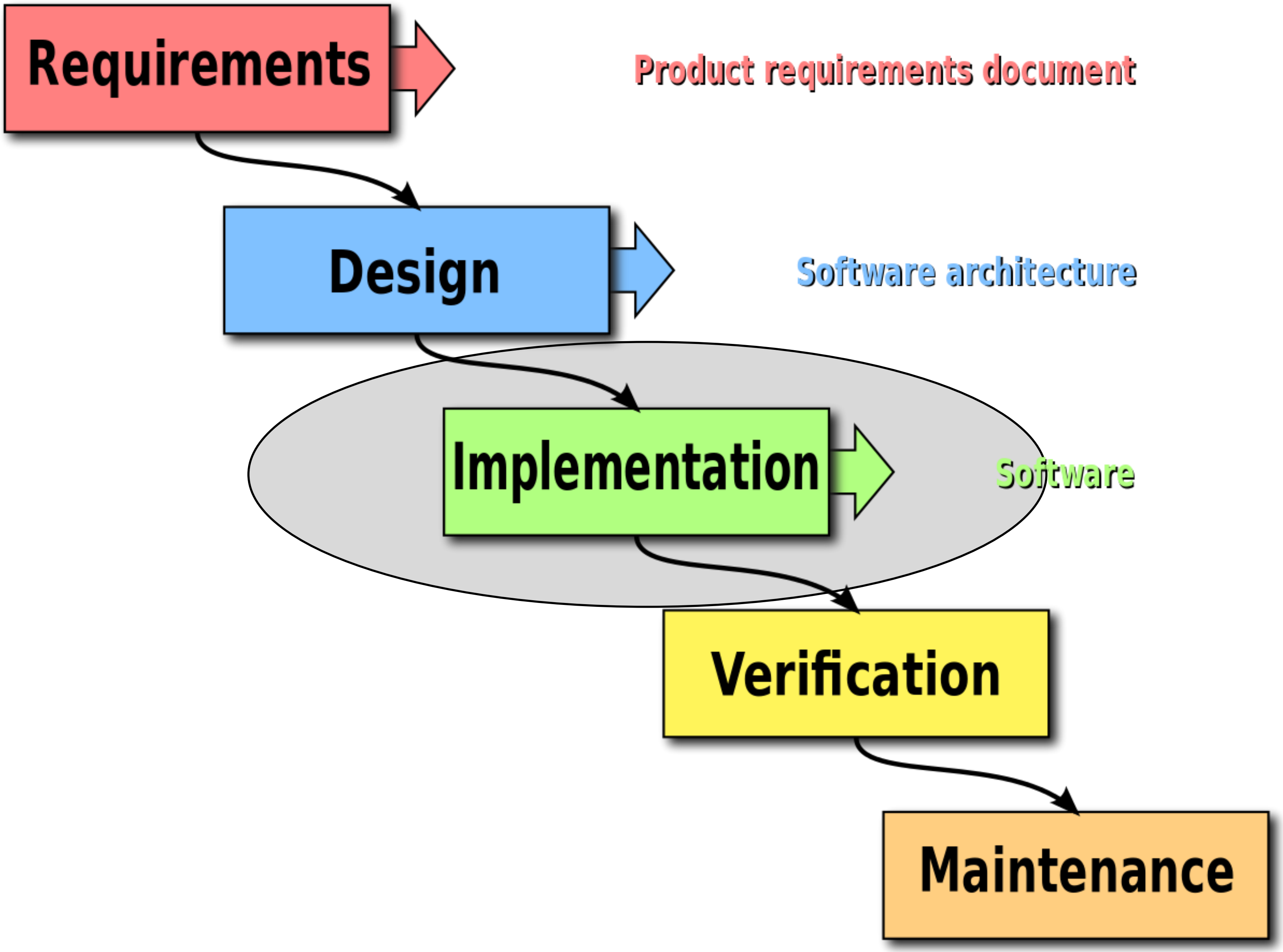
Software Engineering



Software Engineering

- A multi-person construction of multi-version software.

significance of communication…

David Parnas

- An engineering discipline whose focus is the **cost-effective development** of high-quality software systems.

captures the reality of the business environment …

- An engineering discipline that is concerned with all aspects of software production from the early stages of system specification to maintaining the system after it has gone into use.

Ian Sommerville

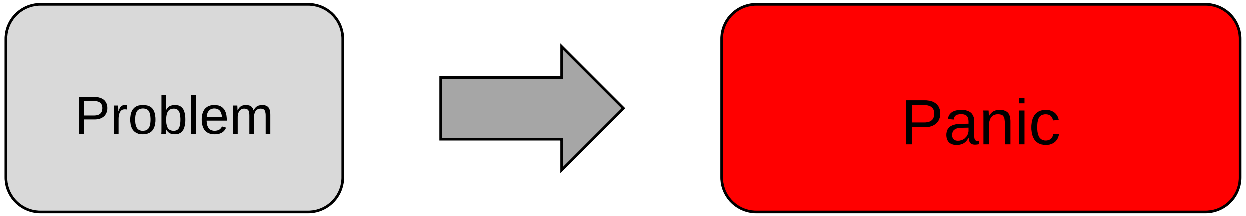
captures the software life cycle …

- The application of computing tools to solving problems.

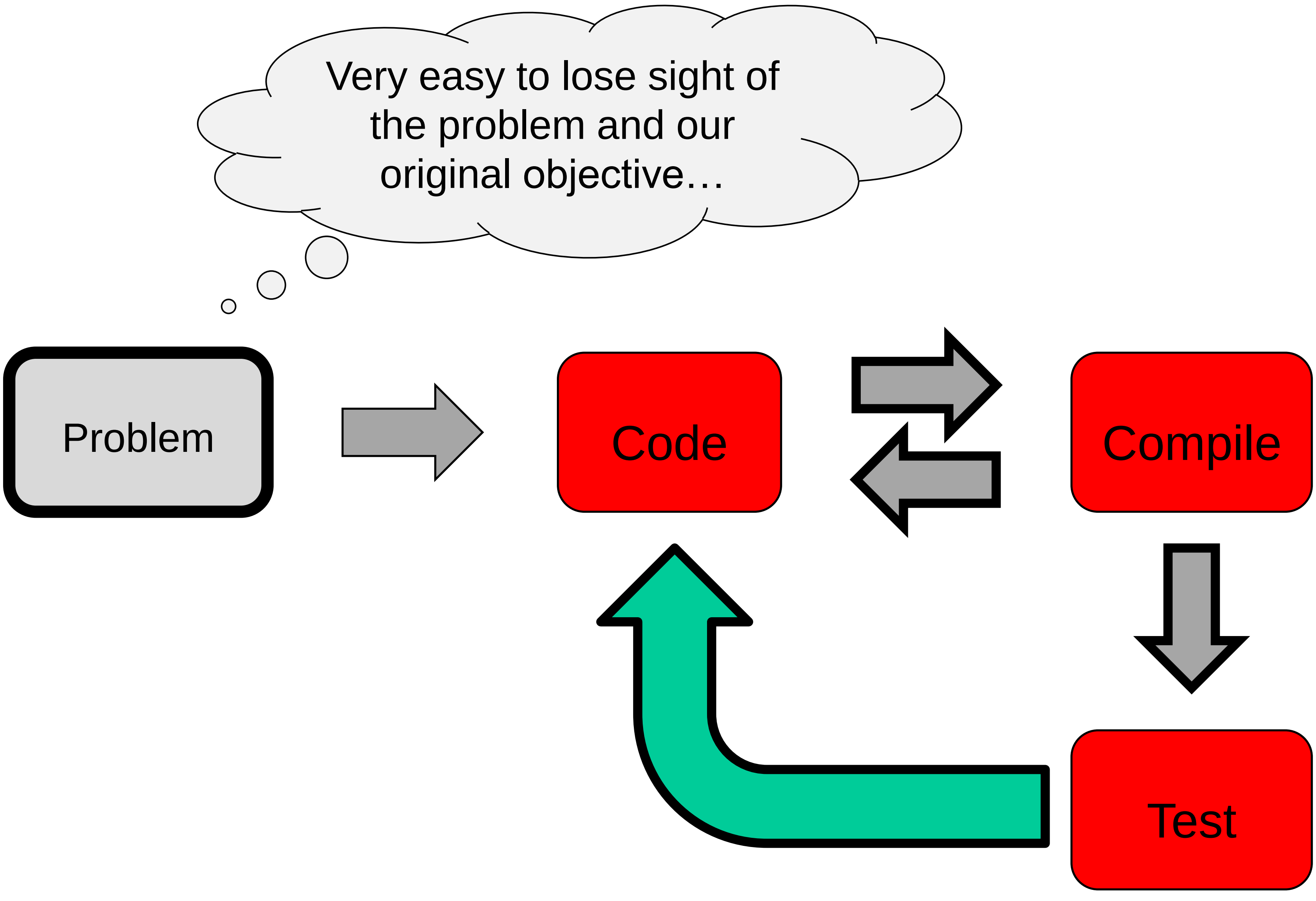
essence of technology…

Shan Pfleeger

How we program…



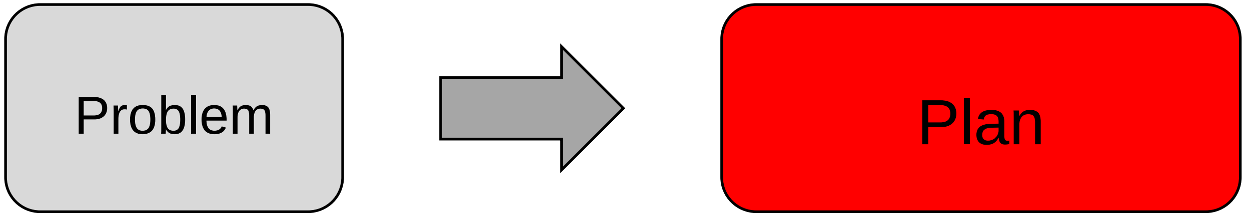
How we program…



How we **should** program…

Problem → Think

How we **should** program…



Principles of Software Engineering

Are there natural laws or *principles* in software engineering, similar to those we find in other scientific disciplines (i.e. *laws of motion*, *thermodynamics*, etc.), such that if we follow them, we will construct quality software?

Early Principles

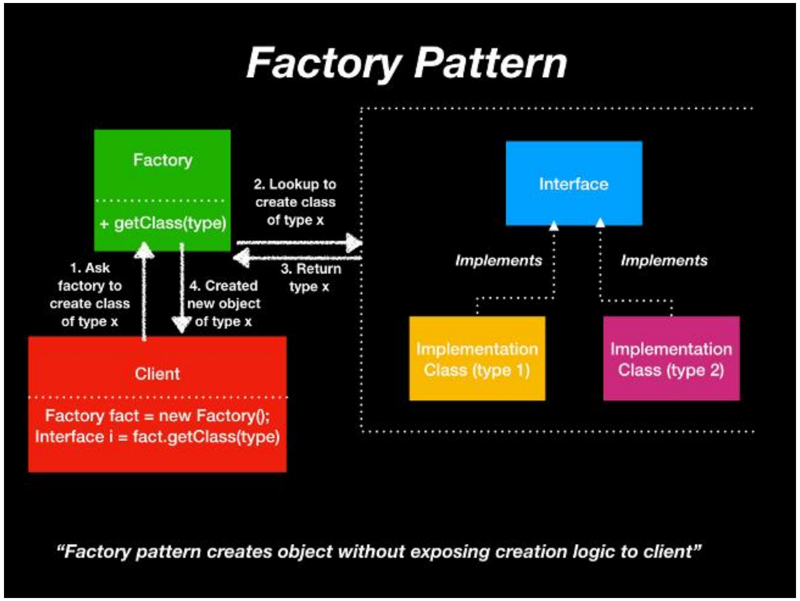
Alan Davis (1994)

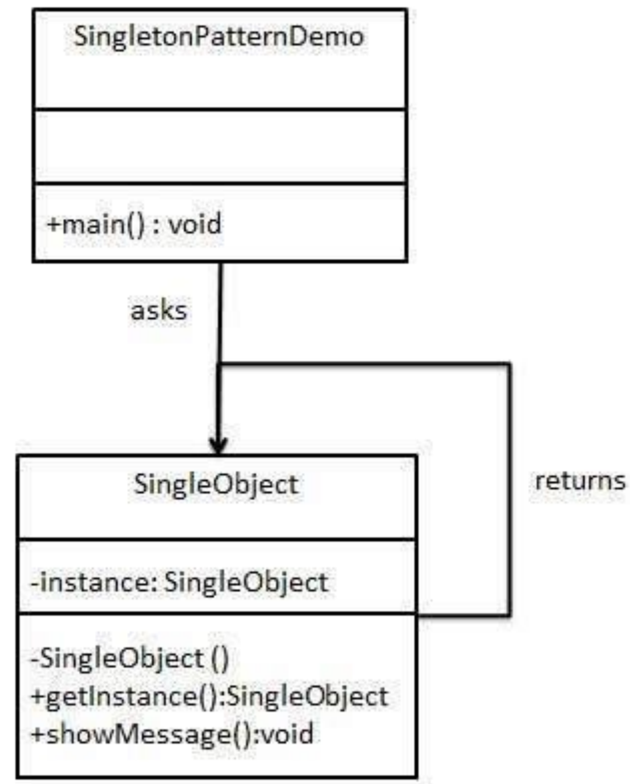
- Make quality number 1
- High quality is possible
- Get products to your customers early
- Understand the problem before developing requirements
- Consider your alternatives
- Choose your process model
- Put technique before tools
- Get it right before you make it faster
- Inspect and evaluate your code
- Good management is more important than good technology
- People are the key to success
- **Take responsibility**

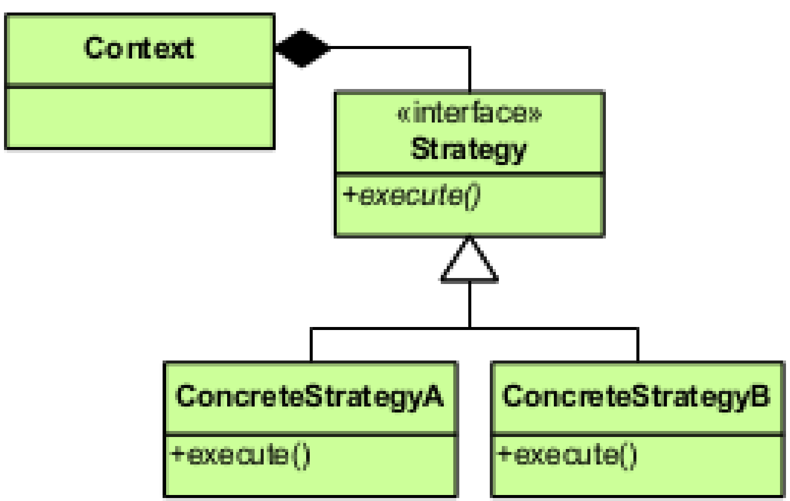
Software Engineering code of ethics..

- Software engineers shall act consistently with the public interest.
- Software engineers shall act in a manner that is in the best interest of their client and employer…
- Software engineers shall ensure that their products and related modifications meet the highest professional standards possible.
- Software engineers shall maintain integrity and independence in their professional judgement.
- Software engineering managers shall subscribe to and promote an ethical approach to the management of software development and maintenance.

Software Design Patterns





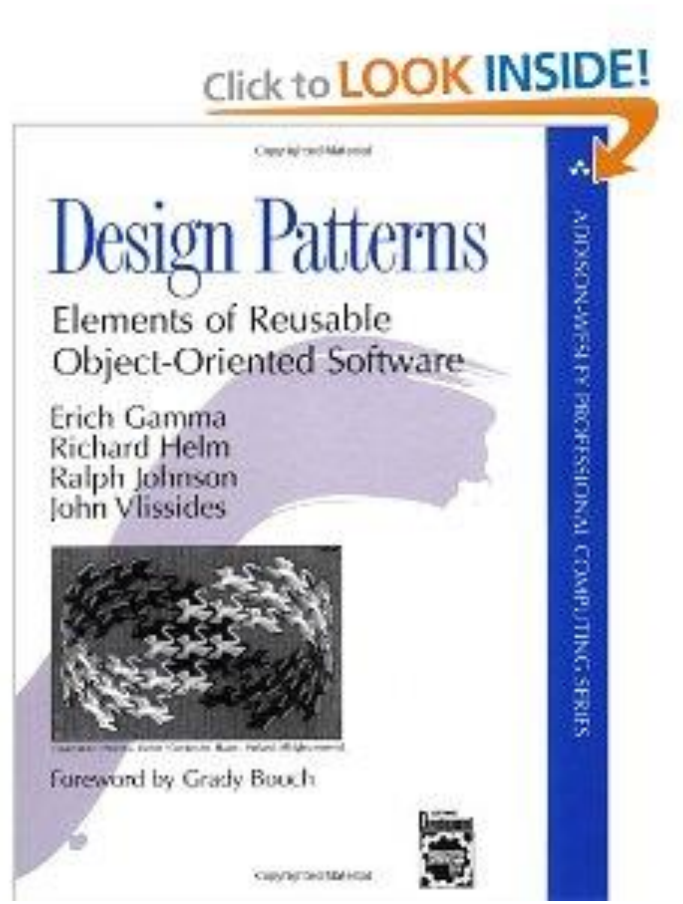


Design Patterns:
Elements of Reusable Object Oriented Software

In 1990 a group called the:
Gang of Four or "GoF“:

- Erich Gamma,
- Richard Helm,
- Ralph Johnson,
- John Vlissides,

compile a catalog of design patterns in this 1995 classic book!



Design Patterns:

In 1990 a grou
Gang of Four

- Erich Gamm
- Richard Hel
- Ralph Johns
- John Vlissid

compile a cata
patterns in this
book!

Motivation for Design Patterns

- Understanding the principles of Object Oriented Design does not guarantee that we will design high quality re-useable software.

- Good design comes with experience!
  - Over many projects you begin to see that certain designs work well in different situation and, a good software engineer develops a database of these design solutions.
  - Once you find a good solution to a particular type of problem you will use it over and over again.
  - If you study many complex systems you will find recurring patterns of classes and object hierarchies.

- Design patterns attempt to *formalize* experience!
  - *Catalog experiences so they can be reused and followed when designing Object Oriented Software*.

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.

Power of a Shared Vocabulary

- The *pattern name* allows you to communicate a set of ***all*** the qualities, characteristics and constraints that the pattern represents.
- Other developers know immediately precisely the design you have in mind.
- Allows developers to stay focused on design and not on implementation.
- Allows developers to have a common understanding about the design approach so that there are less misunderstandings amongst programming teams.
- Allows for younger developers to get up to speed more efficiently.

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.

Describing Design Patterns

as defined in: *Elements of Reusable OOS book*

- Design Patterns are described using a consistent format or template which provides a uniform structure to the information, each pattern to be more easily learned, used and compared.

