# COMP 3111
# SOFTWARE ENGINEERING

## LECTURE 20
## SYSTEM ANALYSIS AND DESIGN

# SYSTEM DESIGN OUTLINE

✓ System Design Overview
- Life Cycle Role
- The Purpose and Importance of System Design
- Realizing Design Goals
- Dealing with the Implementation Environment

➔ **System Analysis and Design Activities**
- √ Architectural Analysis and Design
- √ Use-case Analysis
- √ Class Design
- √ Object Behaviour Analysis: State Machine Diagrams
- ☞ **Design Patterns**
- Anti Patterns

# DESIGN PATTERNS

**A *design pattern* is a general reusable solution to a commonly occurring problem in software design.**

- Represents a *solution* to a common *problem* that arises within a particular *context* when developing software.

  ☞ design pattern ≡ problem/solution pairs *in a context*

  ☞ **Particularly useful for describing how and why to resolve *nonfunctional requirements.***

- It is a description or template for how to solve a certain problem that can be used in many different situations.

  ☞ **It is not a finished design that can be transformed directly into code.**

# DESIGN PATTERNS

**A *design pattern* is a general reusable solution to a commonly occurring problem in software design.**

- It typically shows relationships and interactions between classes or objects, without specifying the final application classes or objects that are involved.

  ☞ **Design patterns make extensive use of inheritance and delegation.**

- Facilitates reuse of successful software designs.

  ☞ Helps novices to learn by example to behave more like experts.

  ☞ A pattern catalog documents design patterns that are useful in a certain context.

Remove slide before posting!

# HOW TO BECOME A CHESS MASTER

- First learn rules and physical requirements
  - e.g., names of pieces, legal movements, chess board geometry and orientation, etc.
- Then learn principles
  - e.g., relative value of certain pieces, strategic value of center squares, power of a threat, etc.
- However, to become a chess master, one must study the games of other masters.
  - These games contain *patterns* that must be understood, memorized and applied repeatedly and correctly.

***There are hundreds of these patterns!***

# HOW TO BECOME A SOFTWARE DESIGN MASTER


- First learn the rules
  - e.g., the algorithms, data structures and languages of software.

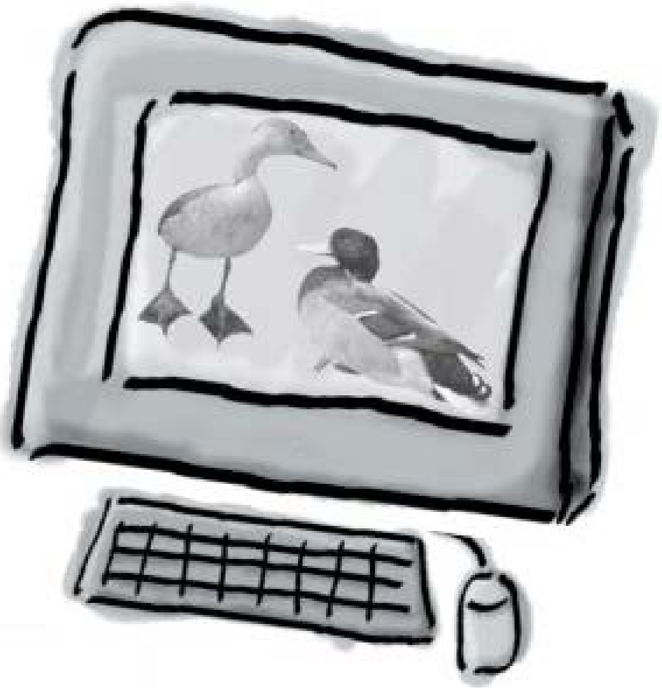
- Then learn the principles
  - e.g., structured programming, modular programming, object-oriented programming, generic programming, etc.

- However, to truly master software design, one must study the designs of other masters.
  - These designs contain *patterns* that must be understood, memorized and applied repeatedly and correctly.

***There are hundreds of these patterns!***

# INTRODUCTION TO DESIGN PATTERNS: SIMUDUCK

Joe works on SimUDuck.app



Meet Joe


... a successful duck pond simulation game.

