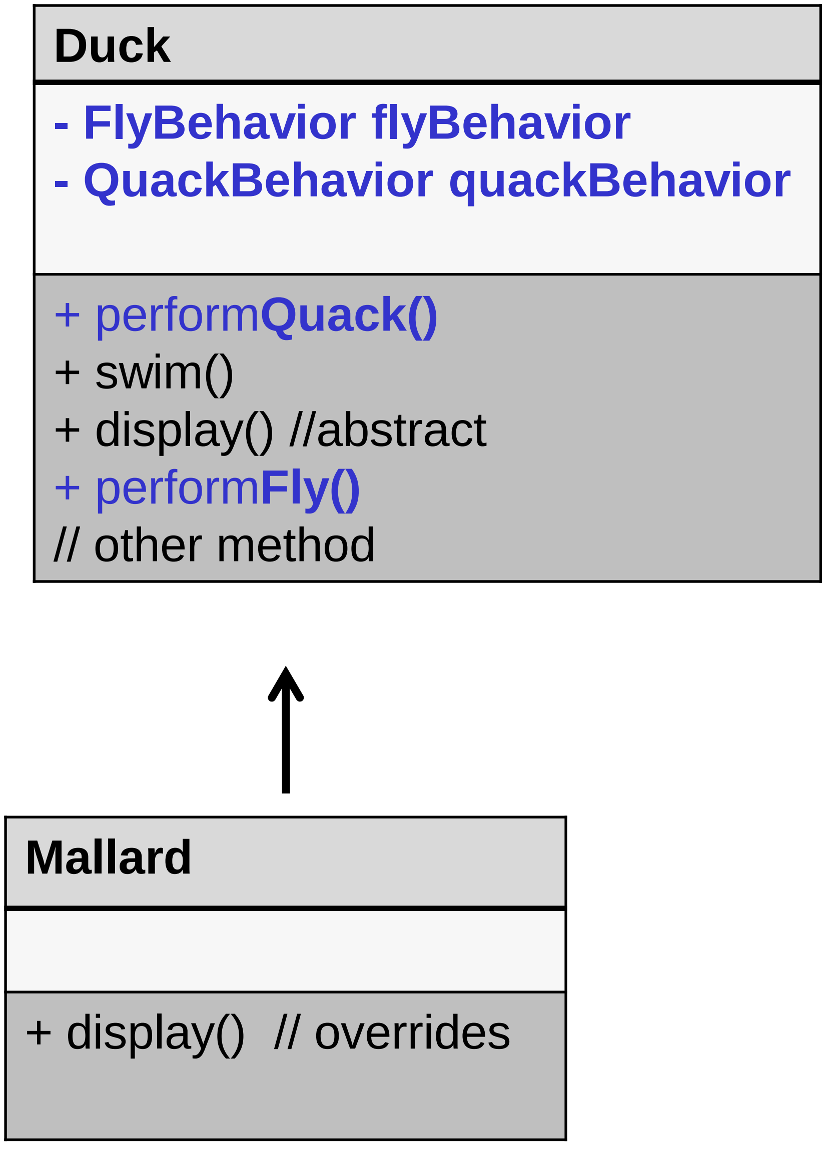
Duck
- FlyBehavior flyBehavior **- QuackBehavior quackBehavior**

+ perform**Quack()**
+ swim()
+ display() //abstract
+ perform**Fly()**
// other method




```
public abstract class Duck {
   protected FlyBehavior flyBehavior;
   protected QuackBehavior quackBehavior;
   ...
   public void performQuack() {
      quackBehavior.quack();
   }
   public void performFly() {
       flyBehavior.fly();
   }
}
```

Duck class revisited…



Duck class revisited…

Duck

- FlyBehavior flyBehavior
- QuackBehavior quackBehavior

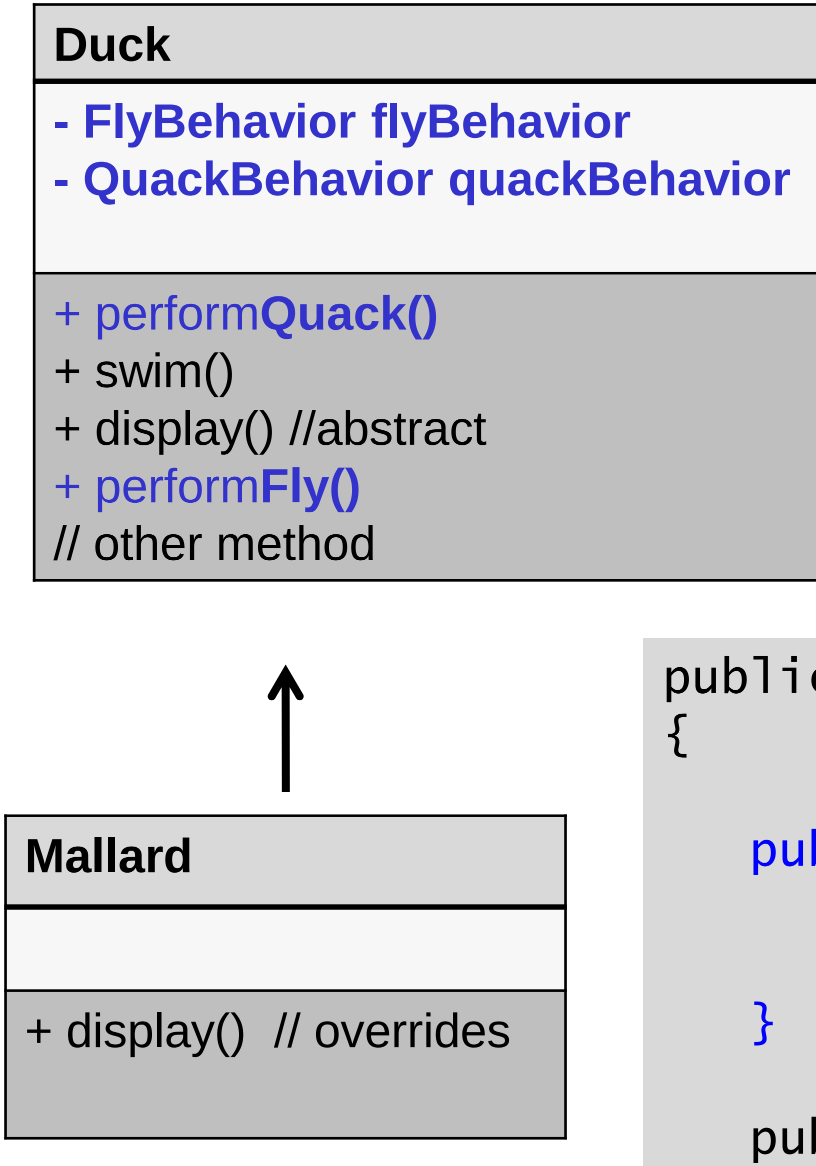
+ perform**Quack()**
+ swim()
+ display() //abstract
+ perform**Fly()**
// other method

Mallard

+ display() // overrides

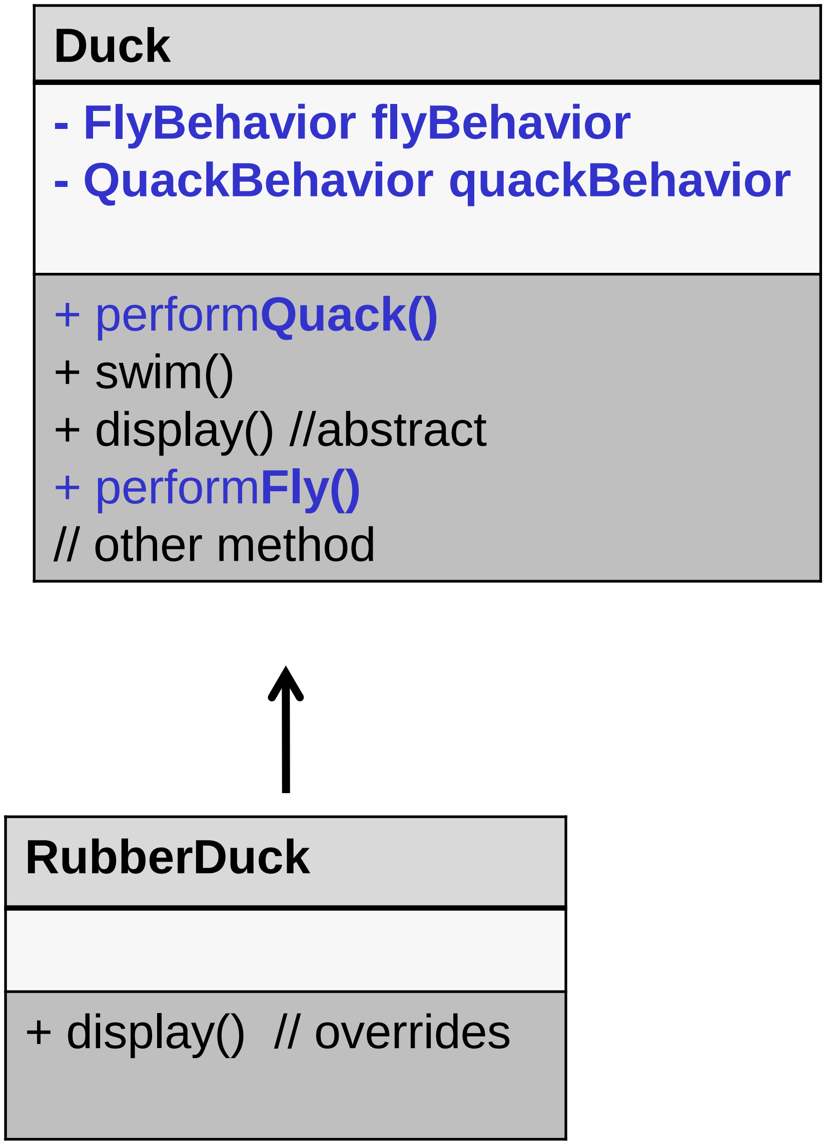
```
public class MallardDuck extends Duck
{

   public MallardDuck() {
      quackBehavior = new Quack();
      flyBehavior = FlyWithWings();
   }

   public display() { ... }
}
```

Duck class revisited…

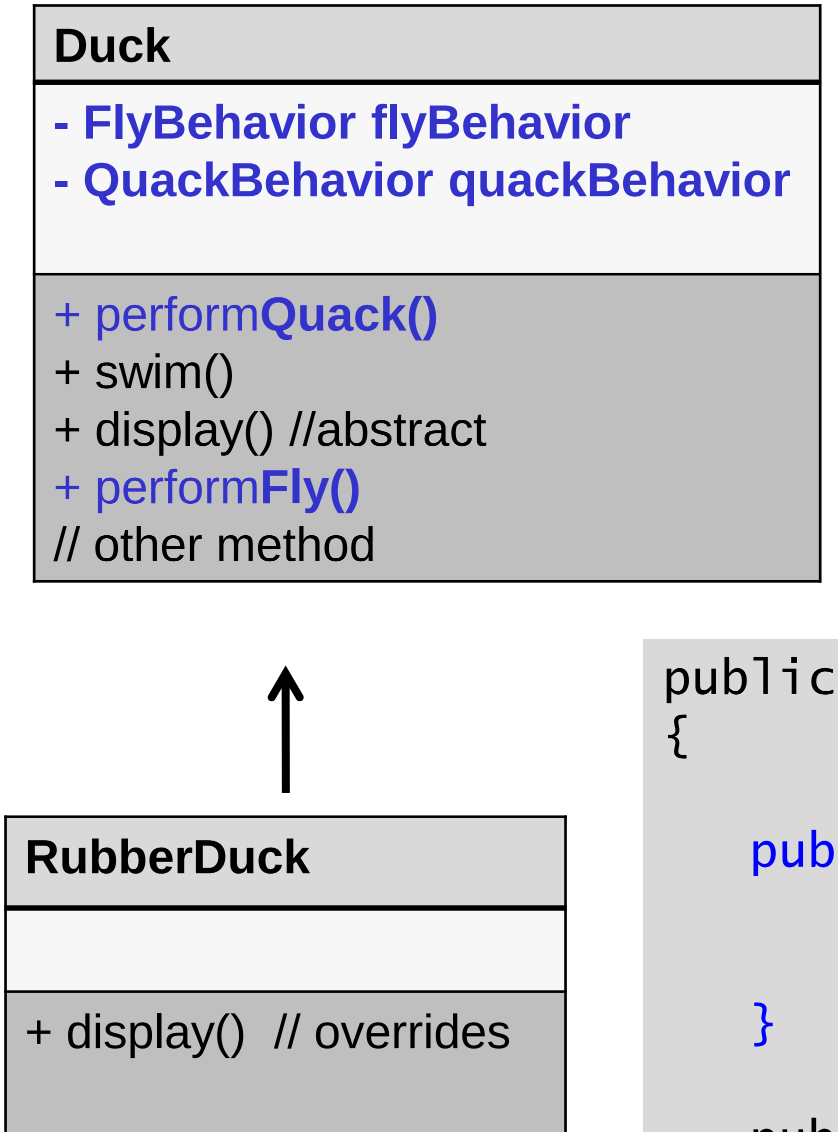



```
public class MallardDuck extends Duck
{

   public MallardDuck() {
      quackBehavior = new Quack();
      flyBehavior = new FlyWithWings();
   }

   public display() { ... }
}
```

Duck class revisited…



Duck class revisited…




```
public class RubberDuck extends Duck
{

   public RubberDuck() {
      quackBehavior = new Squeak();
      flyBehavior = new FlyNoFly();
   }

   public display() { ... }
}
```

Duck class revisited…

```java
public class DuckSimulator {

   public static void main( String[] a ) {
      Duck mallard = new MallardDuck();
      mallard.performQuack();
      mallard.performFly();

      Duck rubberDuckie = new RubberDuck();
      rubberDuckie.performQuack();
      rubberDuckie.performFly();
   }

}
```

Duck class revisited…

```
public class DuckSimulator {

   public static void main( String[] a ) {
      Duck mallard = new MallardDuck();
      mallard.performQuack();
      mallard.performFly();

      Duck rubberDuckie = new RubberDuck();
      rubberDuckie.performQuack();
      rubberDuckie.performFly();
   }

}
```

Duck class revisited…

```
public class DuckSimulator {

   public static void main( String[] a ) {
      Duck mallard = new MallardDuck();
      mallard.performQuack();
      mallard.performFly();

      Duck rubberDuckie = new RubberDuck();
      rubberDuckie.performQuack();
      rubberDuckie.performFly();
   }

}
```

Duck class revisited…

```
public class DuckSimulator {

   public static void main( String[] a ) {
      Duck mallard = new MallardDuck();
      mallard.performQuack();
      mallard.performFly();

      Duck rubberDuckie = new RubberDuck();
      rubberDuckie.performQuack();
      rubberDuckie.performFly();
   }

}
```

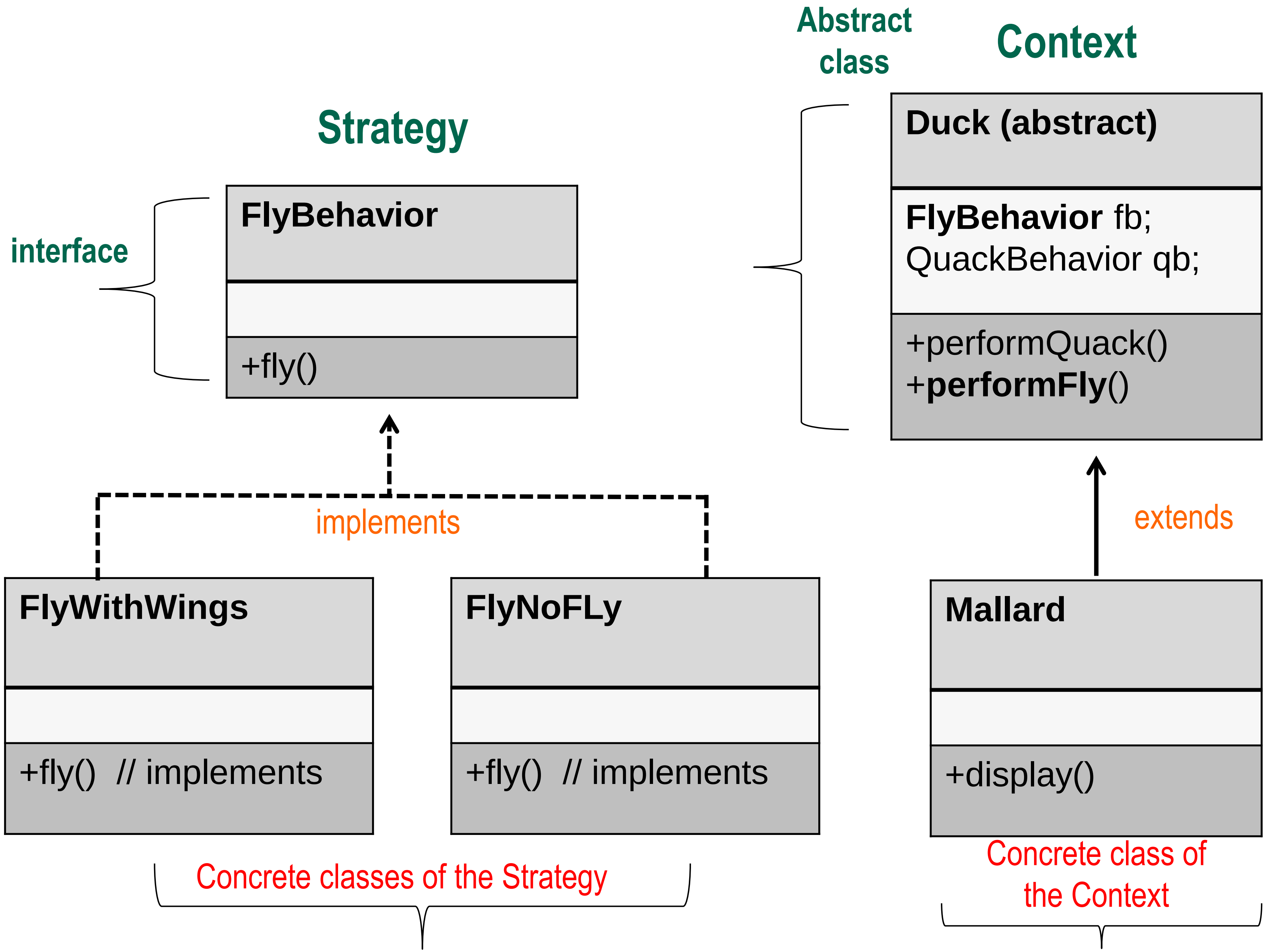
Duck class revisited…

```
public class DuckSimulator {

   public static void main( String[] a ) {
      Duck mallard = new MallardDuck();
      mallard.performQuack();
      mallard.performFly();

      Duck rubberDuckie = new RubberDuck();
      rubberDuckie.performQuack();
      rubberDuckie.performFly();
   }

}
```

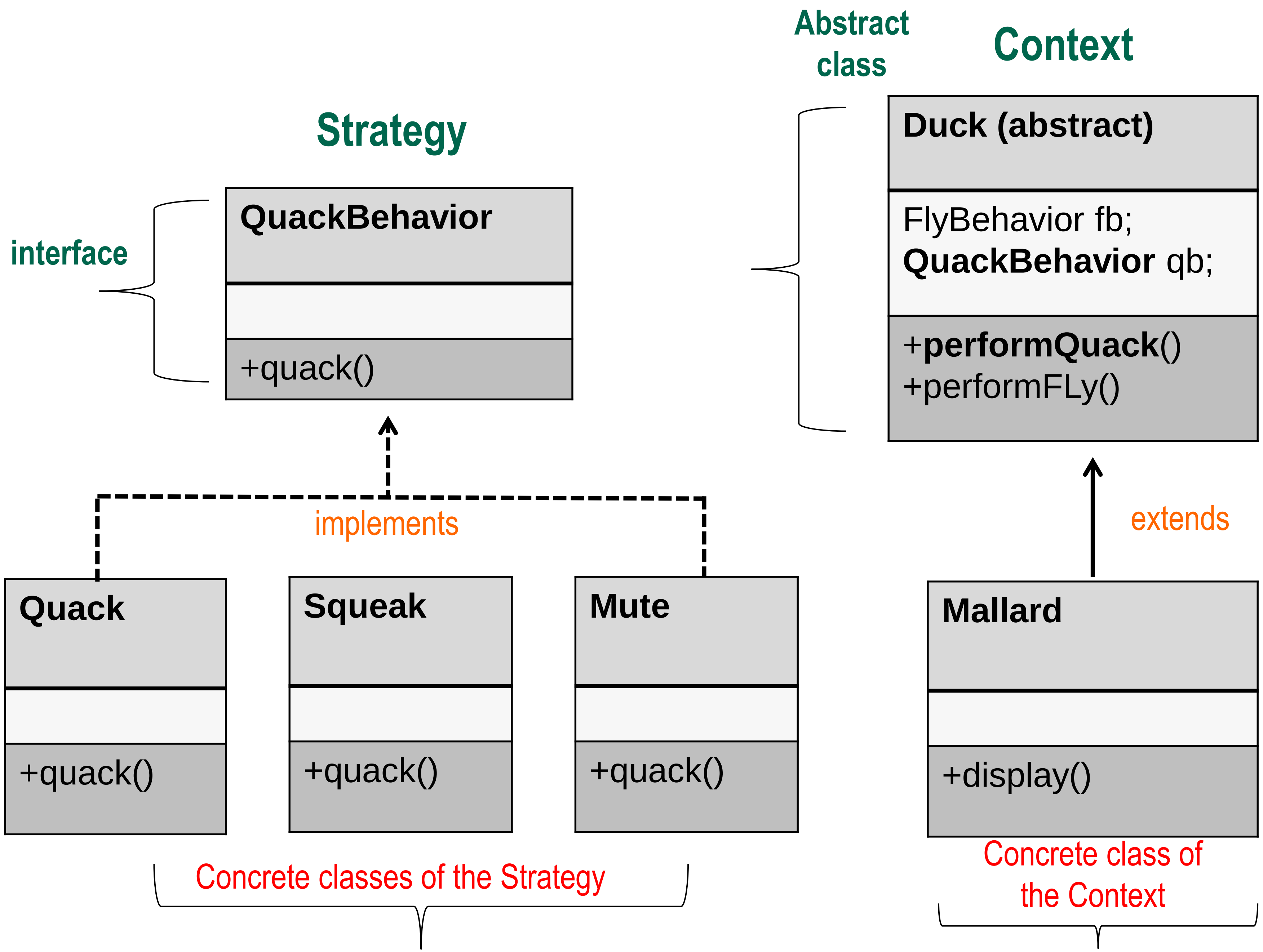
Duck class revisited…

```
public class DuckSimulator {

   public static void main( String[] a ) {
      Duck mallard = new MallardDuck();
      mallard.performQuack();
      mallard.performFly();

      Duck rubberDuckie = new RubberDuck();
      rubberDuckie.performQuack();
      rubberDuckie.performFly();
   }

}
```

Structure of our Example…



Structure of our Example…



Strategy Pattern:
Elements of Reusable OO Software

- **Intent**: Define a family of algorithms, encapsulate each one, and make them interchangeable. *Strategy lets the algorithm vary independently from clients that use it.*

- **Motivation** and Applicability: Many algorithms exist for the same task (i.e. sort).
  - Clients should be allowed to only use the algorithms that make sense for them.
  - Different algorithms will be appropriate at different times.
  - Want to encapsulate different behavior for different objects.
  - Many related classes differ only in their behavior. Strategies provide a way to configure a class with one of many behaviors.
  - You need different variants of an algorithm.
  - A class defines many behaviors, and these are addressed through use of multiple conditional logic. Instead each branch of a conditional logic can be its own strategy.

Strategy Pattern:
Elements of Reusable OO Software

- Intent*:* Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

- **Motivation** and Applicability: *Many algorithms exist for the same task (i.e. sort).*
  - Clients should be allowed to only use the algorithms that make sense for them.
  - Different algorithms will be appropriate at different times.
  - **Want to encapsulate different behavior for different objects.**
  - Many related classes differ only in their behavior. Strategies provide a way to configure a class with one of many behaviors.
  - You need different variants of an algorithm.
  - A class defines many behaviors, and these are addressed through use of multiple conditional logic. Instead each branch of a conditional logic can be its own strategy.

Strategy Pattern:
Elements of Reusable OO Software

- Intent*:* Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

- Motivation *and Applicability*: *Many algorithms exist for the same task (i.e. sort).*
  - Clients should be allowed to only use the algorithms that make sense for them.
  - Different algorithms will be appropriate at different times.
  - Want to encapsulate different behavior for different objects.
  - Many related classes differ only in their behavior. ***Strategies provide a way to configure a class with one of many behaviors.***
  - You need different variants of an algorithm.
  - A class defines many behaviors, and these are addressed through use of multiple conditional logic. Instead each branch of a conditional logic can be its own strategy.

Strategy Pattern:
Elements of Reusable OO Software

- Intent*:* Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

- Motivation *and Applicability*: *Many algorithms exist for the same task (i.e. sort).*
  - Clients should be allowed to only use the algorithms that make sense for them.
  - Different algorithms will be appropriate at different times.
  - Want to encapsulate different behavior for different objects.
  - Many related classes differ only in their behavior. *Strategies provide a way to configure a class with one of many behaviors.*
  - You need different variants of an algorithm.
  - **A class defines many behaviors, and these are addressed through use of multiple conditional logic. Instead each branch of a conditional logic can be its own strategy.**

Strategy Pattern:
Elements of Reusable OO Software

- **Consequences**: The Strategy Patter has the following benefits and drawbacks:
  1. Can create families of related algorithms.
  2. Provides an alternative to sub-classing.
  3. Can eliminate deep conditional logic.
  4. Provide different implementations of the same behavior.
  5. Increases communication overhead between Strategy and the specific Context that you are applying it on.
  6. Increases the number of objects as each algorithm is an instance of a class.

Strategy Pattern:
Elements of Reusable OO Software

- **Consequences**: The Strategy Patter has the following **benefits** and drawbacks:
  1. Can create families of related algorithms.
  2. Provides an alternative to sub-classing.
  3. Can eliminate deep conditional logic.
  4. Provide different implementations of the same behavior.
  5. Increases communication overhead between Strategy and the specific Context that you are applying it on.
  6. Increases the number of objects as each algorithm is an instance of a class.

Strategy Pattern:

Elements of Reusable OO Software

- **Consequences**: The Strategy Patter has the following benefits and ***drawbacks***:
  1. Can create families of related algorithms.
  2. Provides an alternative to sub-classing.
  3. Can eliminate deep conditional logic.
  4. Provide different implementations of the same behavior.
  5. **Increases communication overhead between Strategy and the specific Context that you are applying it on.**
  6. Increases the number of objects as each algorithm is an instance of a class.

Implementation (Issues):

Elements of Reusable OO Software

- The Strategy and Concrete Interfaces must give a *concrete strategy* efficient access to any data it needs from a context and vice versa. Approaches:
  - The context passes to the strategy the information it needs.
  - The context passes itself as an argument to the strategy.
  - A reference to the context object is established when the strategy is created.

Implementation (Issues):

Elements of Reusable OO Software

- The Strategy and Concrete Interfaces must give a *concrete strategy* efficient access to any data it needs from a context and vice versa. Approaches:
  - **The context passes to the strategy the information it needs.**
  - The context passes itself as an argument to the strategy
  - A reference to the context object is established when the strategy is created.

```
public class DuckSimulator {

   public static void main( String[] a ) {
      Duck mallard = new MallardDuck();
      mallard.performQuack( .. .. );
      mallard.performFly();
   }
}
```

Loosest Coupling

Implementation (Issues):

Elements of Reusable OO Software

- The Strategy and Concrete Interfaces must give a *concrete strategy* efficient access to any data it needs from a context and vice versa. Approaches:
  - **The context passes to the strategy the information it needs.**
  - The context passes itself as an argument to the strategy
  - A reference to the context object is established when the strategy is created.

```
public abstract class Duck {
...
   public void performQuack() {
      quackBehavior.quack( .. .. );
   }
   public void performFly() {
       flyBehavior.fly();
   }
}
```

Loose Coupling

Implementation (Issues):

Elements of Reusable OO Software

- The Strategy and Concrete Interfaces must give a *concrete strategy* efficient access to any data it needs from a context and vice versa. Approaches:
  - The context passes to the strategy the information it needs.
  - **The context passes itself as an argument to the strategy.**
  - A reference to the context object is established when the strategy is created.

```
public class DuckSimulator {

   public static void main( String[] a ) {
      Duck mallard = new MallardDuck();
      mallard.performQuack( mallard );
      mallard.performFly();
   }
}
```

Implicit in the call

Stronger Coupling

Implementation (Issues):

Elements of Reusable OO Software

- The St *concrete strateg* a context and vic
  - The needs.
  - **The strategy.**
  - A re the strat

```
public abstract class Duck {
    protected FlyBehavior flyBehavior;
    protected QuackBehavior quackBehavior;
    ...
    public void performQuack() {
        quackBehavior.quack(this);
    }
    public void performFly() {
        flyBehavior.fly();
    }
}
```

```
public class DuckSimulator {

    public static void main( String[] a ) {
        Duck mallard = new MallardDuck();
        mallard.performQuack();
        mallard.performFly();
    }
}
```

Stronger Coupling

Implementation (Issues):

Elements of Reusable OO Software

- The Strategy and Concrete Interfaces must give a *concrete strategy* efficient access to any data it needs from a context and vice versa. Approaches:
  - The context passes to the strategy the information it needs.
  - The context passes itself as an argument to the strategy
  - **A reference to the context object is established when the strategy is created.**

```
public class MallardDuck extends Duck {

   public MallardDuck() {
      quackBehavior = new Quack();
      flyBehavior = new FlyWithWings(this);
   }
   .. ..
}
```

Strongest Coupling

Implementation (Issues):

Elements of Reusable OO Software

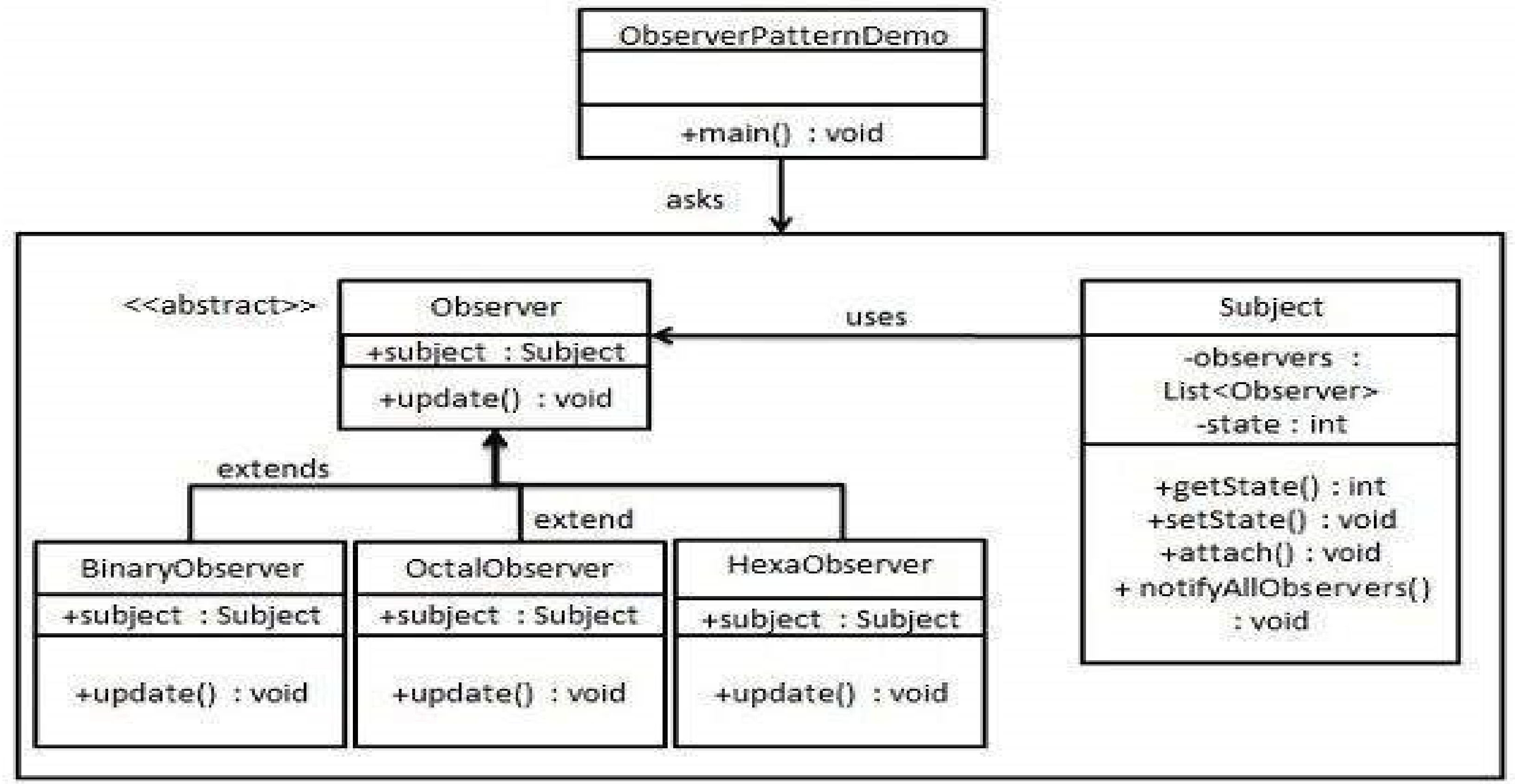
- The Strategy and Concrete Interfaces must give a *concrete strategy* efficient access to any data it needs from a context and vice versa. Approaches:
  - The context passes to the strategy the information it needs.
  - The context passes itself as an argument to the strategy
  - A reference to the context object is established when the strategy is created.

```
public class MallardDuck extends Duck {

   public MallardDuck() {
      quackBehavior = new Quack();
      flyBehavior = new FlyWithWings(.. ..);
   }
   .. ..
}
```

Loose Coupling

Observer Pattern

Intent: Define a **one-to-many** dependency between objects so that when one object changes state, all its dependencies are notified and updated automatically.