    - **Pattern Name and Classification**
    - **Intent**
    - Also Known as…
    - **Motivation and**
    - **Applicability**
    - **Structure**
    - Participants
    - Collaborations
    - **Consequences**
    - Implementation
    - Sample Code
    - Known Uses
    - Related Patterns

Describing Design Patterns

as defined in: *Elements of Reusable OOS book*

- Pattern Name and *Classification*
  - uniquely identifies the pattern
  - conveys the essence of the pattern
  - *categorizes the pattern into one of three purposes:*
    - ***Creational***
    - ***Structural***
    - ***Behavioral***

- Intent is a short statement that addresses the questions:
  - What does the design do?
  - What is its rationale and intent?
  - What particular design issue or problem does it address?

- Motivation is and Applicability addresses the questions:
  - A scenario that illustrates a design problem and a description of how the object structure of the design pattern addresses it.
  - Which situation can the design pattern be applied?
  - What examples of poor design does the pattern address?
  - How can you recognize these situations?

- Structure is a graphical representation of the classes in the pattern using a standard class notation (UML diagrams).

- **Consequences** addresses the questions:
  - How does the pattern support the objective
  - What are the trade-offs and results of using the pattern.

Describing Design Patterns

as defined in: *Elements of Reusable OOS book*

- Pattern Name and Classification
  - uniquely identifies the pattern
  - conveys the essence of the pattern
  - categorizes the pattern into one of three purposes:
    - **Creational**
    - **Structural**
    - **Behavioral**

- Intent is a short statement that addresses the questions:
  - What does the design do?
  - What is its rationale and intent?
  - *What particular design issue or problem does it address?*

- Motivation is and Applicability addresses the questions:
  - A scenario that illustrates a design problem and a description of how the object structure of the design pattern addresses it.
  - Which situation can the design pattern be applied?
  - What examples of poor design does the pattern address?
  - How can you recognize these situations?

- Structure is a graphical representation of the classes in the pattern using a standard class notation (UML diagrams).

- **Consequences** addresses the questions:
  - How does the pattern support the objective
  - What are the trade-offs and results of using the pattern.

Describing Design Patterns

as defined in: *Elements of Reusable OOS book*

- Pattern Name and Classification
  - and results of using the pattern.
  - uniquely identifies the pattern
  - conveys the essence of the pattern
  - categorizes the pattern into one of three purposes:
    - **Creational**
    - **Structural**
    - **Behavioral**

- Intent is a short statement that addresses the questions:
  - What does the design do?
  - What is its rationale and intent?
  - What particular design issue or problem does it address?

- Motivation offers and Applicability addresses the questions:
  - *A scenario that illustrates a design problem and a description of how the object structure of the design pattern addresses it.*
  - Which situation can the design pattern be applied?
  - What examples of poor design does the pattern address?
  - How can you recognize these situations?

- Structure is a graphical representation of the classes in the pattern using a standard class notation (UML diagrams).

- **Consequences** addresses the questions:
  - How does the pattern support the objective

Describing Design Patterns

as defined in: *Elements of Reusable OOS book*

- Pattern Name and Classification
  - and results of using the pattern.
  - uniquely identifies the pattern
  - conveys the essence of the pattern
  - categorizes the pattern into one of three purposes:
    - **Creational**
    - **Structural**
    - **Behavioral**

- Intent is a short statement that addresses the questions:
  - What does the design do?
  - What is its rationale and intent?
  - What particular design issue or problem does it address?

- Motivation offers and Applicability addresses the questions:
  - A scenario that illustrates a design problem and a description of how the object structure of the design pattern addresses it.
  - Which situation can the design pattern be applied?
  - ***What examples of poor design does the pattern address?***
  - How can you recognize these situations?

- Structure is a graphical representation of the classes in the pattern using a standard class notation (UML diagrams).

- **Consequences** addresses the questions:
  - How does the pattern support the objective

Describing Design Patterns

as defined in: *Elements of Reusable OOS book*

- Pattern Name and Classification
  - and results of using the pattern.
  - uniquely identifies the pattern
  - conveys the essence of the pattern
  - categorizes the pattern into one of three purposes:
    - **Creational**
    - **Structural**
    - **Behavioral**

- Intent is a short statement that addresses the questions:
  - What does the design do?
  - What is its rationale and intent?
  - What particular design issue or problem does it address?

- Motivation offers and Applicability addresses the questions:
  - A scenario that illustrates a design problem and a description of how the object structure of the design pattern addresses it.
  - Which situation can the design pattern be applied?
  - What examples of poor design does the pattern address?
  - How can you recognize these situations?

- Structure is a graphical representation of the classes in the pattern using a standard class notation (UML diagrams).

- **Consequences** addresses the questions:
  - How does the pattern support the objective

Describing Design Patterns

as defined in: *Elements of Reusable OOS book*

- Pattern Name and Classification
  - uniquely identifies the pattern
  - conveys the essence of the pattern
  - categorizes the pattern into one of three purposes:
    - Creational
    - Structural
    - Behavioral

- Intent is a short statement that addresses the questions:
  - What does the design do?
  - What is its rationale and intent?
  - What particular design issue or problem does it address?

- Motivation offers and Applicability addresses the questions:
  - A scenario that illustrates a design problem and a description of how the object structure of the design pattern addresses it.
  - Which situation can the design pattern be applied?
  - What examples of poor design does the pattern address?
  - How can you recognize these situations?

- Structure is a graphical representation of the classes in the pattern using a standard class notation (UML diagrams).

- Consequences addresses the questions:
  - How does the pattern support the objective
  - *What are the trade-offs and results of using the pattern.*

Catalog of Design Patterns:

as defined in: *Elements of Reusable OOS book*

- Factory and
  - Abstract Factory
- Adaptor
- Bridge
- Builder
- Chain of Responsibility
- Command
- Composite
- Decorator
- Façade
- Flyweight
- Interpreter
- Flyweight
- Iterator
- Mediator
- Memento
- Observer
- Prototype
- Proxy
- Singleton
- State
- Strategy
- Template
- Visitor

Categories of Design Patterns

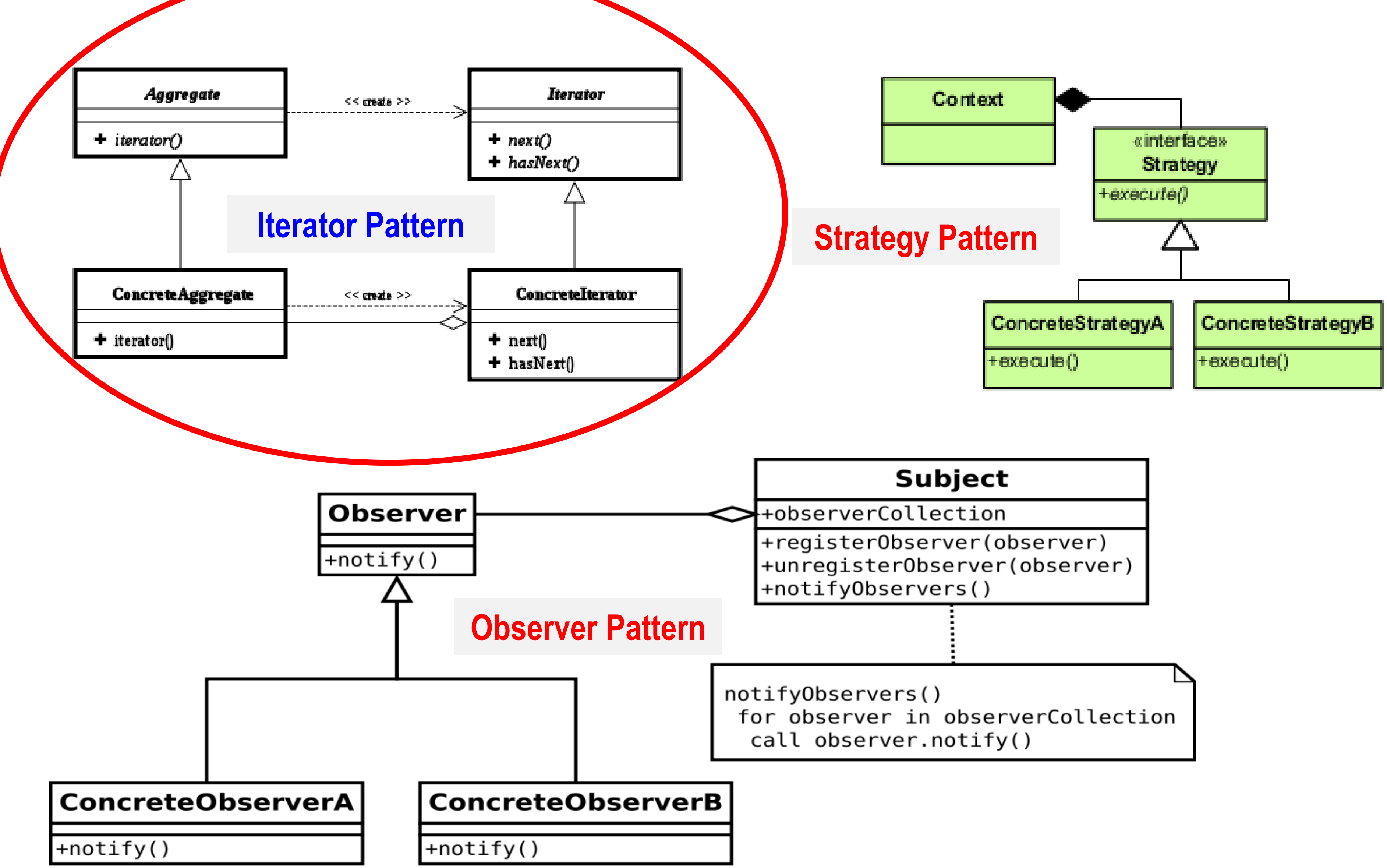
- **Creational Patterns** (abstracting the object-instantiation process)
  - *Factory Method* **Abstract Factory** *Singleton*
  - Builder Prototype

- **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:

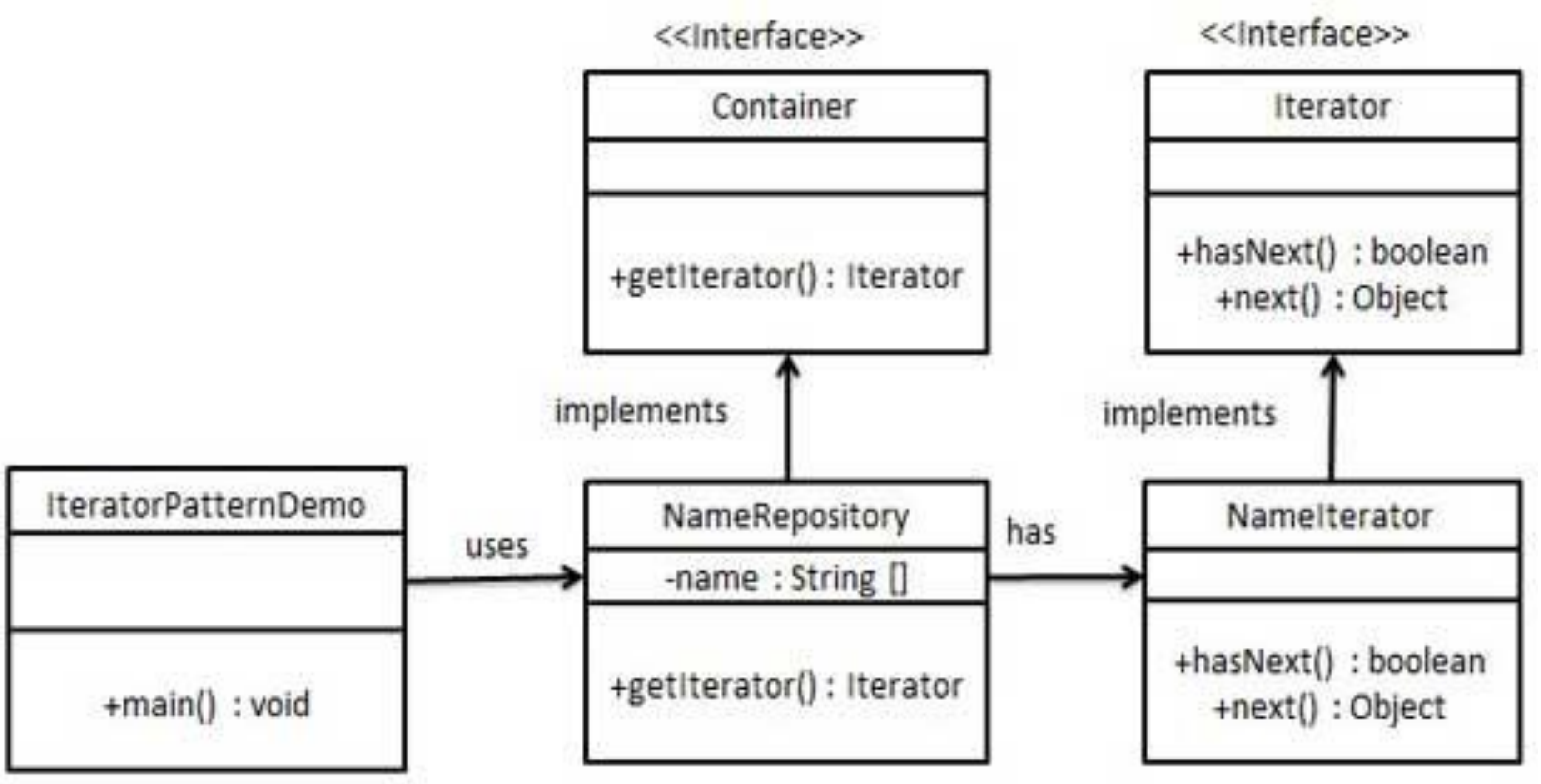
Elements of Reusable OO Software

- **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*.

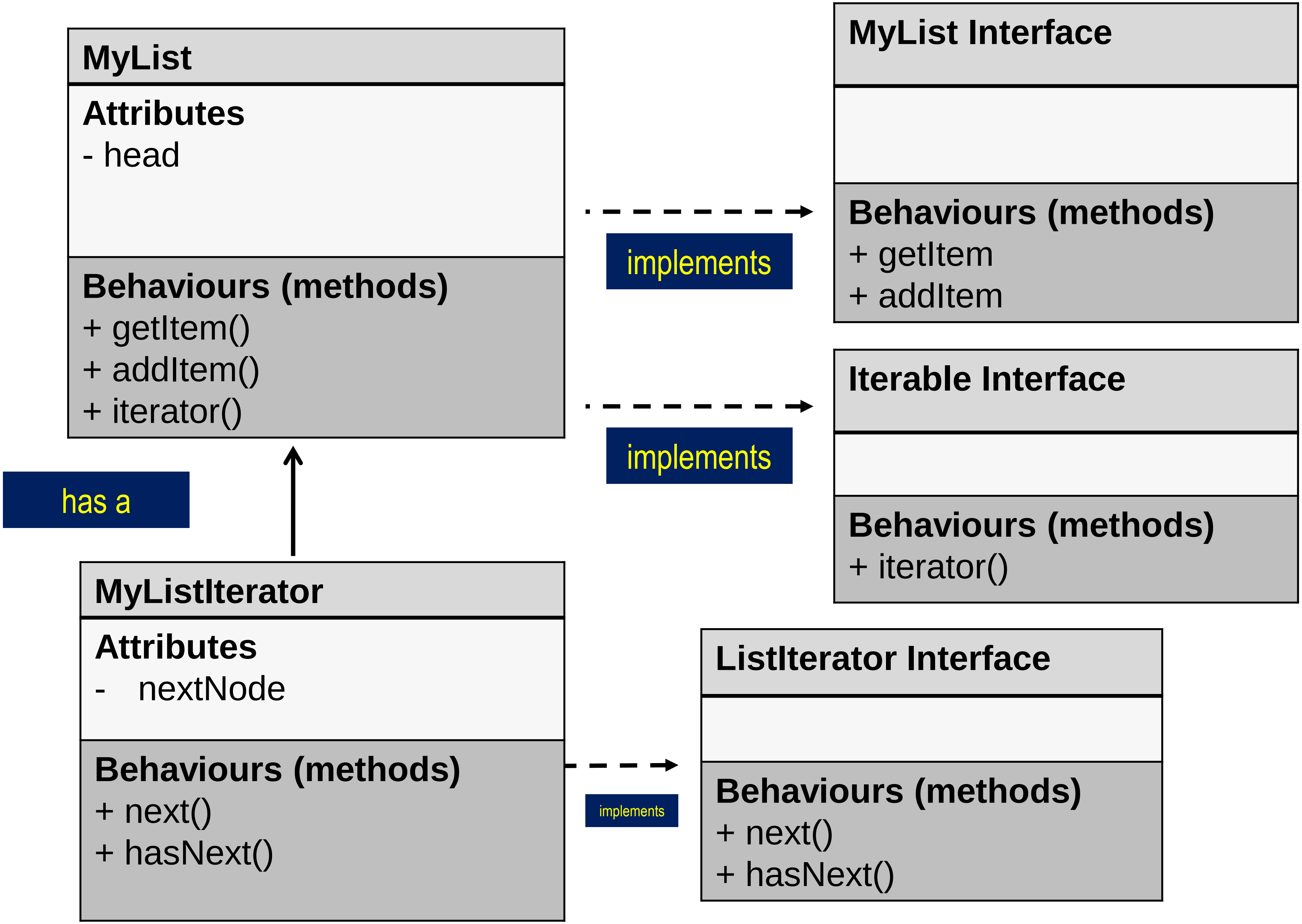
Iterator Pattern

- **Structure**

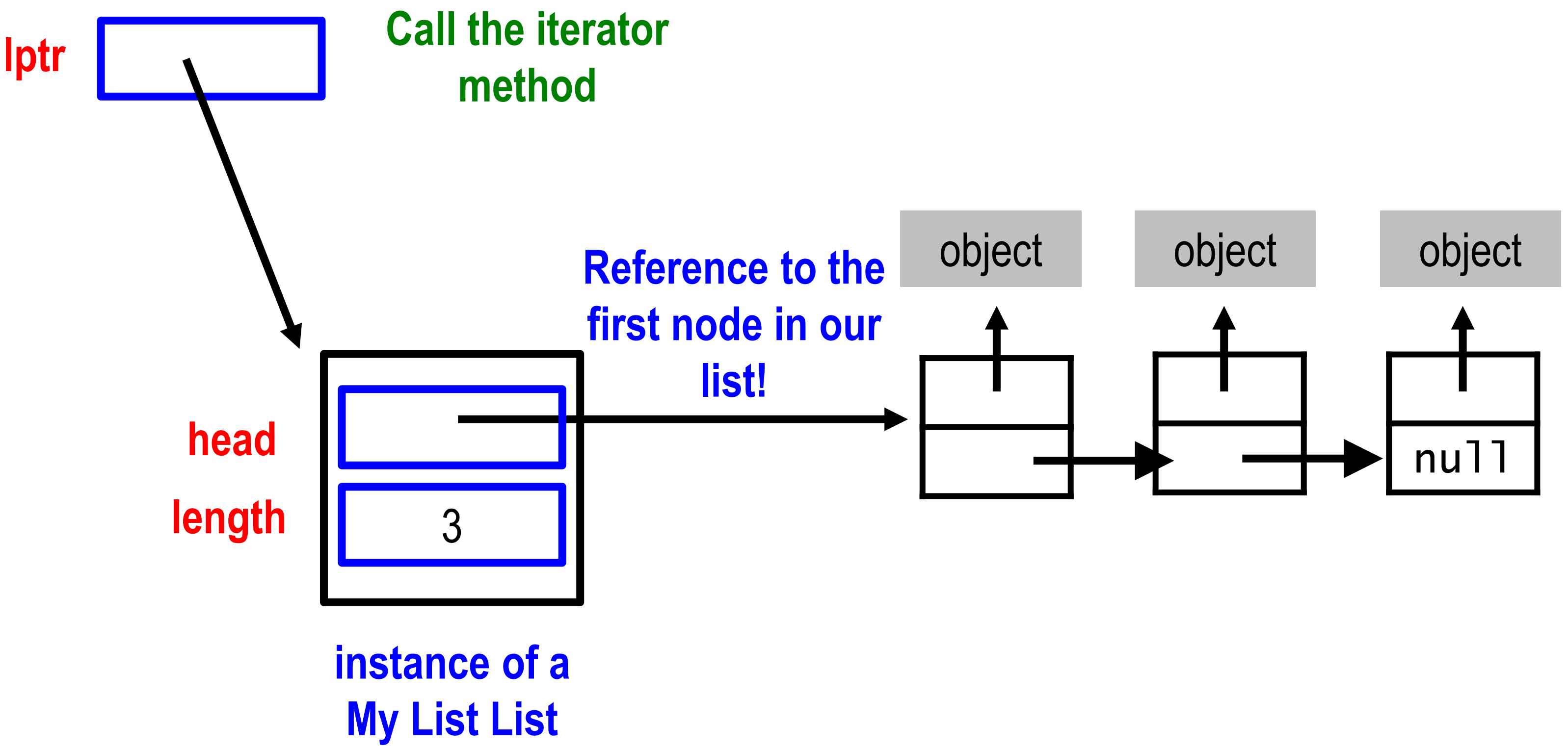


[Tutorials Point]