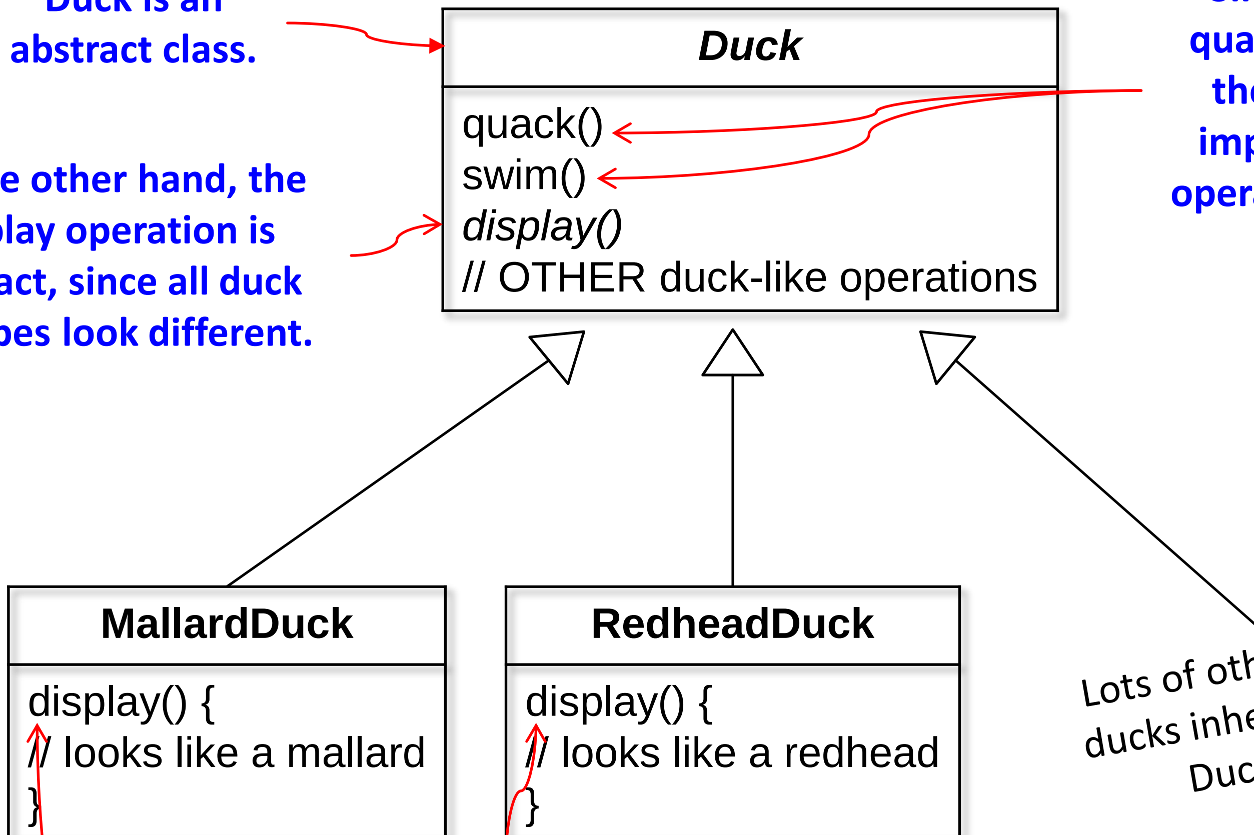
SimUDuck example from Tom Zimmermann, Microsoft Research

# SIMUDUCK IS OO

Duck is an abstract class.

Since all ducks quack and swim, the superclass implements the operation method.

On the other hand, the display operation is abstract, since all duck subtypes look different.




Lots of other types of ducks inherit from the Duck class.