Tutorials Point

Observer Pattern:
Elements of Reusable OO Software

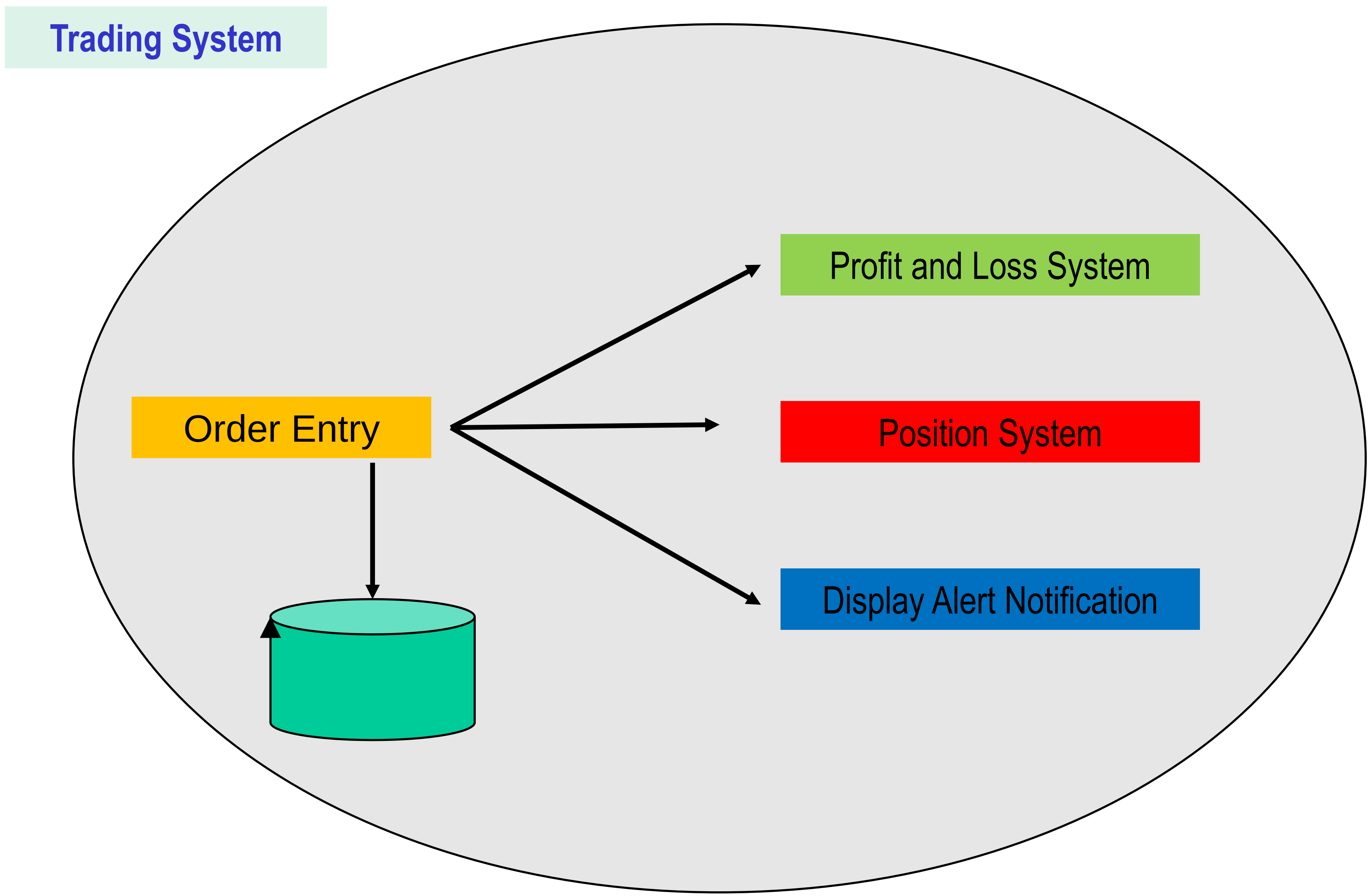
- **Motivation** and Applicability: A common side-effect of partitioning a system into a collection of cooperating classes is the need to maintain consistency between related objects – without creating a tightly coupled behavior amongst them.

Observer Pattern:

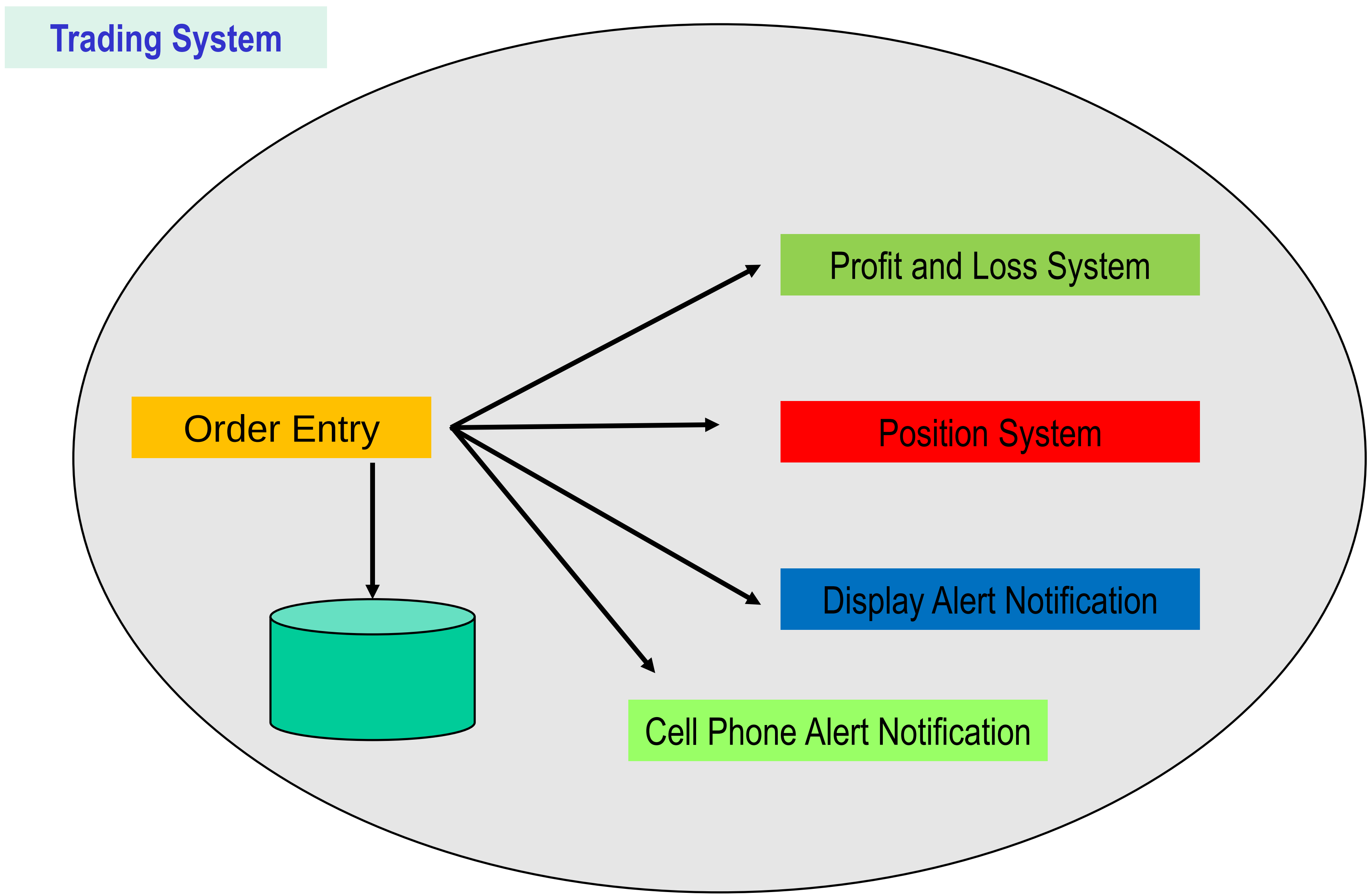
Elements of Reusable OO Software

- Motivation and Applicability: A common side-effect of partitioning a system into a collection of cooperating classes is the need to maintain consistency between related objects – without creating a tightly coupled behavior amongst them.
  - When an abstraction has two aspects, one dependent on the other.
  - When a change to one object requires changing others, and you don’t know how many objects need to be changed.
  - *When an object should be able to notify other objects without making assumptions about who the objects are.*

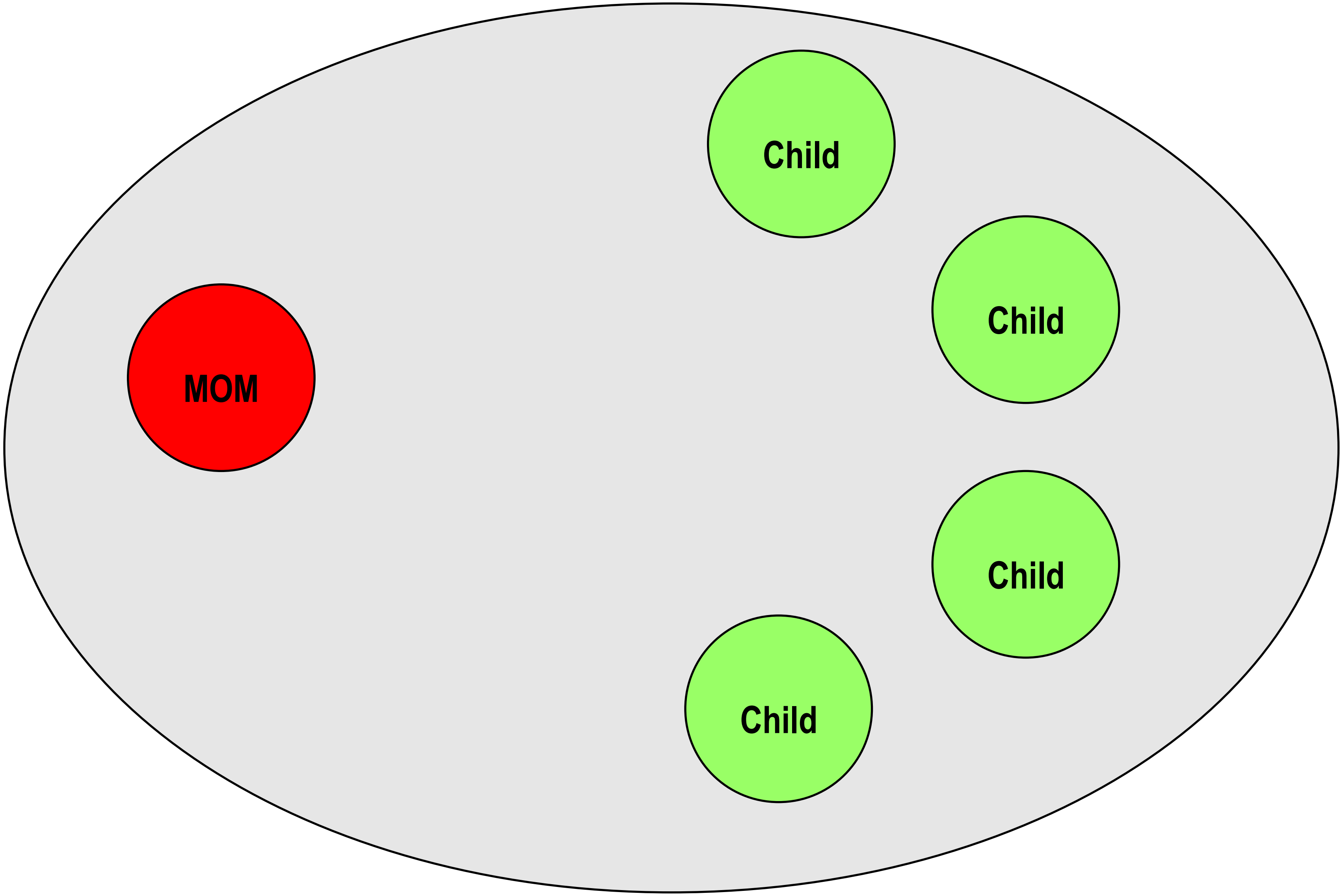
Observer Pattern:



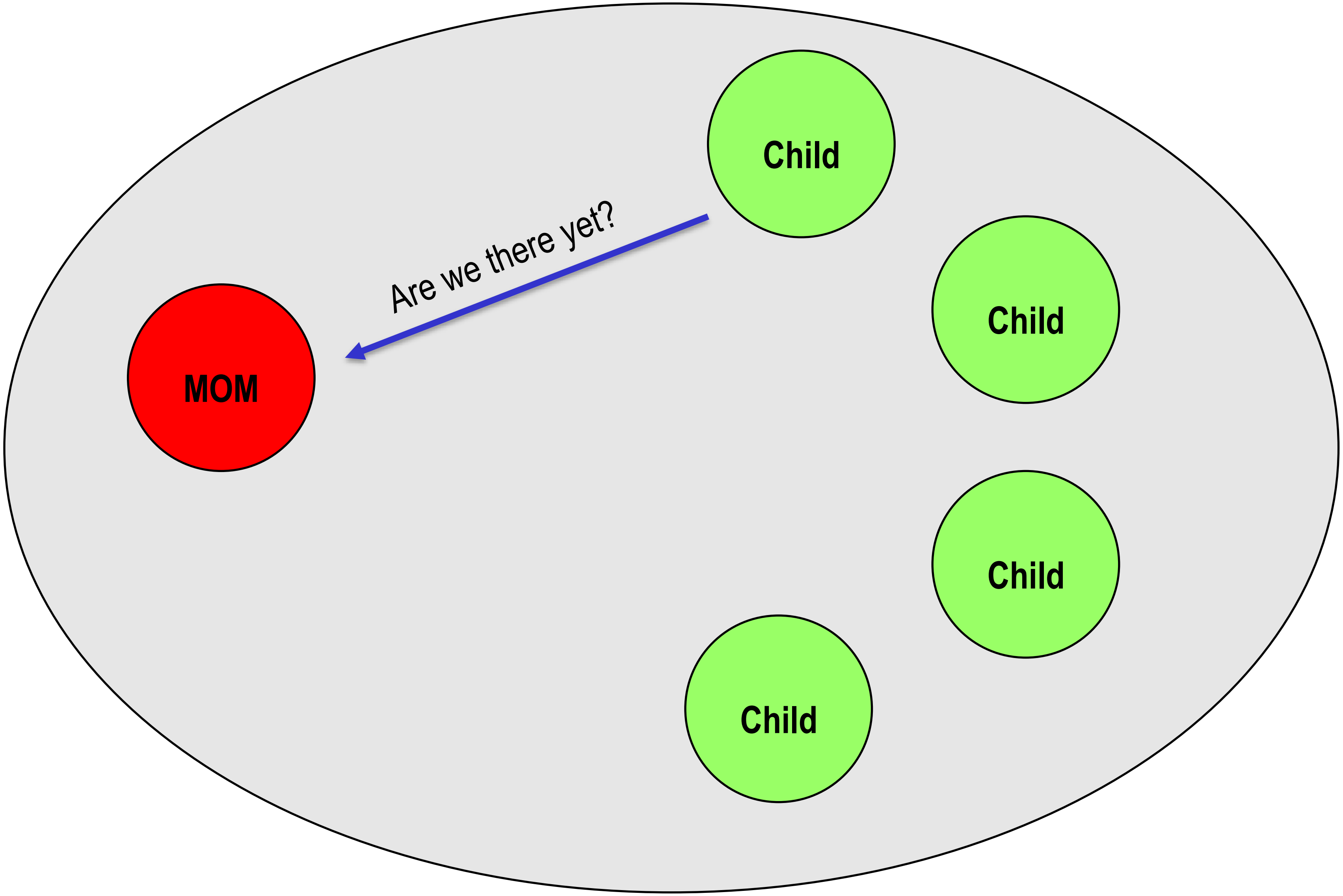
Observer Pattern:



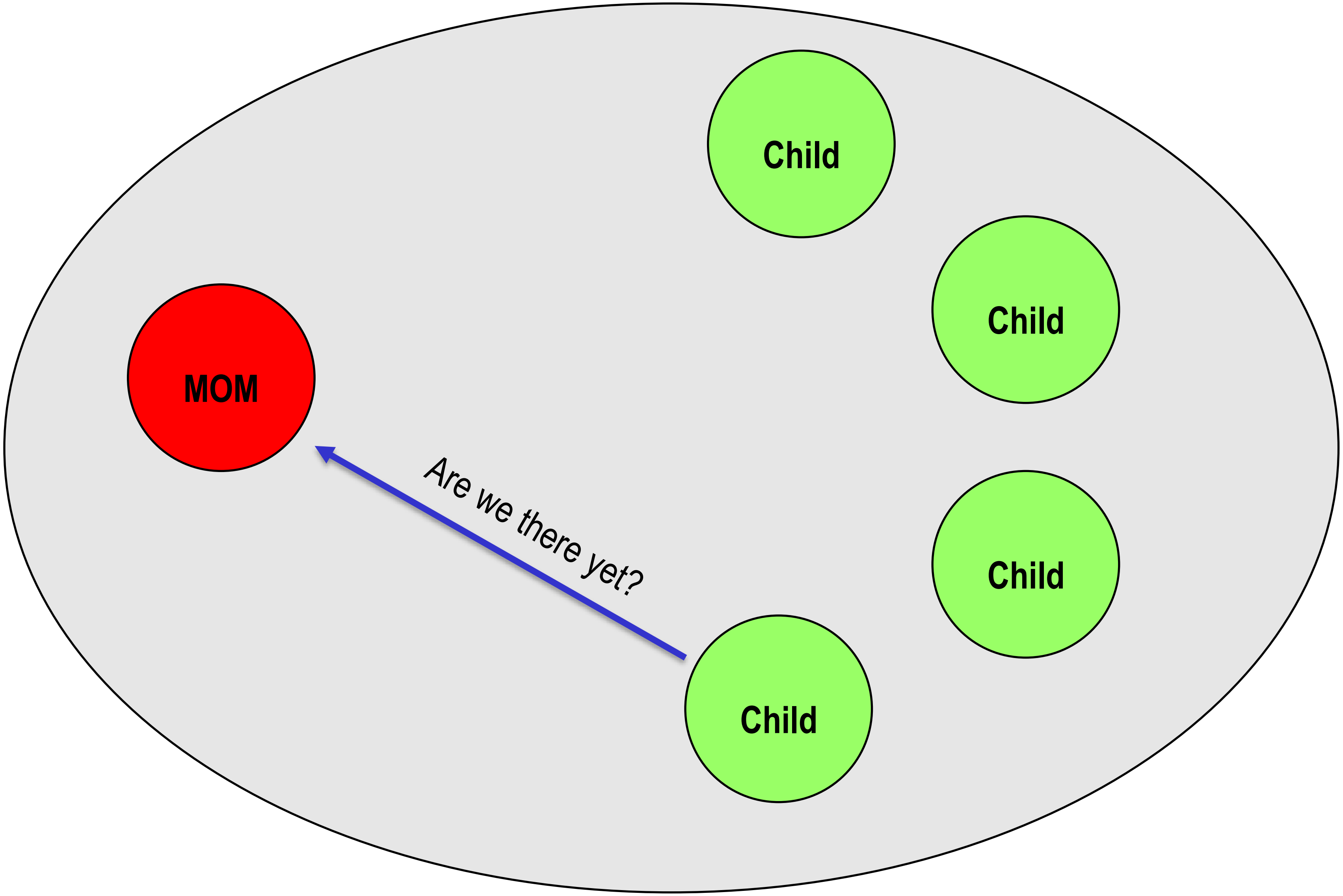
Observer Pattern



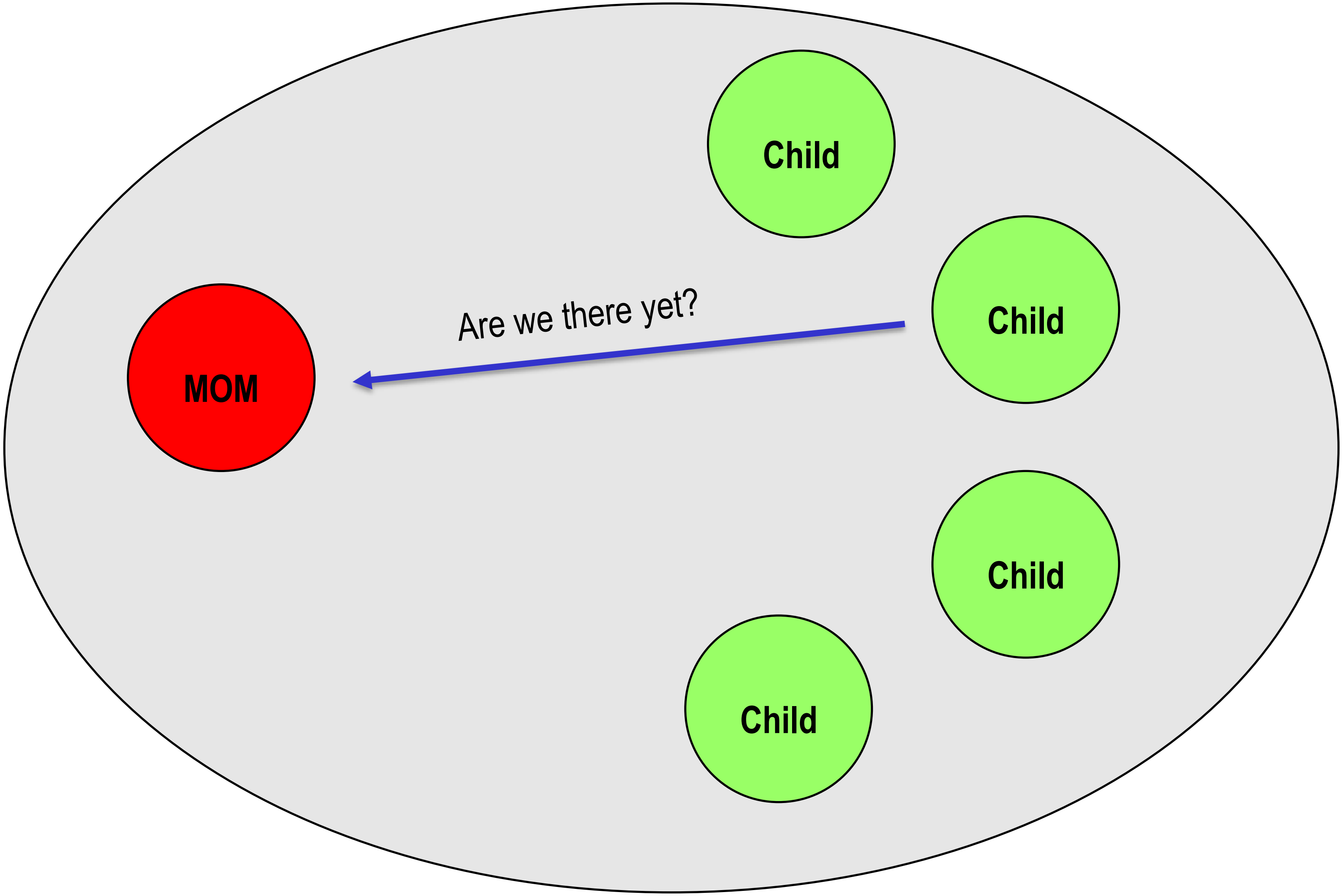
Observer Pattern



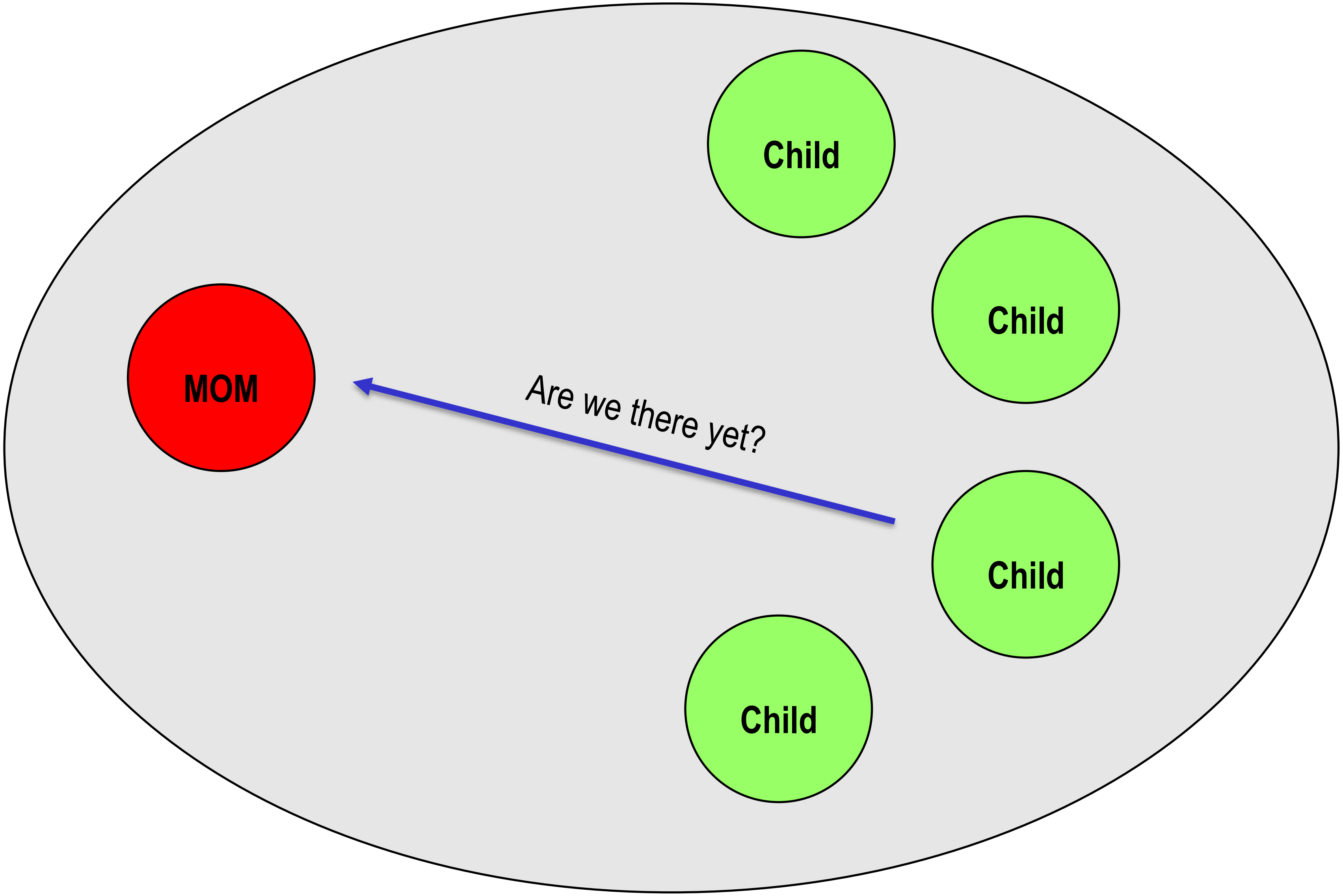
Observer Pattern



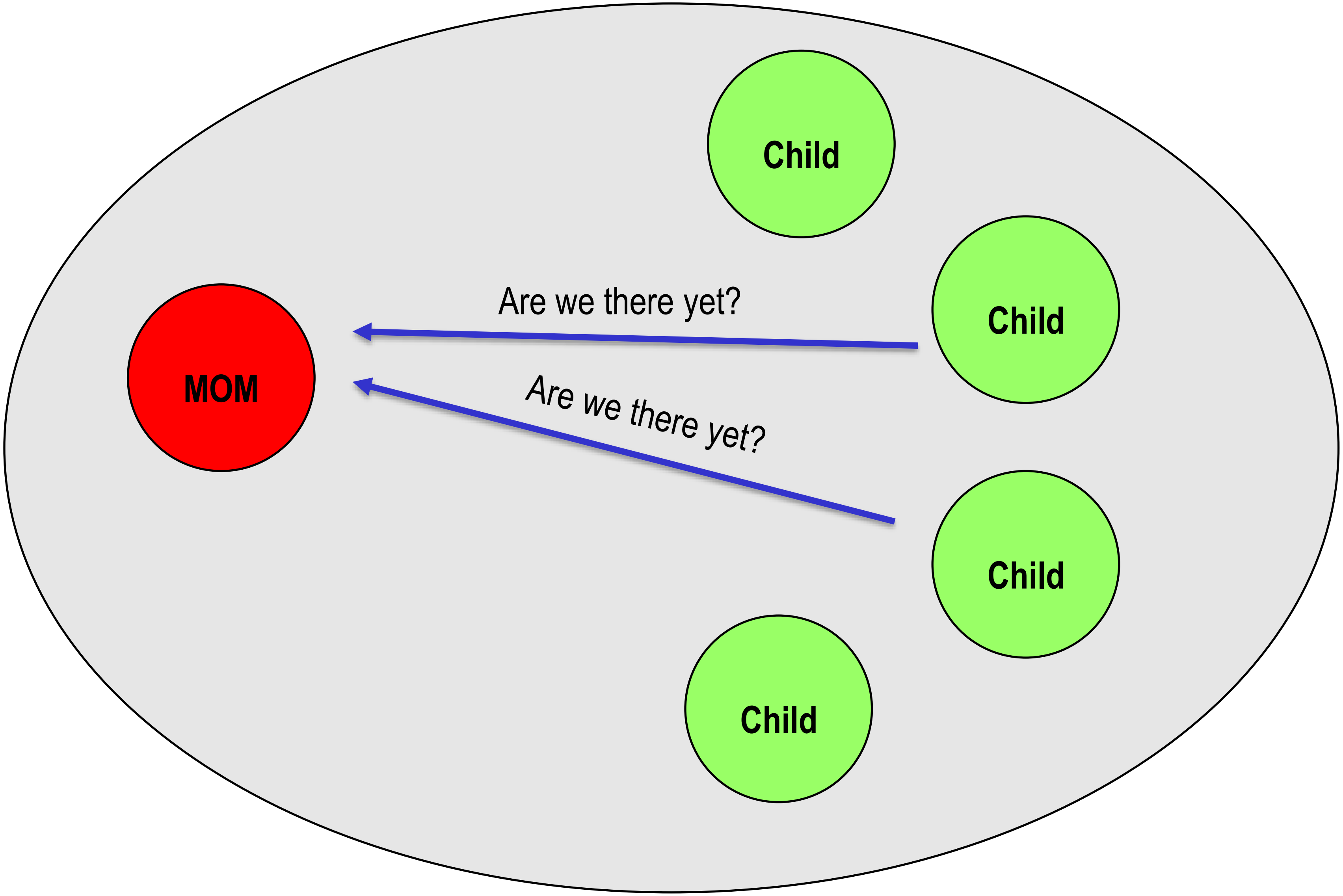
Observer Pattern



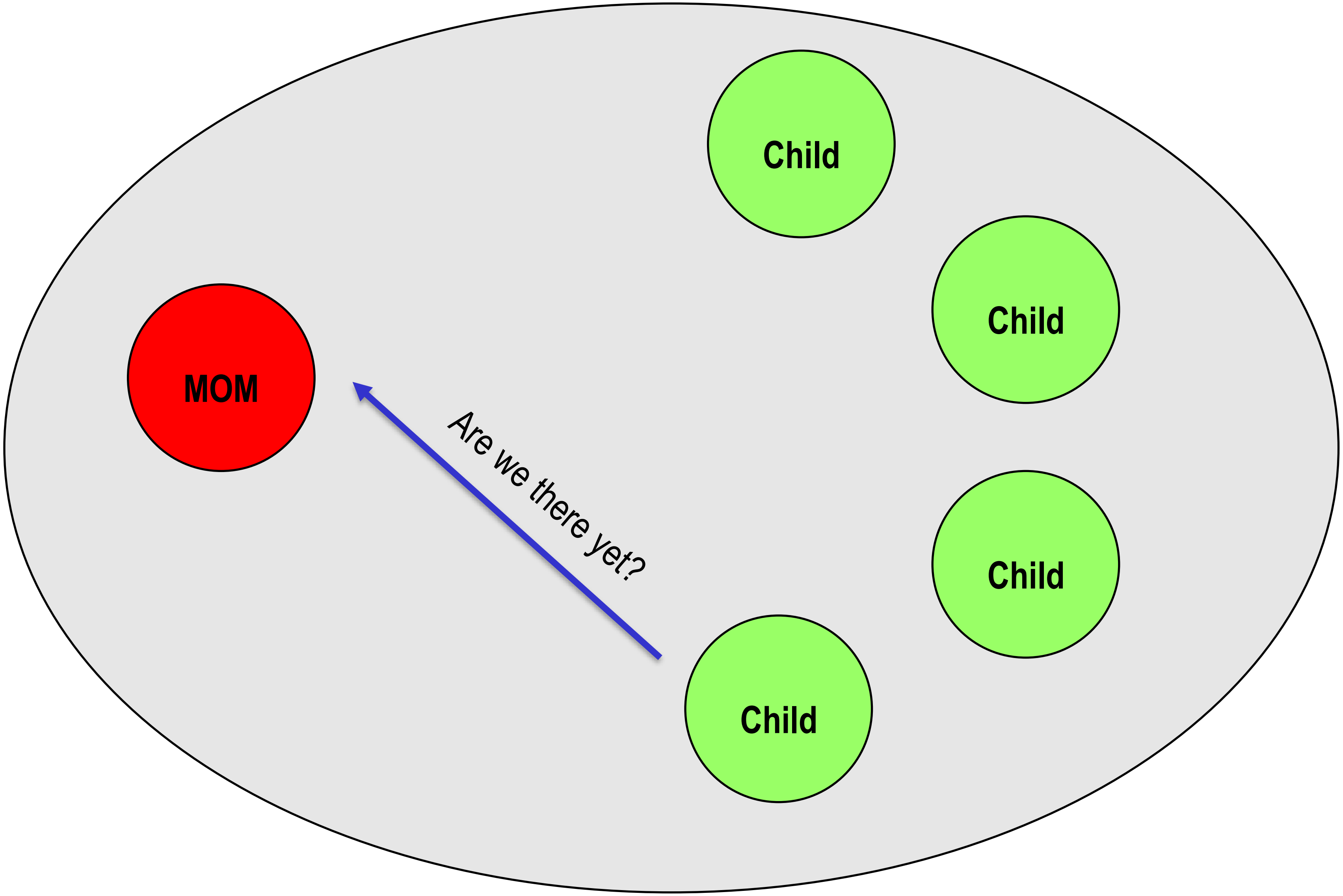
Observer Pattern



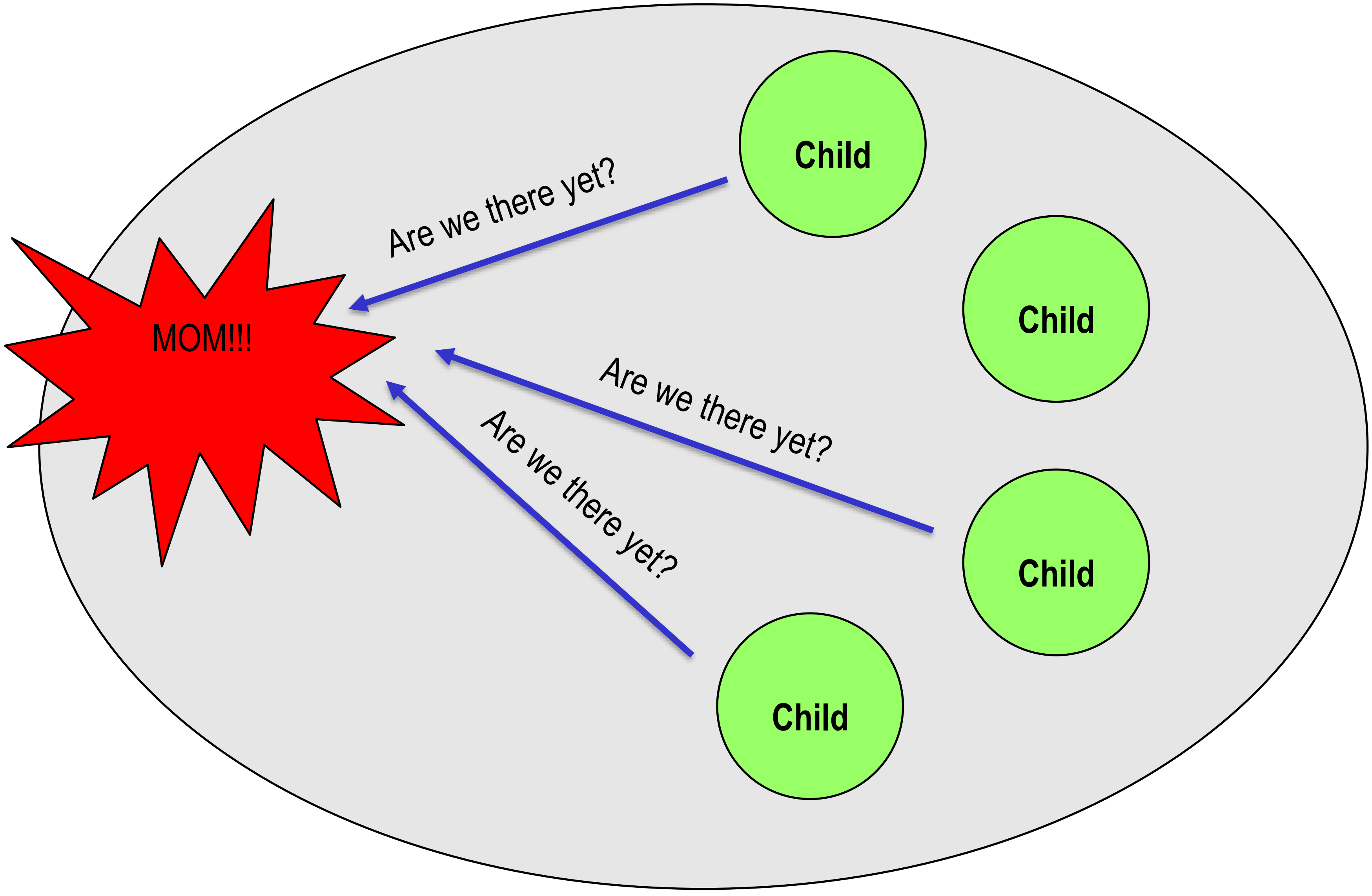
Observer Pattern



Observer Pattern

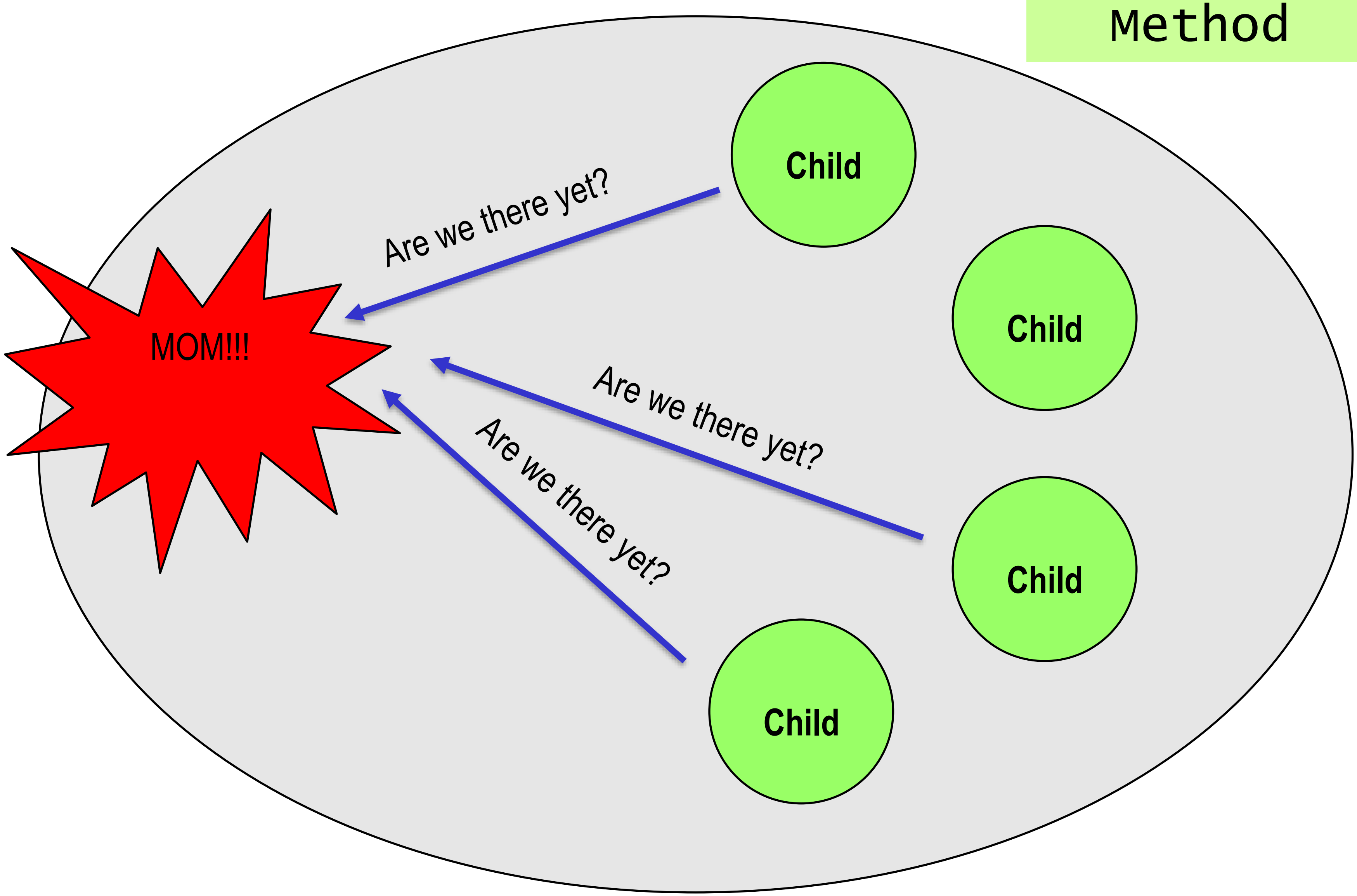


Observer Pattern

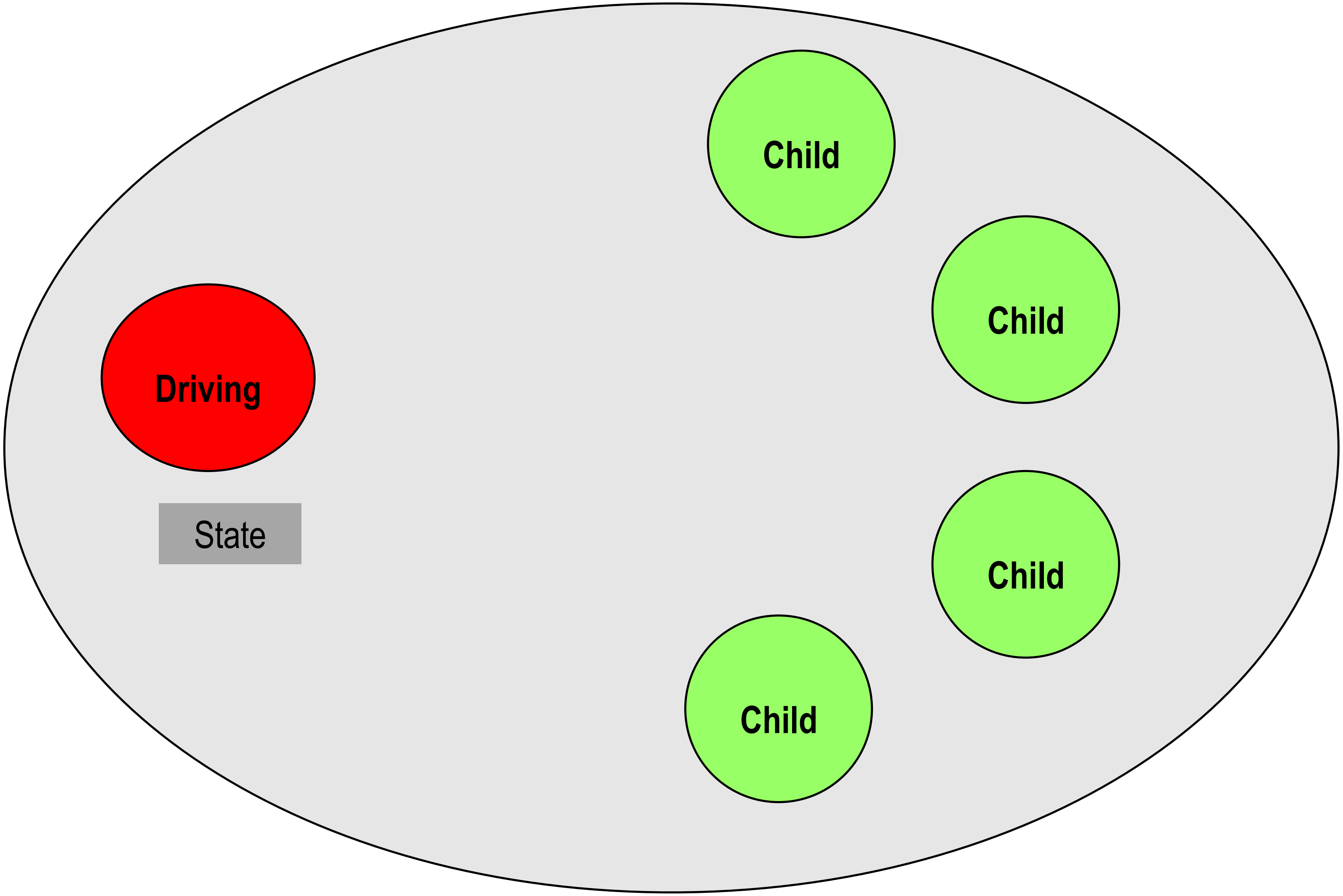


Observer Pattern

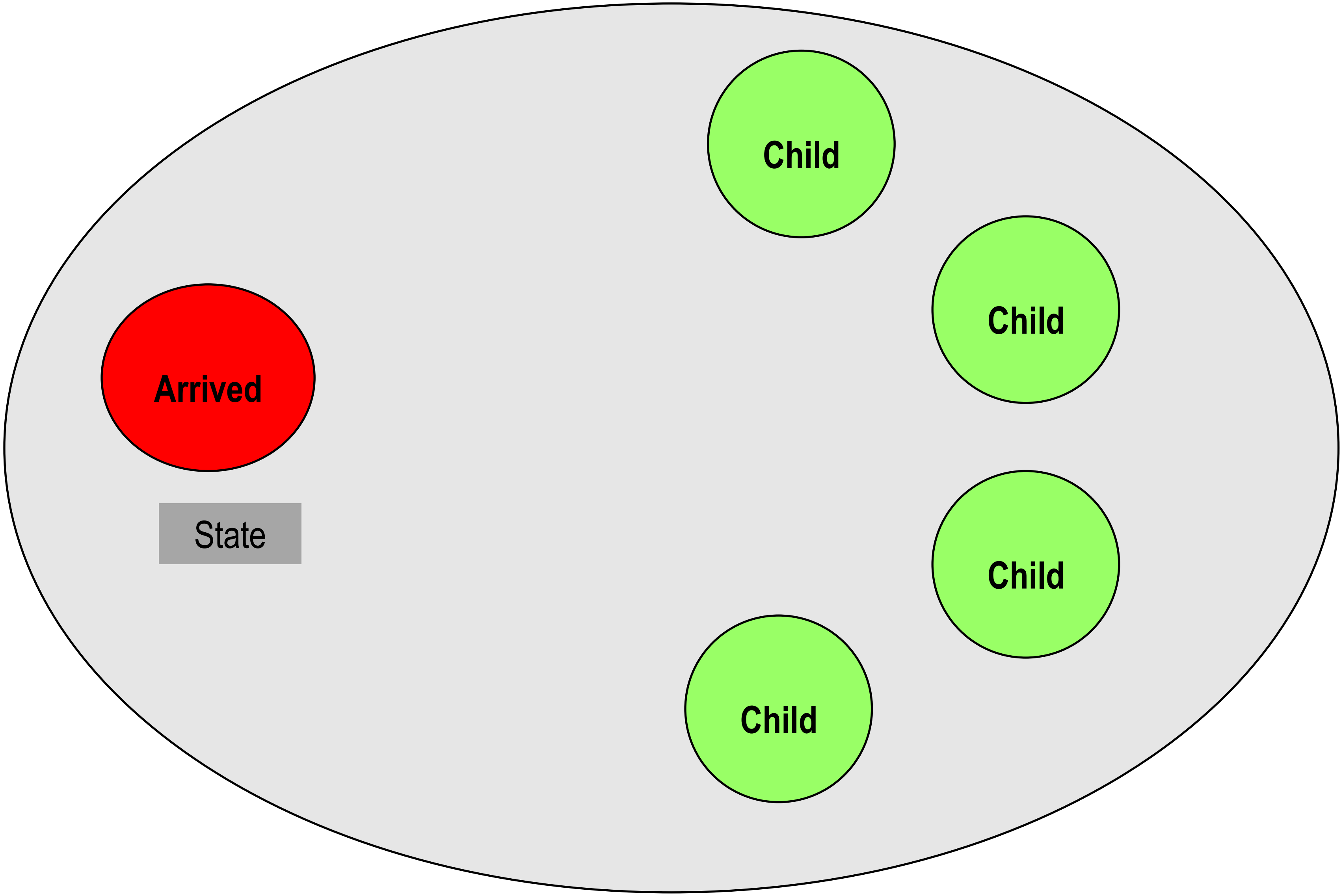
Poll Method



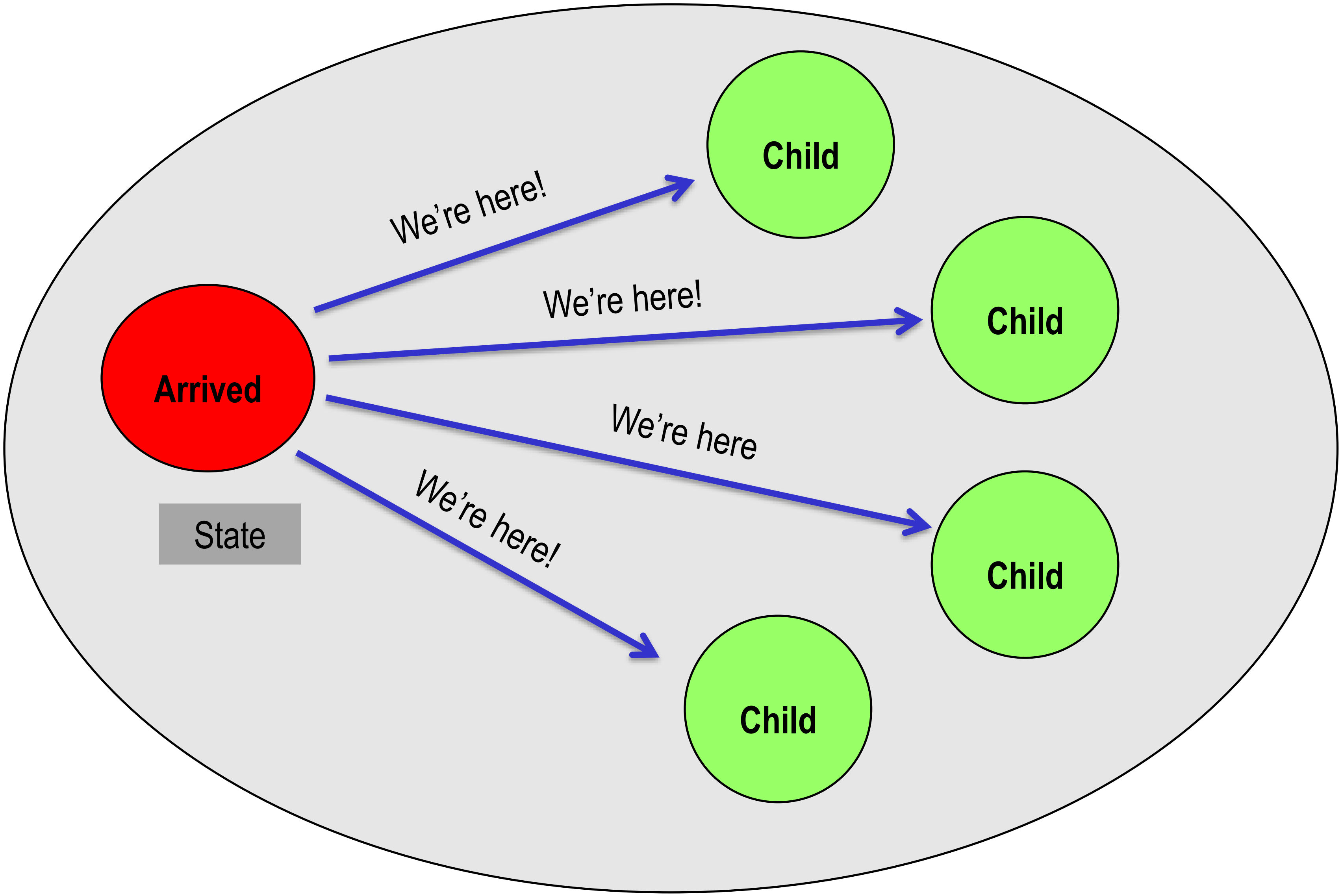
Observer Pattern