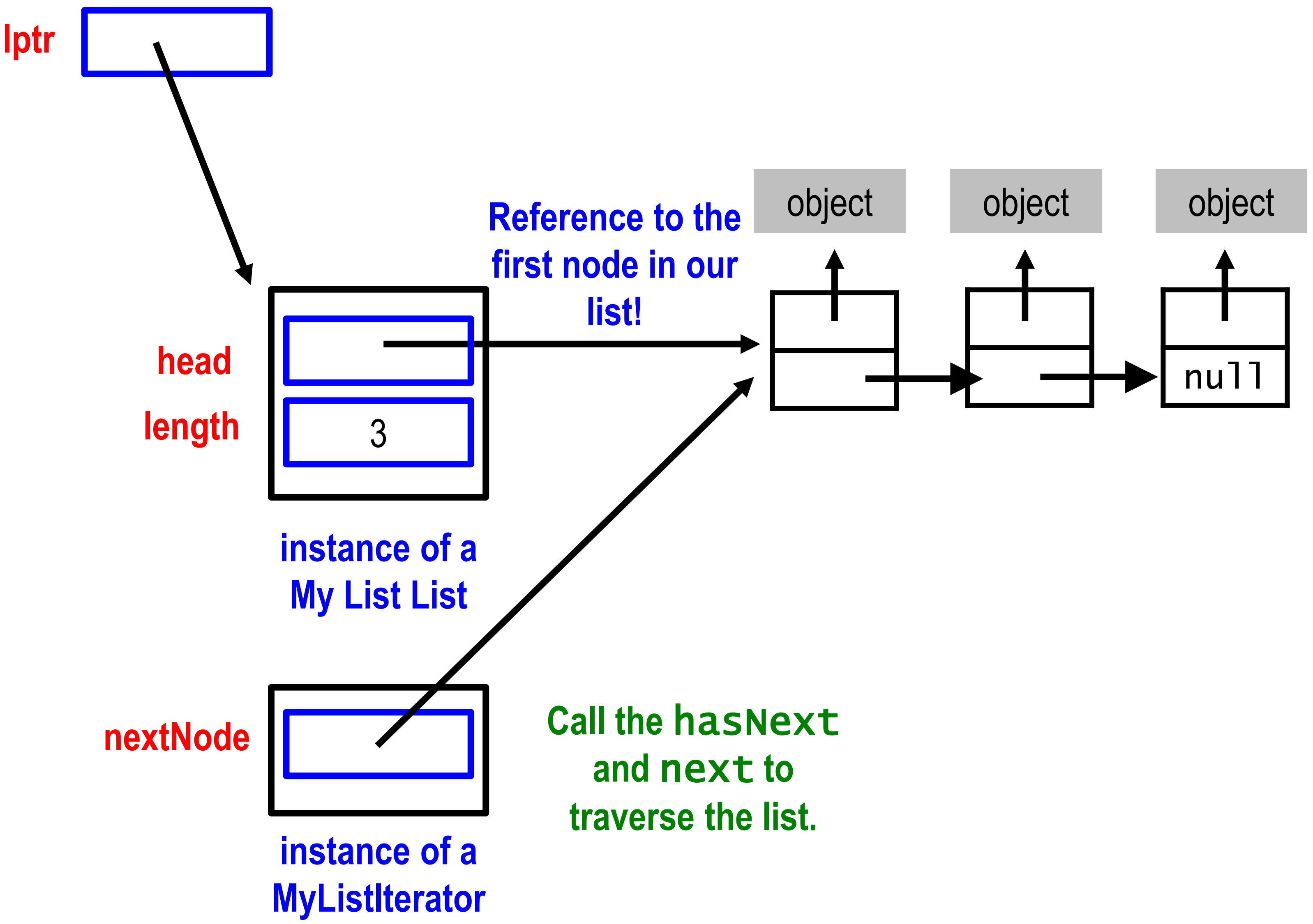
Recall out Iterator Example



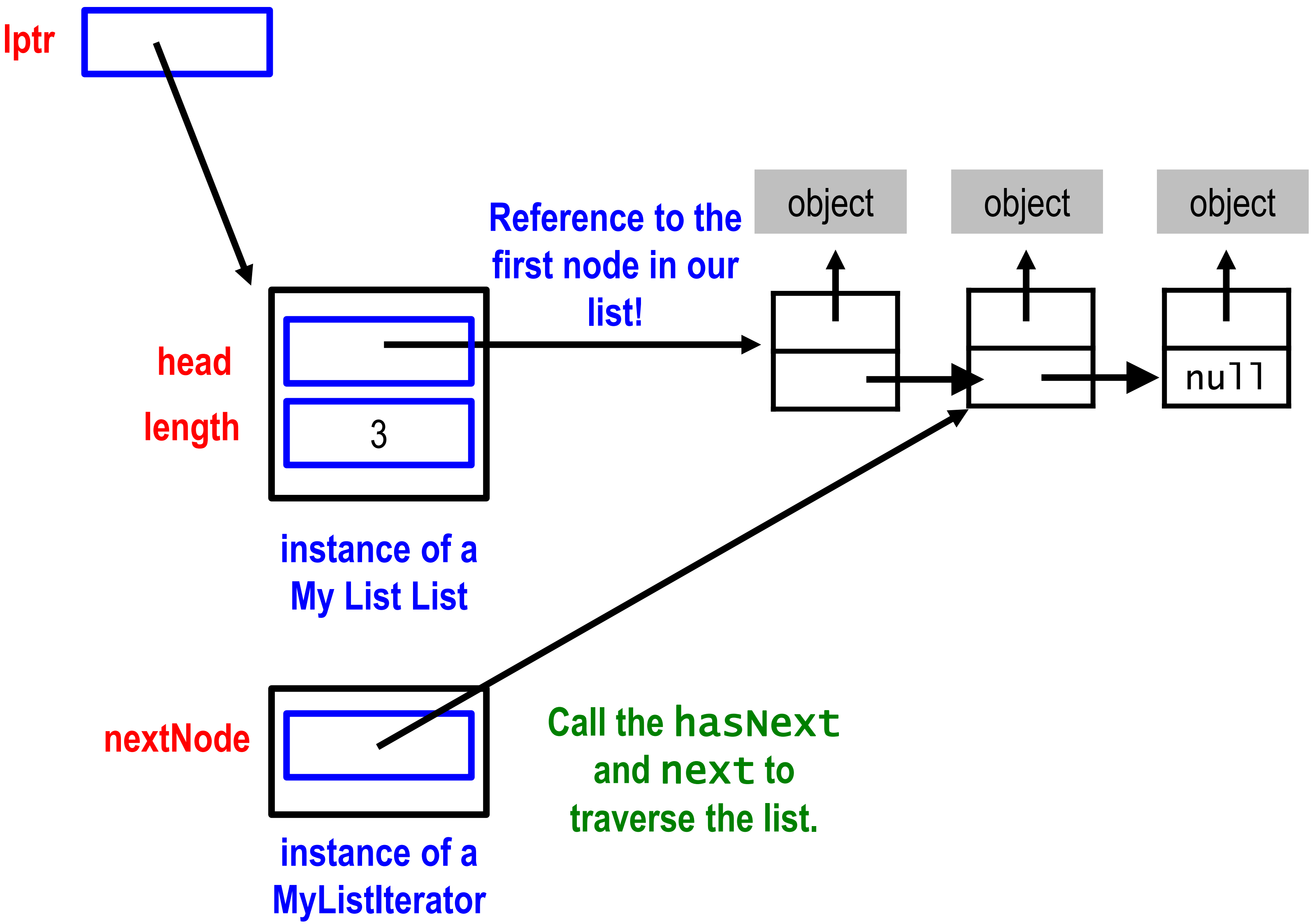
MyList Class



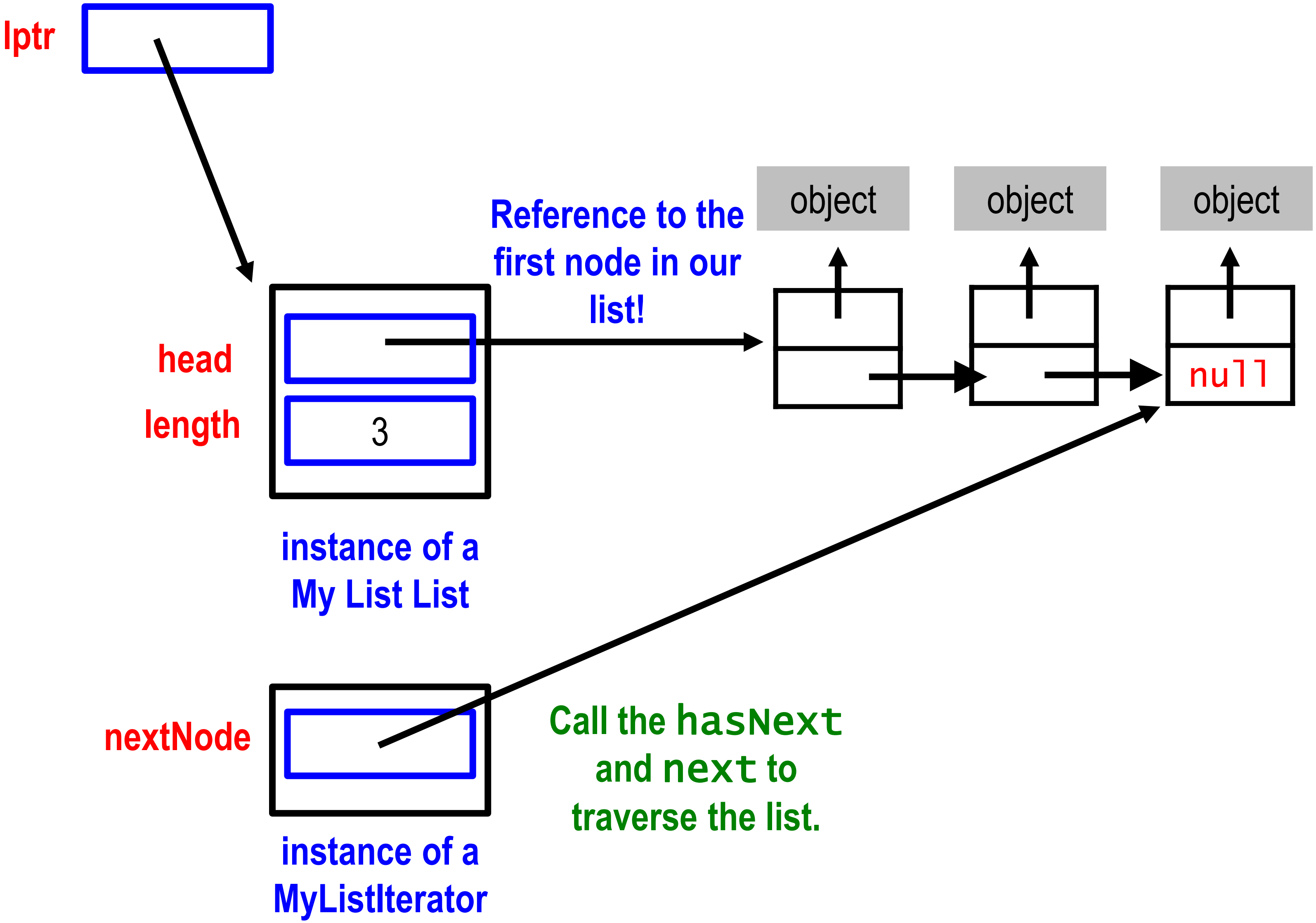
MyList Class



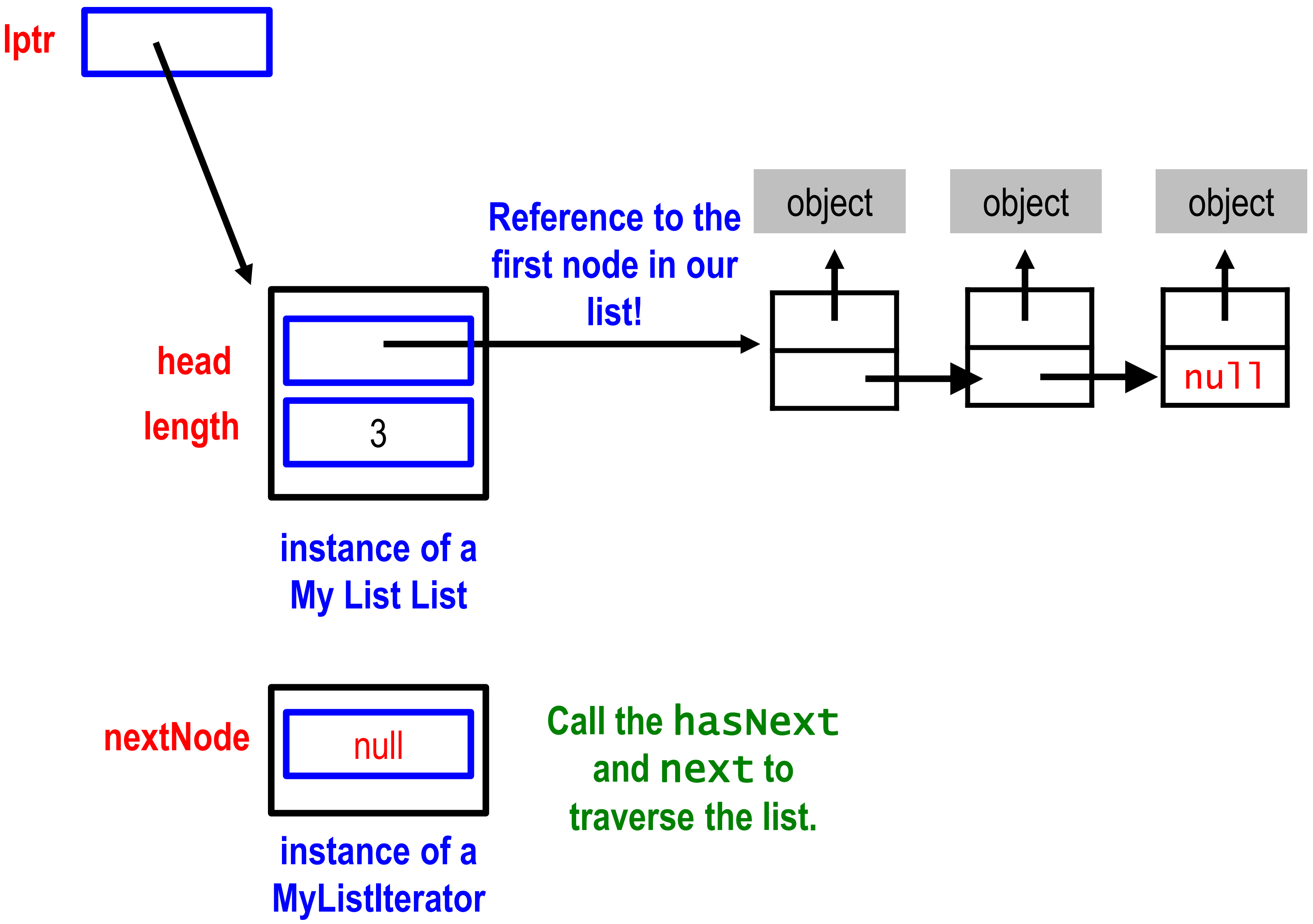
MyList Class



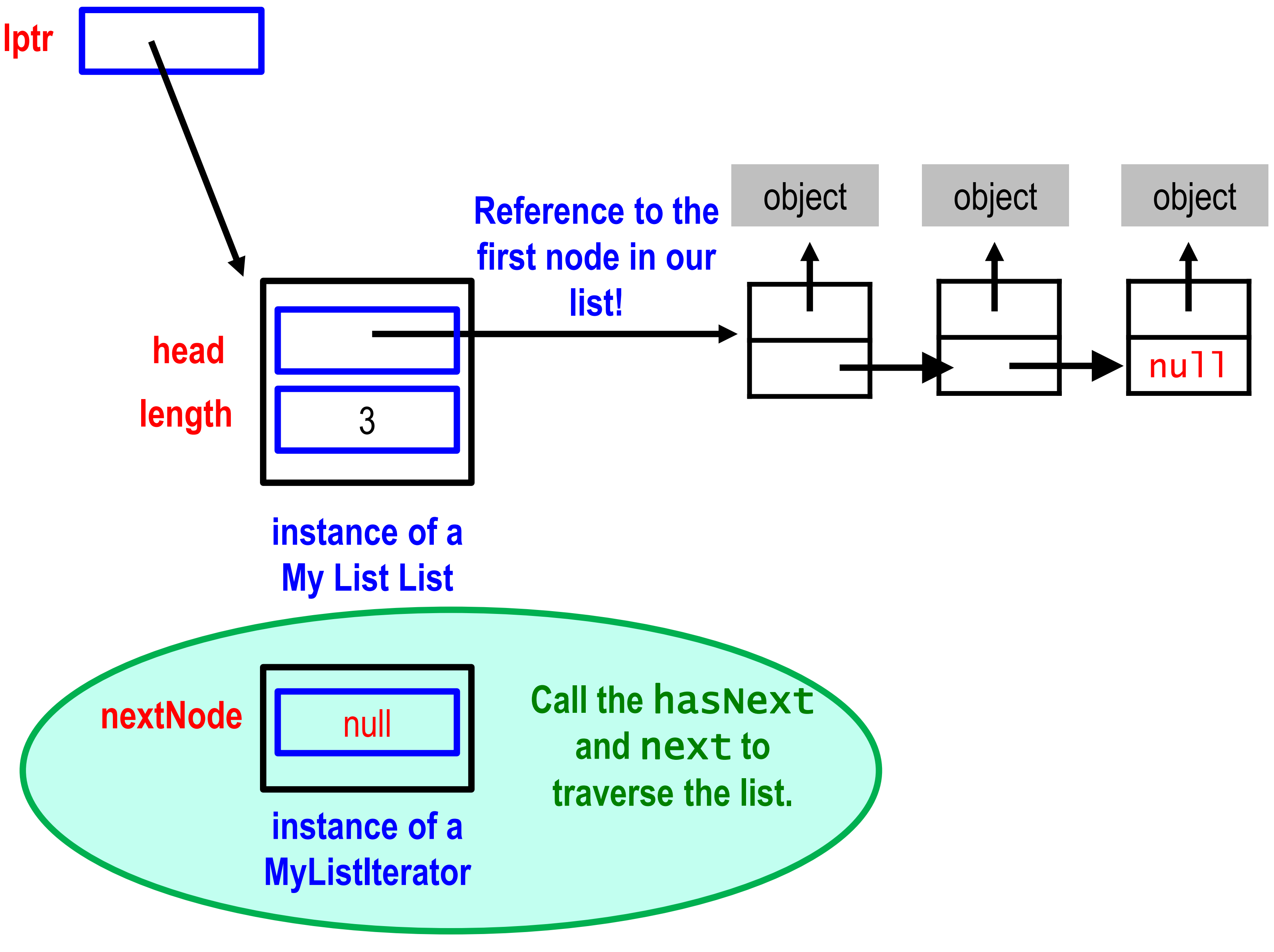
MyList Class



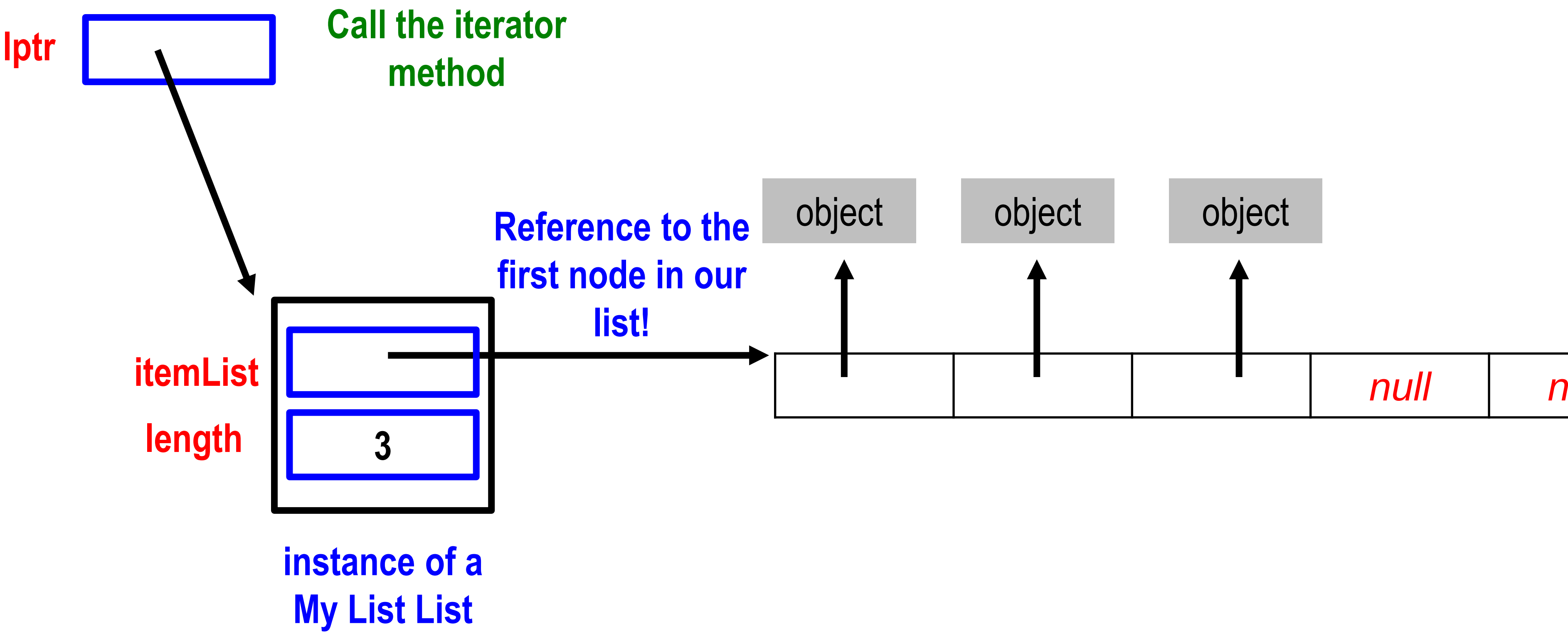