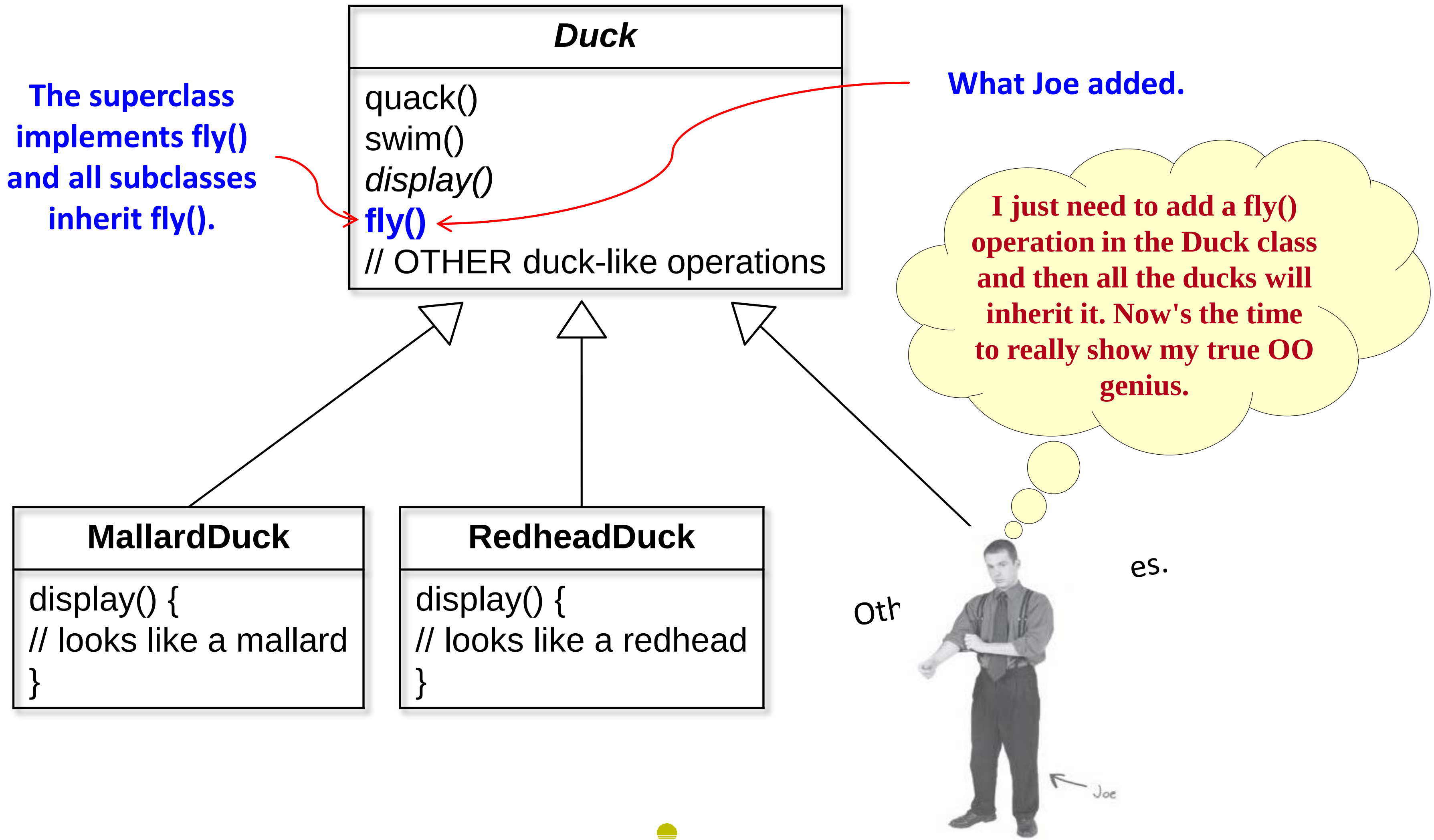
Each duck subtype is responsible for implementing its own display() operation for how it looks on the screen.