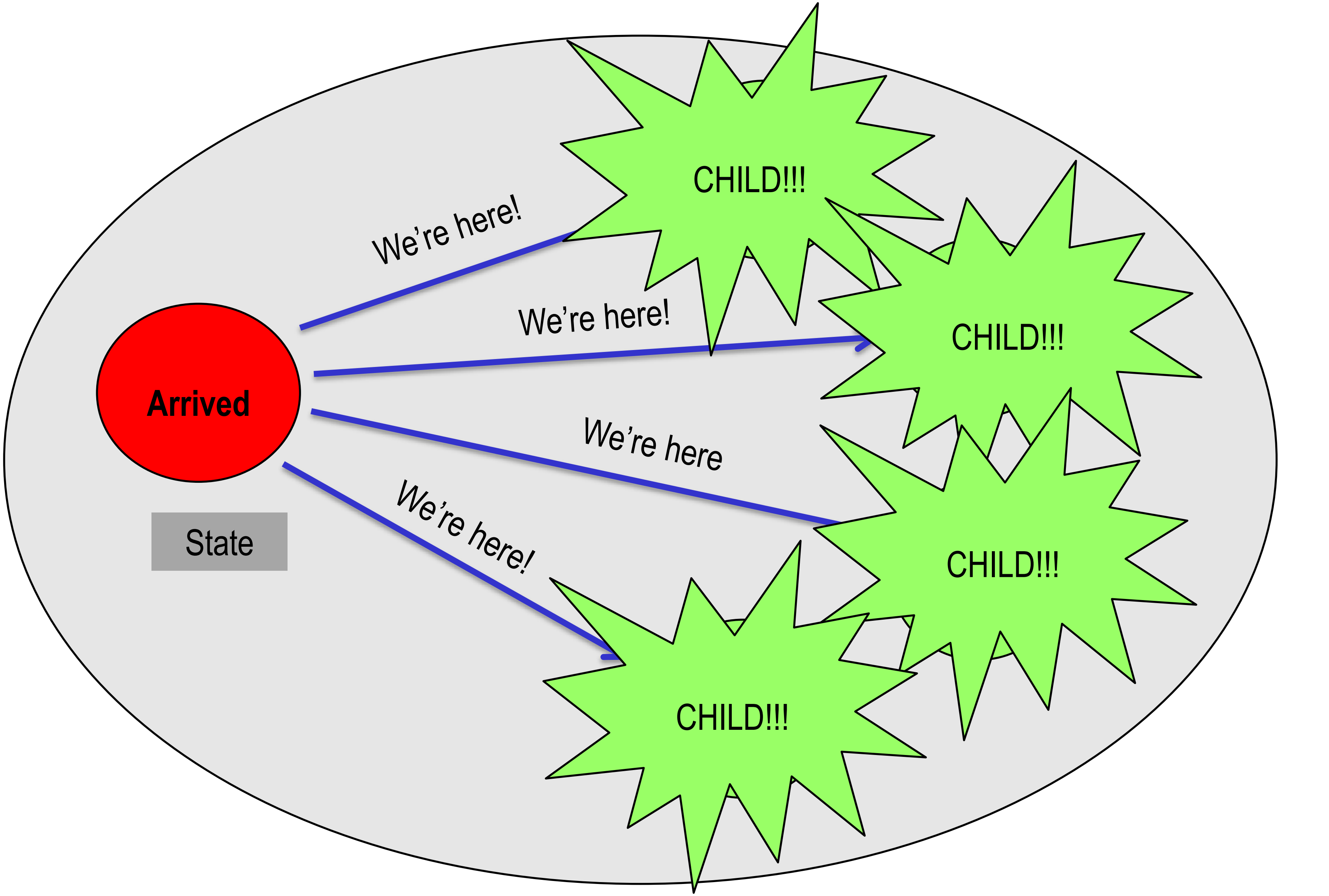
Observer Pattern



Observer Pattern

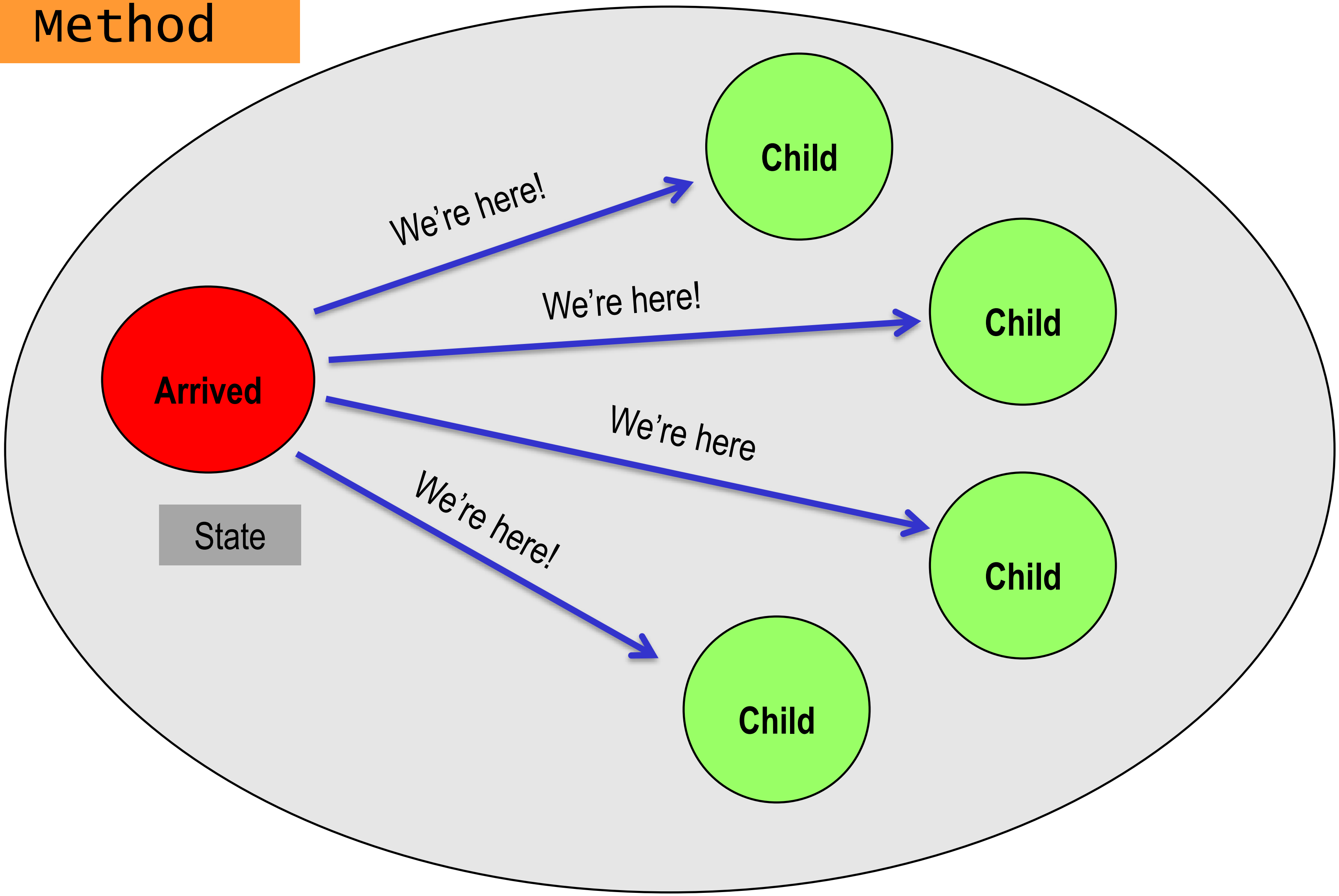


Observer Pattern

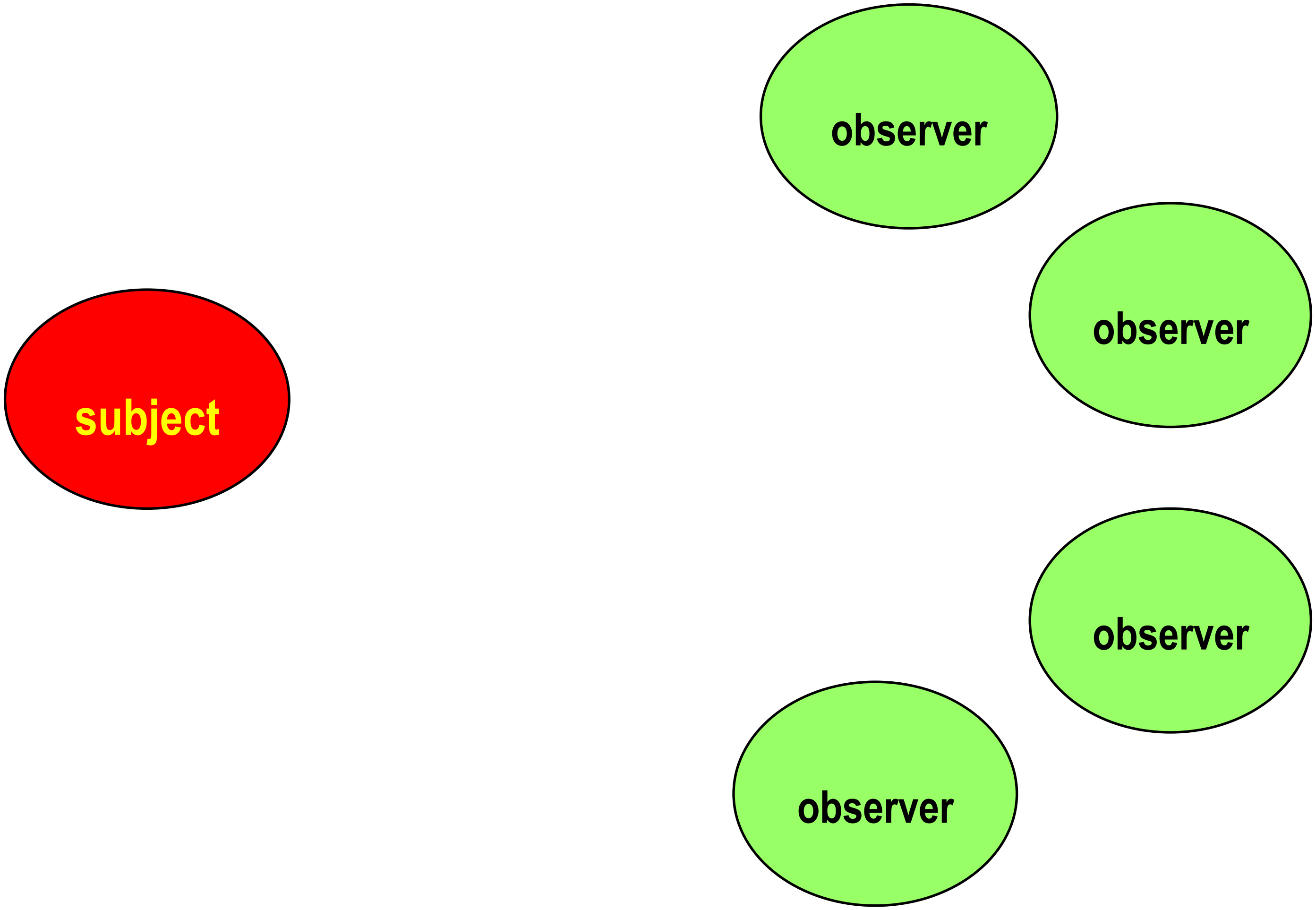


Observer Pattern

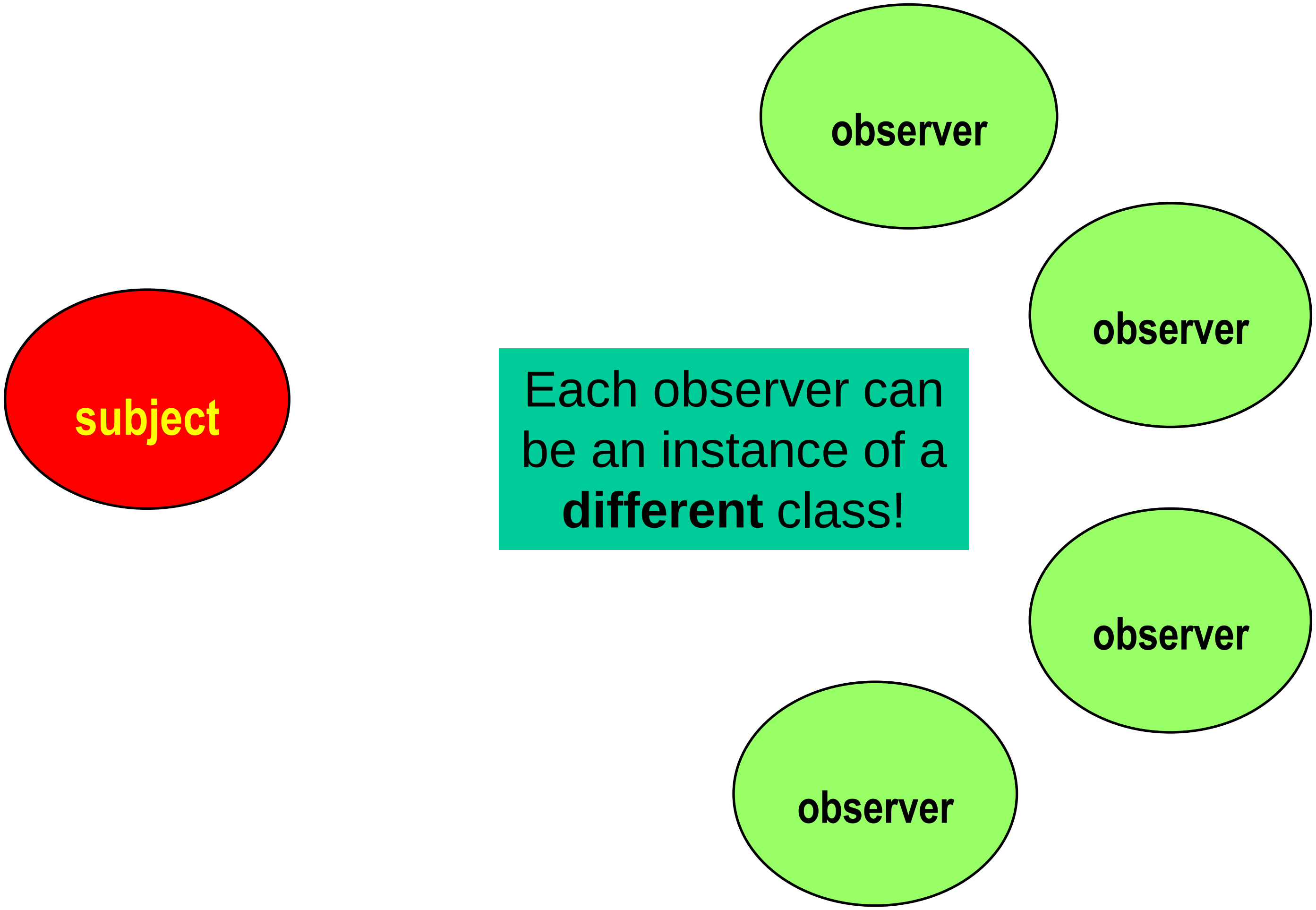
Push Method



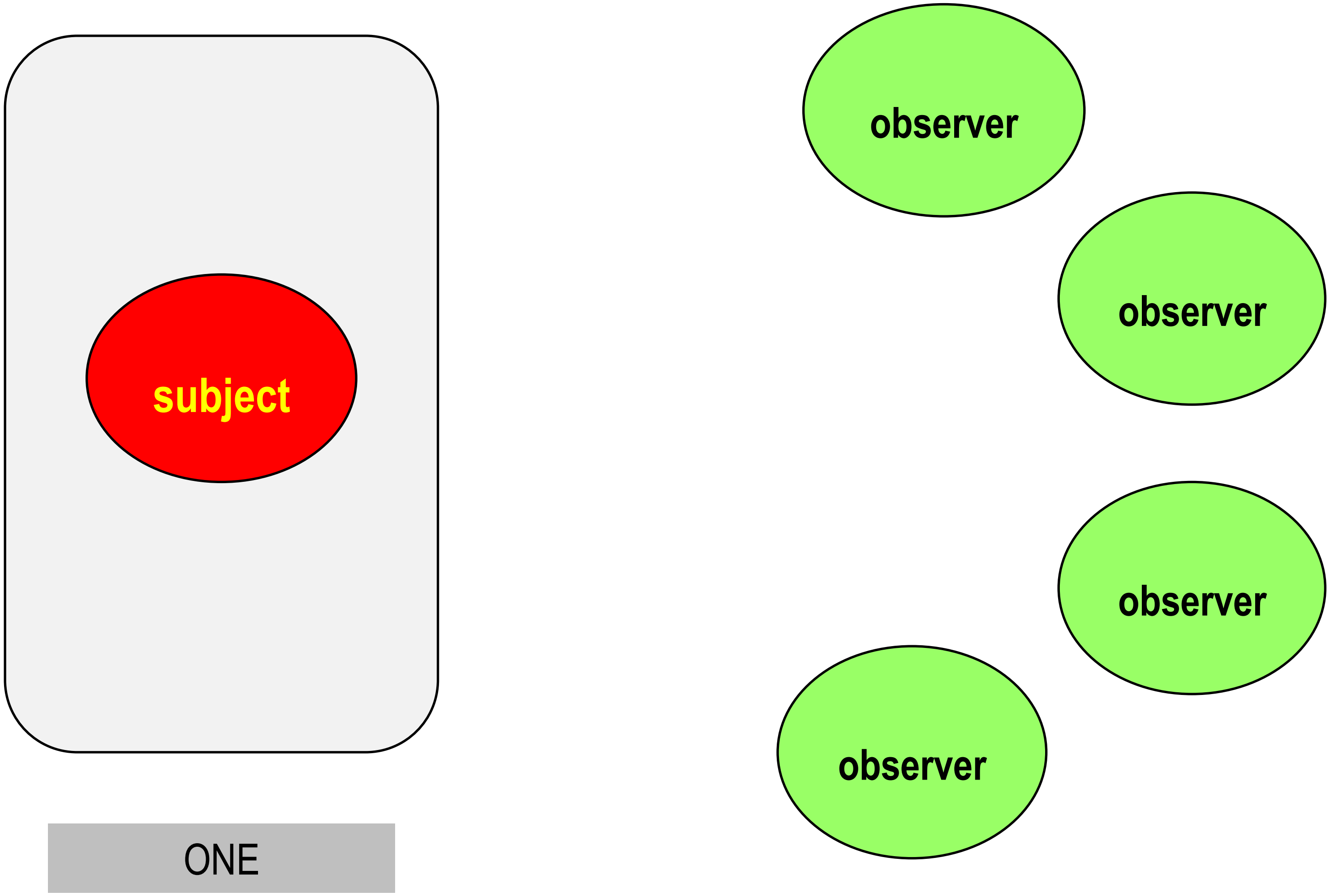
Observer Pattern



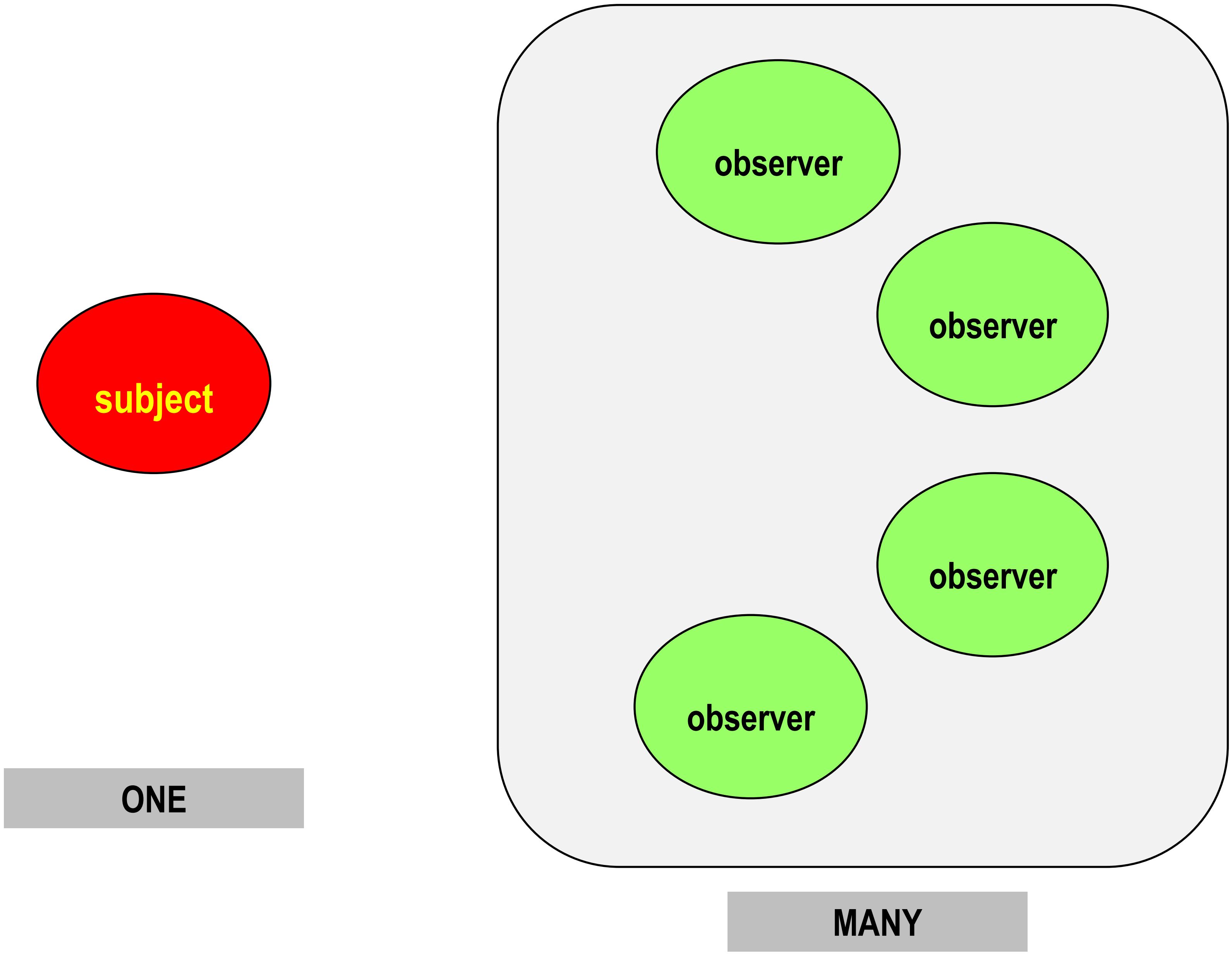
Observer Pattern



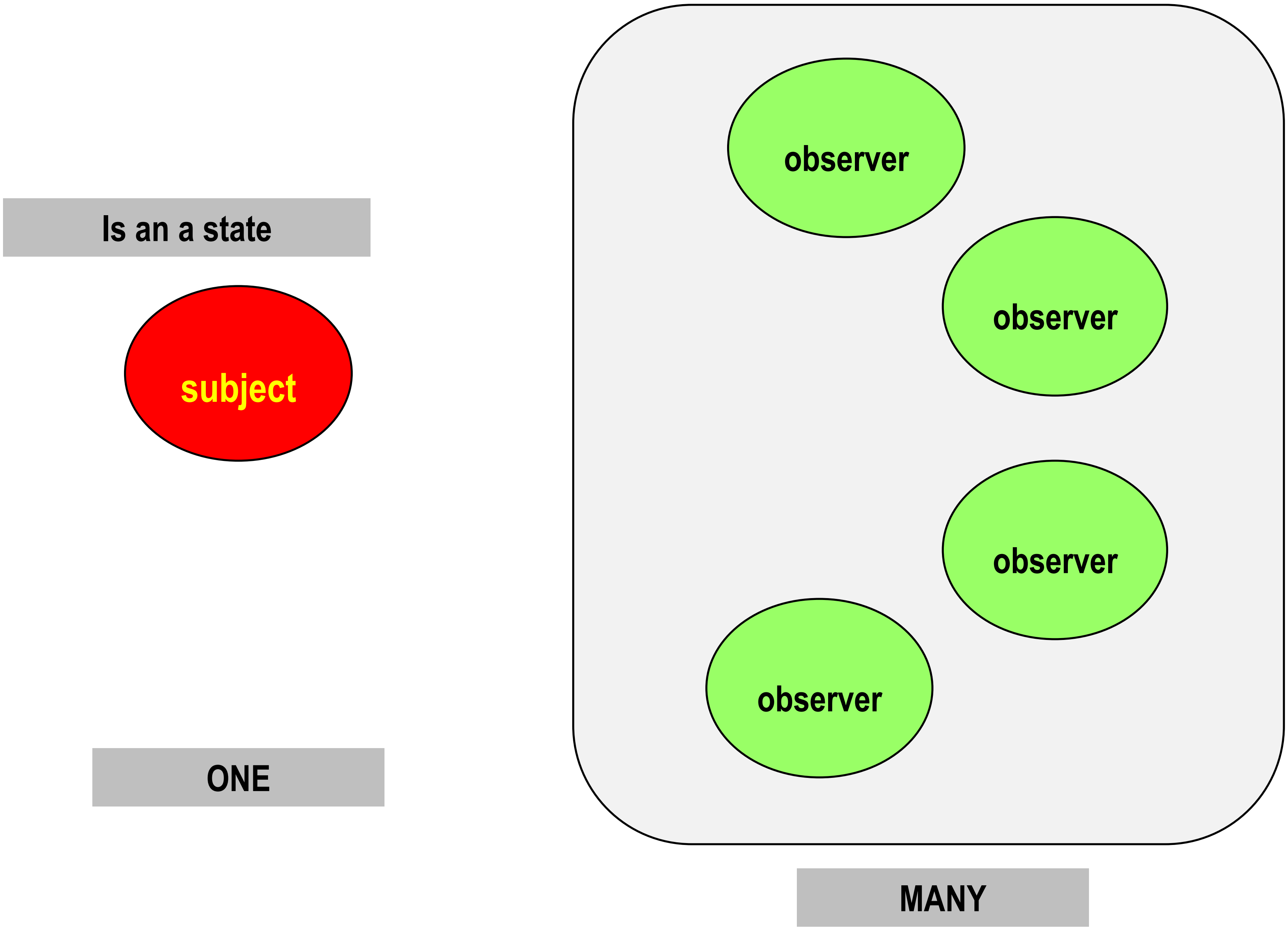
Observer Pattern



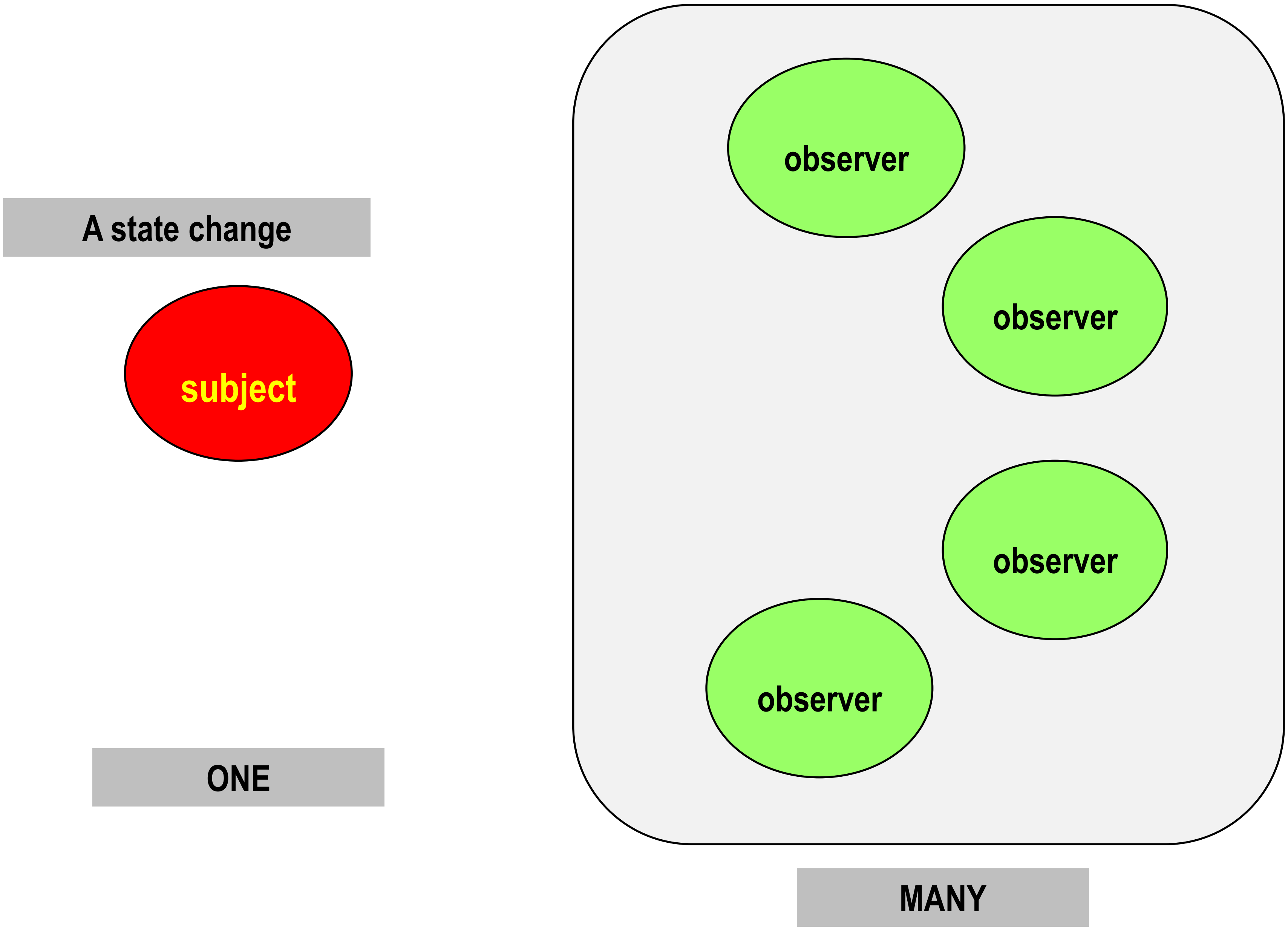
Observer Pattern



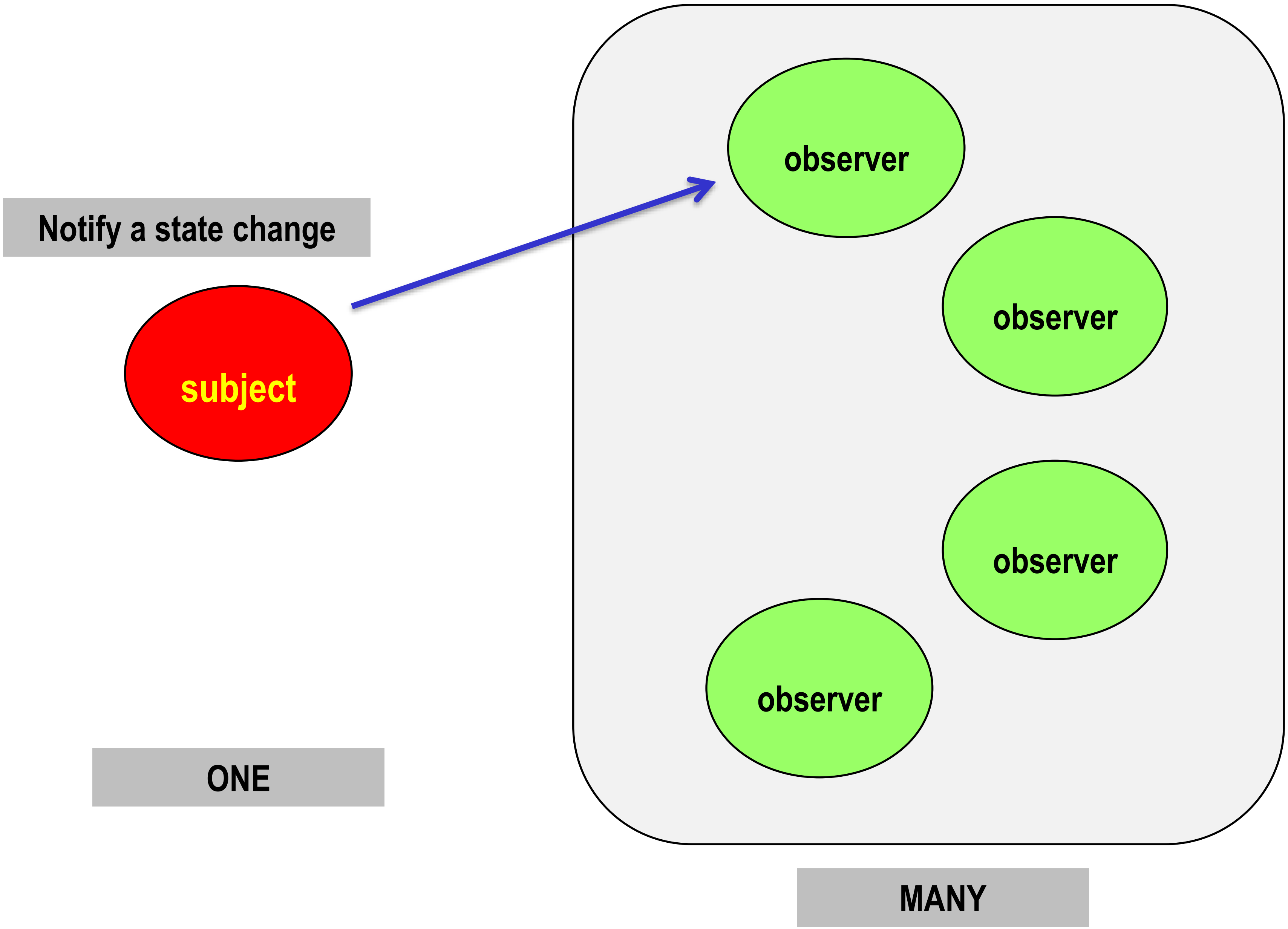
Observer Pattern



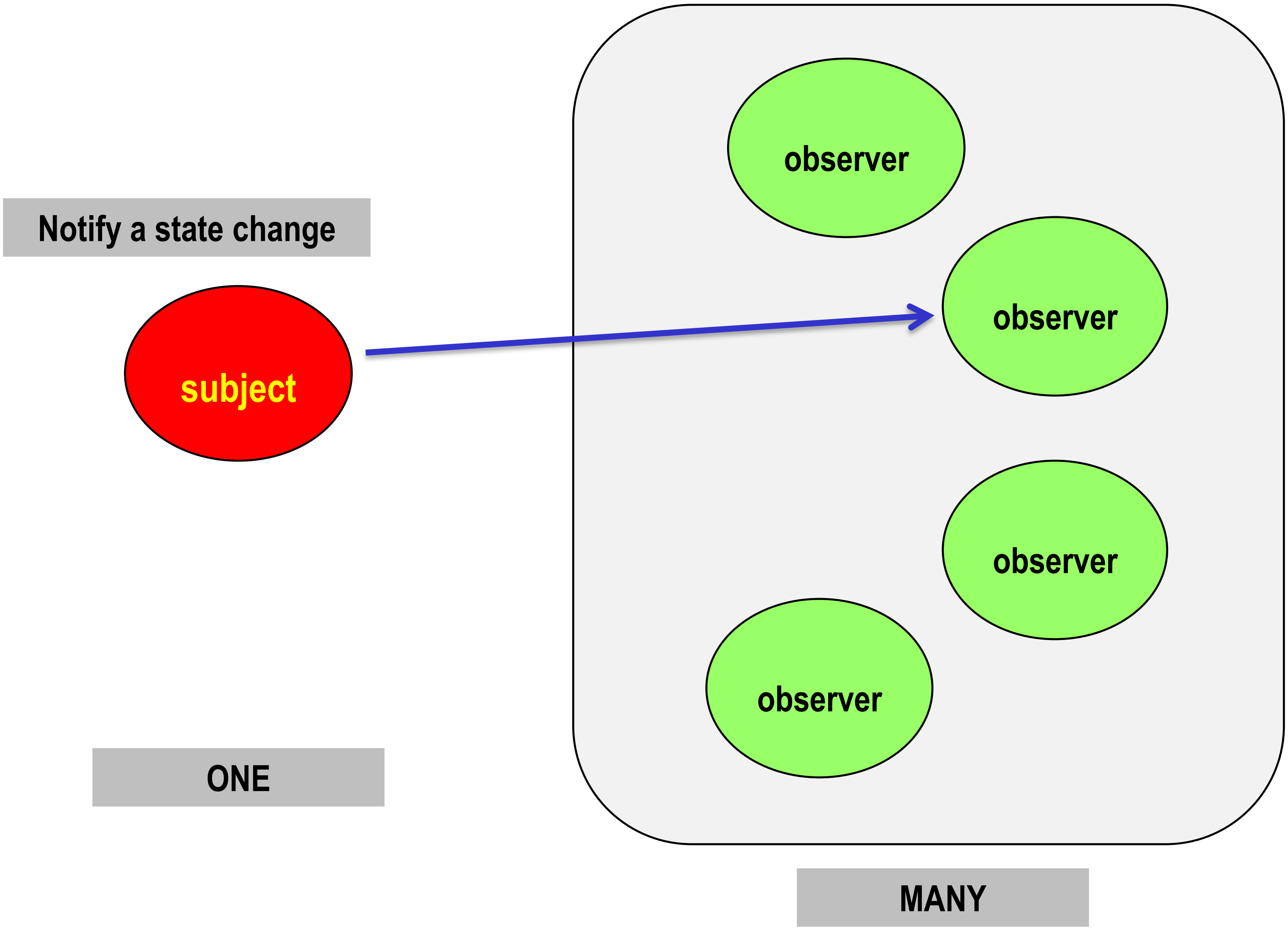
Observer Pattern