MyList Class



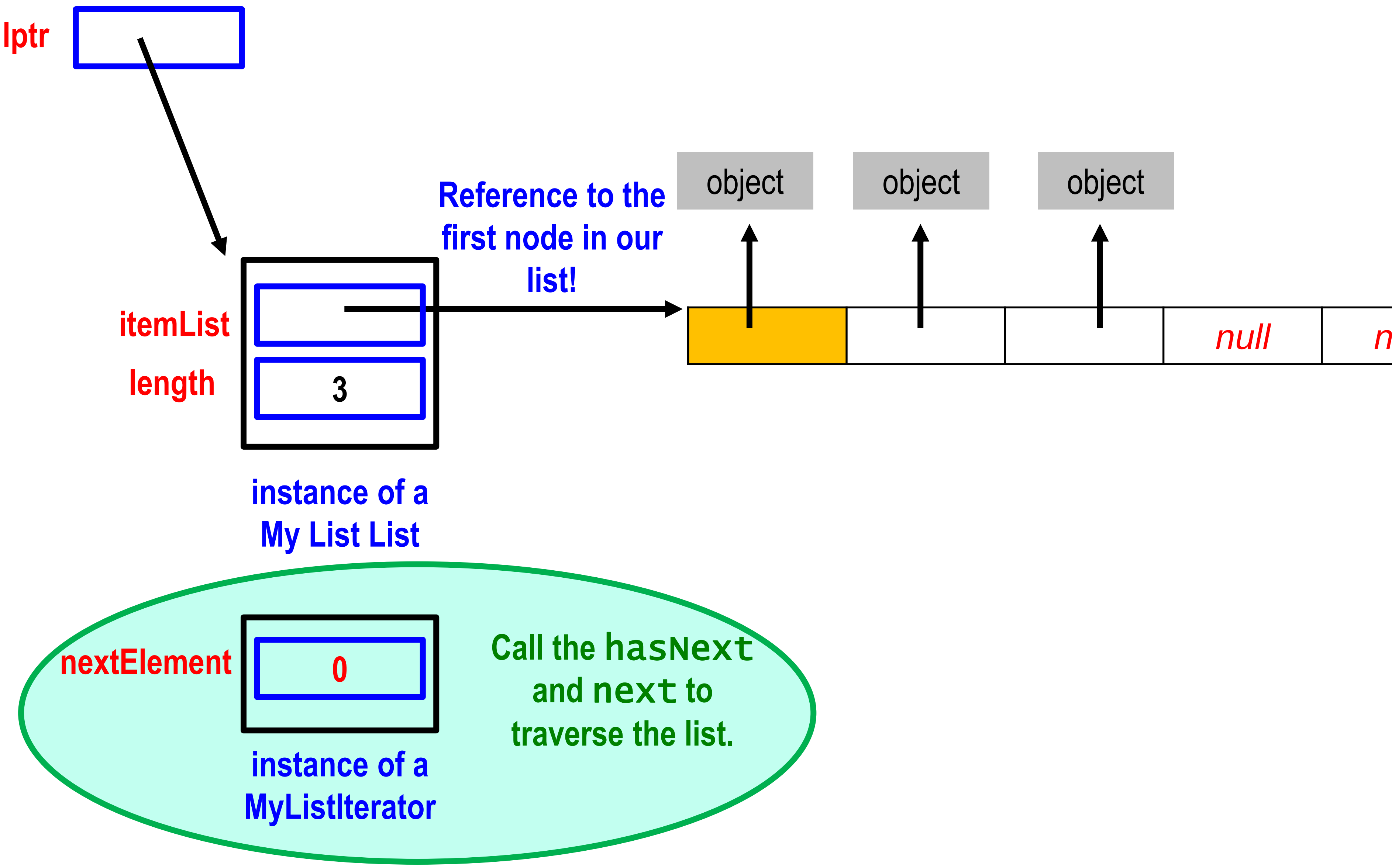
MyList Class



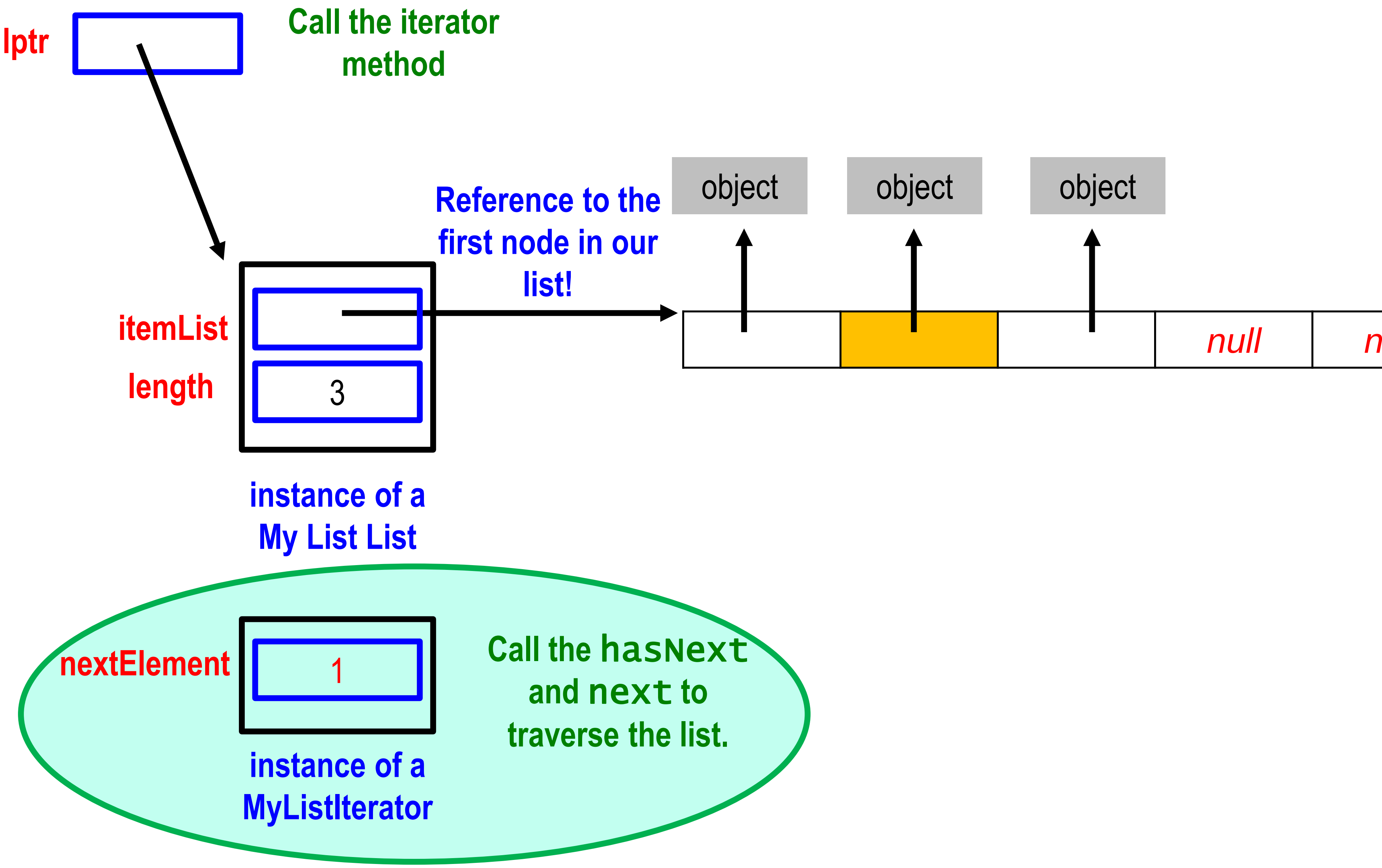
My(**Array**)List Class



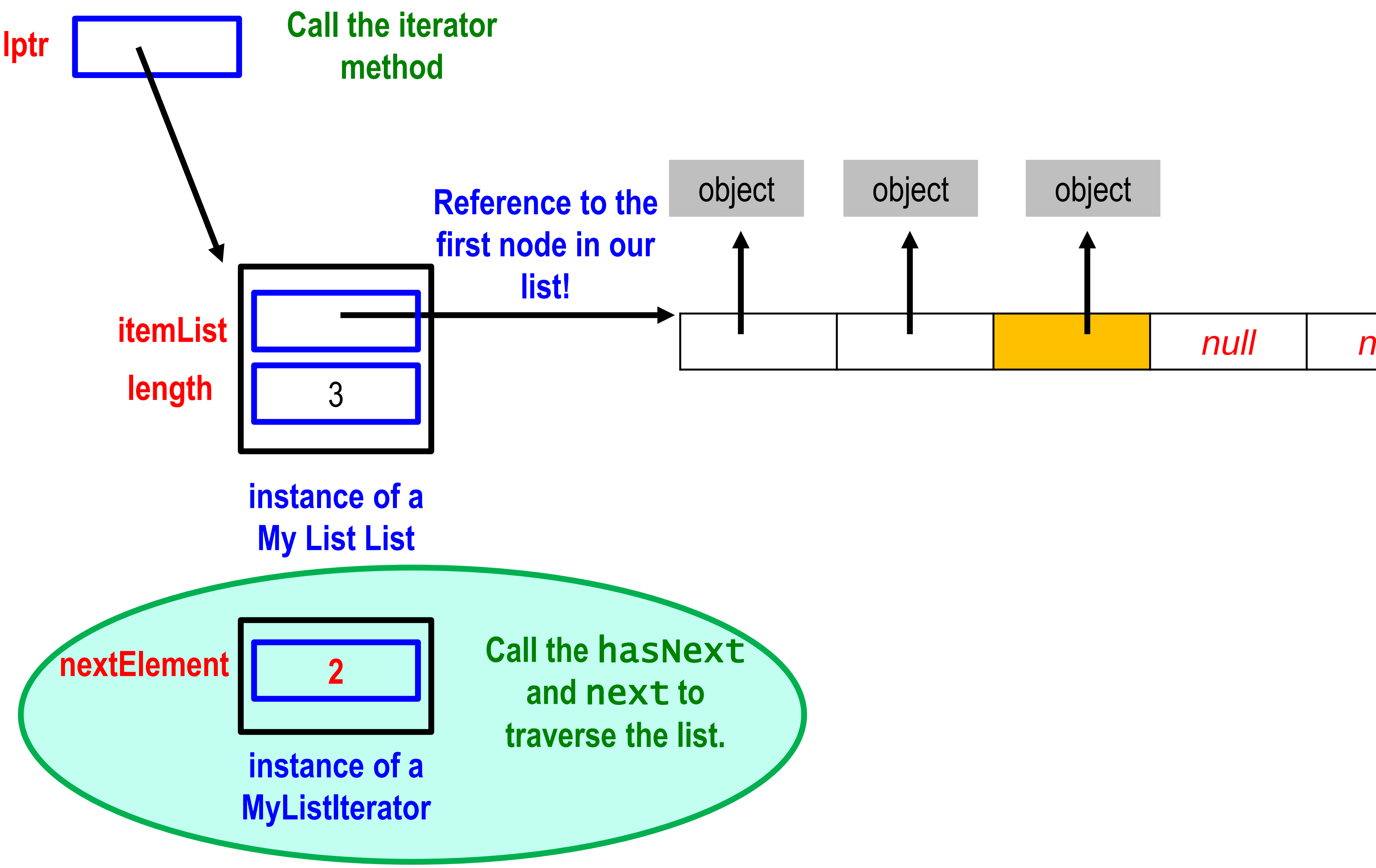
My(Array)List Class



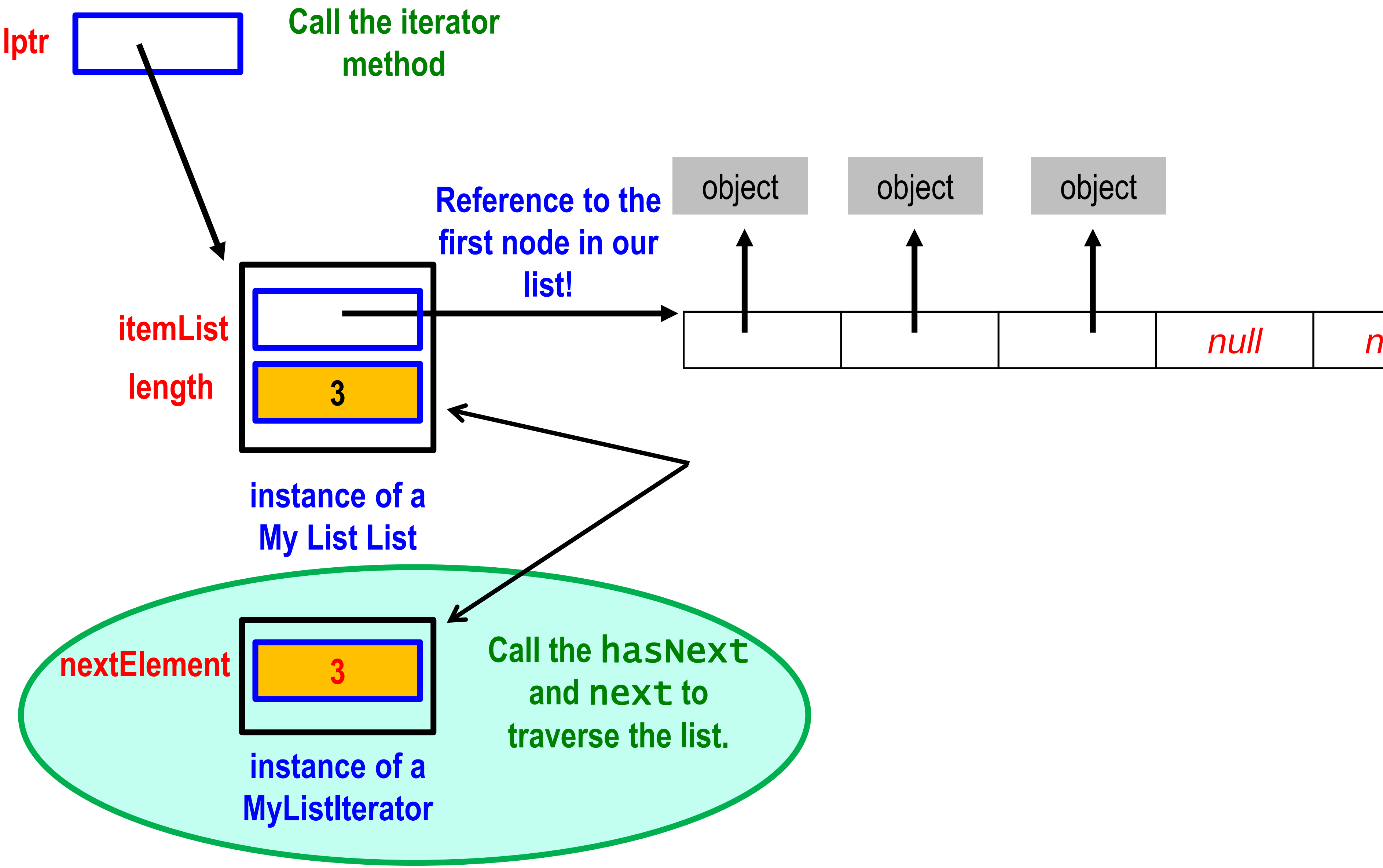
My(Array)List Class



My(Array)List Class



My(Array)List Class



Iterator Pattern:

Elements of Reusable OO Software

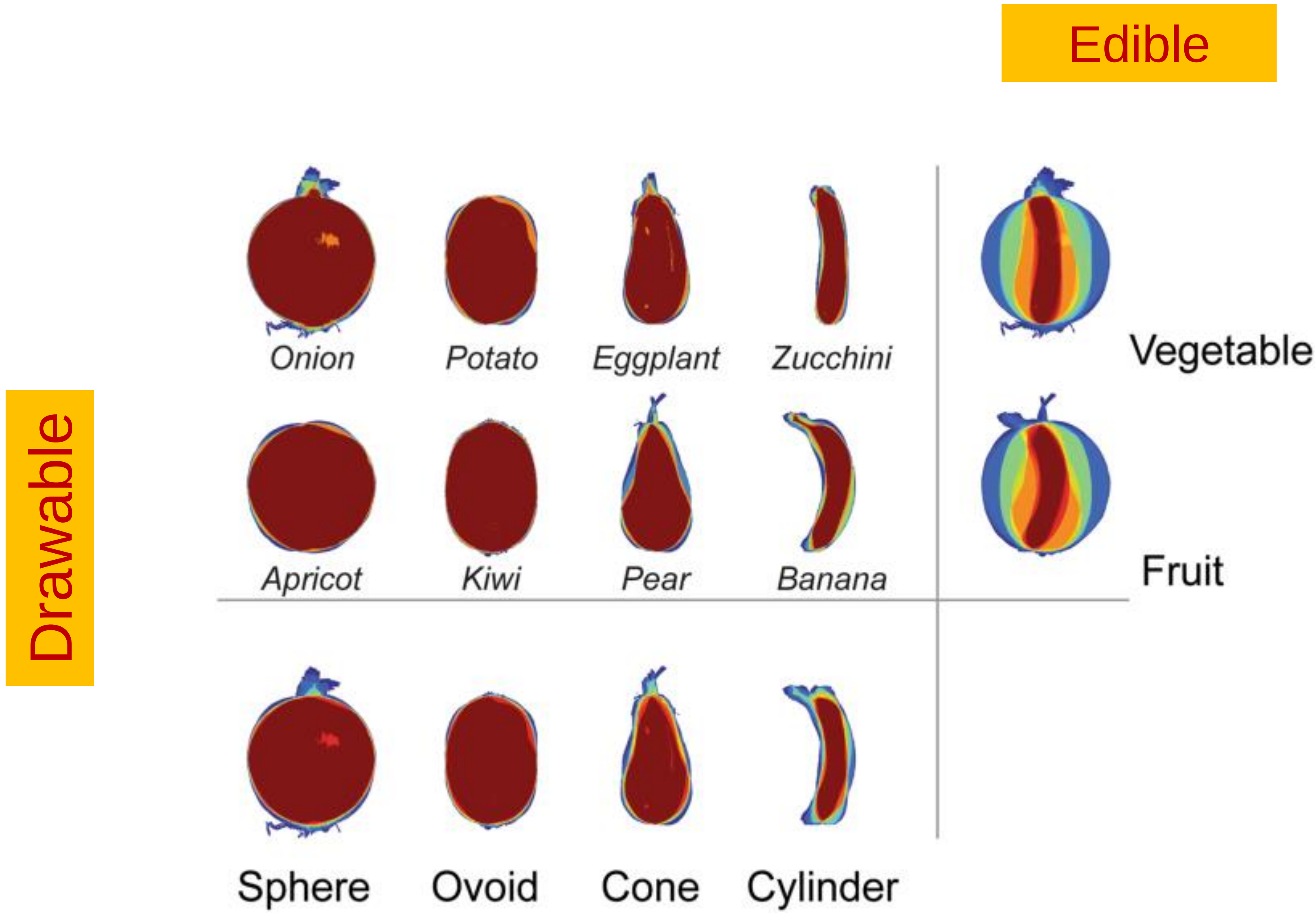
- **Consequences:** This pattern has *three* important stated consequences:
  - It separates the traversal from the Collection.
  - It supports variations in the traversal of a Collection, example: ***preorder***, ***postorder***, ***inorder***. Depending on which tree traversal we are interested in, 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.

Design Principles:
class vs. type



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.

The data members of the object.

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

- An object's class define
  implemented, and it de
- 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.

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

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

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.

- Clearly client programs need to instantiate concrete classes. Following the dictates of design patterns, the creation process has been abstracted out to a series of creational patterns that if followed ensures your program is written in terms of types or interfaces and not concrete implementations.

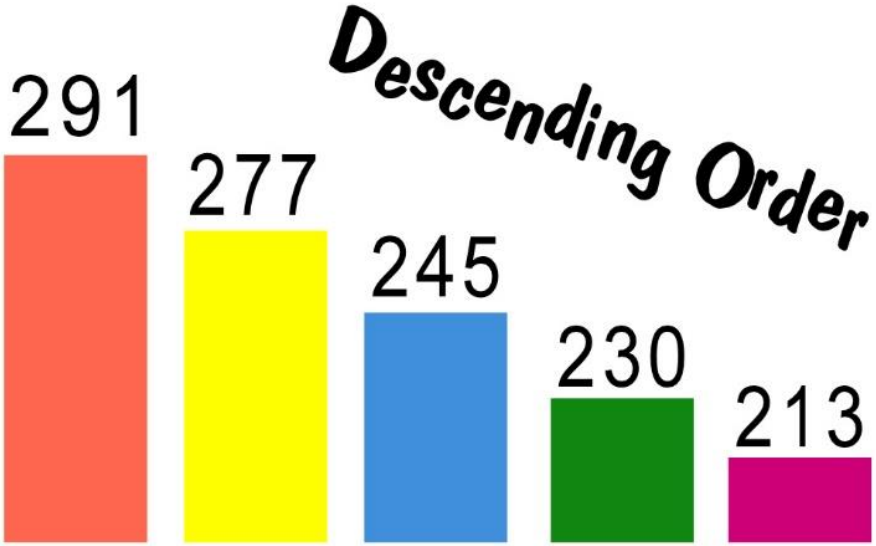
Second Principle of Good Design as stated in:

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.

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

```
Kiwi
Mango
Apples
Banana
Oranges
```

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 s2, String s2){

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


    }
} // class
```

CollectionS Class

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

        Collections.ad


        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 );
    }
}
```

Passing to the method an object of type **Comparator**!

```
Oranges
Apples
Banana
Mango
Kiwi
```

CollectionS Class

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

        Collections.ad


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

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

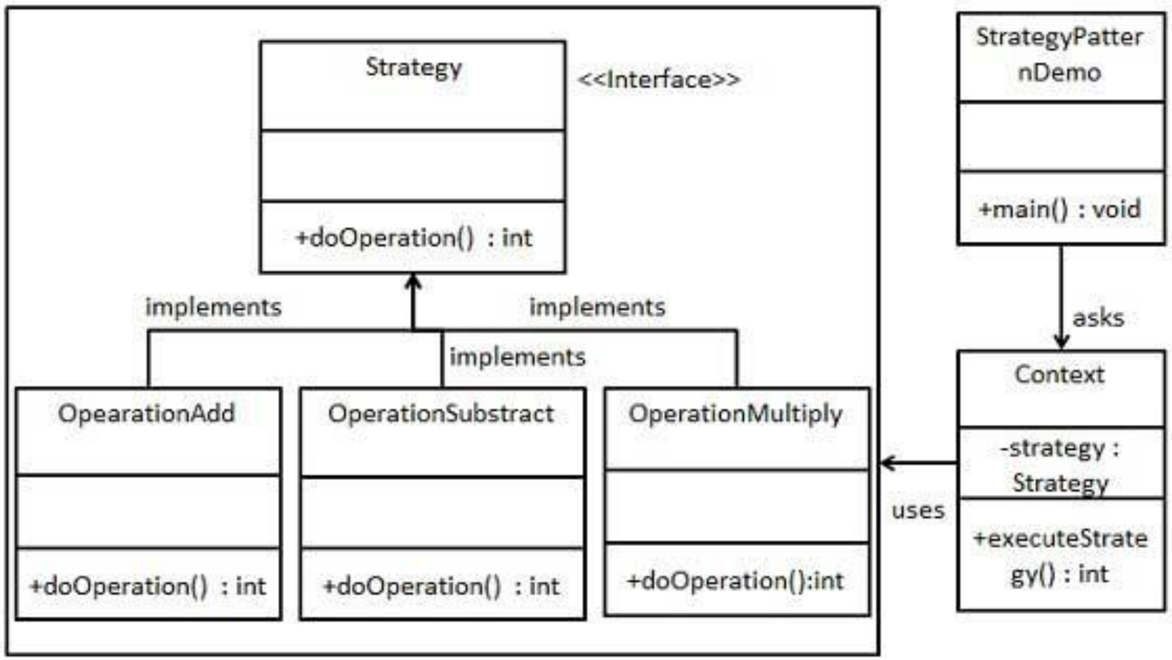
Passing to the method an object of type **Comparator**!

```
Oranges
Apples
Banana
Mango
Kiwi
```

Strategy Pattern:

Reuse through object composition

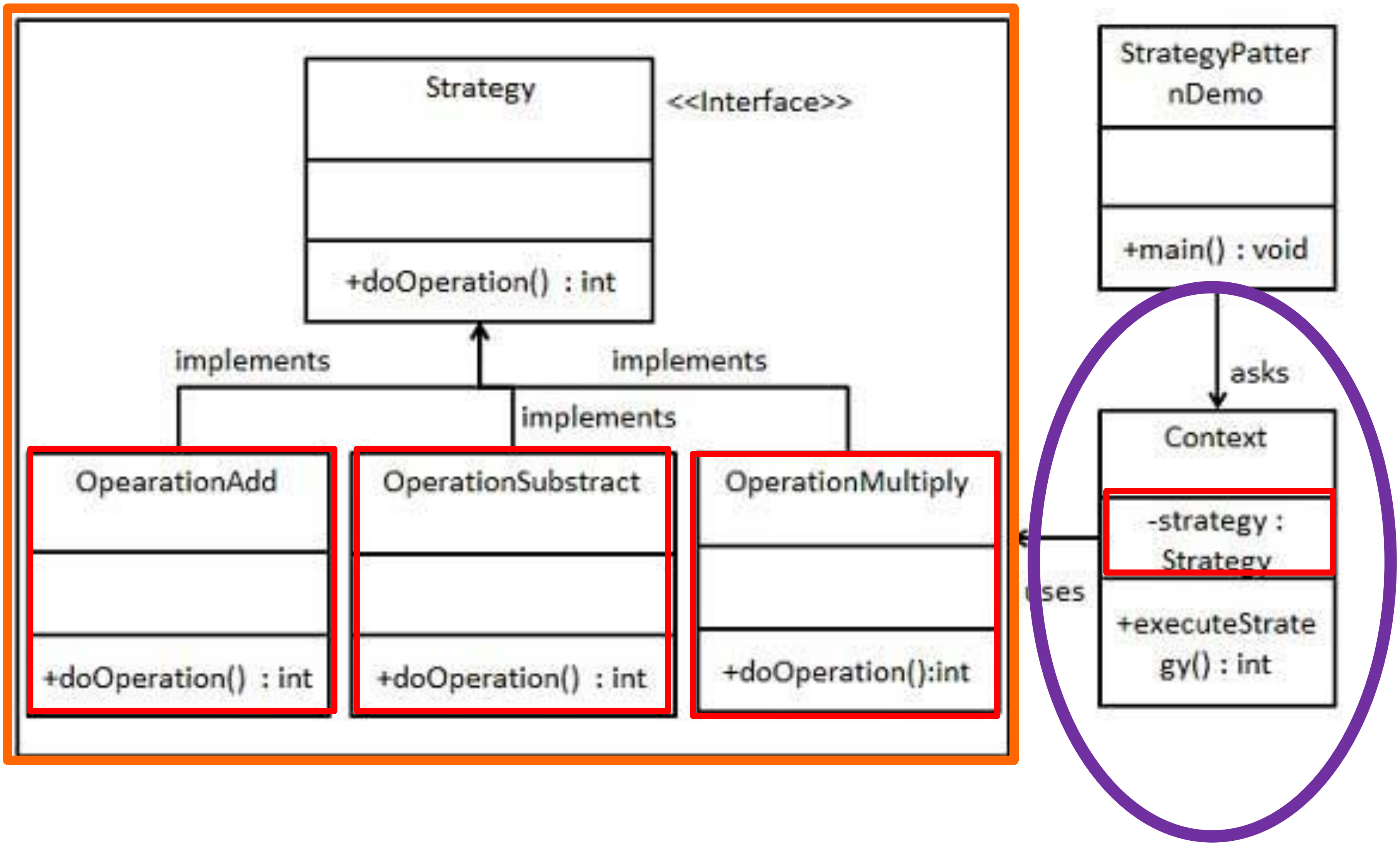
Intent: Define a **family of algorithms**, encapsulate each one, and make them interchangeable.



[Tutorials Point]

Strategy Pattern:

Reuse through object composition

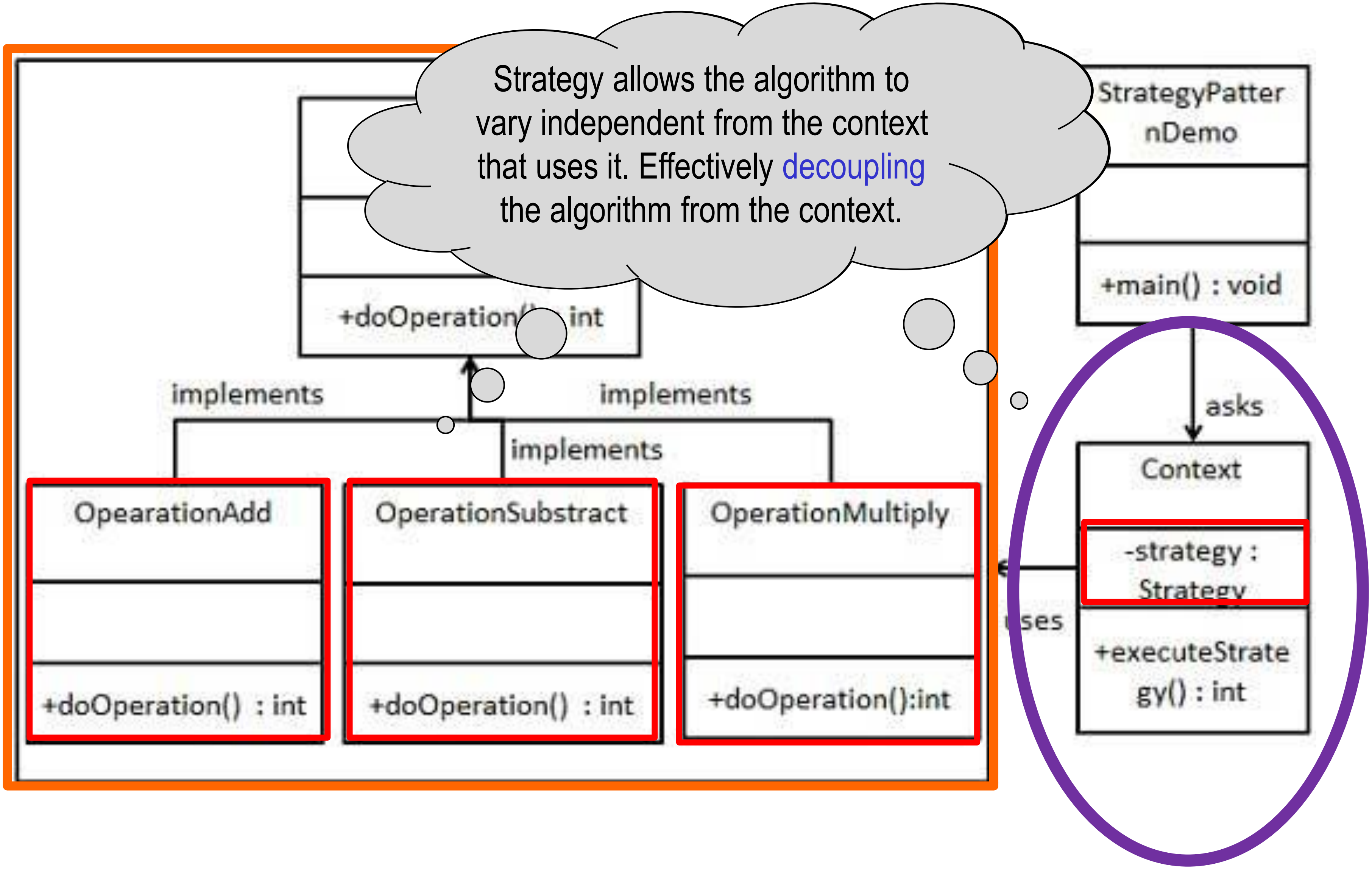


[Tutorials Point](#)

Strategy Pattern:

Reuse through object composition

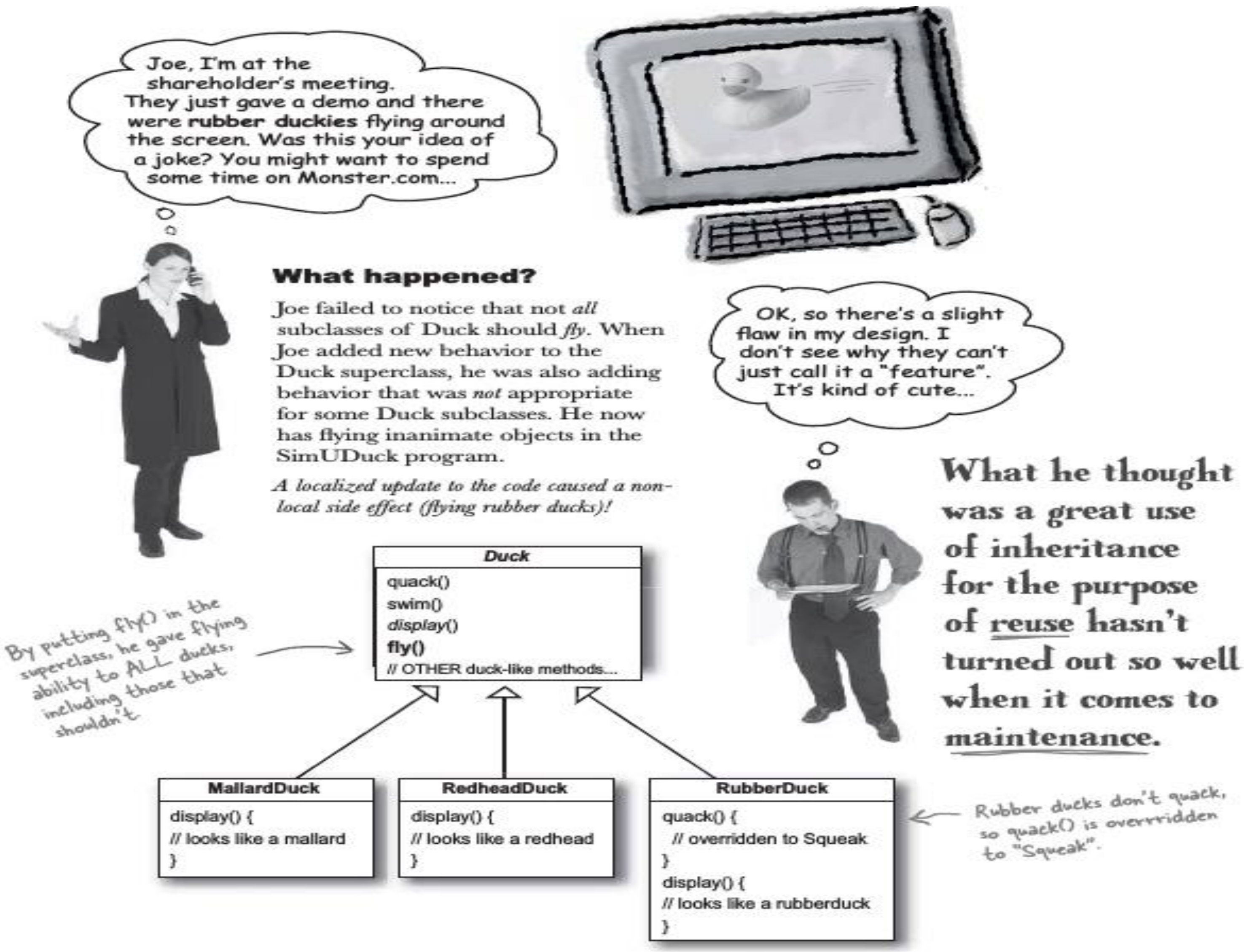
Strategy allows the algorithm to vary independent from the context that uses it. Effectively decoupling the algorithm from the context.