# SIMUDUCK

**Prepare for the big innovation …**


**now ducks need to fly.**

# SIMUDUCK: ADD FLY() OPERATION



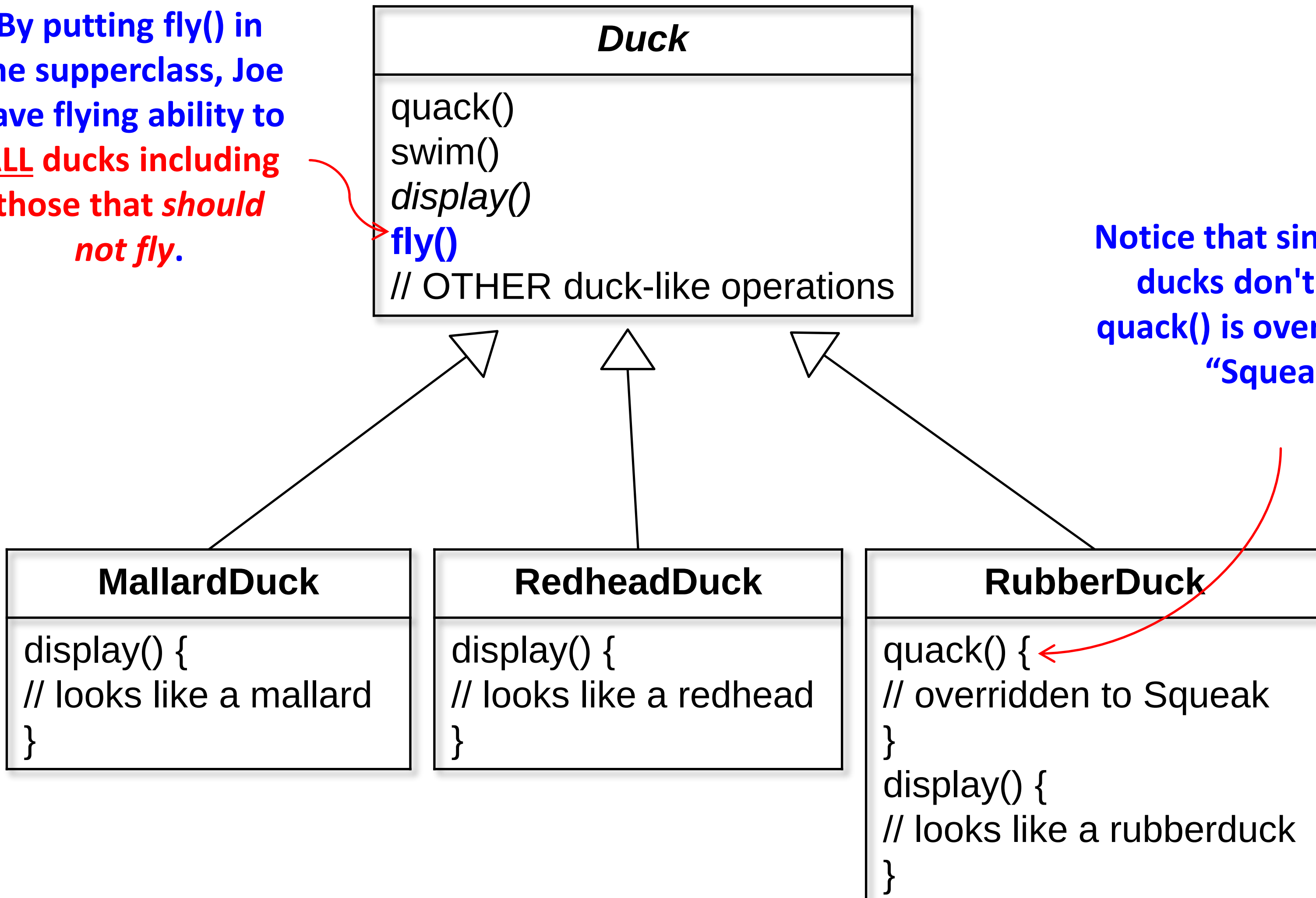
# SIMUDUCK: ADD FLY() OPERATION

But something went wrong …

flying rubber duckies!

# SIMUDUCK: WHAT HAPPENED?

By putting fly() in the supperclass, Joe gave flying ability to ALL ducks including those that *should not fly*.




Notice that since rubber ducks don't quack, quack() is overridden to "Squeak".

# SIMUDUCK: WHAT NOW?

I could always just **override** the fly() operation in rubber duck, the way I do with the quack() operation …

| RubberDuck |
|---|
| quack() {// override to Squeak}<br>display() {// rubberduck}<br>fly() {\\ override to do nothing} |

But then what happens when we add **wooden decoy ducks** to the program? They are not supposed to fly or quack …

Here's another class in the hierarchy. Notice that like RubberDuck, it does not fly, but it also does not quack.

| DecoyDuck |
|---|
| quack() {// override to do nothing}<br>display() {// decoy duck}<br>fly() {\\ override to do nothing} |

Remove slide before posting!

# EXERCISE 1: PUT ON YOUR THINKING CAP


**Which of the following are *disadvantages of using inheritance* to provide Duck behaviour?** **(Choose all that apply.)**

☐ A. Code is duplicated across subclasses.

☐ B. Runtime behaviour changes are difficult.