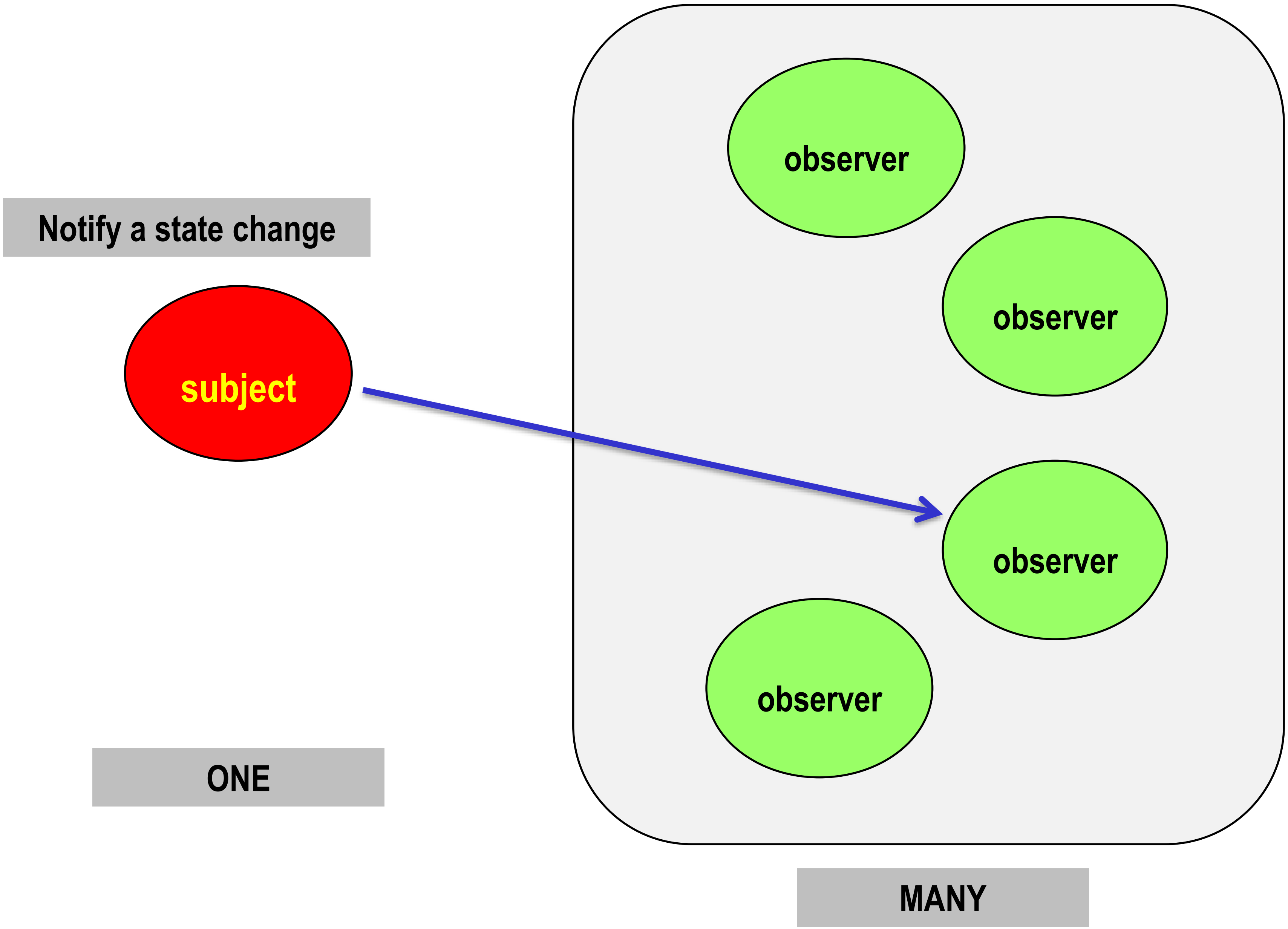
Observer Pattern



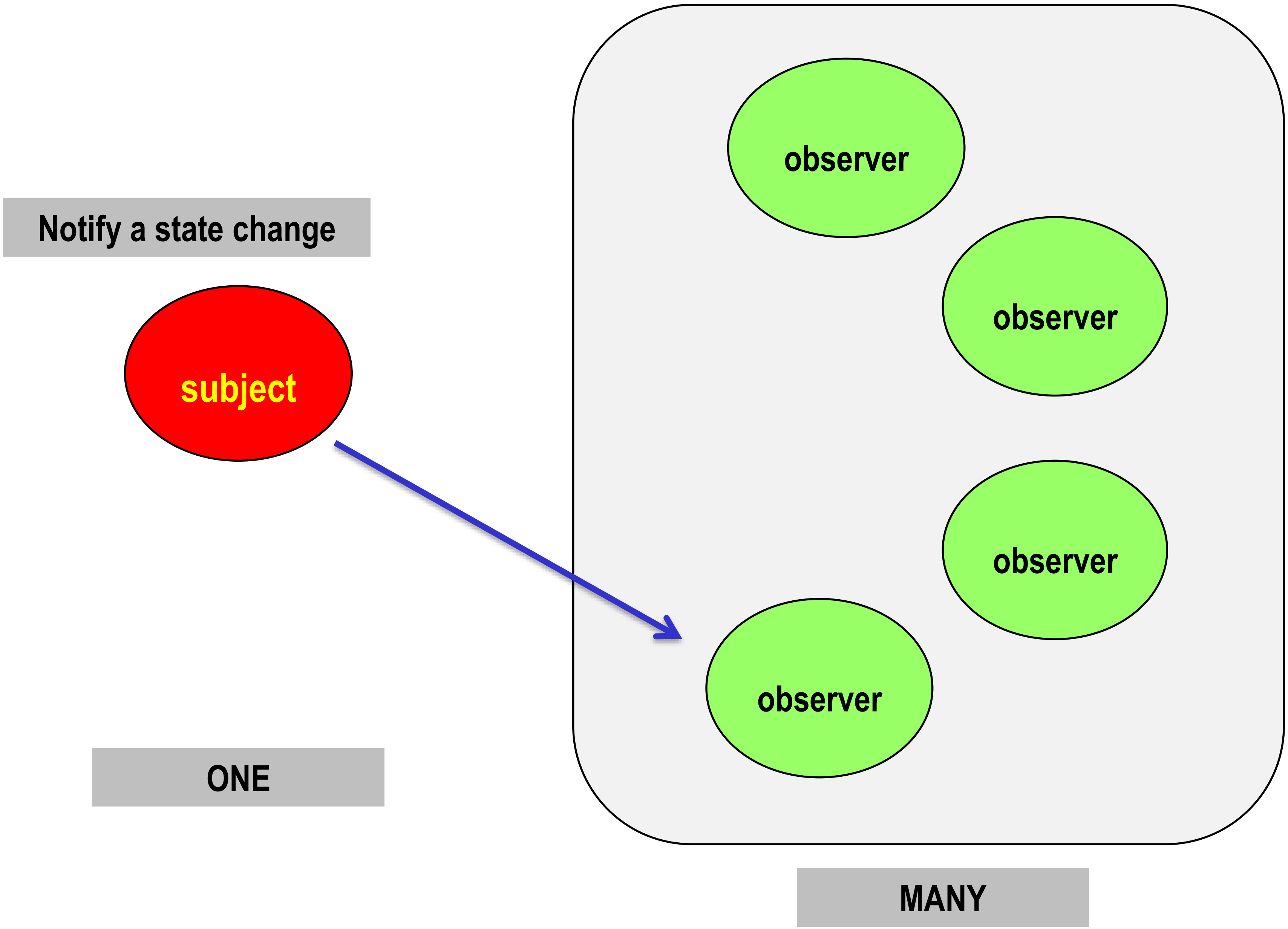
Observer Pattern



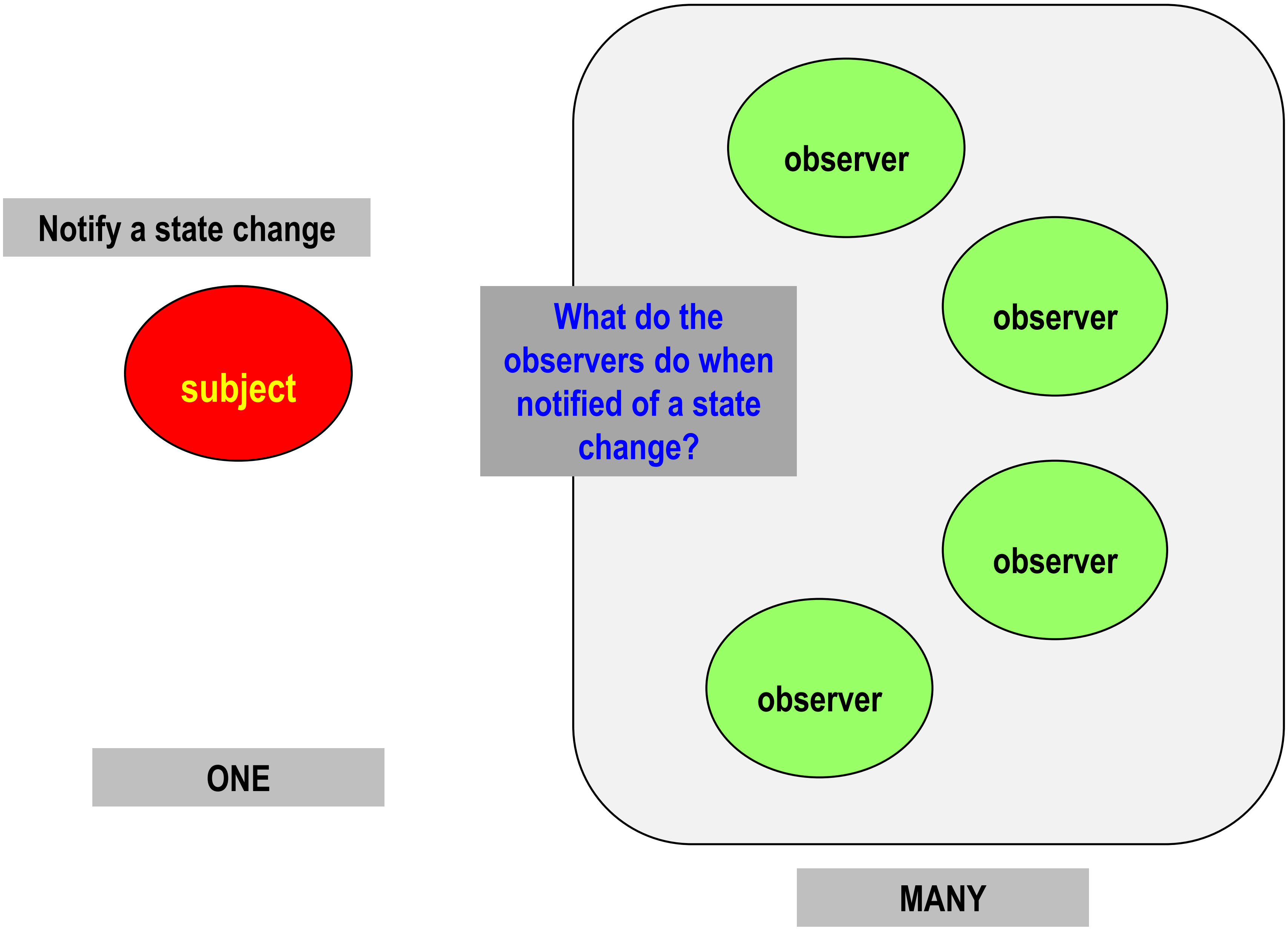
Observer Pattern



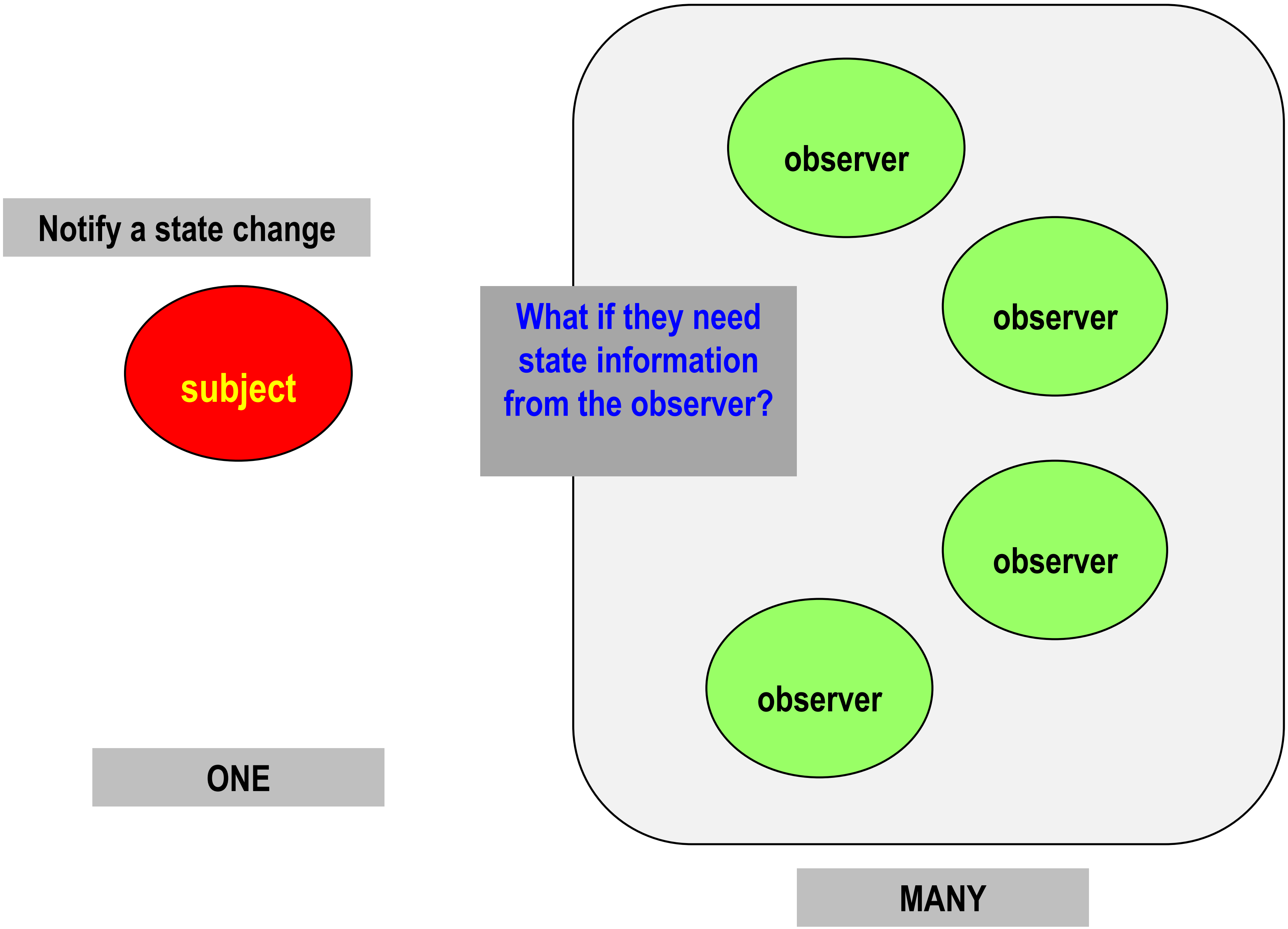
Observer Pattern



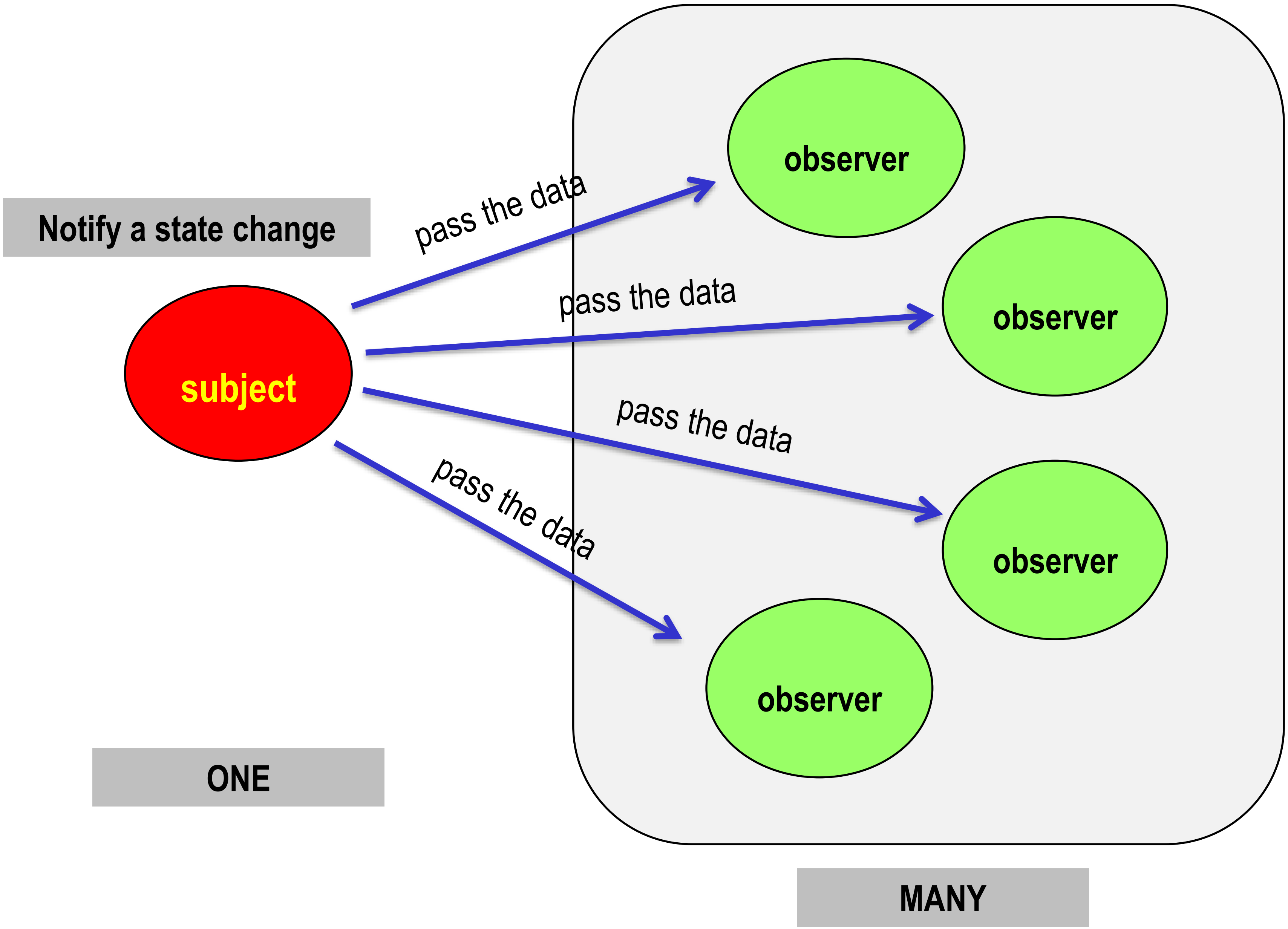
Observer Pattern



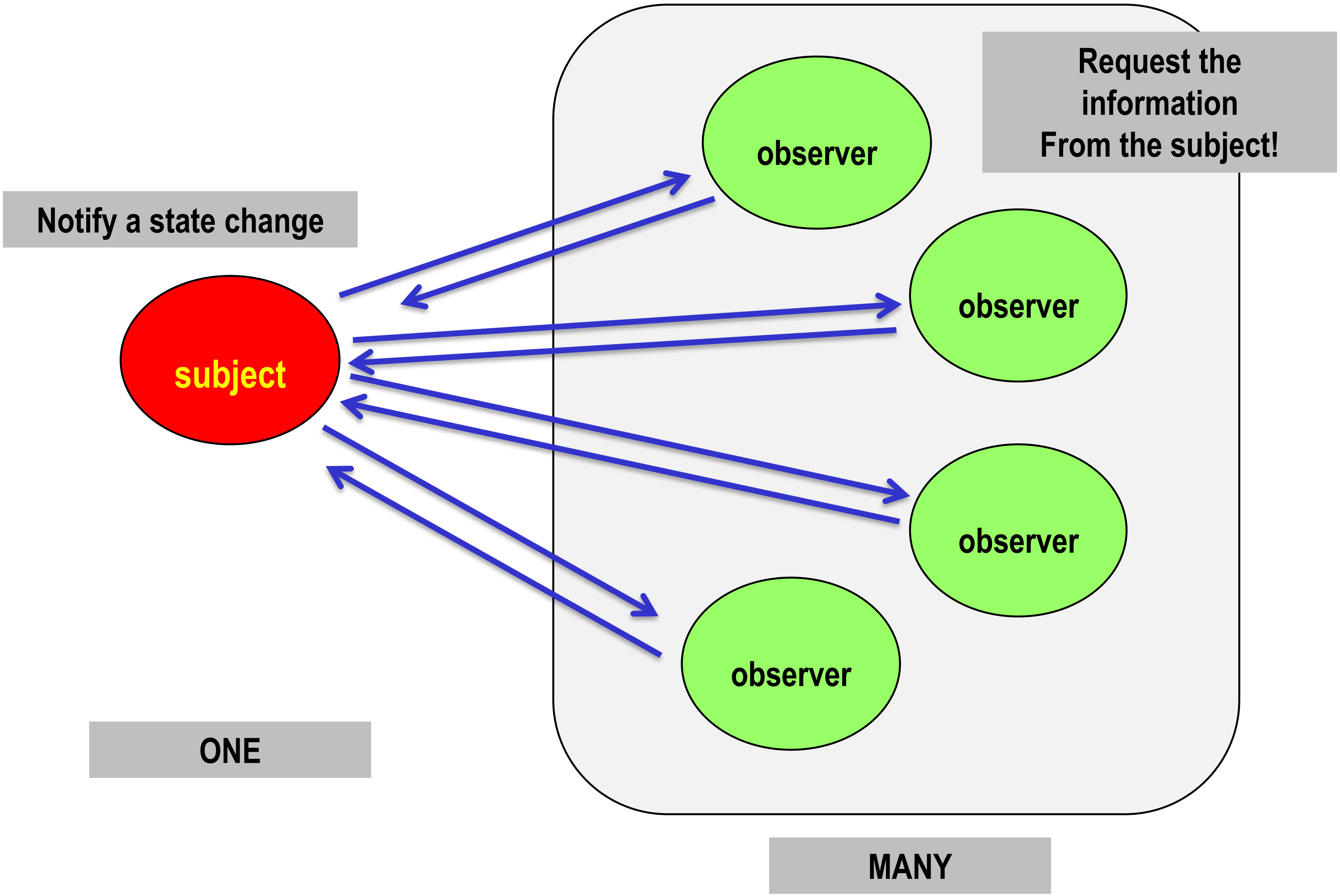
Observer Pattern



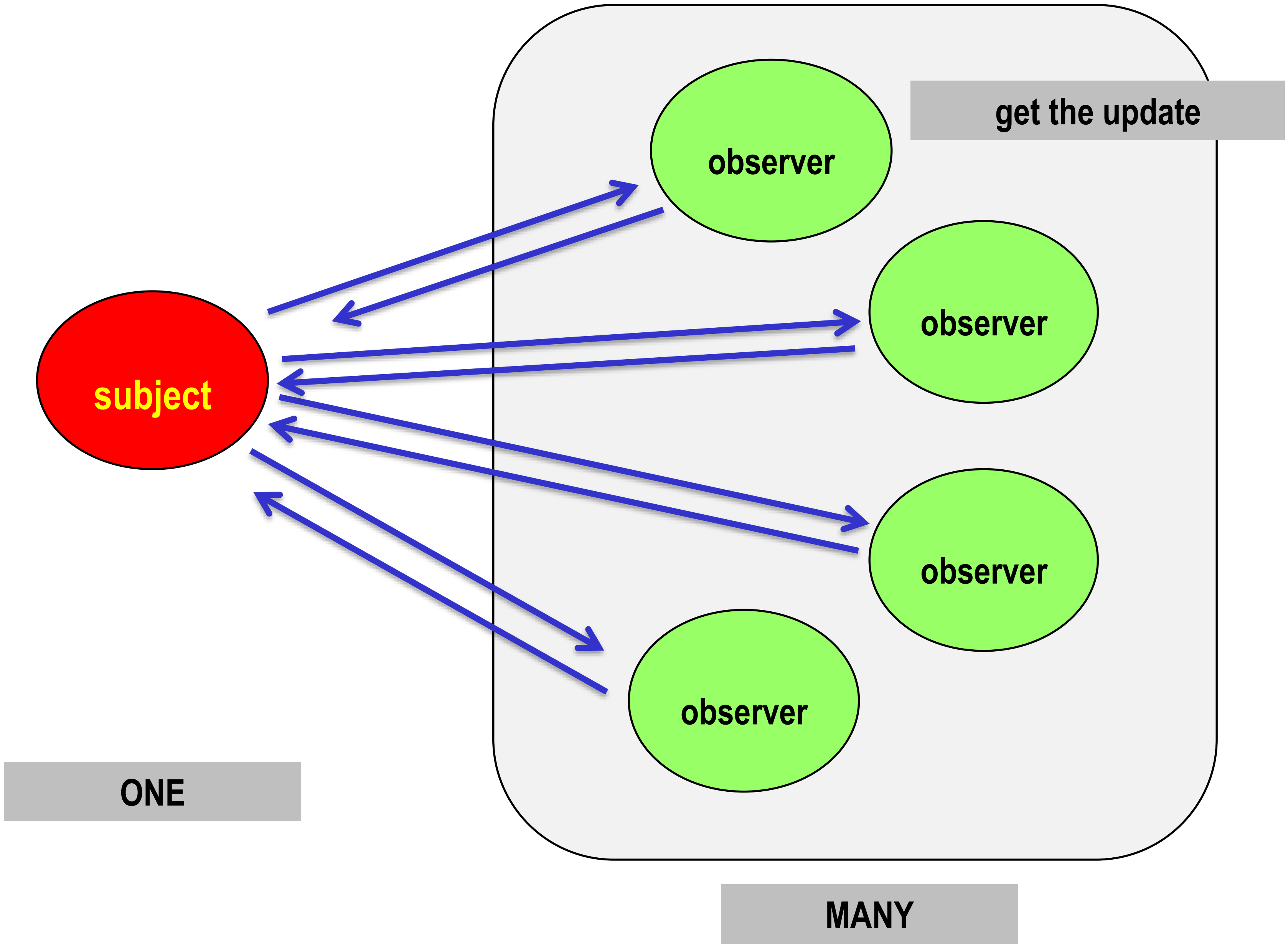
Observer Pattern



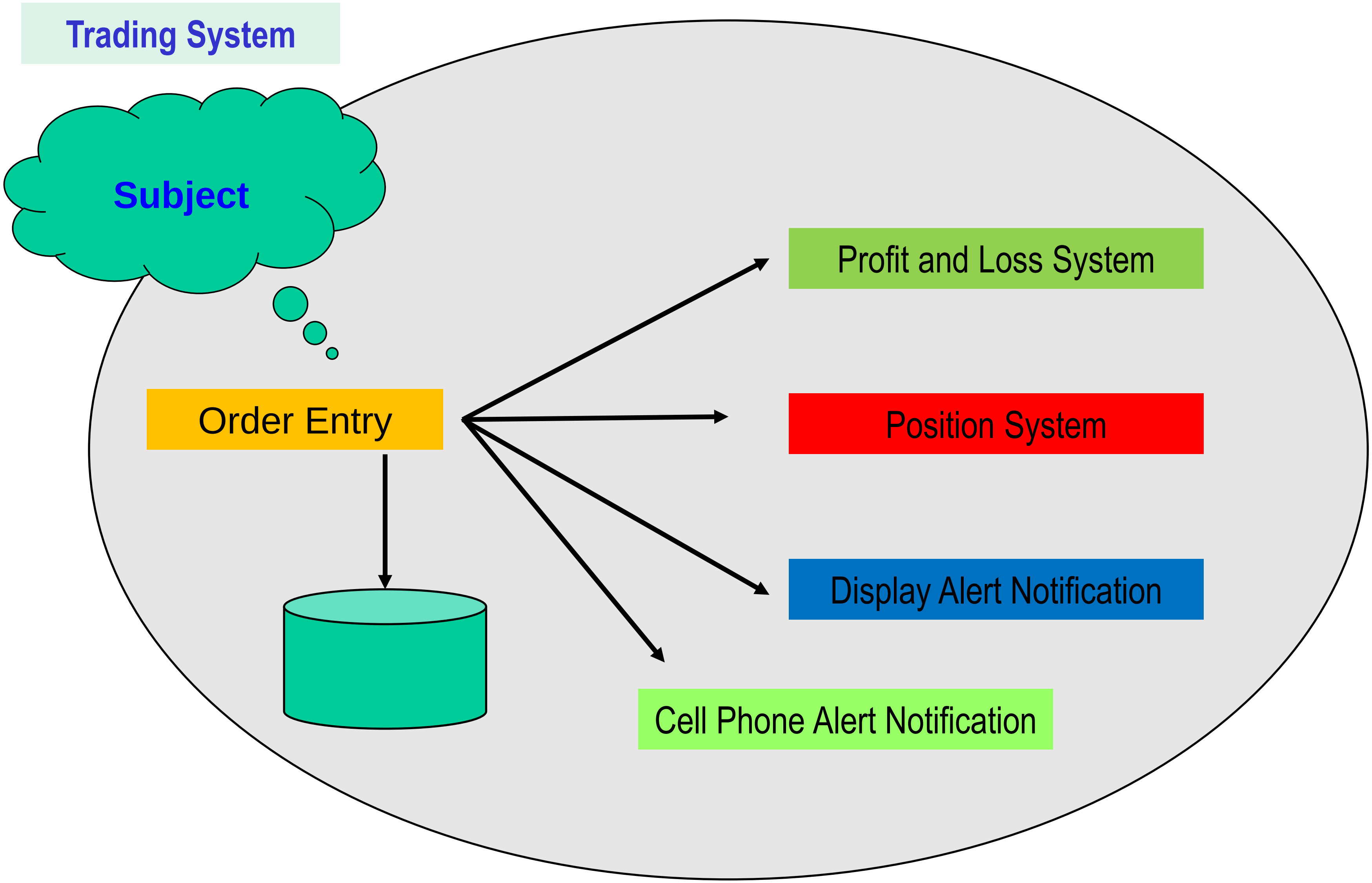
Observer Pattern



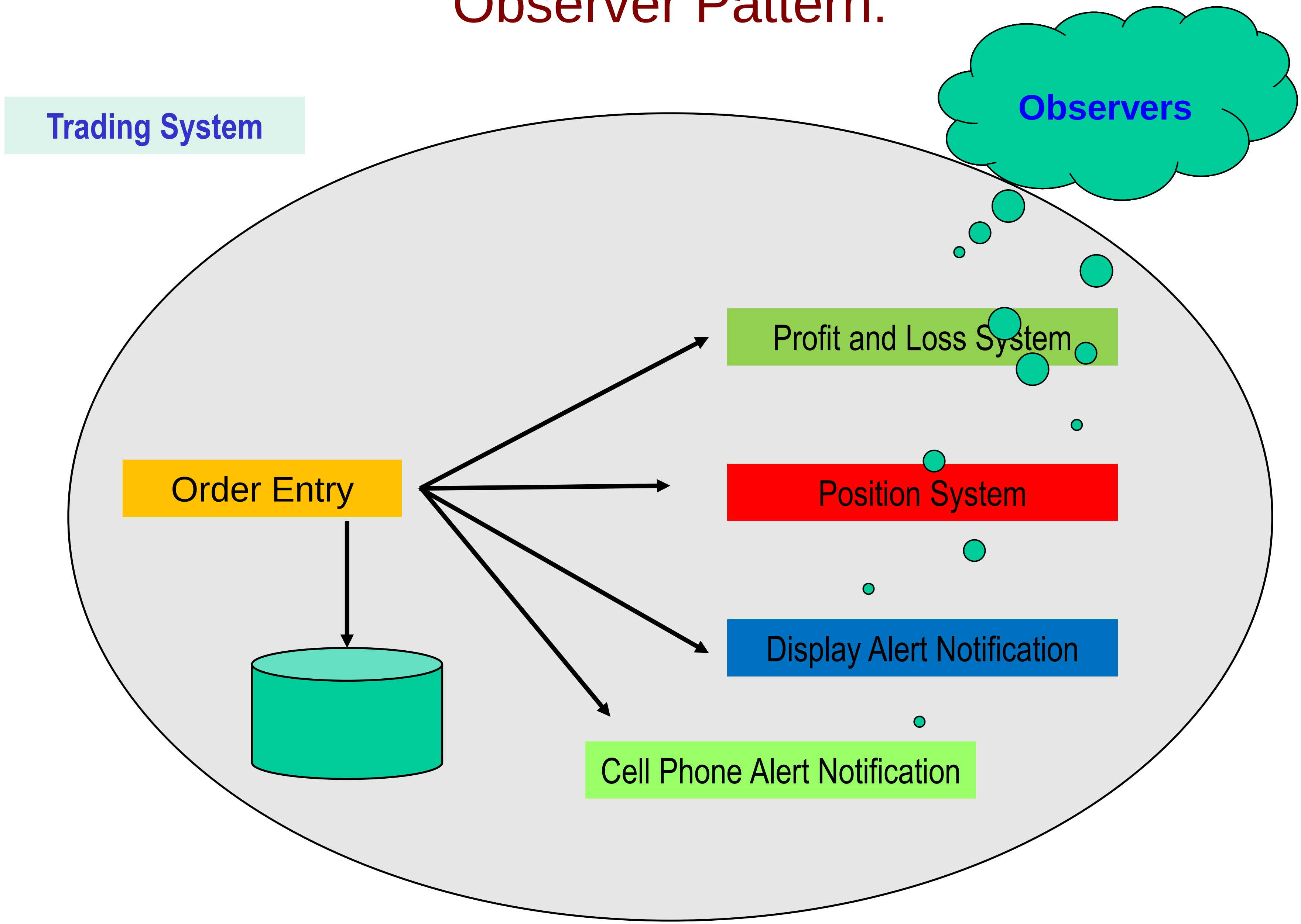
Observer Pattern



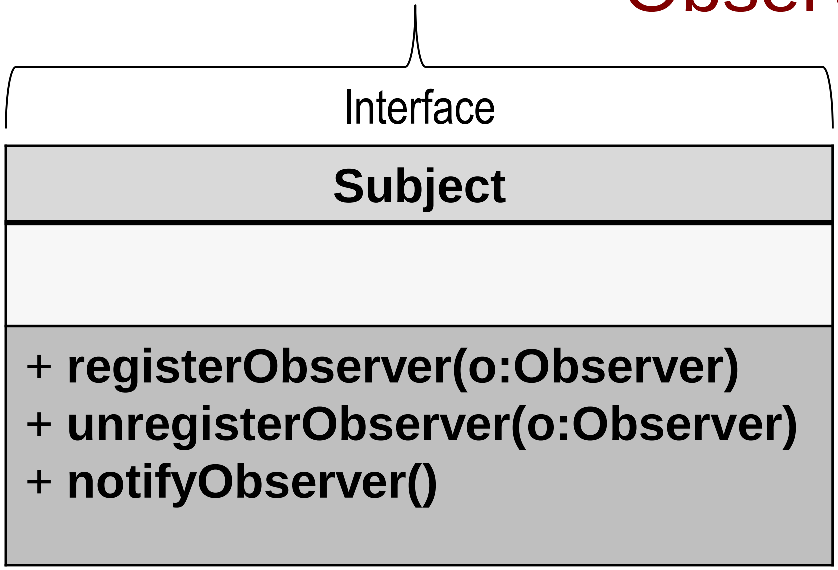
Observer Pattern:



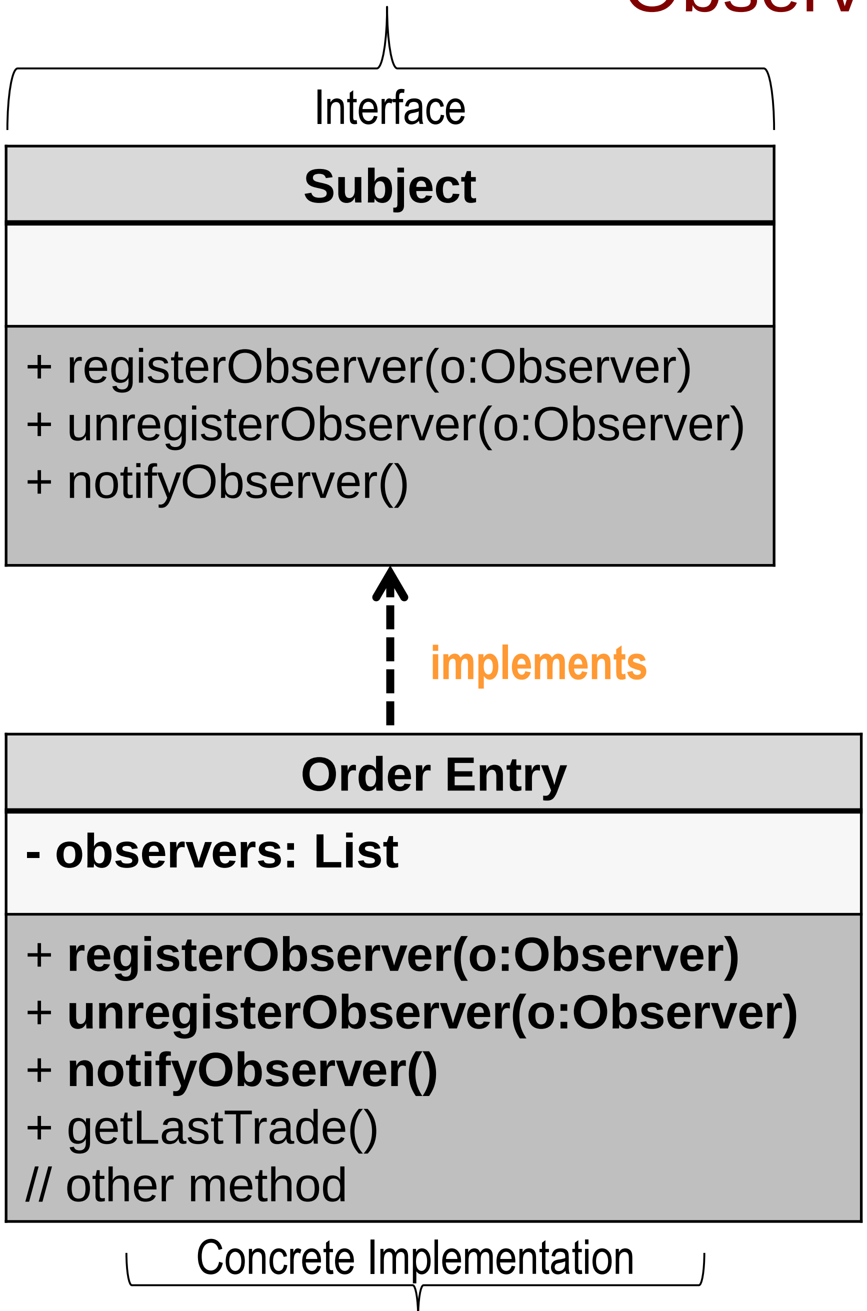
Observer Pattern:



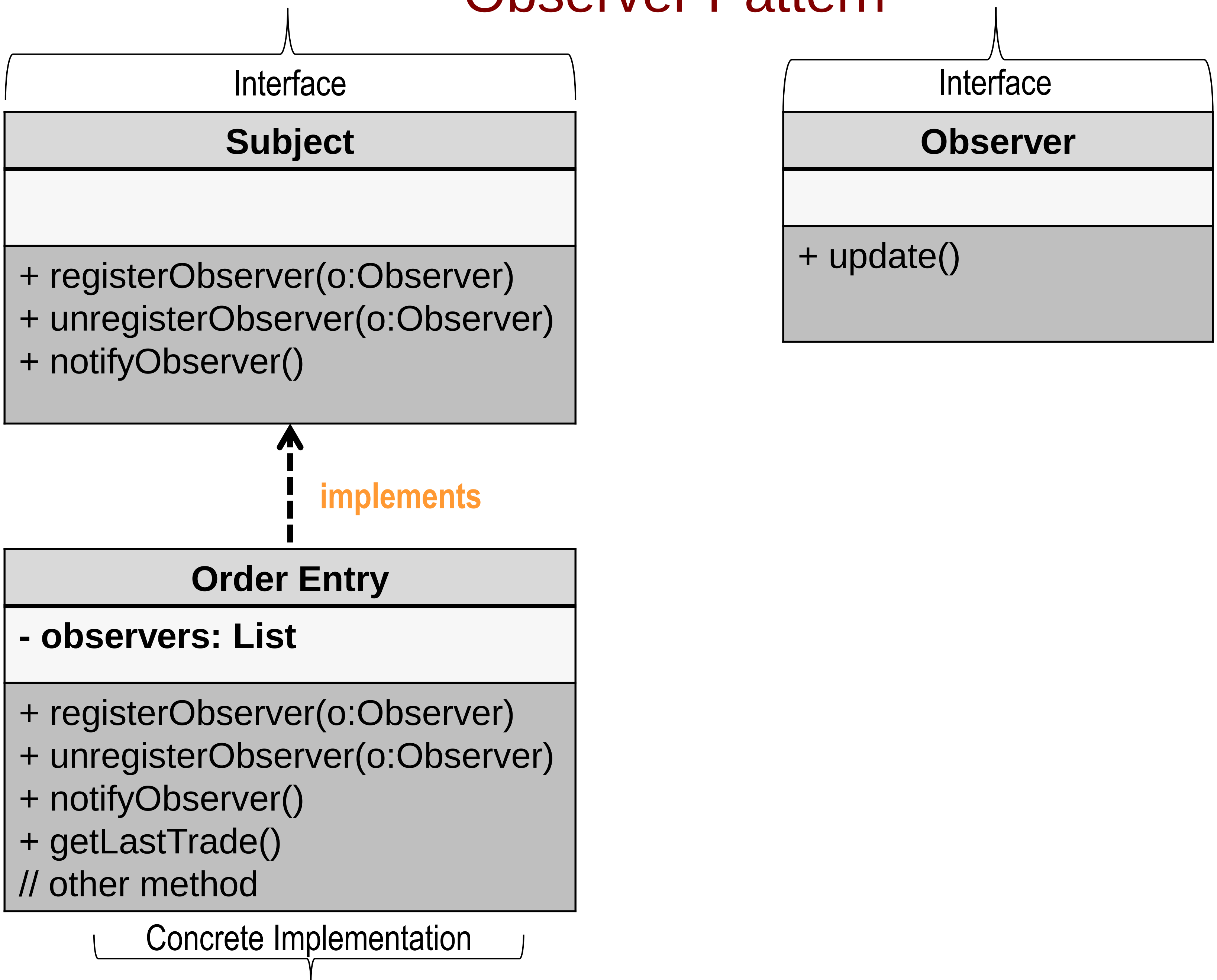
Observer Pattern



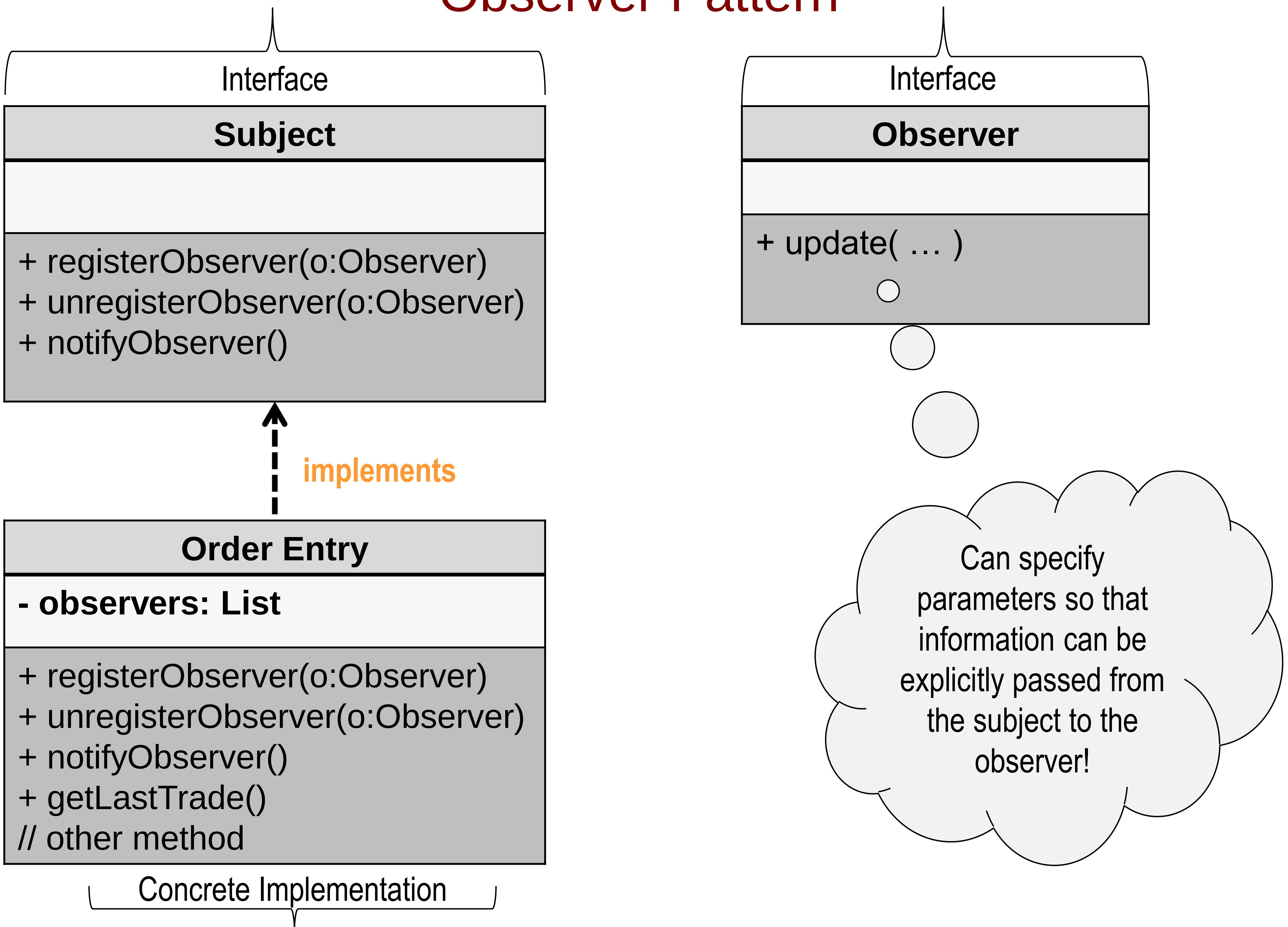
Observer Pattern



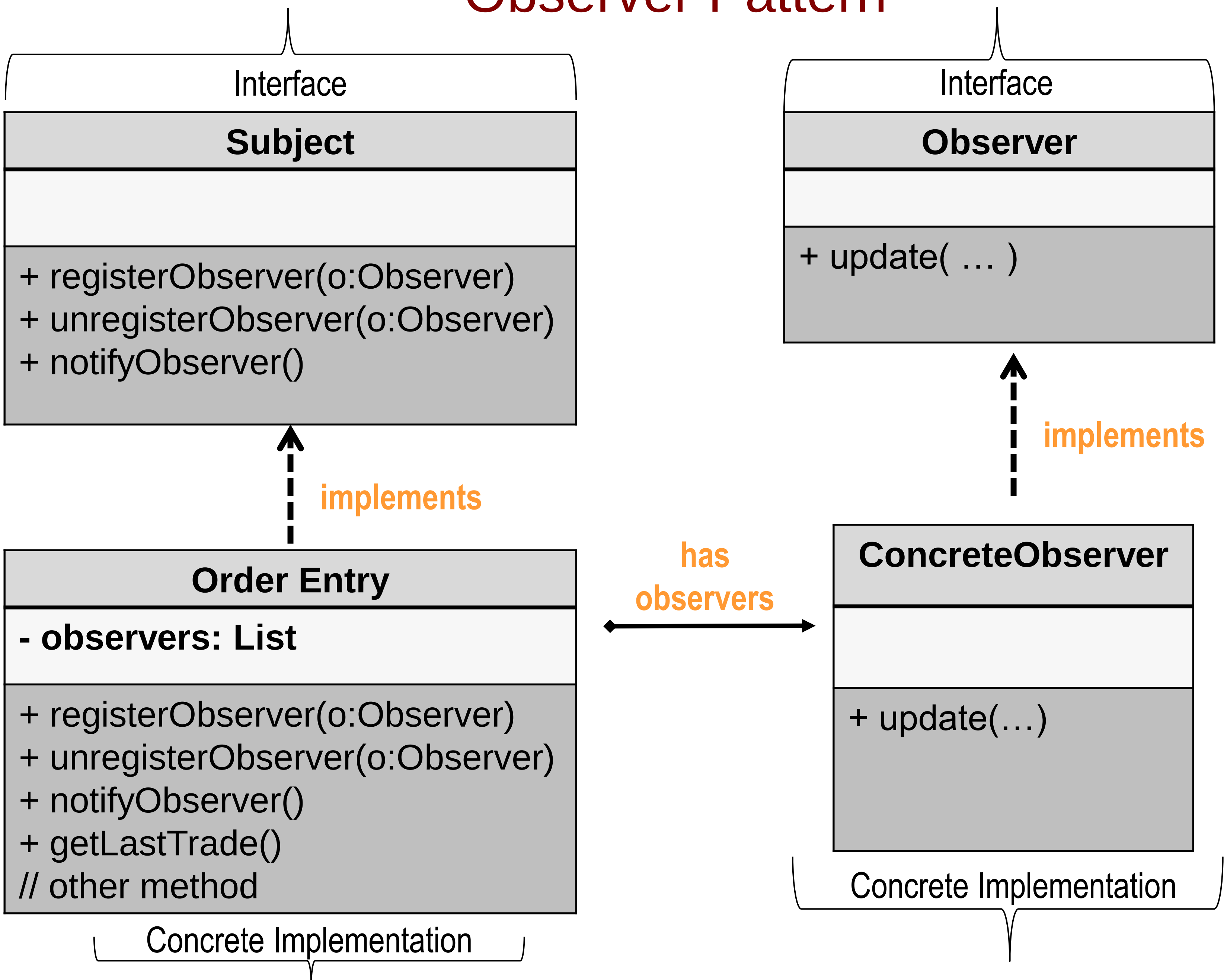
Observer Pattern



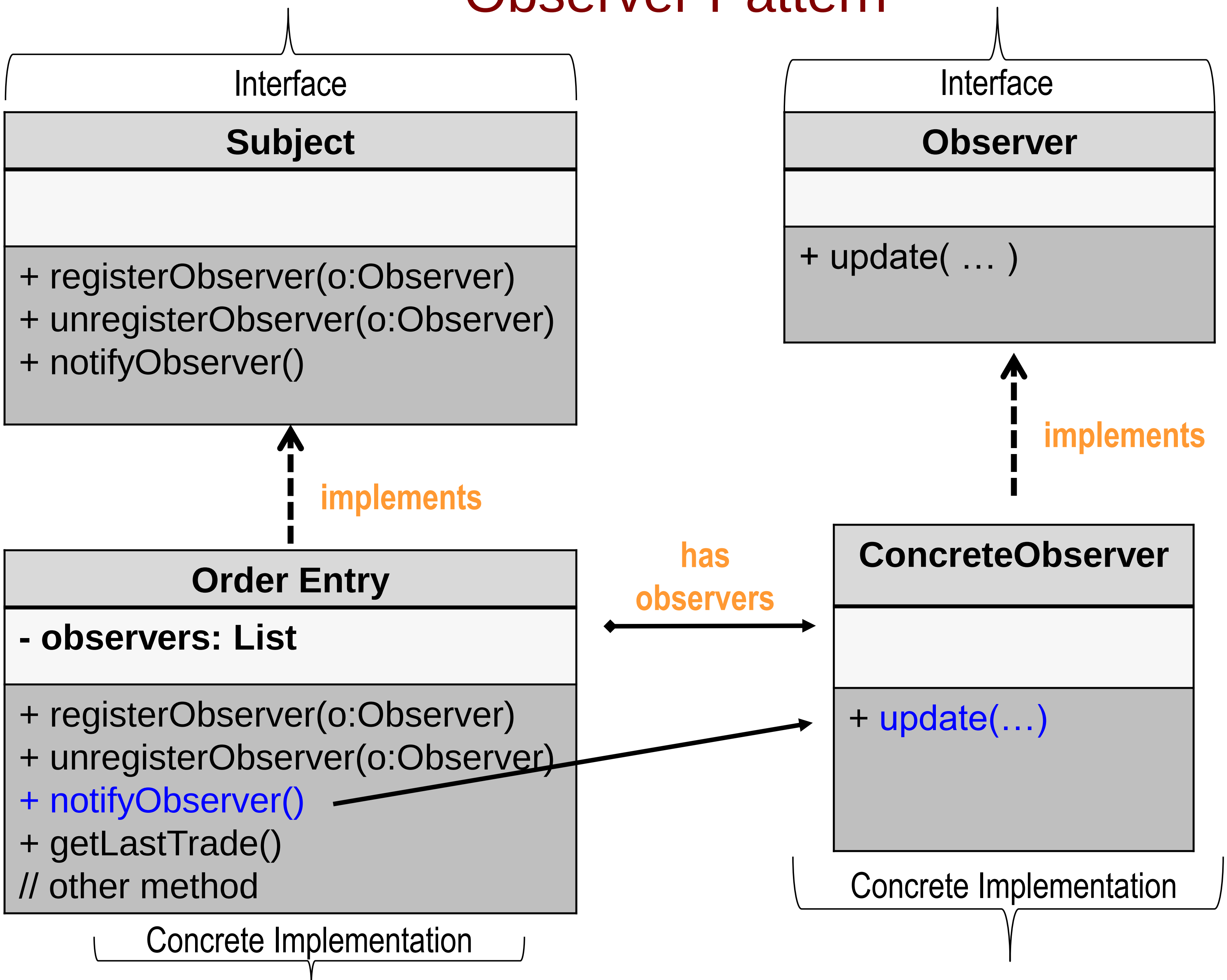
Observer Pattern



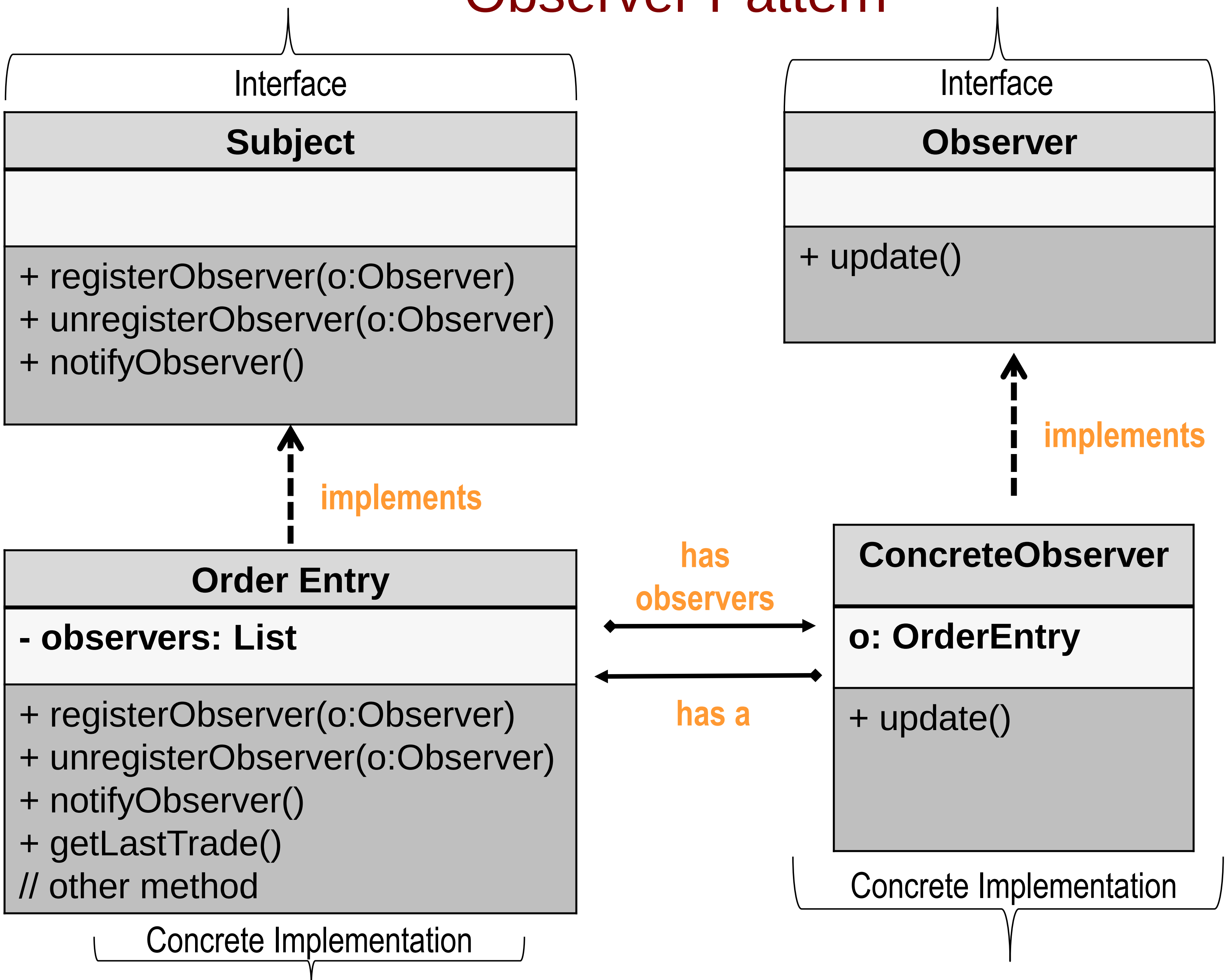
Observer Pattern



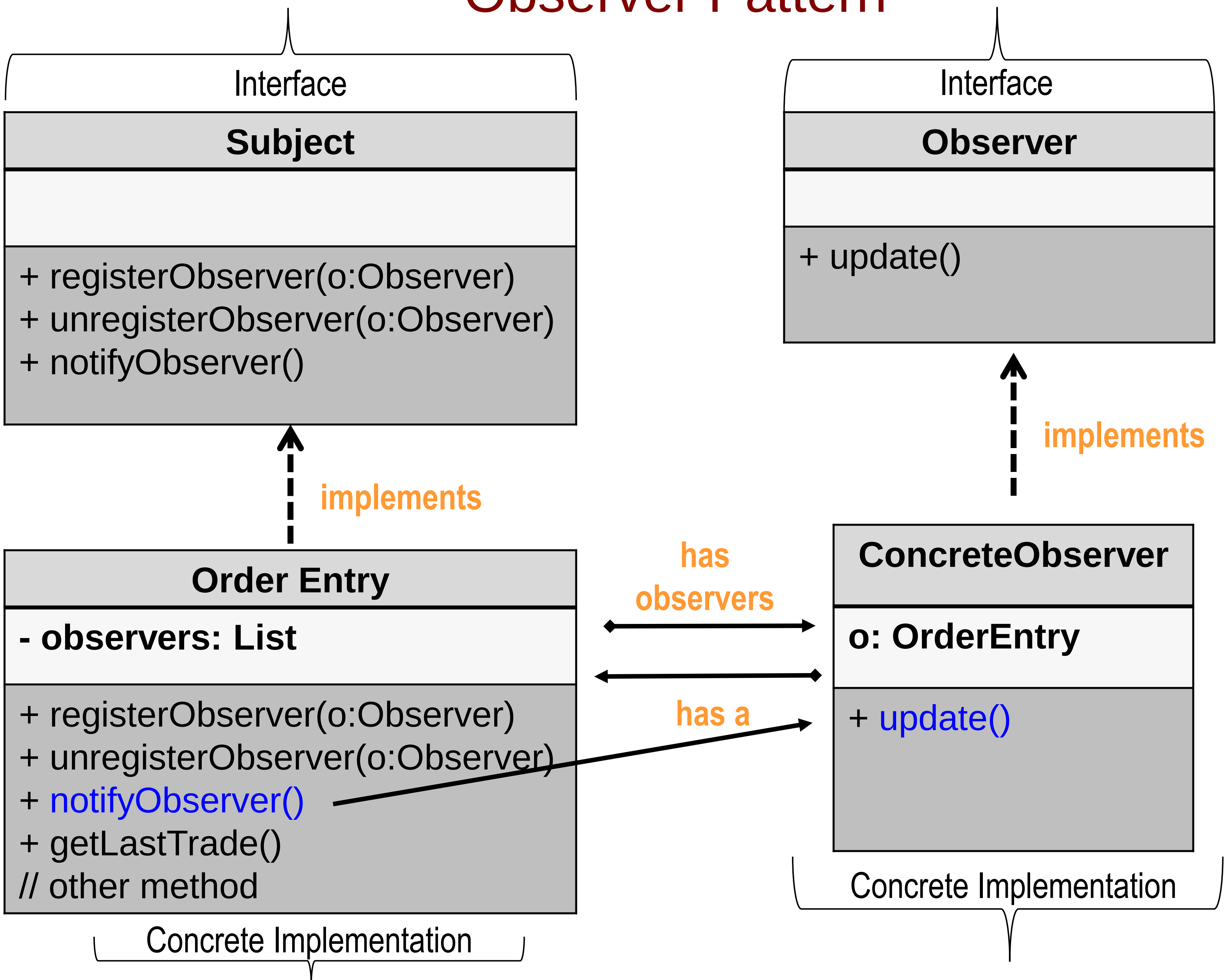
Observer Pattern



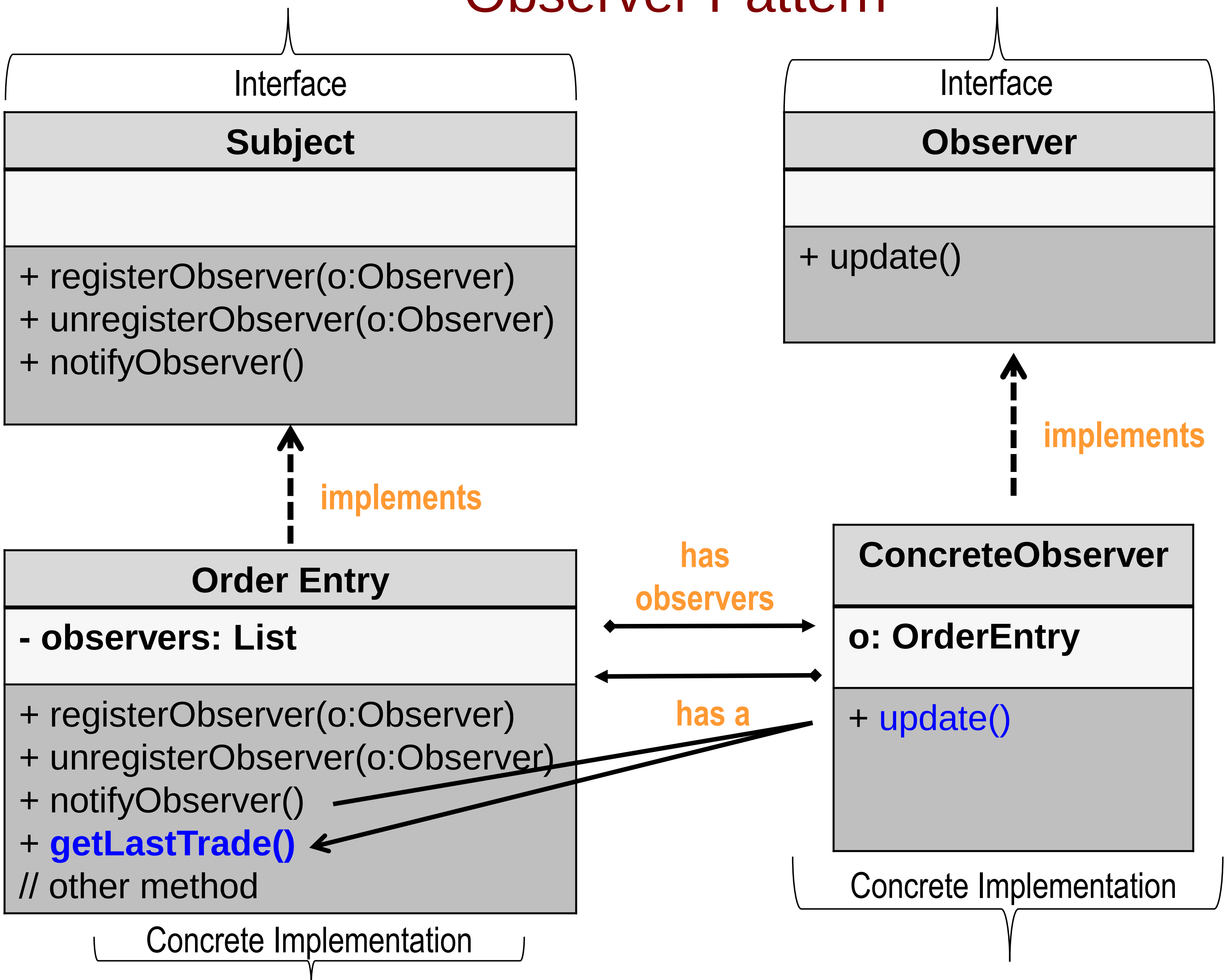
Observer Pattern



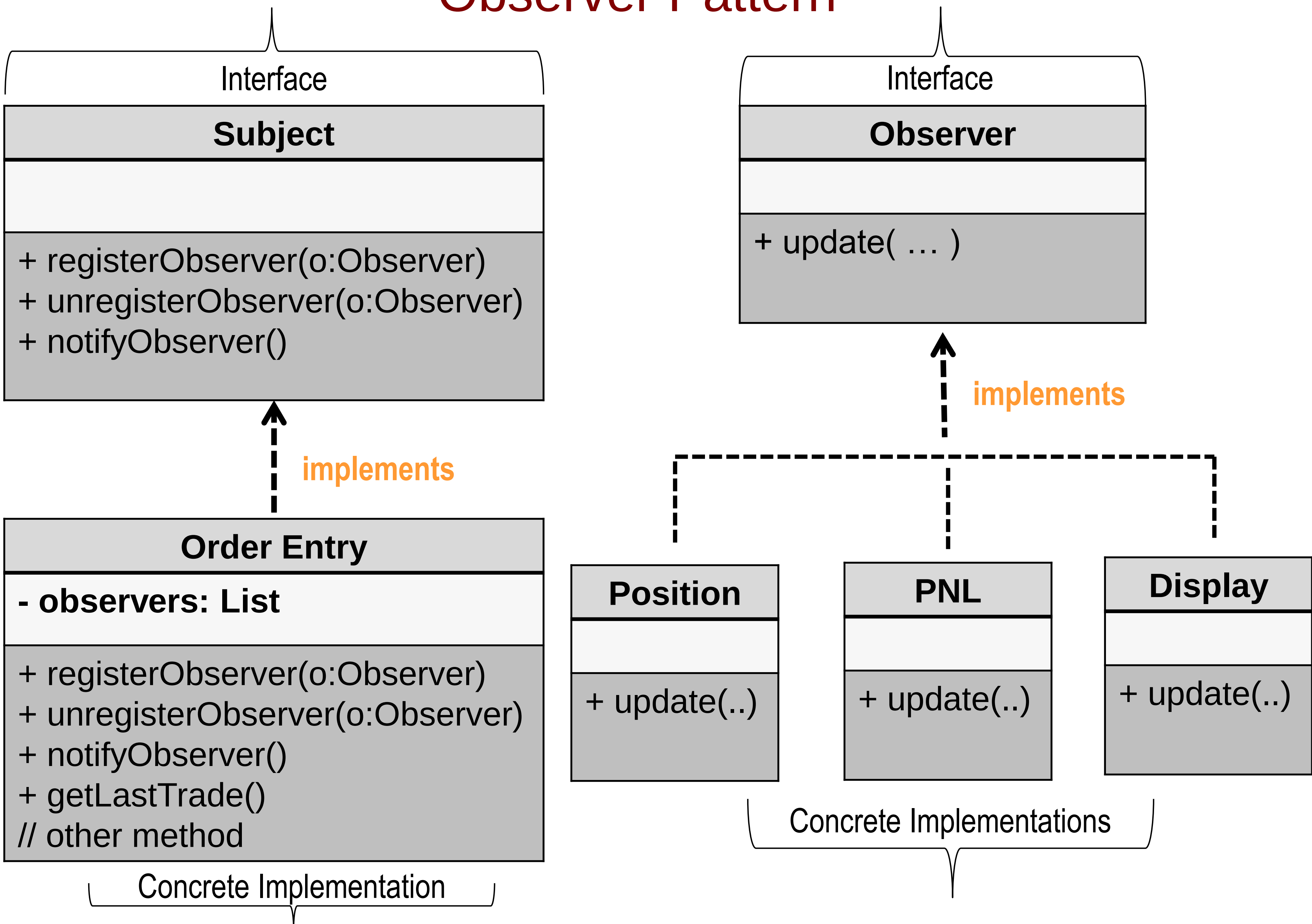
Observer Pattern



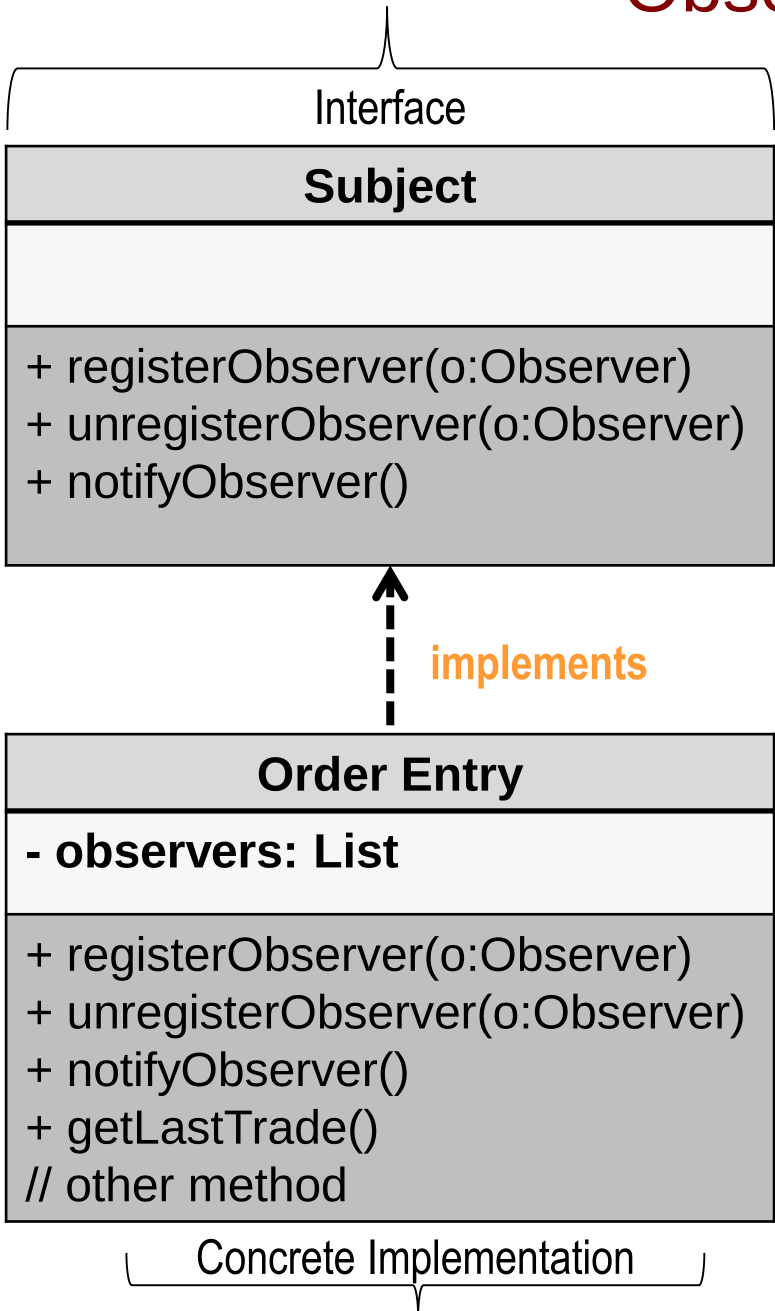
Observer Pattern



Observer Pattern

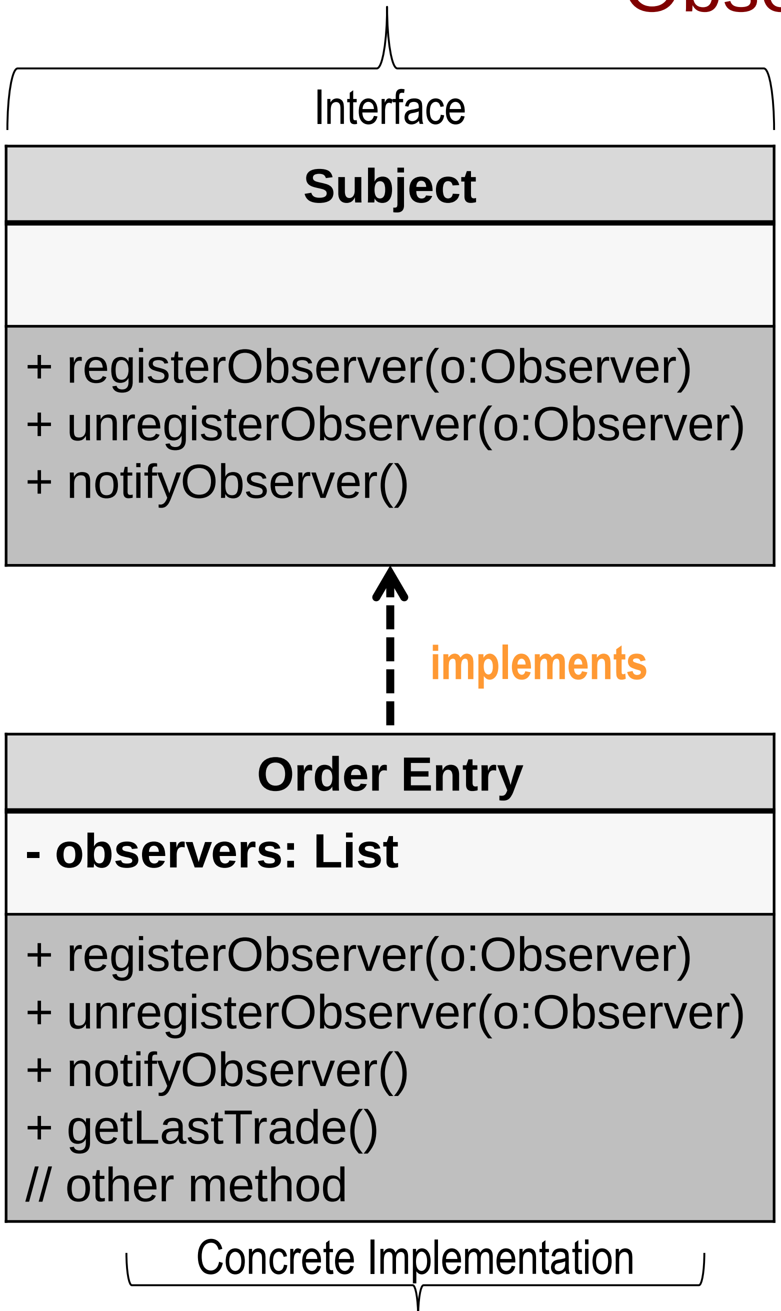


Observer Pattern



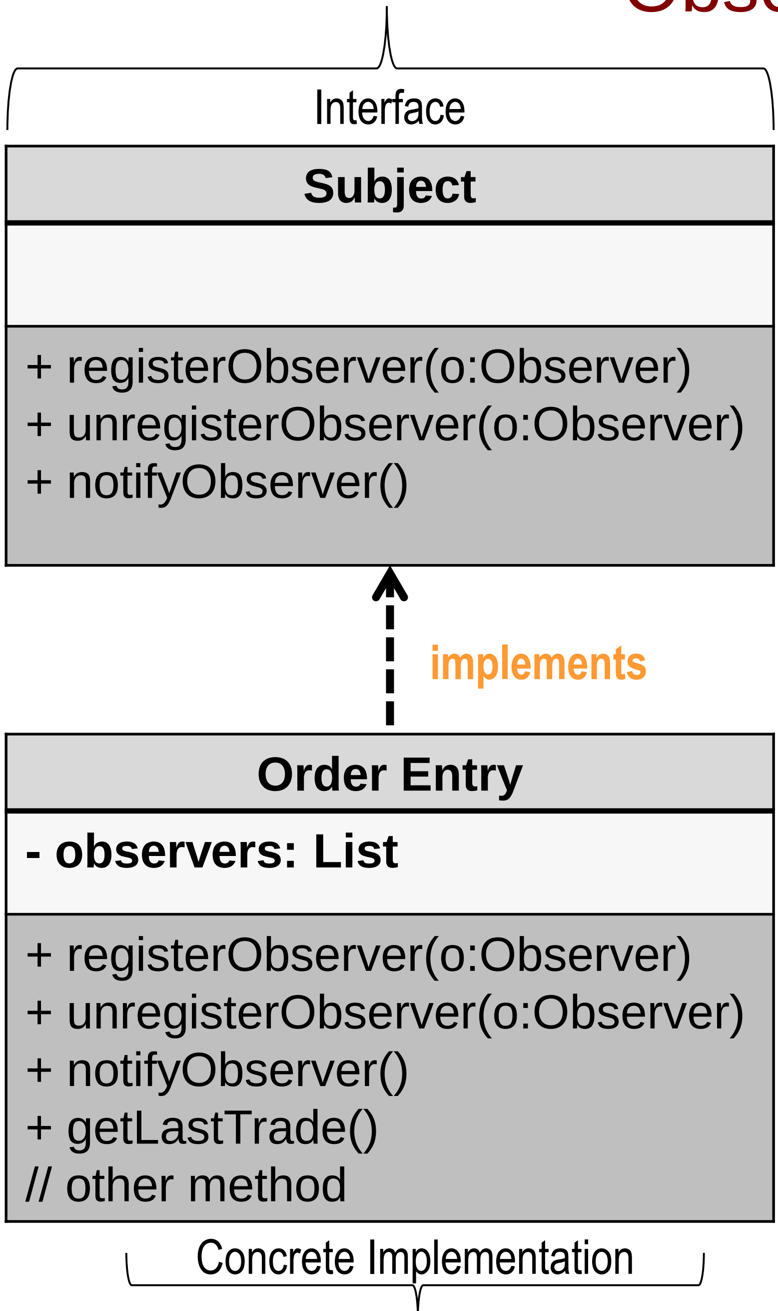

```
class OrderEntry implements Subject {

 ArrayList<Observer> observerList;

 public OrderEntry() {
   observerList =
      new ArrayList<Observer>();
 }

 public void registerObserver(Observer o)
 {
   observerList.add(o);
 }

 public void unregisterObserver( .. o )
 {
   observerList.remove(o);
 }

 notifyObserver() {
    forEach( Observer o : observerList )
       o.update();
 }

} // class
```

Observer Pattern



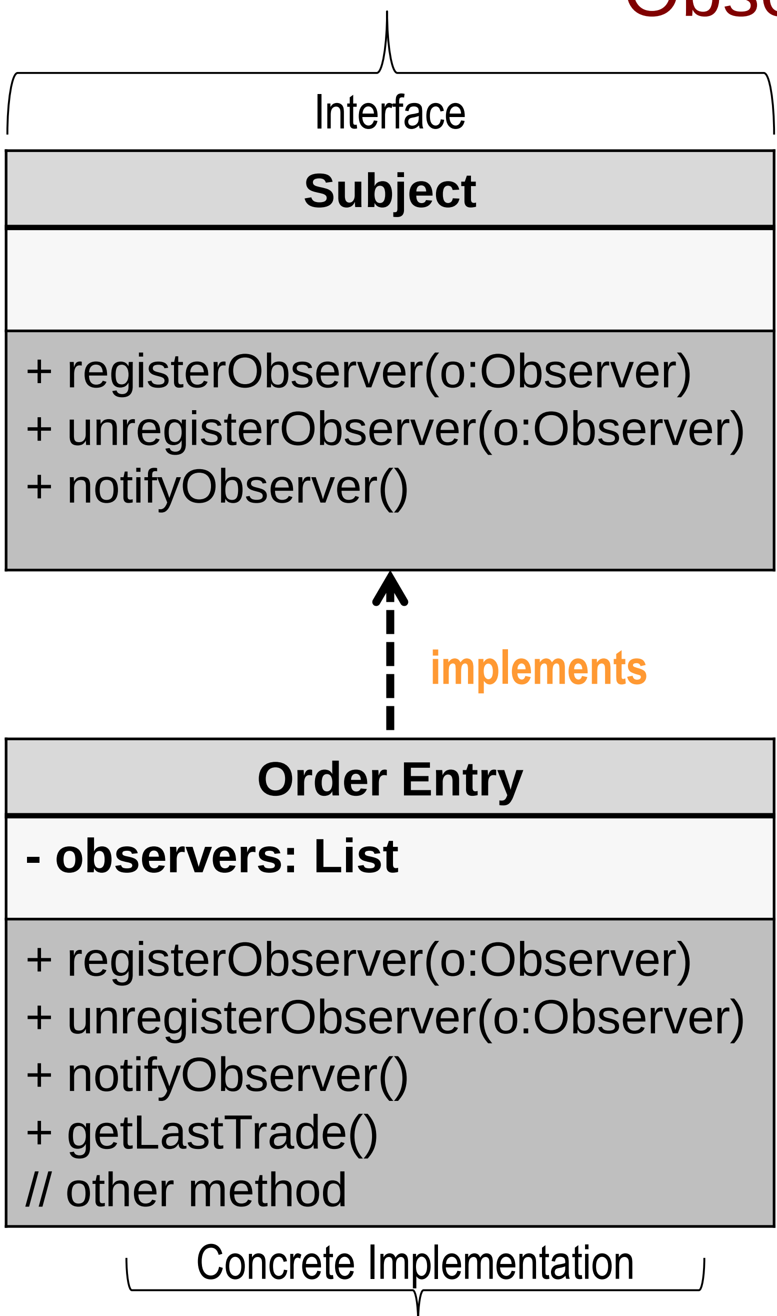

```
class OrderEntry implements Subject {

 ArrayList<Observer> observerList;

 public OrderEntry() {
   observerList =
      new ArrayList<Observer>();
 }

 public void registerObserver(Observer o)
 {
   observerList.add(o);
 }

 public void unregisterObserver( .. o )
 {
   observerList.remove(o);
 }

 notifyObserver() {
    forEach( Observer o : observerList )
       o.update();
 }

} // class
```

Observer Pattern



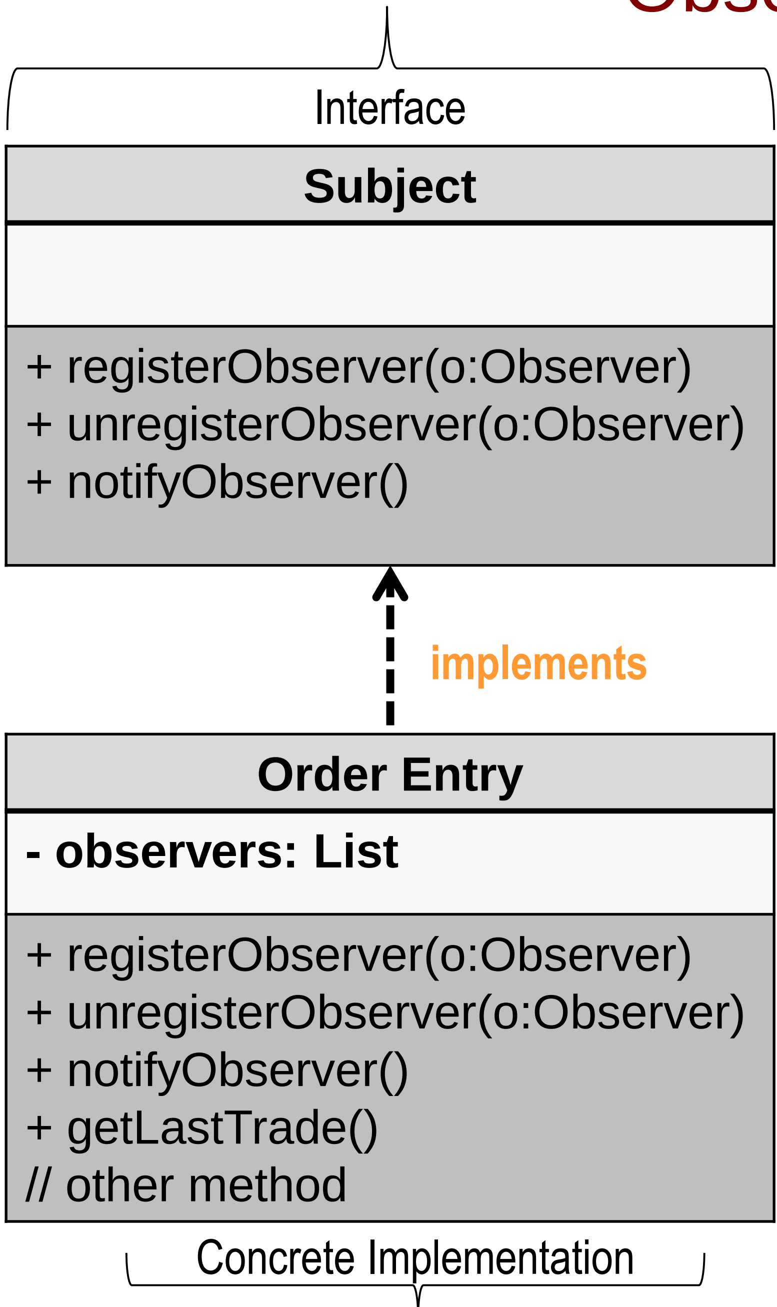

```
class OrderEntry implements Subject {

 ArrayList<Observer> observerList;

 public OrderEntry() {
   observerList =
       new ArrayList<Observer>();
 }

 public void registerObserver(Observer o)
 {
   observerList.add(o);
 }

 public void unregisterObserver( .. o )
 {
   observerList.remove(o);
 }

 notifyObserver() {
    forEach( Observer o : observerList )
       o.update();
 }

} // class
```

Observer Pattern



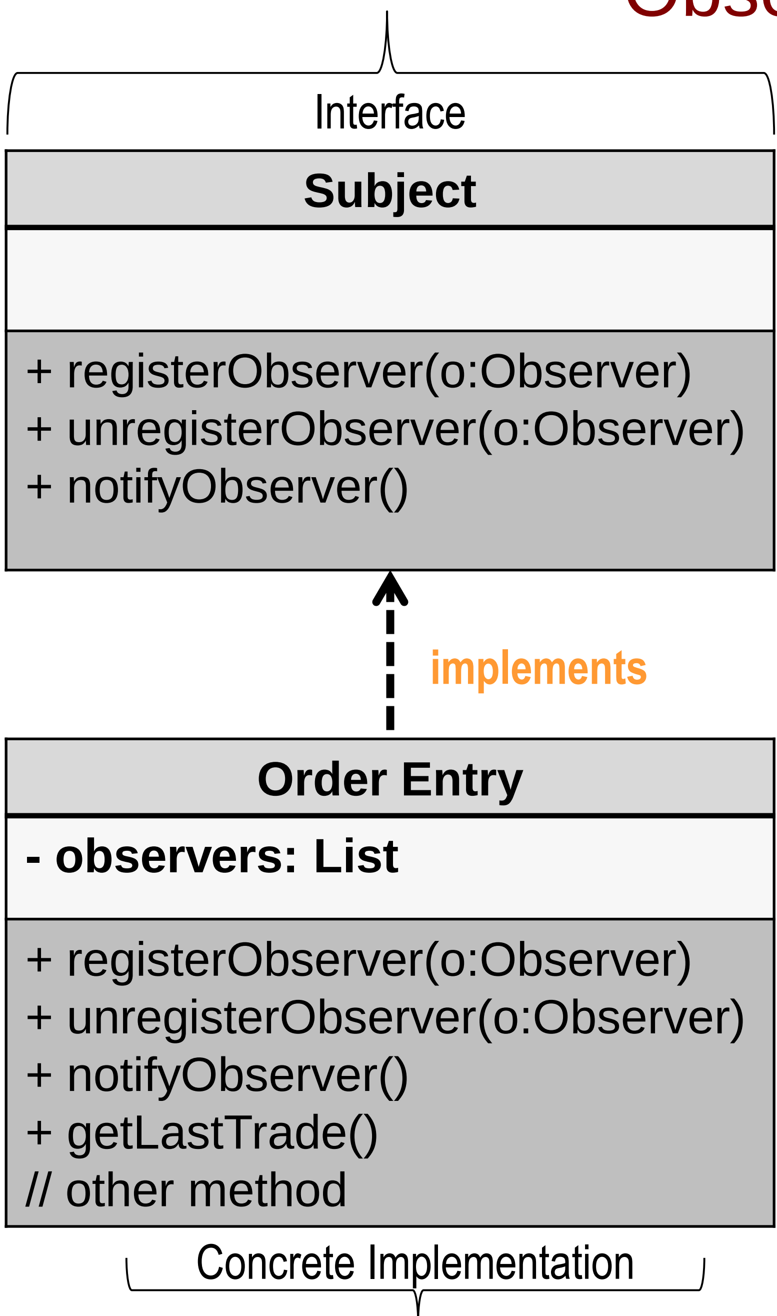

```
class OrderEntry implements Subject {

 ArrayList<Observer> observerList;

 public OrderEntry() {
   observerList =
      new ArrayList<Observer>();
 }

 public void registerObserver(Observer o)
 {
   observerList.add(o);
 }

 public void unregisterObserver( .. o )
 {
   observerList.remove(o);
 }

 notifyObserver() {
    forEach( Observer o : observerList )
       o.update();
 }

} // class
```

Observer Pattern




```
class OrderEntry implements Subject {

 ArrayList<Observer> observerList;

 public OrderEntry() {
   observerList =
      new ArrayList<Observer>();
 }

 public void registerObserver(Observer o)
 {
   observerList.add(o);
 }

 public void unregisterObserver( .. o )
 {
   observerList.remove(o);
 }

 notifyObserver() {
    forEach( Observer o : observerList )
       o.update();
 }

} // class
```

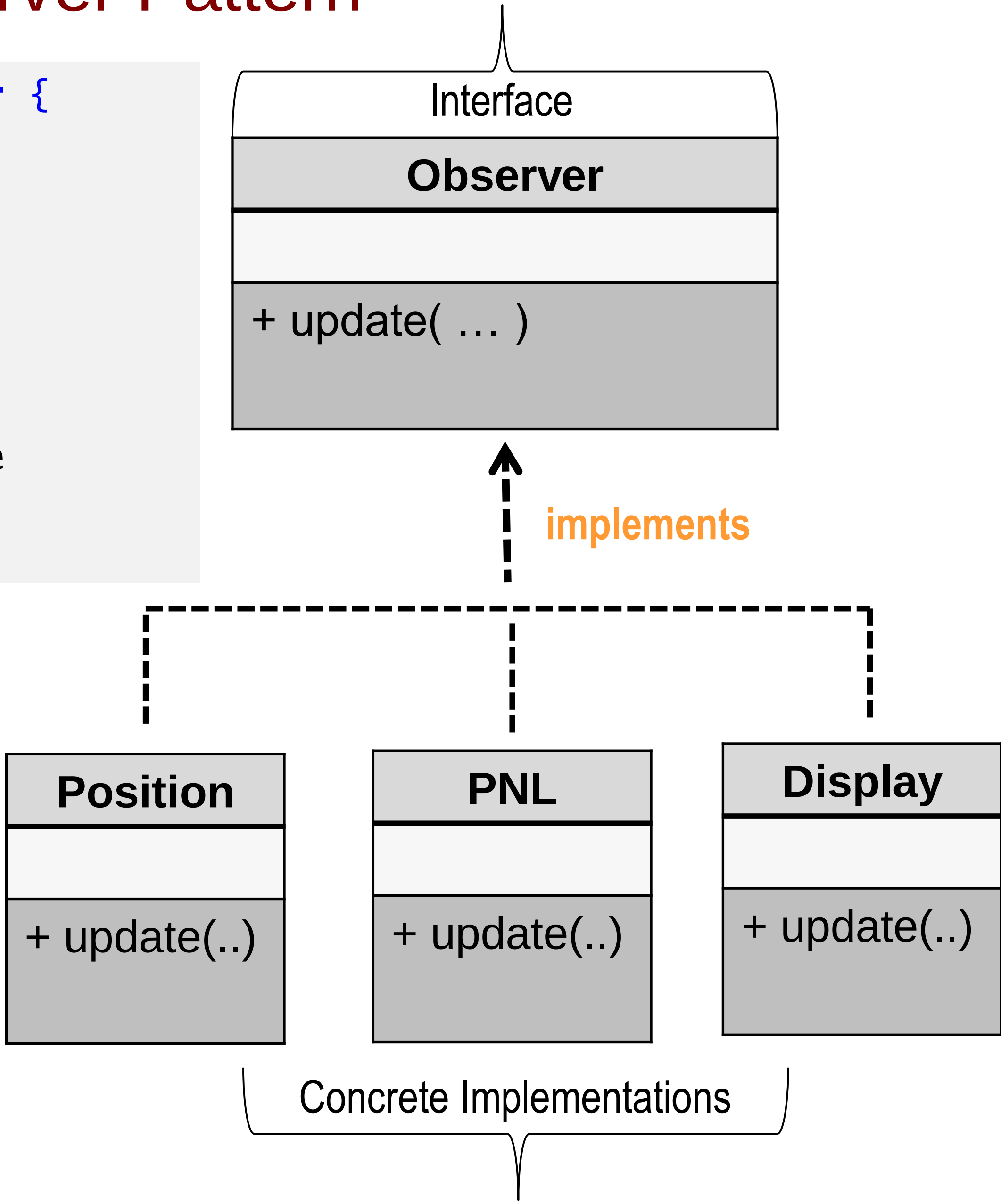
Observer Pattern




```
class OrderEntry implements Subject {

 ArrayList<Observer> observerList;

 public OrderEntry() {
   observerList =
      new ArrayList<Observer>();
 }

 public void registerObserver(Observer o)
 {
   observerList.add(o);
 }

 public void unregisterObserver( .. o )
 {
   observerList.remove(o);
 }

 notifyObserver() {
    forEach( Observer o : observerList )
       o.update( ... );
 }

} // class
```

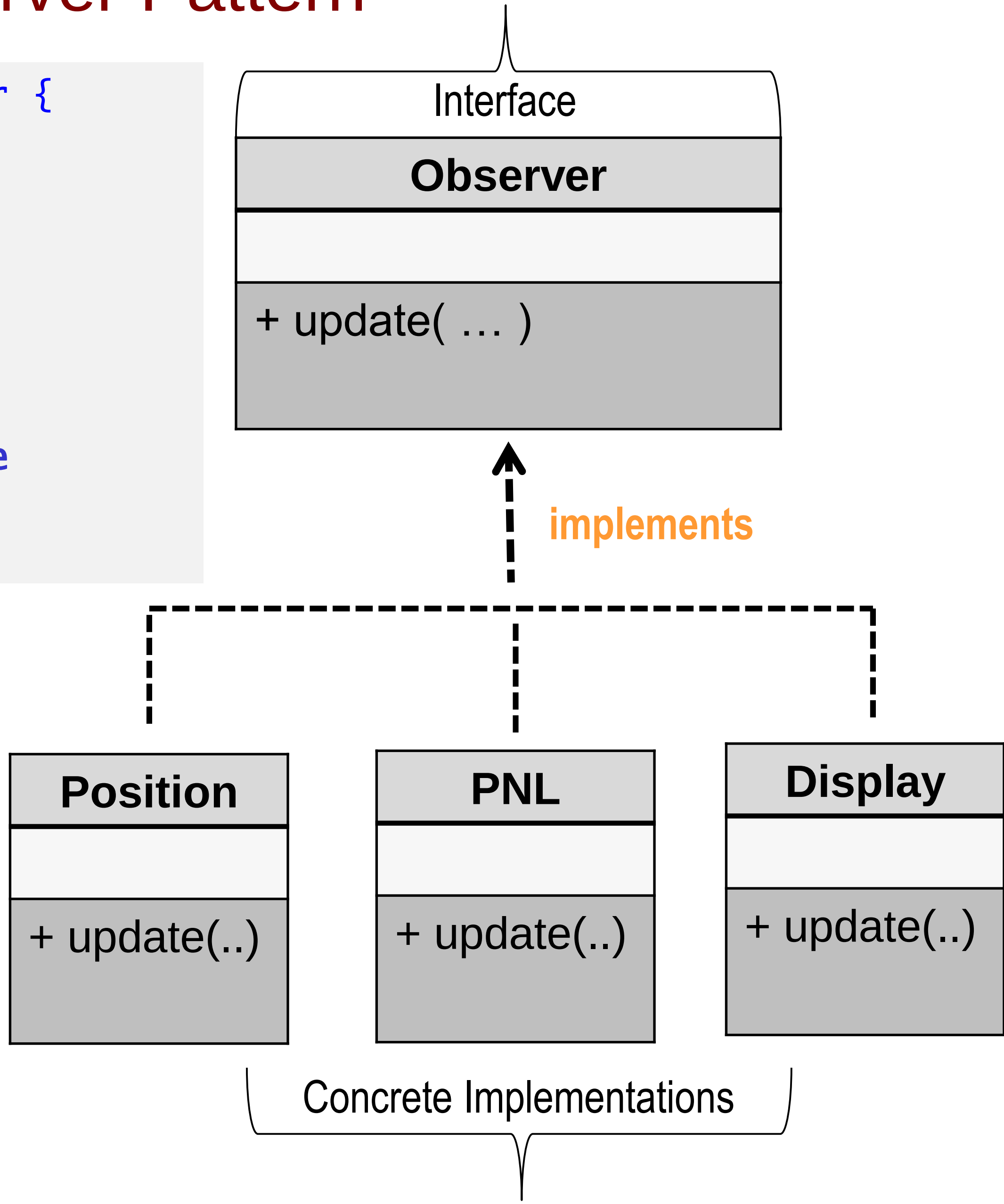
Observer Pattern

```
class Position implements Observer {


 public Position() {
    // constructor
 }

 public update( ... ) {
    // performs an internal update
 }
} // class
```