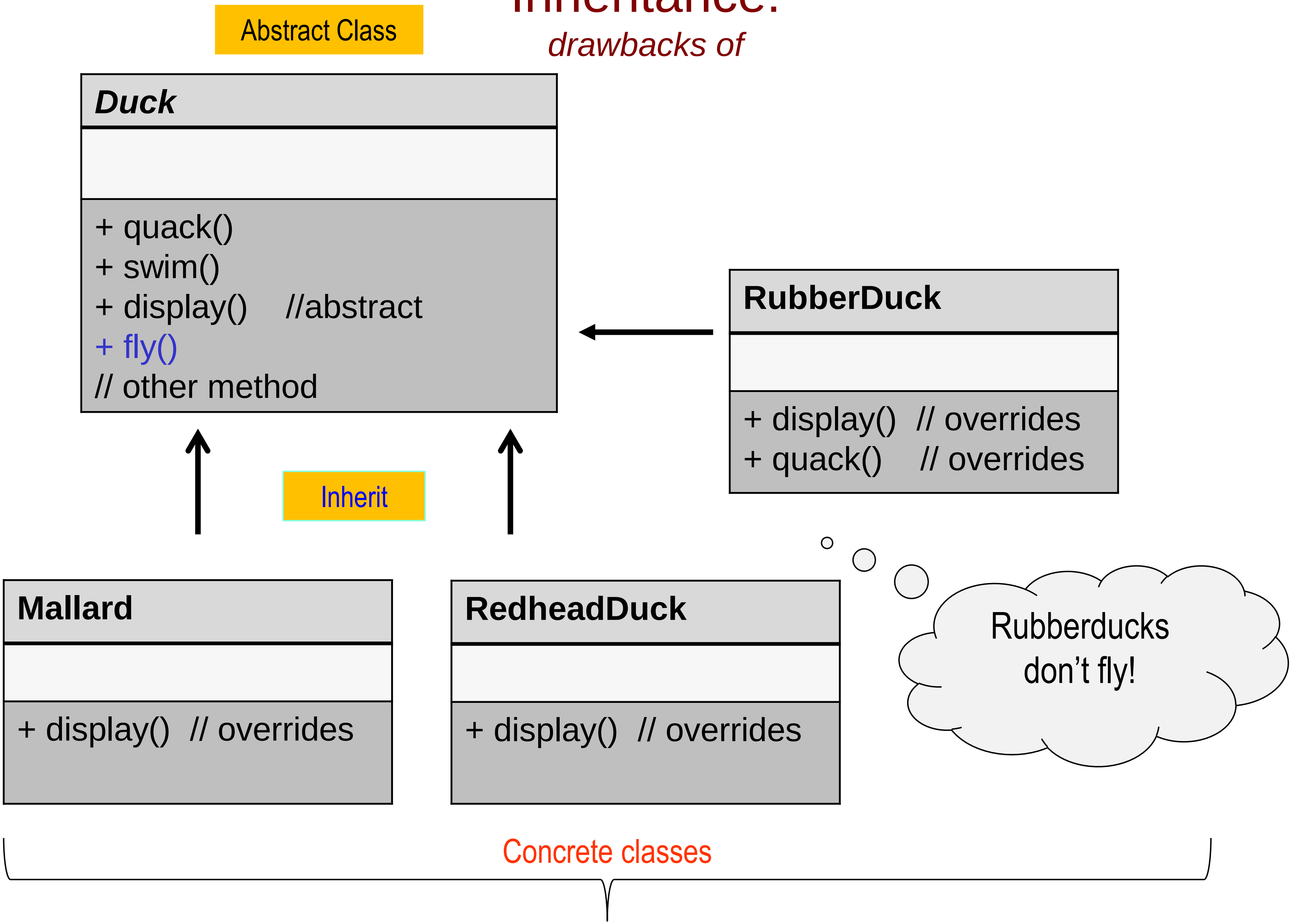
[Tutorials Point]

Strategy Pattern:

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

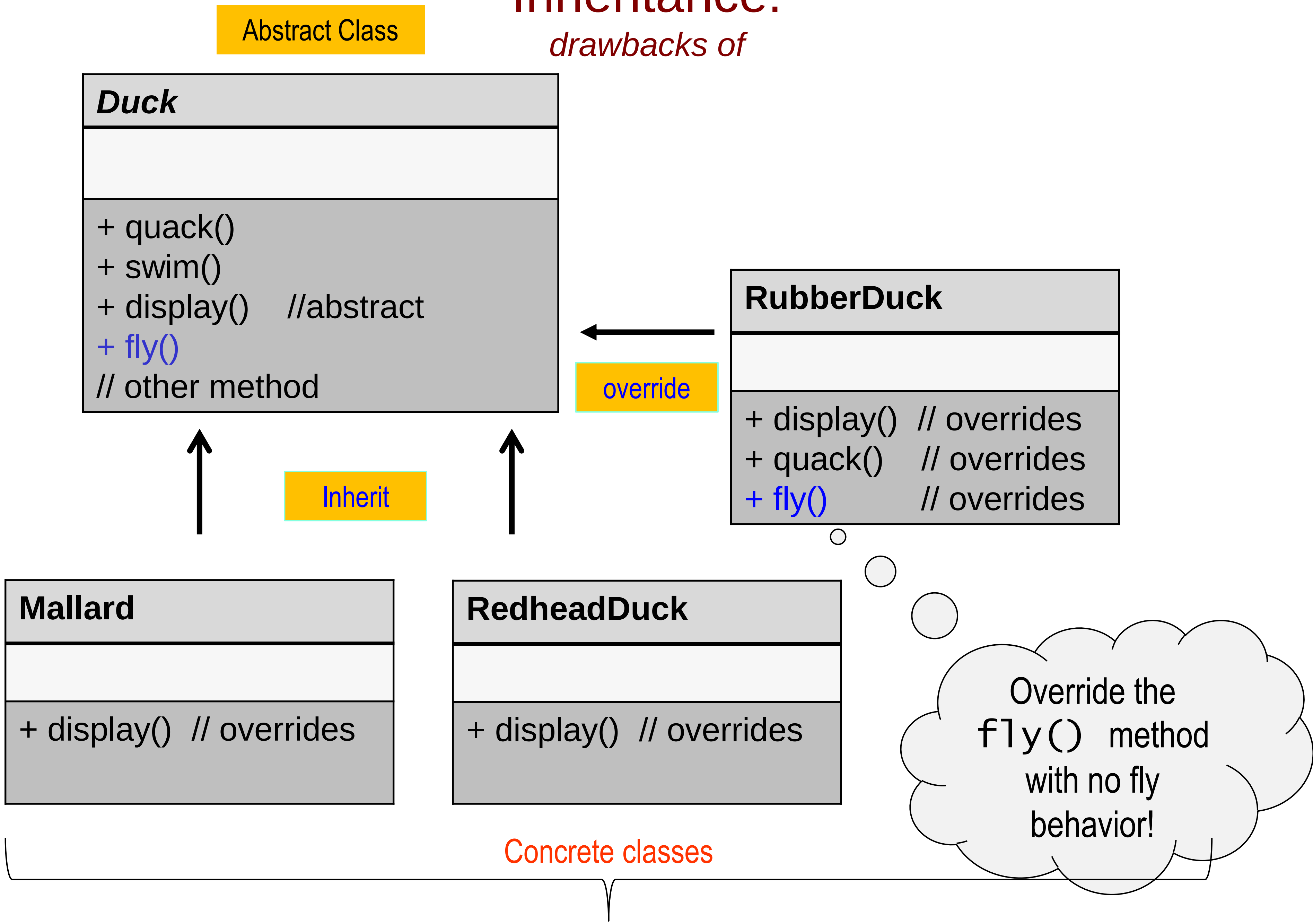


Inheritance:
drawbacks of



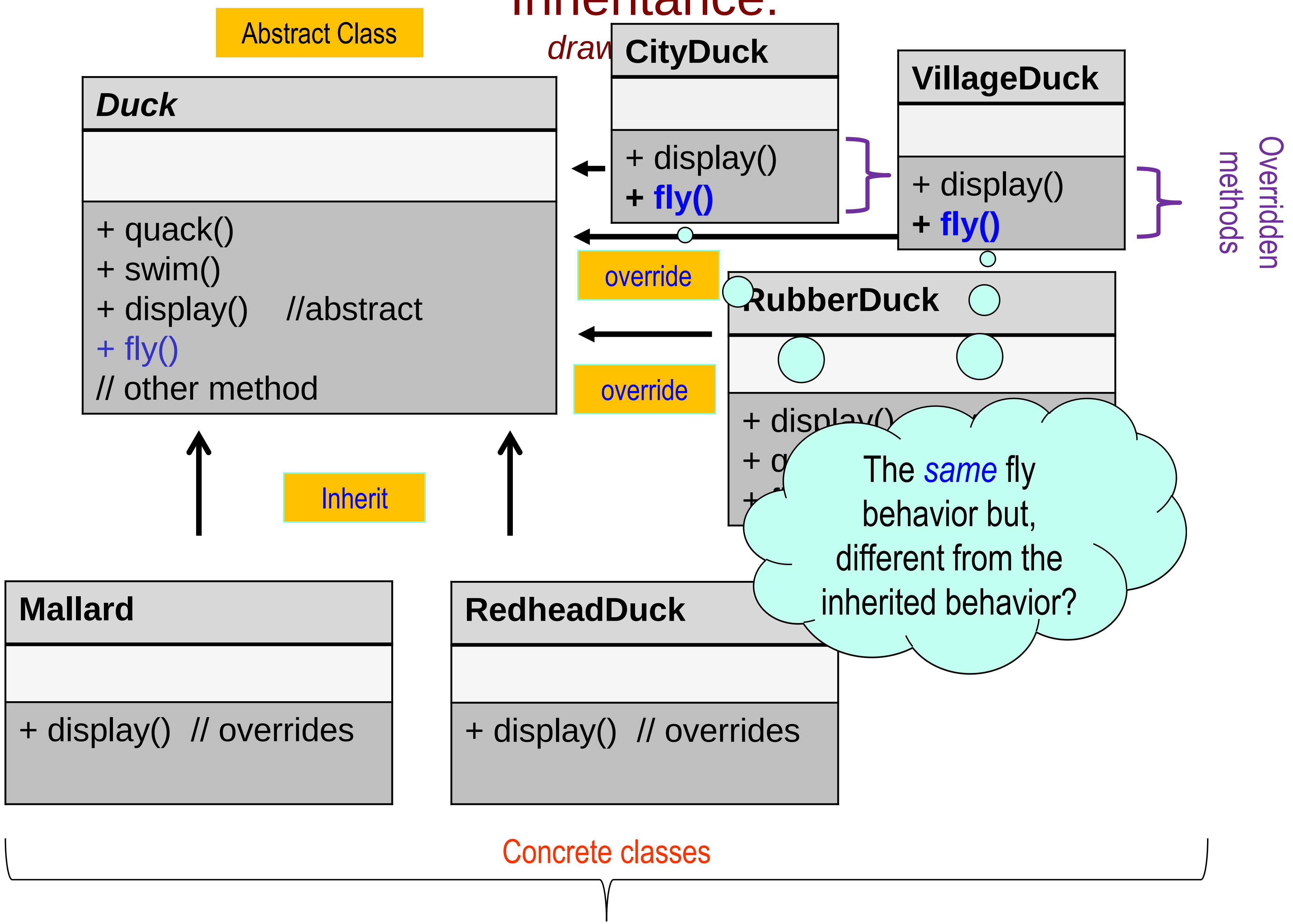
Inheritance:

drawbacks of

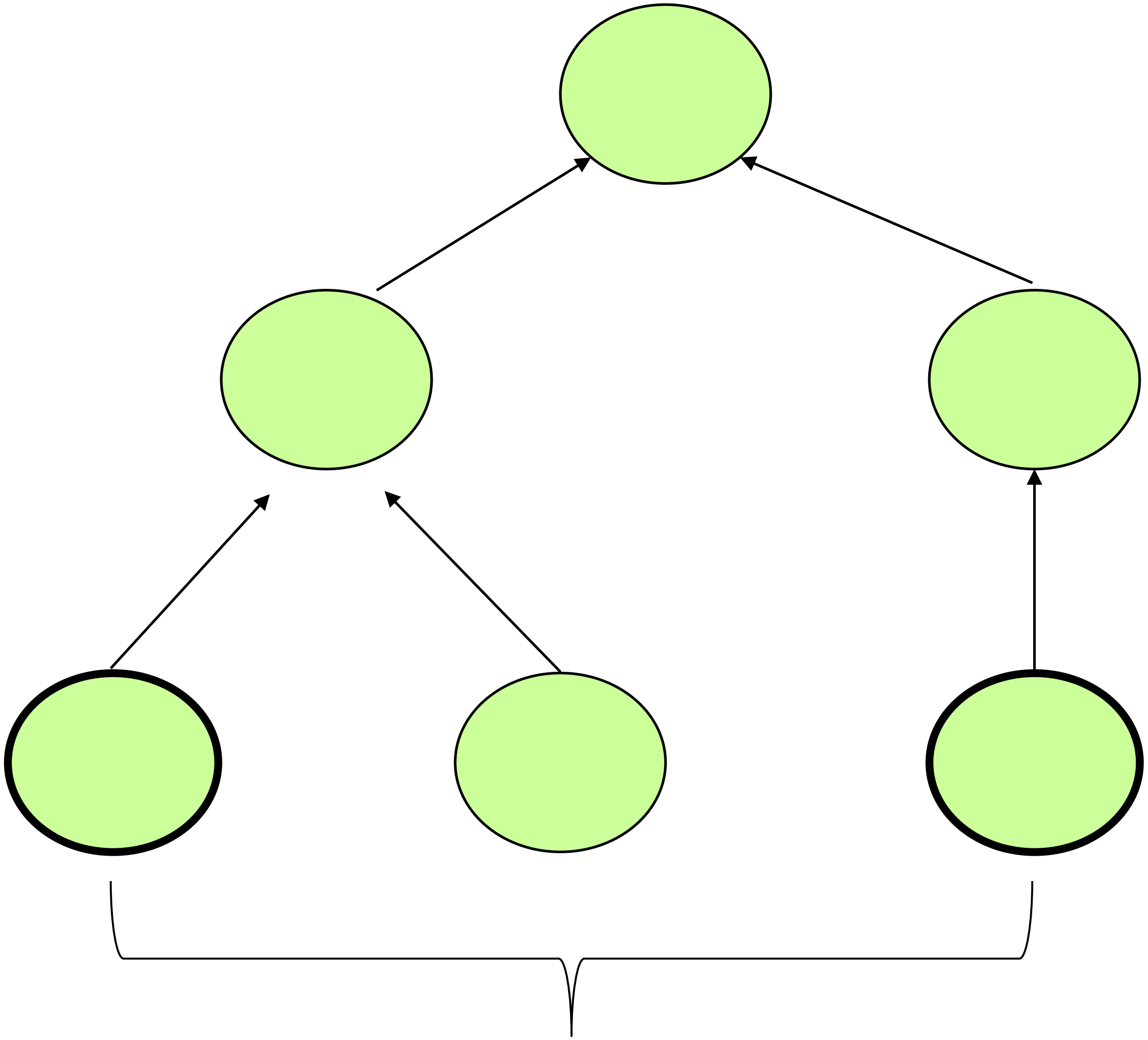


Inheritance:

draw

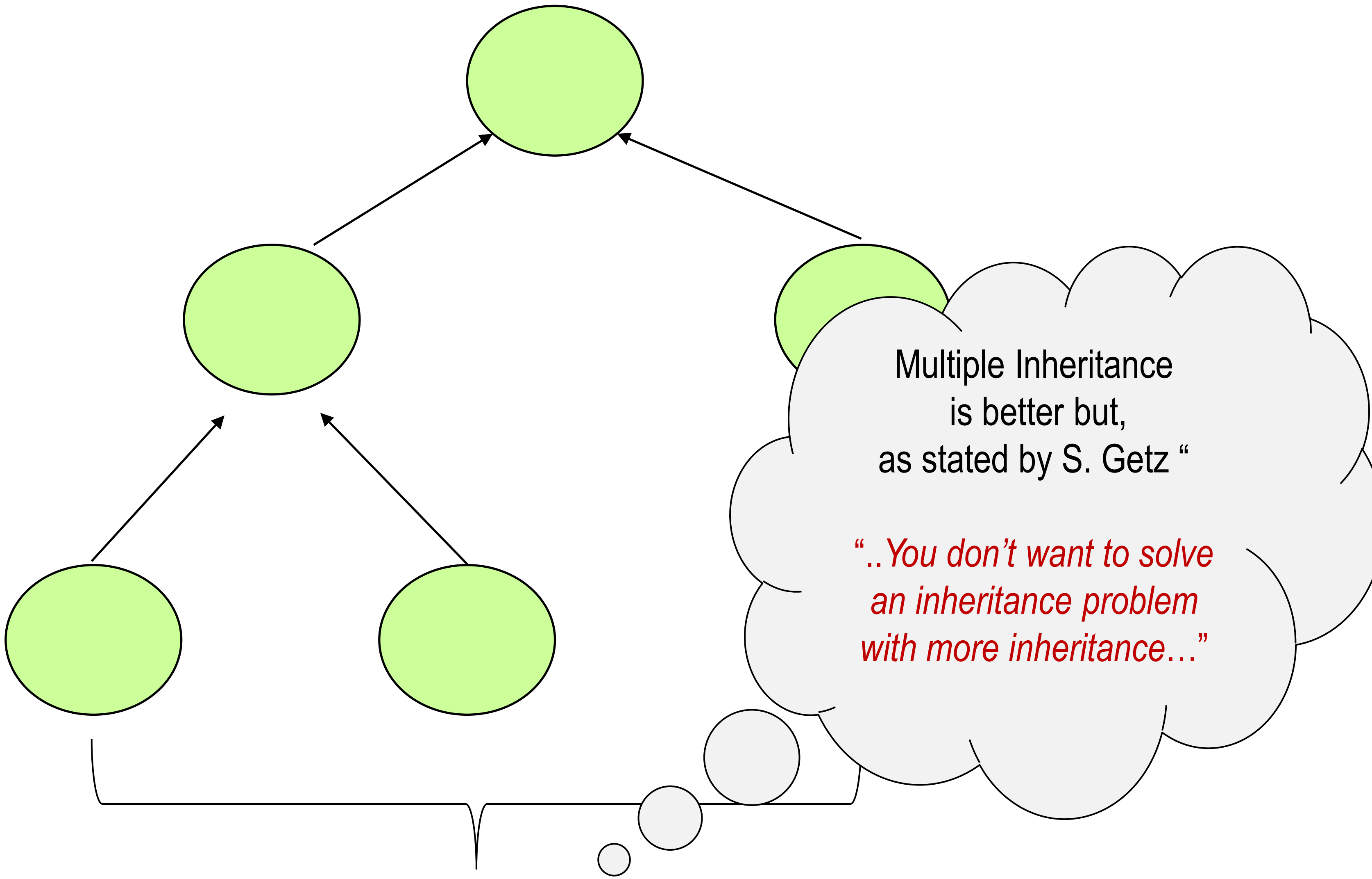


Problem with (Single) Inheritance



Sharing behavior *horizontally* is
problematic in Single Inheritance

Problem with (Single) Inheritance



Sharing behavior *horizontally* is problematic in Single Inheritance

An alternative:

an interface

Interface

Flyable

+ fly()

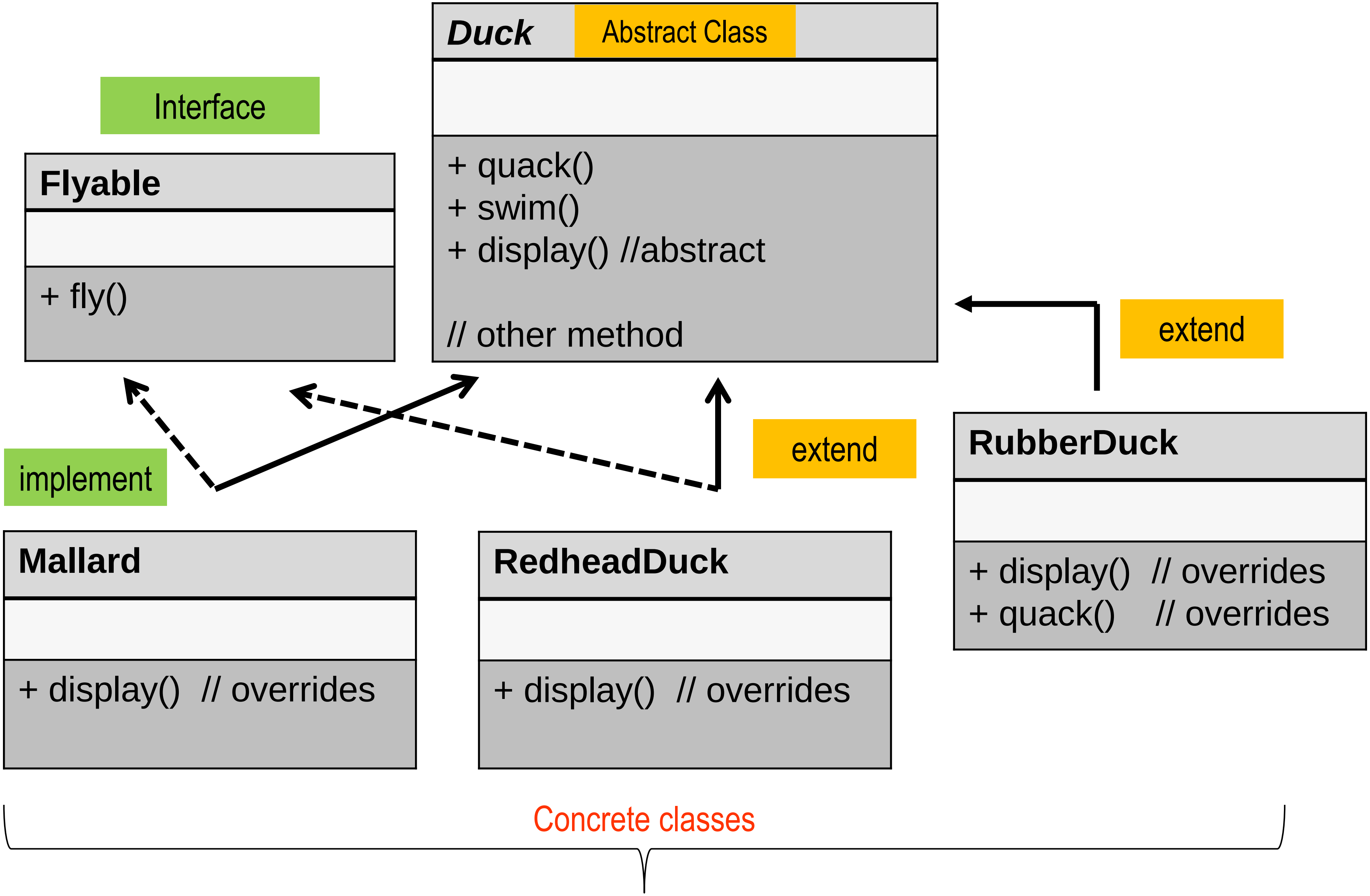
Duck Abstract Class

+ quack()
+ swim()
+ display() //abstract

// other method

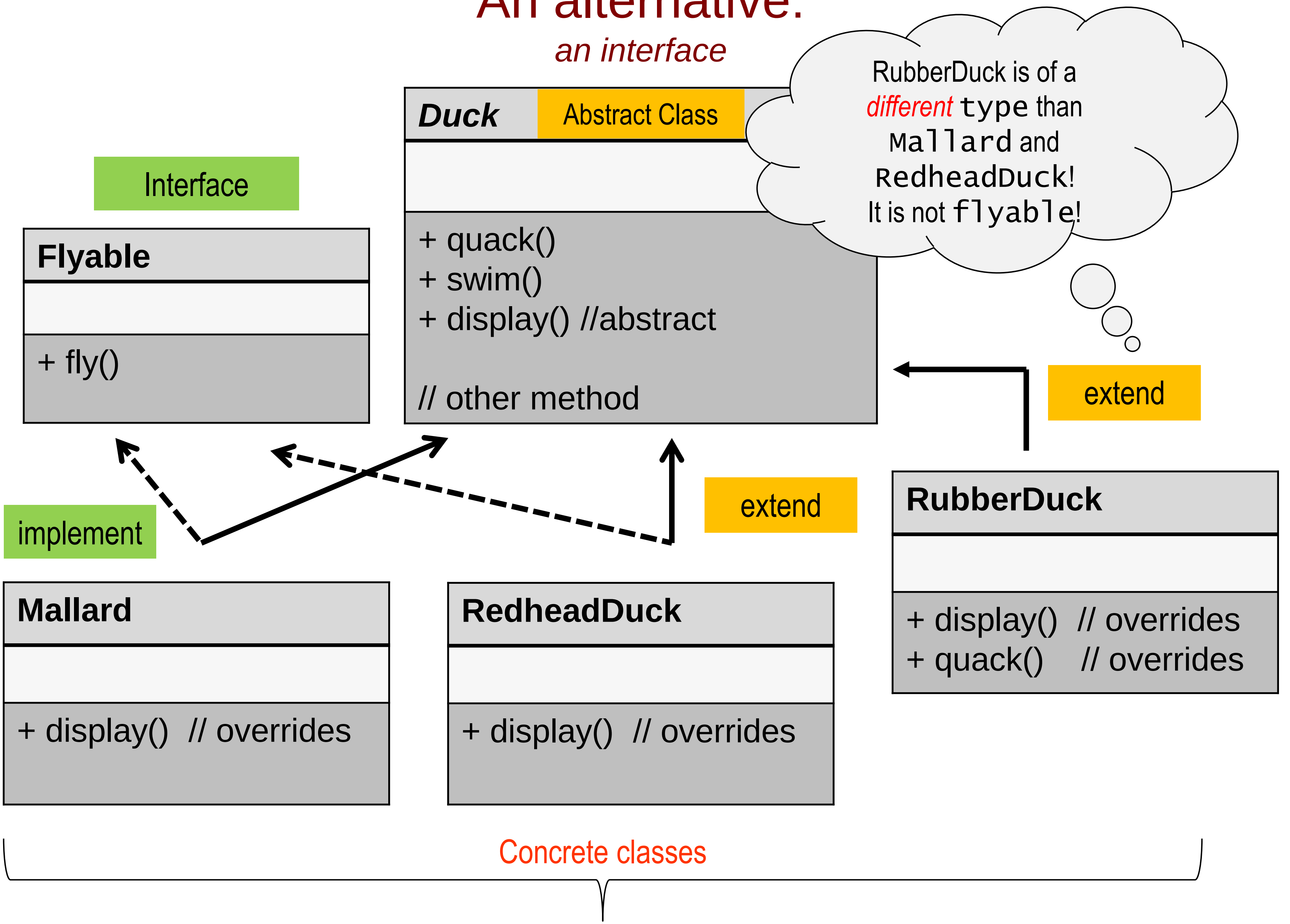
An alternative:

an interface



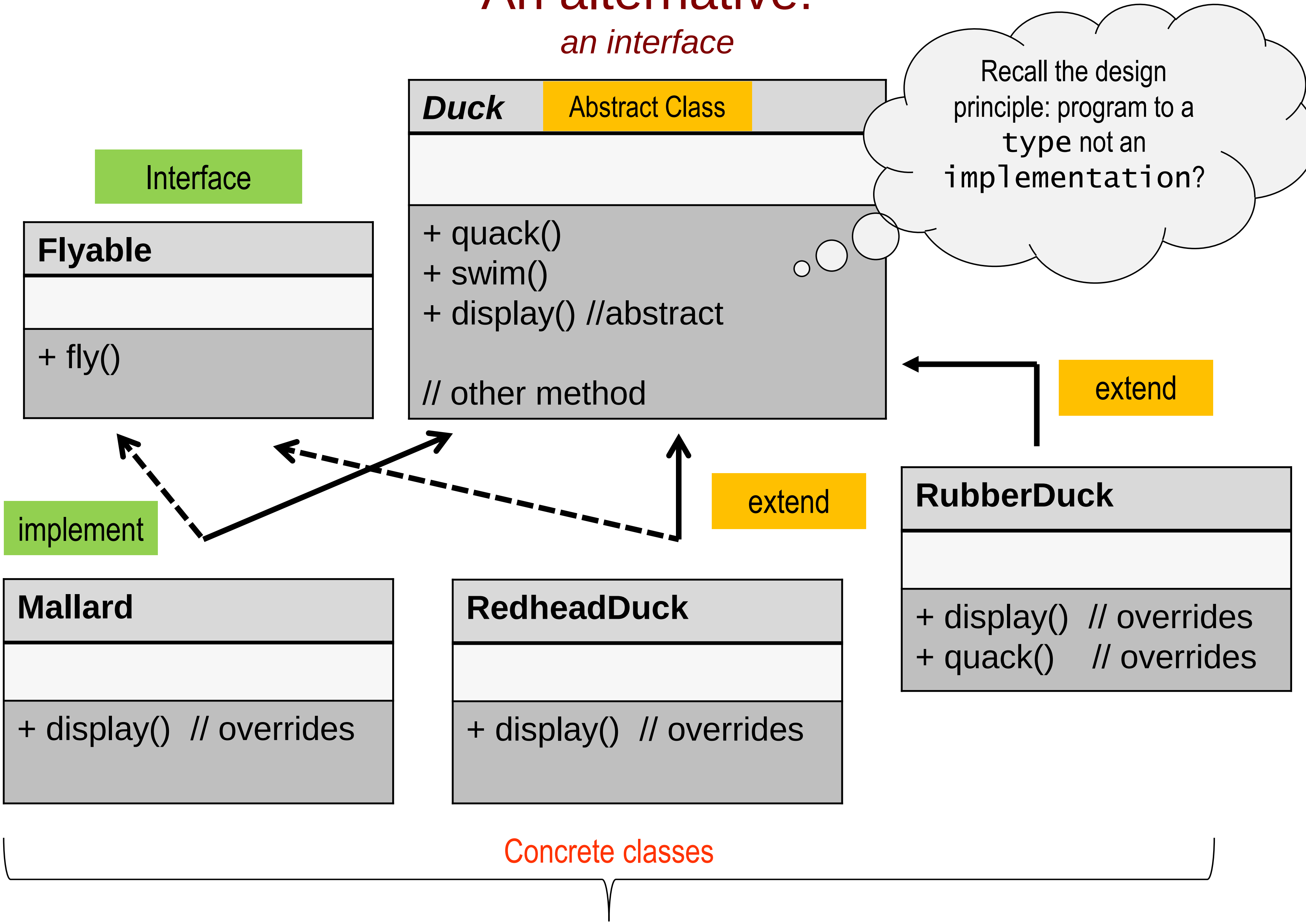
An alternative:

an interface



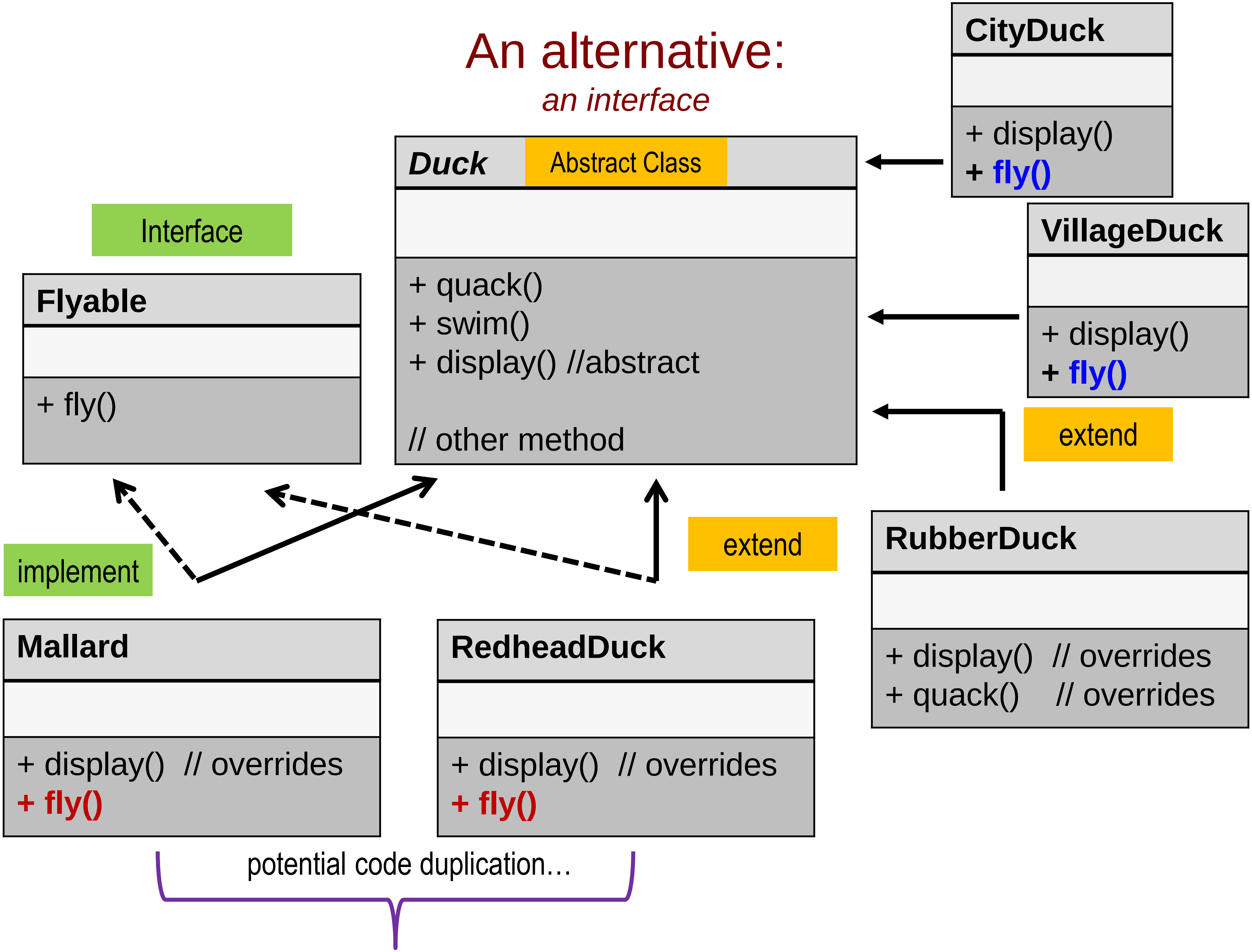
An alternative:

an interface



An alternative:

an interface



Core Design Principle

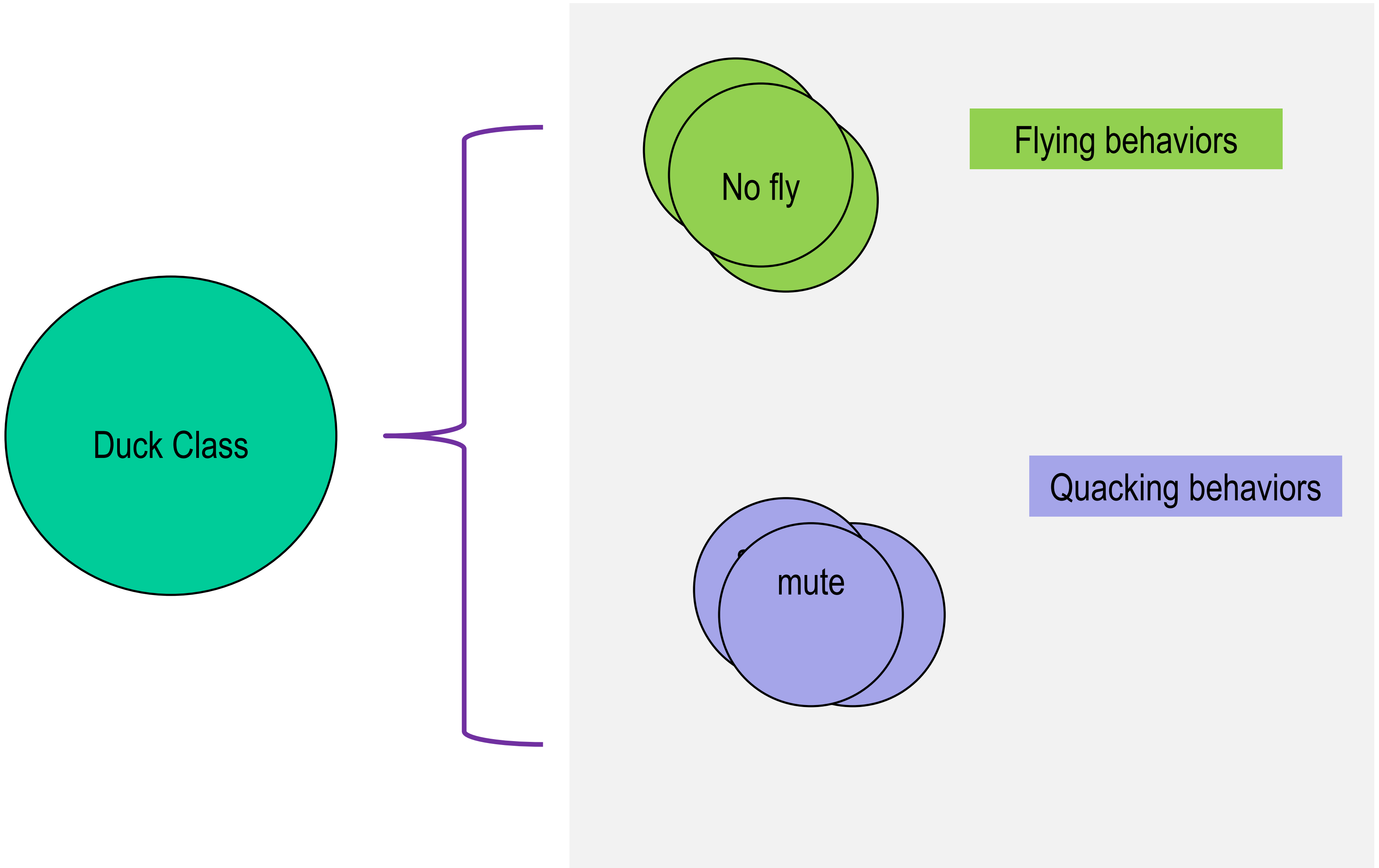
- Separate what changes from what stays the same. This is a core design principle. Recall *Abstraction by Parameterization*. The use of variables allow us to write logically structured code that operates on different variables.

- 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.

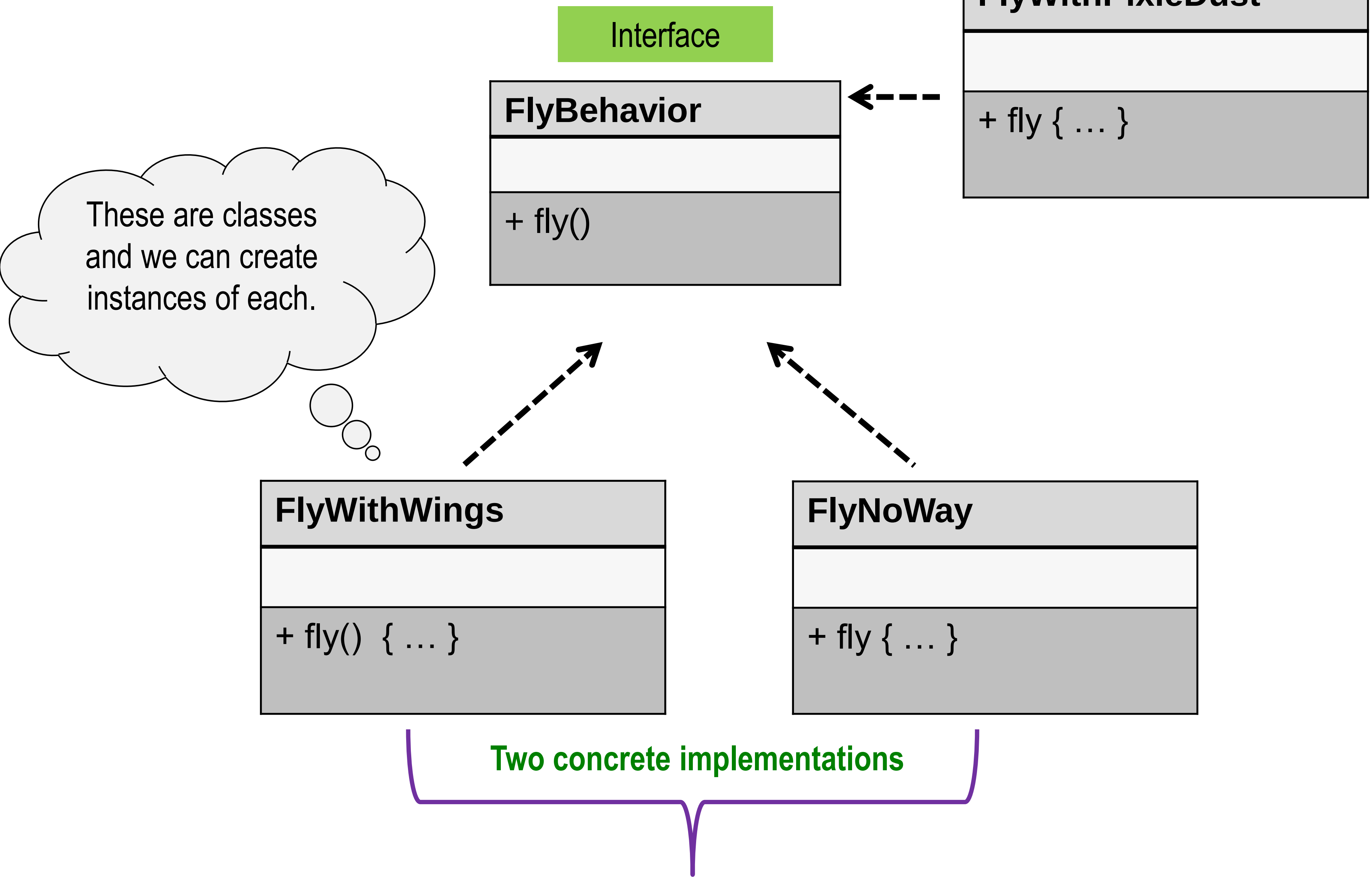
- In our example, the behaviors that can vary are:
  - how ducks fly, and
  - how ducks quack.

Duck class revisited



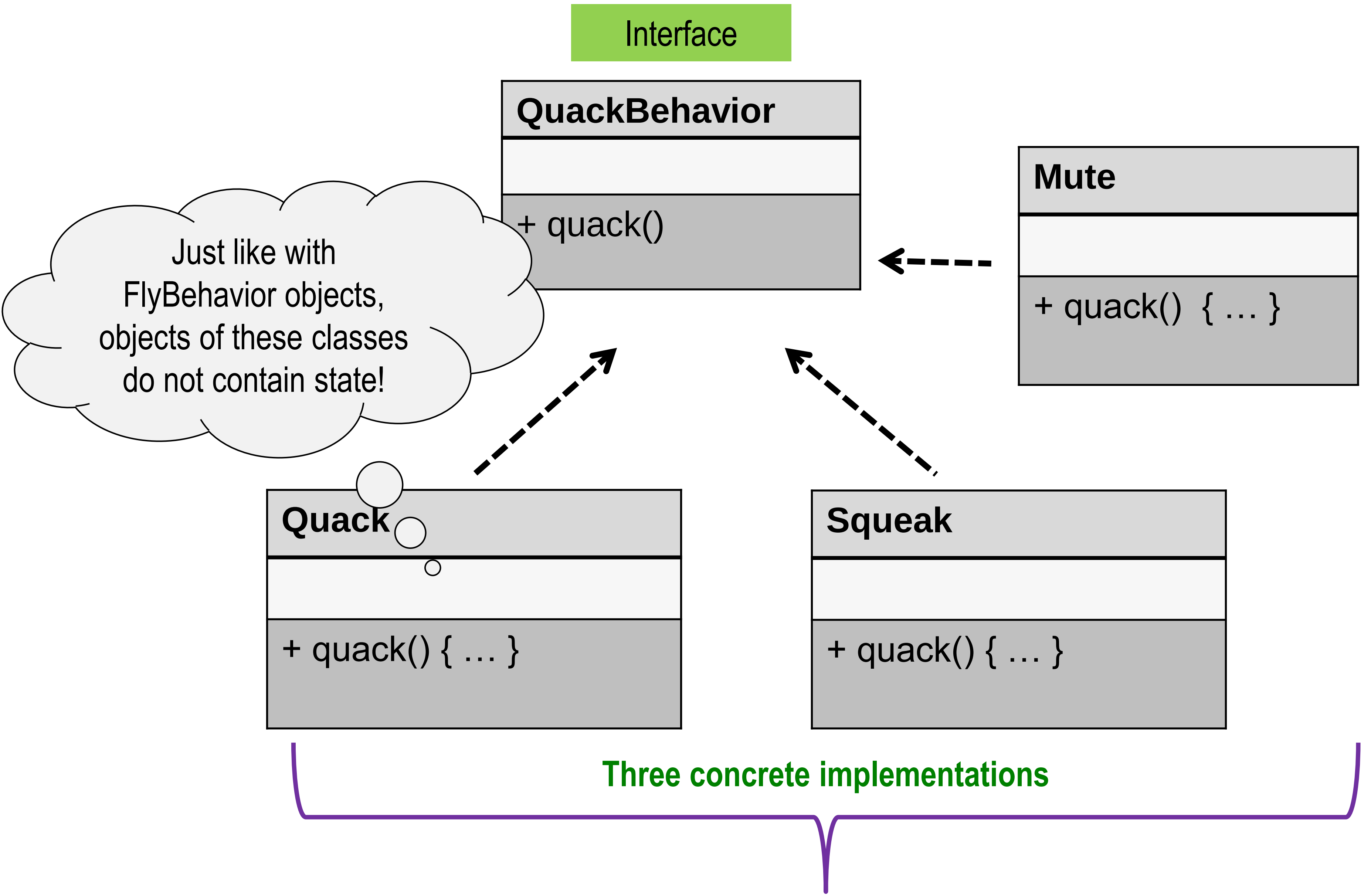
An alternative:

an interface



An alternative:

an interface



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

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

```
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

Mallard

+ display() // overrides

```
public class MallardDuck extends Duck
{

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

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

Duck class revisited…

Duck Abstract Class

- FlyBehavior flyBehavior
- QuackBehavior quackBehavior

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

↑

RubberDuck

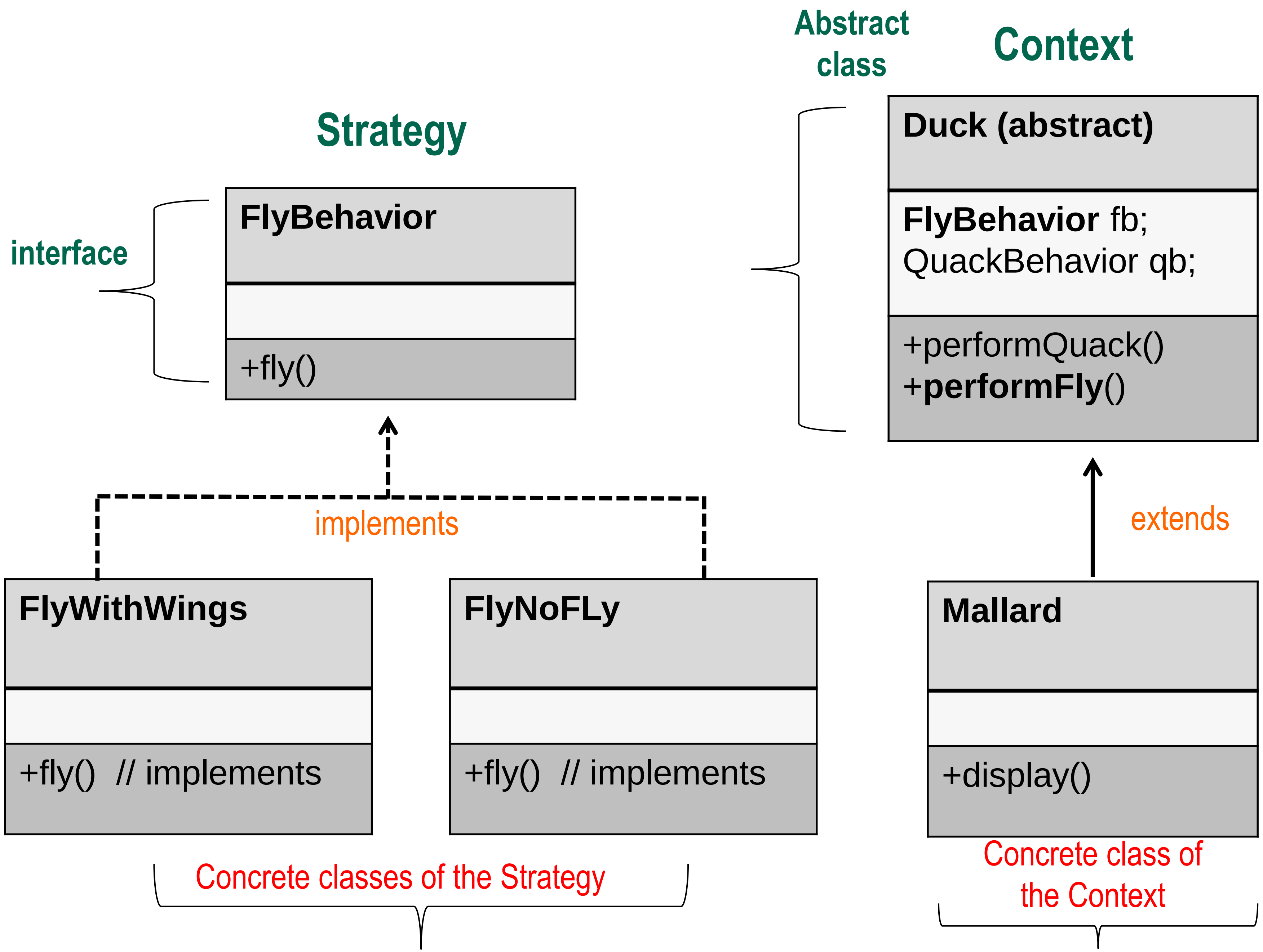
+ display() // overrides

```
public class RubberDuck extends Duck
{

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

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

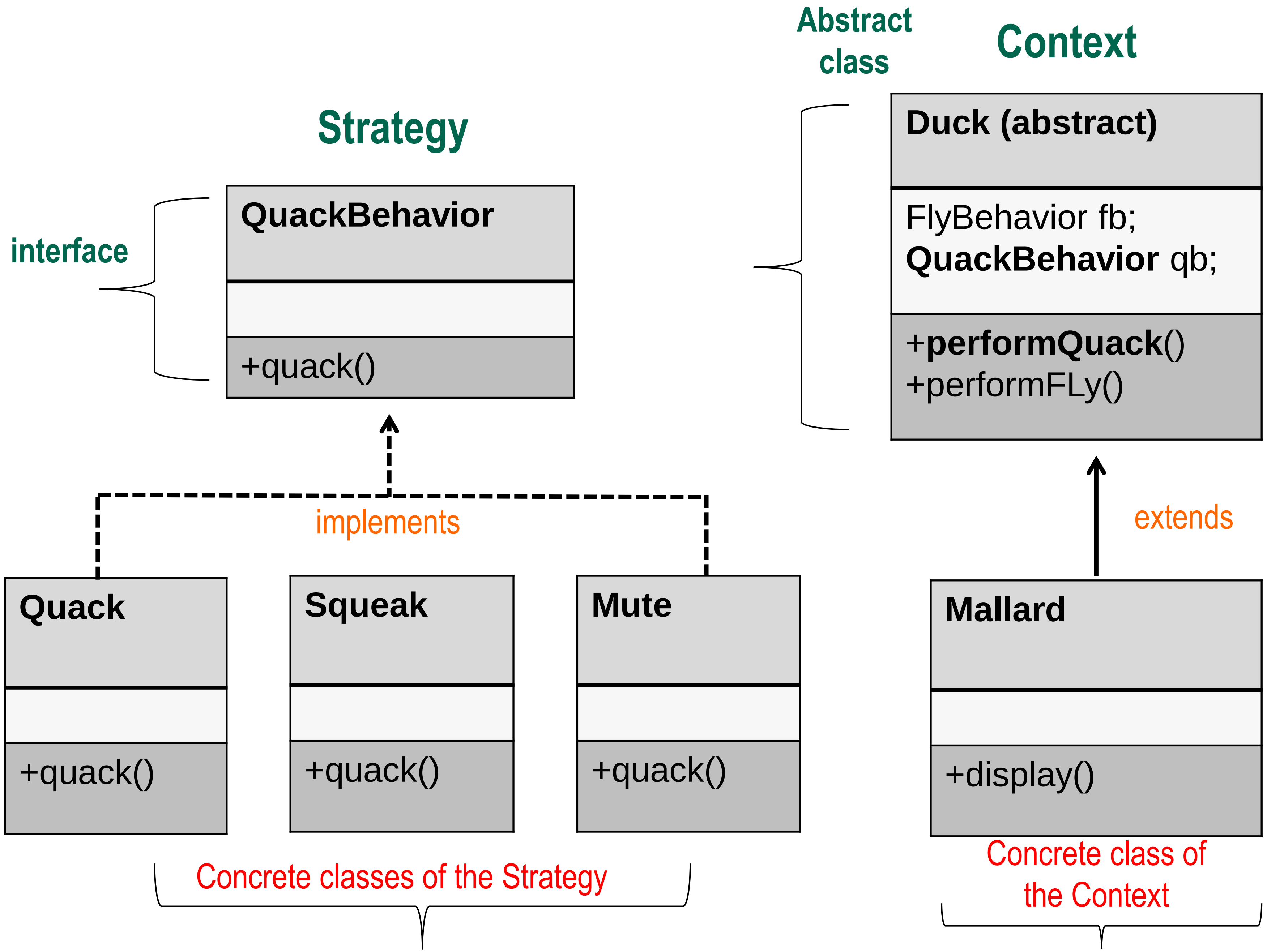
Duck class revisited…

```
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

- **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.