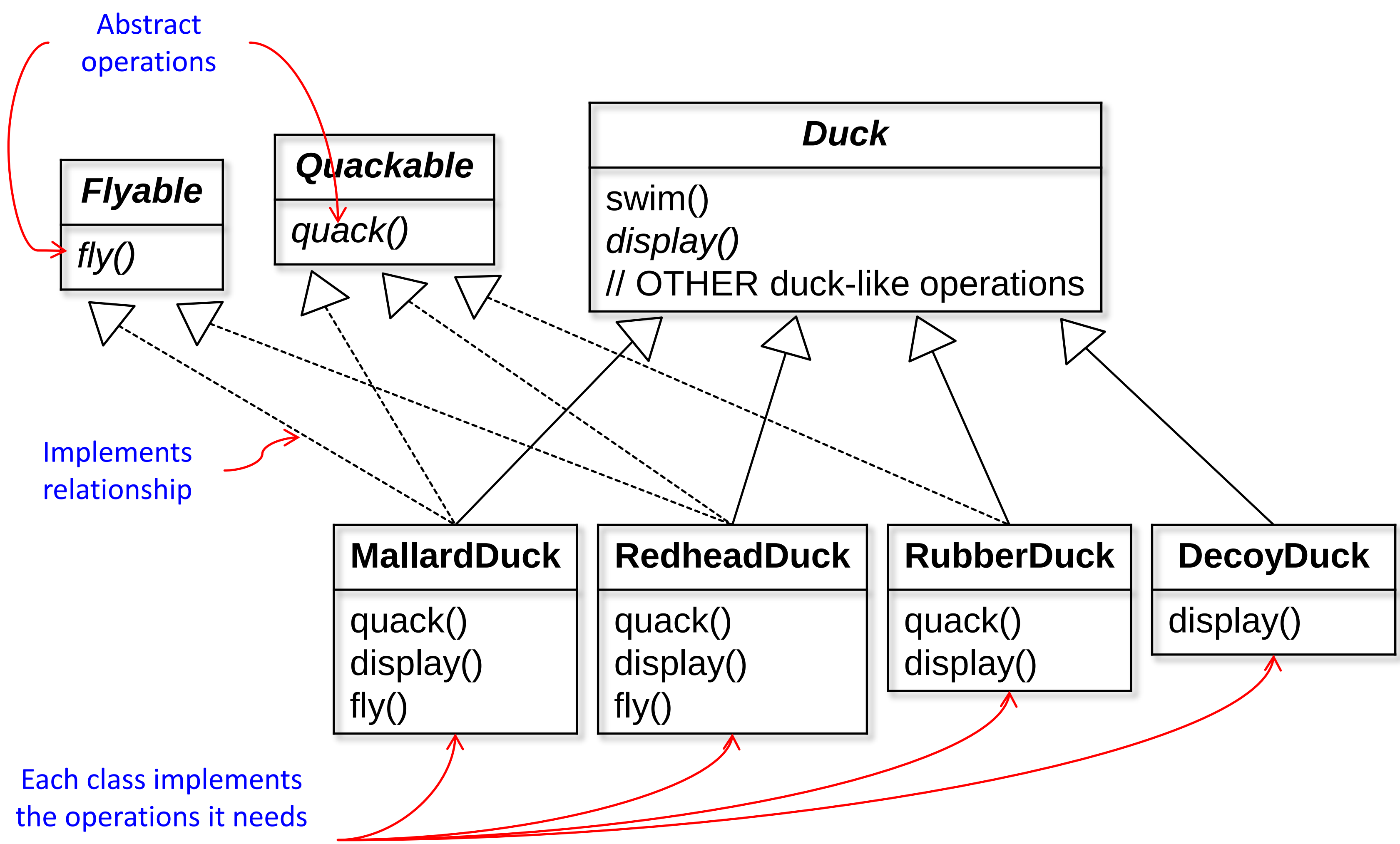
☐ C. We can't make ducks dance.

☐ D. Hard to gain knowledge of all duck behaviours.

☐ E. Ducks can't fly and quack at the same time.

☐ F. Changes can unintentionally affect other ducks.

# SIMUDUCK: HOW ABOUT USING AN INTERFACE?




**What do YOU think about this design?**

Remove slide before posting!

# SIMUDUCK: DUPLICATED CODE

**That is, like, the dumbest idea you have come up with. Can you say "duplicate code"? If you thought having to override a few operations was bad, how are you going to feel when you need to make a little change to the flying behaviour … *in all 48 of the flying Duck subclasses*?!**