Observer Pattern

```
class Position implements Observer {


 public Position() {
    // constructor
 }

 public update( ... ) {
    // performs an internal update
 }
} // class
```



Implementation

```
public class ObserverSimulator {

   public static void main( String[] a ) {
      // create the observers
      Position posn = new Position();
      PNL pnl = new PNL();

      // createt the Subject
      OrderEntry oe = new OrderEntry();

      // Register the observers
      oe.register( (Observer) posn );
      oe.register( (Observer) pnl );

      // kick off trading…

   } // main
} // class
```

Implementation

```
public class ObserverSimulator {

   public static void main( String[] a ) {
      // create the observers
      Position posn = new Position();
      PNL pnl = new PNL();

      // createt the Subject
      OrderEntry oe = new OrderEntry();

      // Register the observers
      oe.register( (Observer) posn );
      oe.register( (Observer) pnl );

      // kick off trading…

   } // main
} // class
```

Implementation

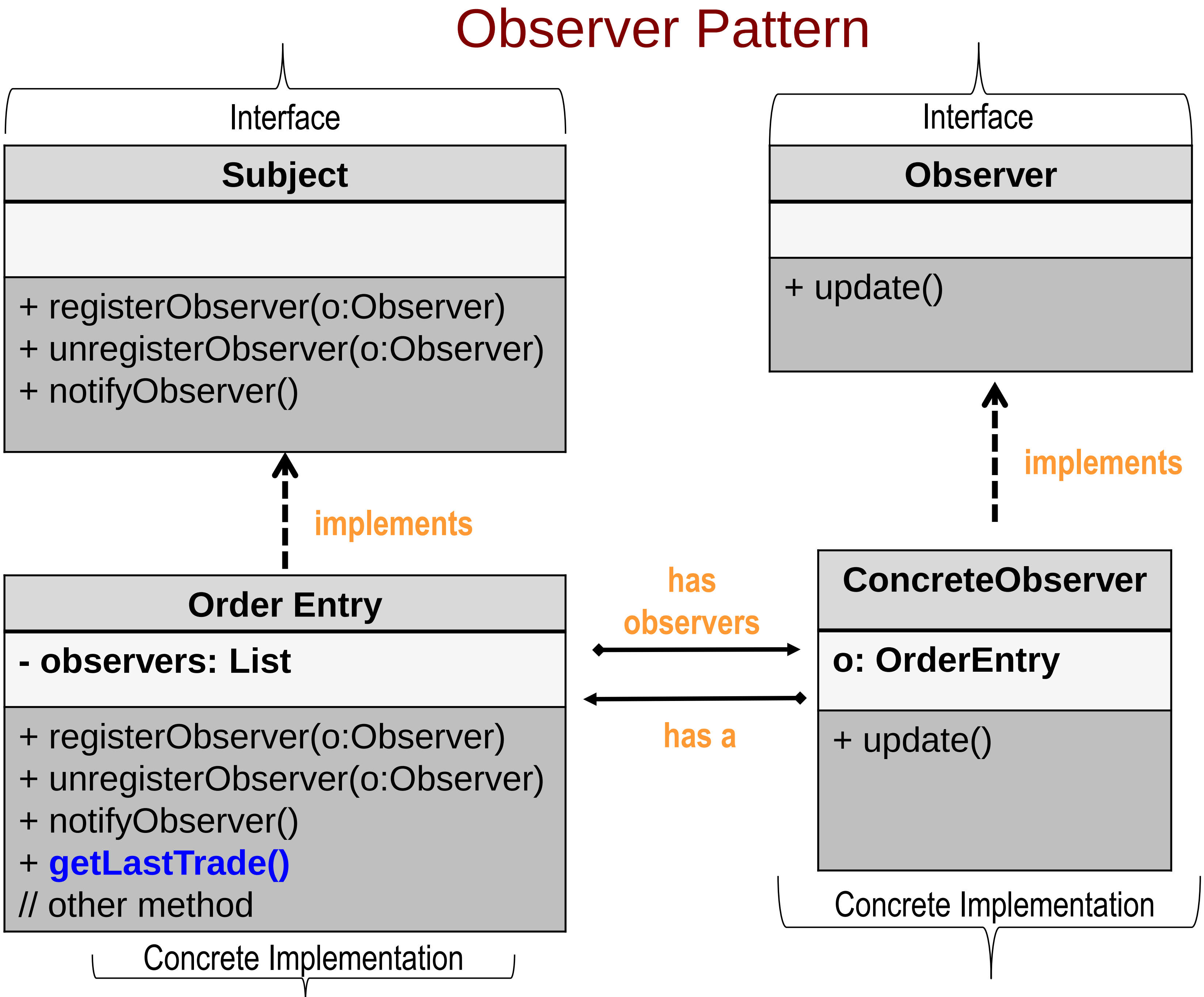
```
public class ObserverSimulator {

   public static void main( String[] a ) {
      // create the observers
      Position posn = new Position();
      PNL pnl = new PNL();

      // createt the Subject
      OrderEntry oe = new OrderEntry();

      // Register the observers
      oe.register( (Observer) posn );
      oe.register( (Observer) pnl );

      // kick off trading…

   } // main
} // class
```

Implementation

```
public class ObserverSimulator {

   public static void main( String[] a ) {
      // create the observers
      Position posn = new Position();
      PNL pnl = new PNL();

      // createt the Subject
      OrderEntry oe = new OrderEntry();

      // Register the observers
      oe.register( (Observer) posn );
      oe.register( (Observer) pnl );

      // kick off trading

   } // main
} // class
```

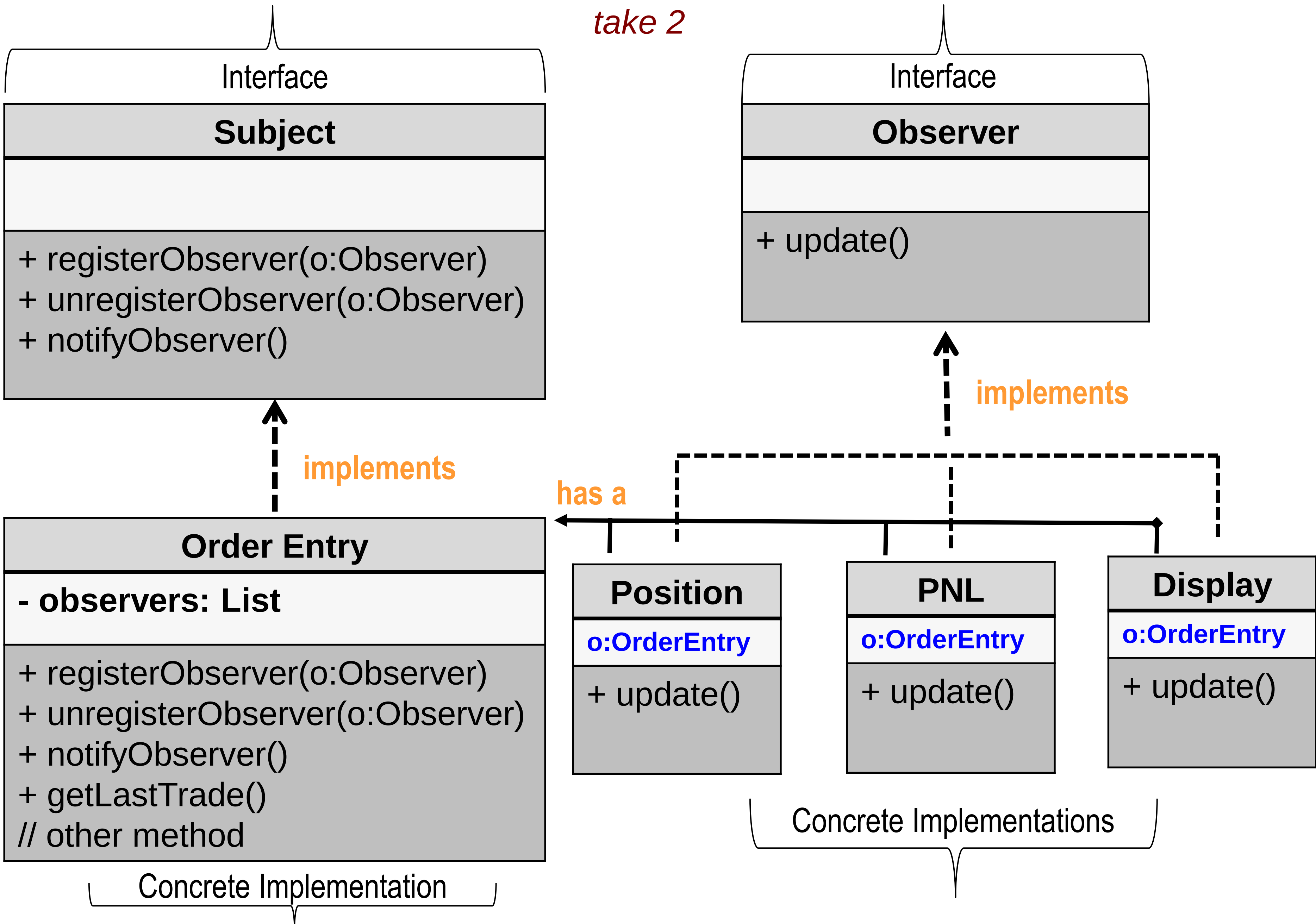
Implementation

```
public class ObserverSimulator {

   public static void main( String[] a ) {
      // create the observers
      Position posn = new Position();
      PNL pnl = new PNL();

      // createt the Subject
      OrderEntry oe = new OrderEntry();

      // Register the observers
      oe.register( (Observer) posn );
      oe.register( (Observer) pnl );

      // kick off trading

   } // main
} // class
```


Observer Pattern
Interface
Subject
+ registerObserver(o:Observer)
+ unregisterObserver(o:Observer)
+ notifyObserver()
Interface
Observer
+ update()
implements
implements
Order Entry
- observers: List
+ registerObserver(o:Observer)
+ unregisterObserver(o:Observer)
+ notifyObserver()
+ getLastTrade()
// other method
Concrete Implementation
has observers
has a
ConcreteObserver
o: OrderEntry
+ update()
Concrete Implementation

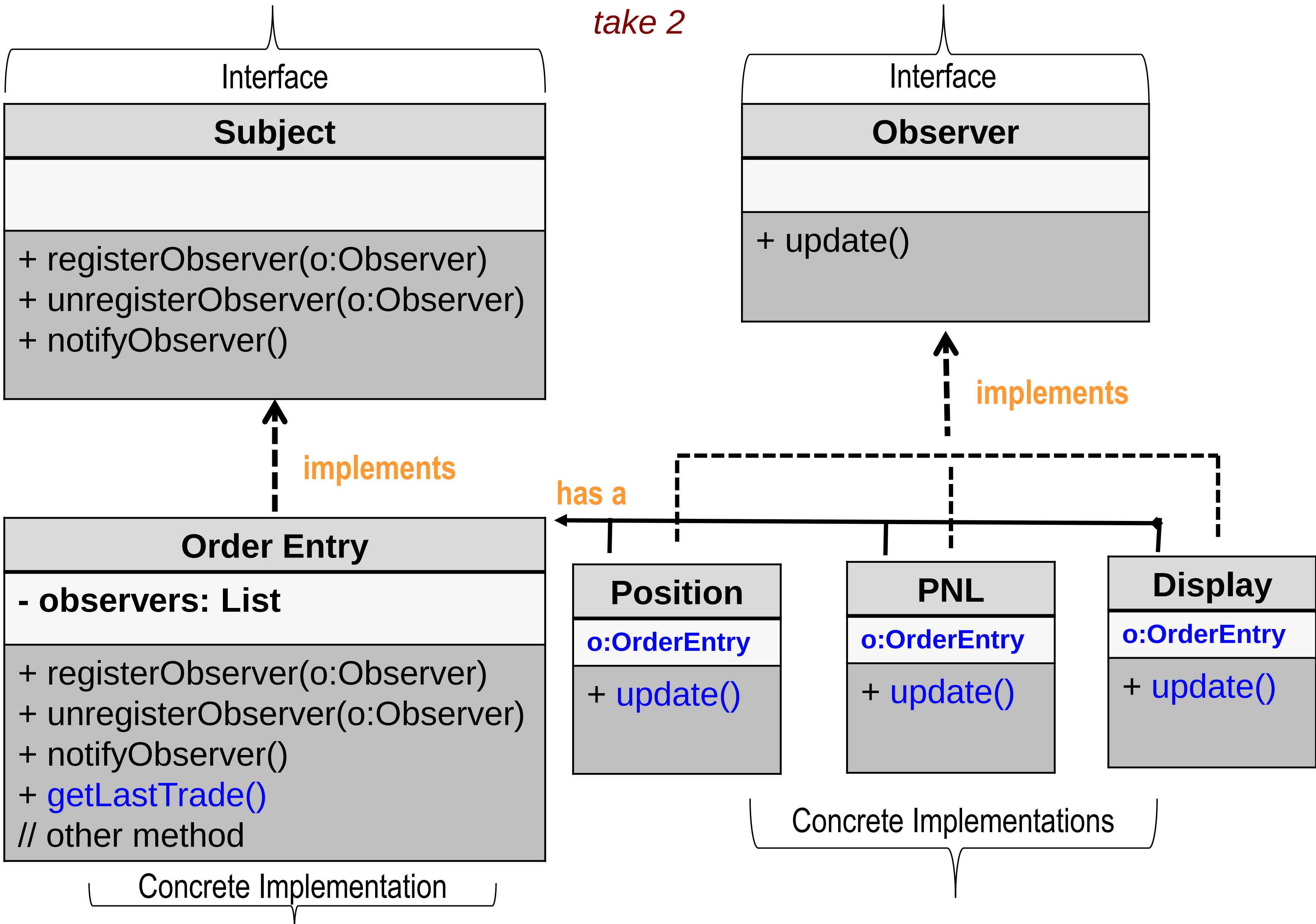
Observer Pattern:

take 2



Observer Pattern:

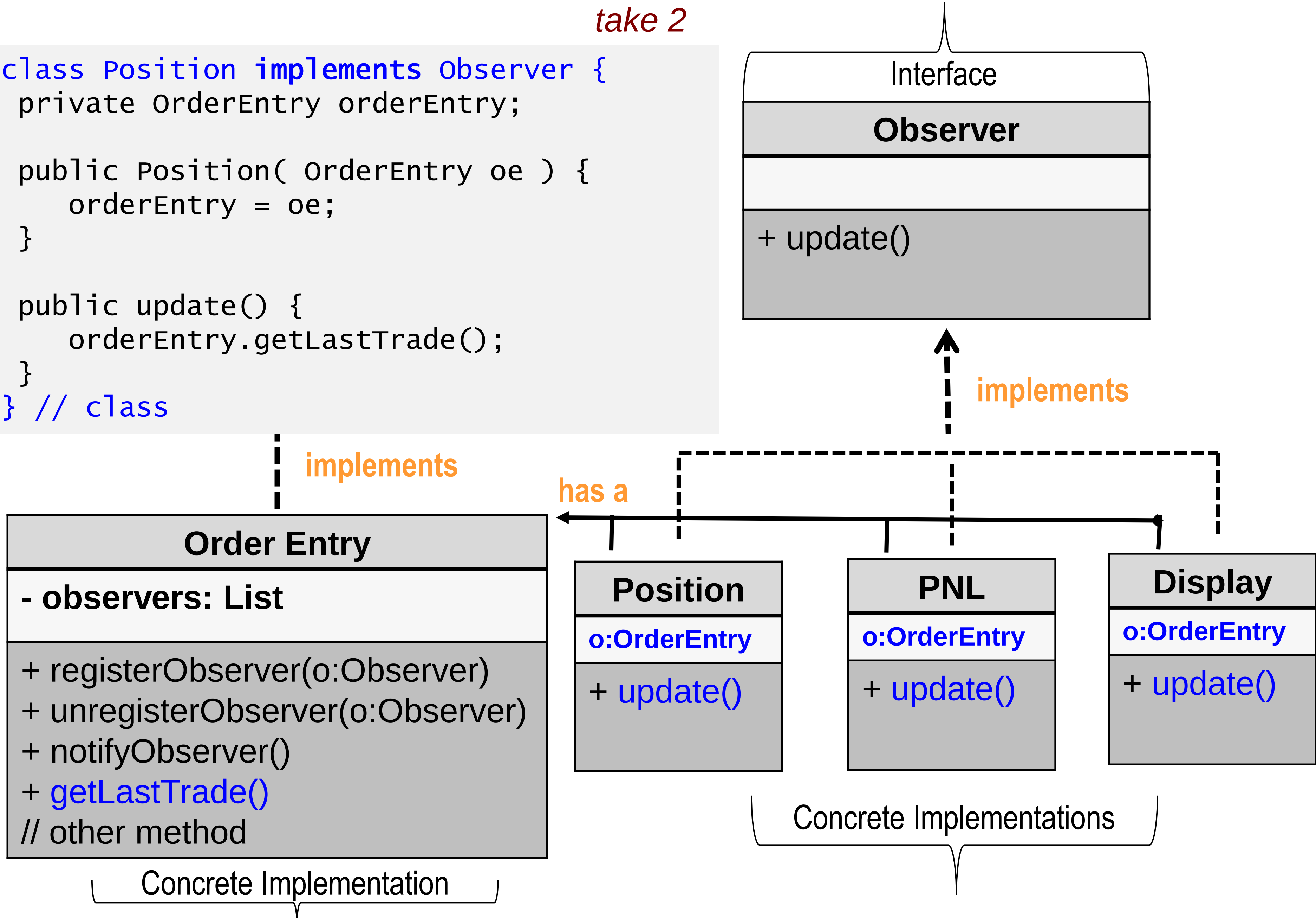
take 2



Observer Pattern:

take 2

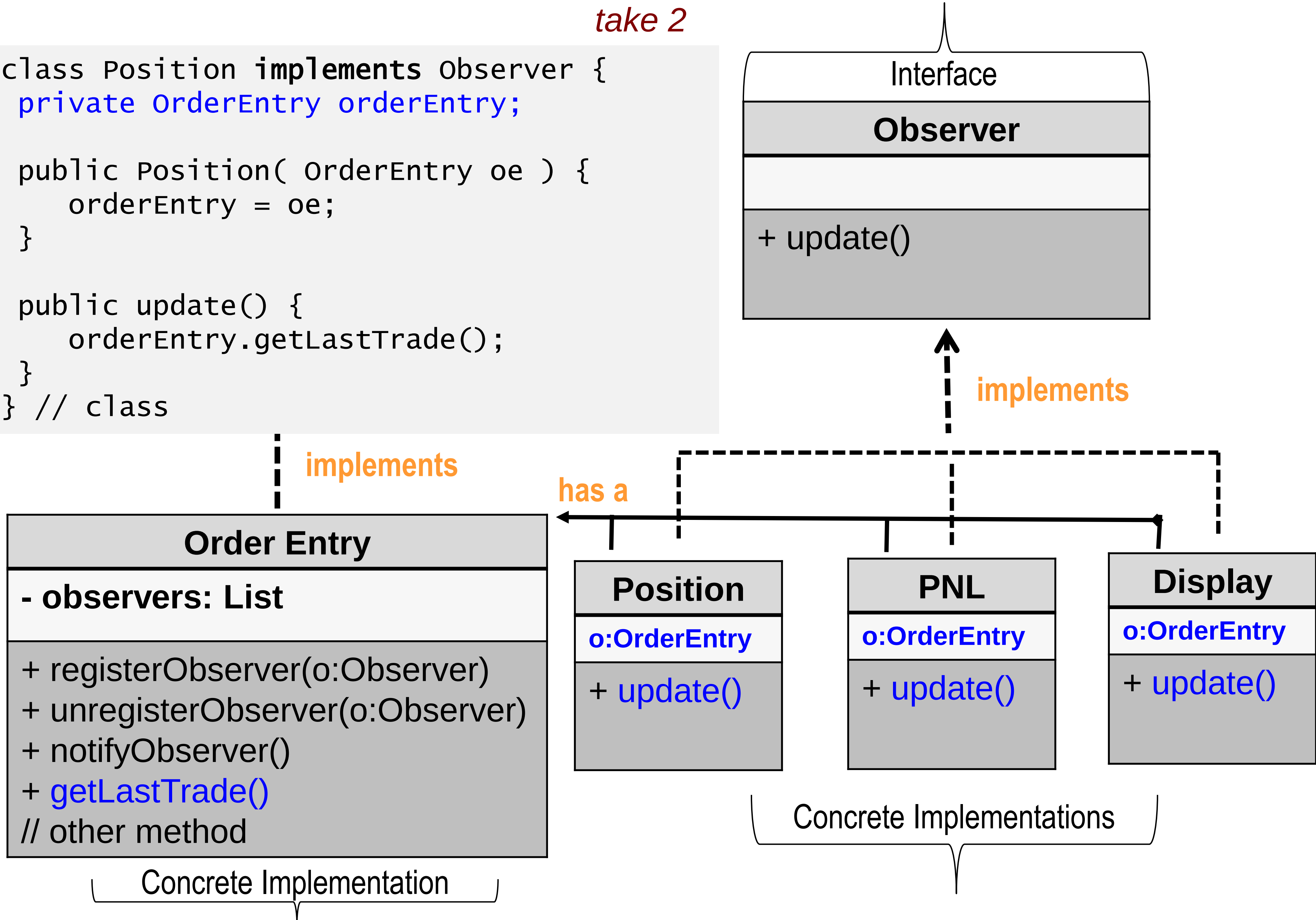
```
class Position implements Observer {
 private OrderEntry orderEntry;

 public Position( OrderEntry oe ) {
    orderEntry = oe;
 }

 public update() {
    orderEntry.getLastTrade();
 }
} // class
```



Observer Pattern:

take 2

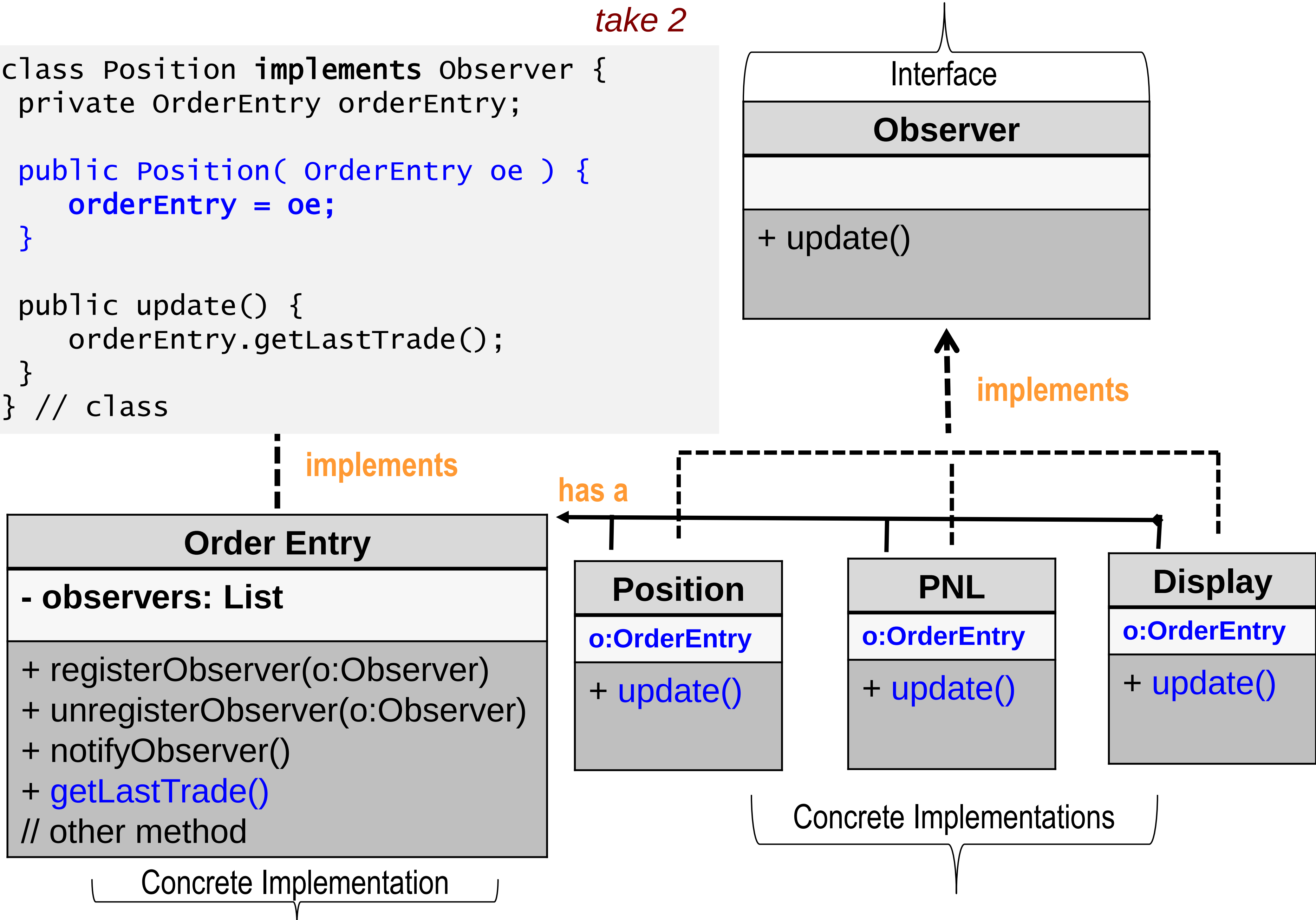
```
class Position implements Observer {
 private OrderEntry orderEntry;

 public Position( OrderEntry oe ) {
    orderEntry = oe;
 }

 public update() {
    orderEntry.getLastTrade();
 }
} // class
```



Observer Pattern:

take 2

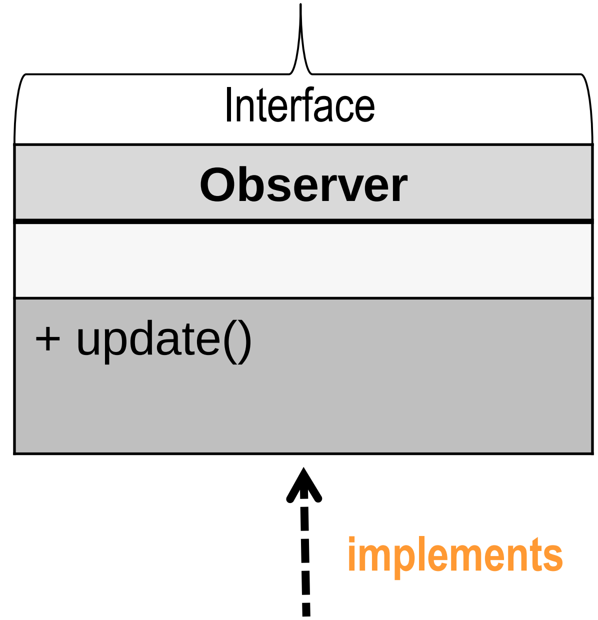
```
class Position implements Observer {
 private OrderEntry orderEntry;

 public Position( OrderEntry oe ) {
    orderEntry = oe;
 }

 public update() {
    orderEntry.getLastTrade();
 }
} // class
```

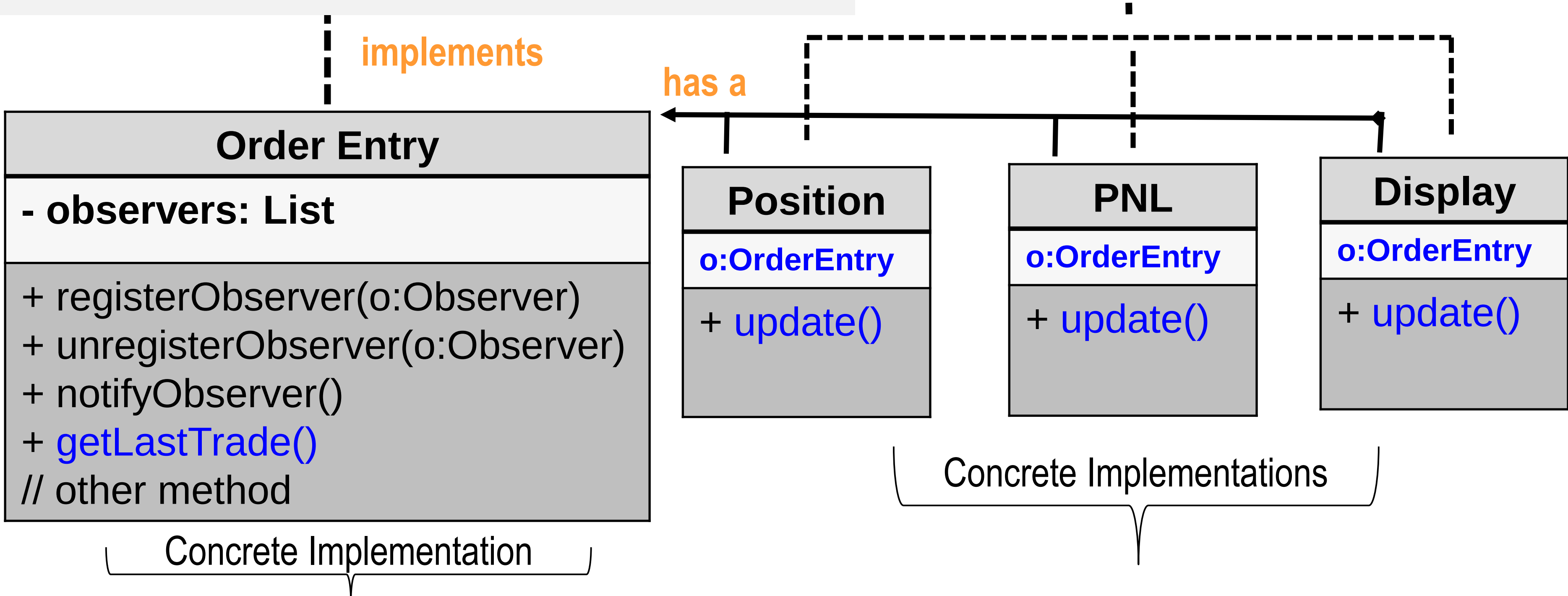


Observer Pattern:

take 2

```
class Position implements Observer {
 private OrderEntry orderEntry;

 public Position( OrderEntry oe ) {
    orderEntry = oe;
 }

 public update() {
    orderEntry.getLastTrade();
 }
} // class
```





Implementation

```
public class ObserverSimulator {

   public static void main( String[] a ) {
      // createt the Subject
      OrderEntry oe = new OrderEntry();

      // create the observers
      Position posn = new Position( oe );
      PNL pnl = new PNL( oe );

      // Register the observers
      oe.register( (Observer) posn );
      oe.register( (Observer) pnl );

      // kick of trading

   } // main
} // class
```

Observer Pattern:

Elements of Reusable OO Software

- **Consequences**: The observer pattern allows you to vary subjects and observers independently. You can reuse subjects without reusing their observers, and vice verse. Allows observes to be added without modifying the subject or other observers.
  - Abstract coupling between Subject and Observer. A subject only knows that it has a list of observers, but knows nothing about the concrete class of each observer.
  - Support for broadcast communication. The subject does know anything about its receivers. Therefore receivers can be added and removed dynamically.

Observer Pattern:
Elements of Reusable OO Software

- **Consequences**: The observer pattern allows you to vary subjects and observers independently. You can reuse subjects without reusing their observers, and vice verse. Allows observes to be added without modifying the subject or other observers.
  - Abstract coupling between Subject and Observer. A subject only knows that it has a list of observers, but knows nothing about the concrete class of each observer.
  - Support for broadcast communication. The subject does know anything about its receivers. Therefore receivers can be added and removed dynamically.
  - The observer pattern violates the **single responsibility rule**. The concrete class of the Subject is now responsible for not only it's own behavior but also notifying its observers of changes.