**Design Principle**

***Identify the aspects of your application that vary and separate them from what stays the same.***

**Take what varies and "encapsulate" it so it won't affect the rest of your code.**

**The result?**

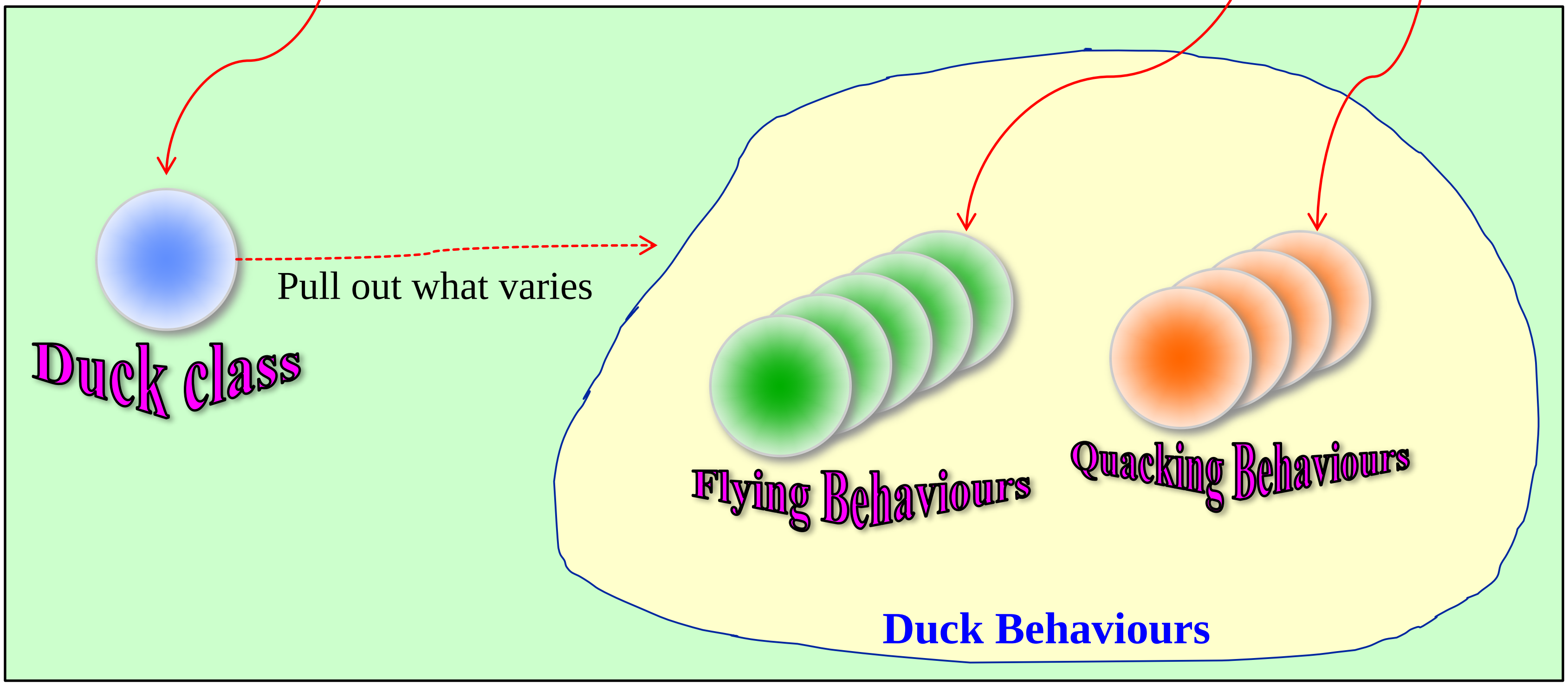
**Fewer unintended consequences from code changes and more flexibility in your systems.**

# SIMUDUCK:
# SEPARATING THE DUCK BEHAVIOURS

The Duck class is still the superclass of all ducks, but we pull out the fly and quack behaviours and put them into another class structure.

Now flying and quacking each get their own set of classes.

Various behaviour implementations are going to live here.

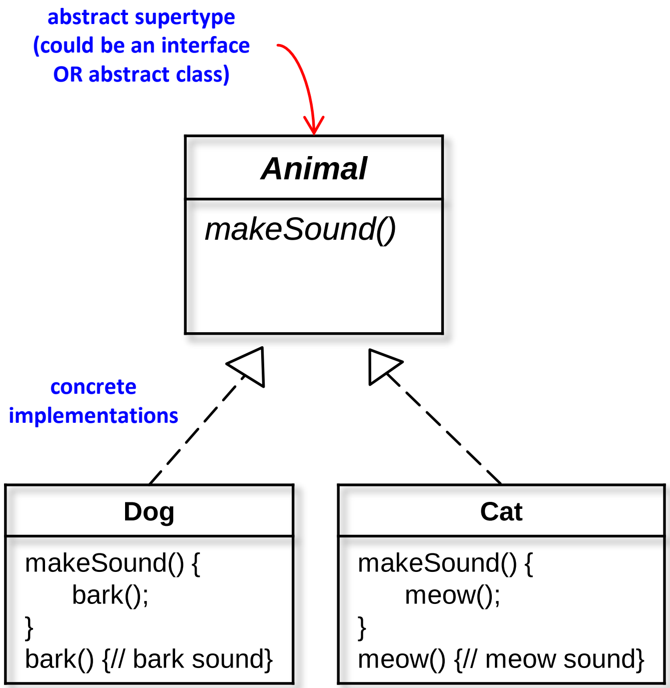


## Design Principle

***Program to an interface, not an implementation.***

**That way, the Duck classes won't need to know any of the implementation details of their behaviours.**

# PROGRAM TO AN INTERFACE/SUPERTYPE




**Programming to an implementation** would be:

```
Dog d = new Dog();
d.bark;
```

Declaring the variable “d” as type Dog (a concrete implementation of Animal) forces us to code to a concrete implementation.

But **programming to an interface/supertype** would be:

```
Animal a = new Dog();
a.makeSound();
```

We know it's a Dog, but we can now use the a reference polymorphically.

Even better, rather than hard-coding the instantiation of the subtype (like `new Dog()`) into the code, **assign the concrete implementation object at runtime**:

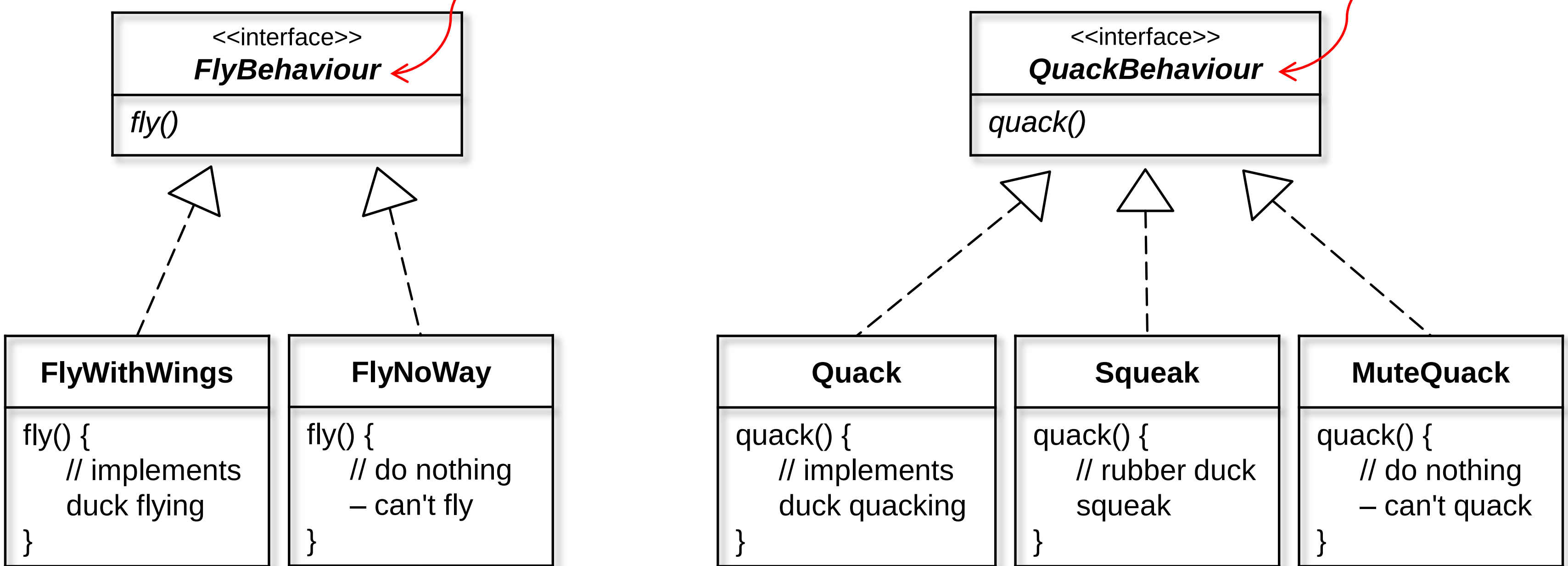
```
Animal a;
a = getAnimal();
a.makeSound();
```

We don't know WHAT the actual animal subtype is … all we care about is that it knows how to respond to makeSound().

# SIMUDUCK: USE INTERFACES FOR BEHAVIOURS

**FlyBehaviour is an interface that all flying classes implement. All new flying classes just need to implement the fly operation.**

**Same thing here for the quack behaviour; we have an interface that just includes a quack() operation that needs to be implemented.**




**Here's the implementation of flying for all ducks that have wings.**

**And here's the implementation for all ducks that can't fly.**

**Quacks that really quack.**

**Quacks that squeak**

**Quacks that make no sound at all.**

# SIMUDUCK: OUR DESIGN ROCKS!

- We can reuse the fly and quack behaviours.
  - They are not hidden inside Duck anymore.

- We can add new behaviours.
  - Without modifying any existing behaviour classes or Duck classes that use behaviours.

- SimUDuck.app now has the **BENEFIT OF REUSE** while still being **MAINTAINABLE**!

# SIMUDUCK: PUT ON YOUR THINKING CAP


1. Using our new design, what would you do if you needed to add rocket-powered flying to the SimUDuck app?

2. Can you think of a class that might want to use the Quack behaviour that is not a duck?

# SIMUDUCK: INTEGRATING THE DUCK BEHAVIOUR

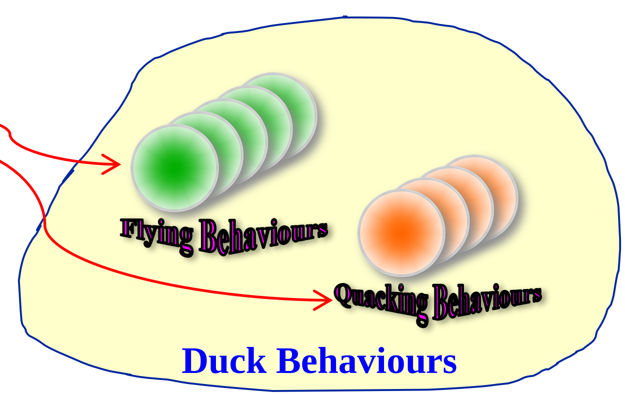
① **First we will add two instance variables to the Duck class.**

The instance variables hold a reference to a specific behaviour at runtime.

The behaviour variables are declared as the behaviour INTERFACE type.

| *Duck* |
|---|
| **FlyBehaviour flyBehaviour**<br>**QuackBehaviour quackBehaviour** |
| **performQuack()**<br>swim()<br>*display()*<br>**performFly()**<br>// OTHER duck-like operations |

These operations replace fly() and quack().



# SIMUDUCK: INTEGRATING THE DUCK BEHAVIOUR

## ② Now we implement performQuack() and performFly().

```
public class Duck {
   FlyBehaviour flyBehaviour;
   QuackBehaviour quackBehaviour;
   // more

   public void performFly() {
     flyBehaviour.fly();
   }

   public void performQuack() {
     quackBehaviour.quack();
   }
}
```

**Each Duck has a reference to something that implements the FlyBehaviour and QuackBehaviour interfaces.**

**Rather than handling the quack and fly behaviour itself, the Duck object delegates that behaviour to the objects referenced by quackBehaviour and flyBehaviour.**

# SIMUDUCK: INTEGRATING THE DUCK BEHAVIOUR

```
public class MallardDuck extends Duck {
    public MallardDuck() {
        quackBehaviour = new Quack();
        flyBehaviour = new FlyWithWings();
    }
```

A MallardDuck uses the Quack class to handle its quack, so when performQuack is called, the responsibility for the quack is delegated to the Quack object and we get a real quack.

Remember, MallardDuck inherits the quackBehaviour and flyBehaviour instance variables from class Duck.

And it uses FlyWithWings as its FlyBehaviour type.

**How about a rubber duck?**

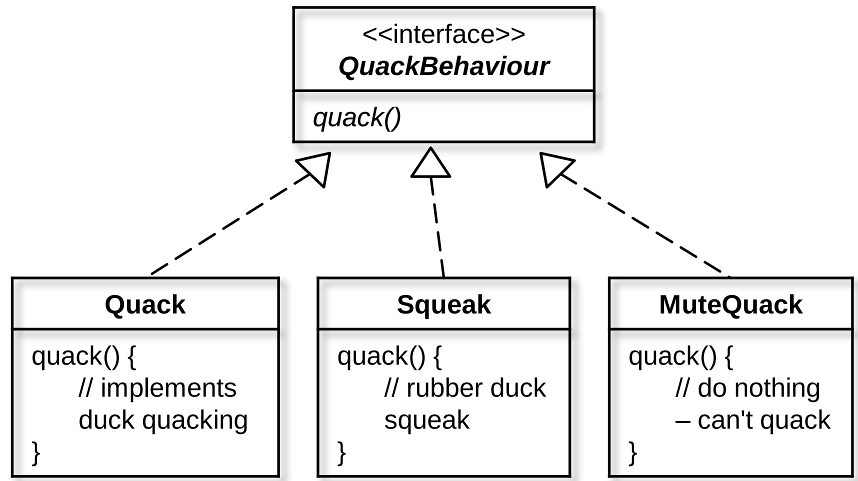
```
public class RubberDuck extends Duck {
    public RubberDuck() {
        quackBehaviour = new
        flyBehaviour = new
    }
```

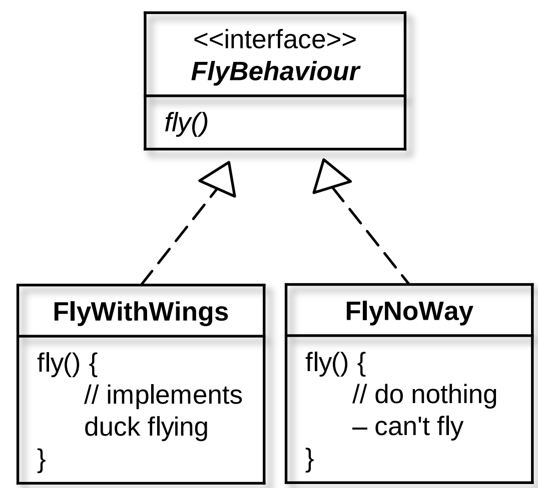
**Initialize the behaviours in subclasses.**

Remove slide before posting!

# SIMUDUCK: INTEGRATING THE DUCK BEHAVIOUR

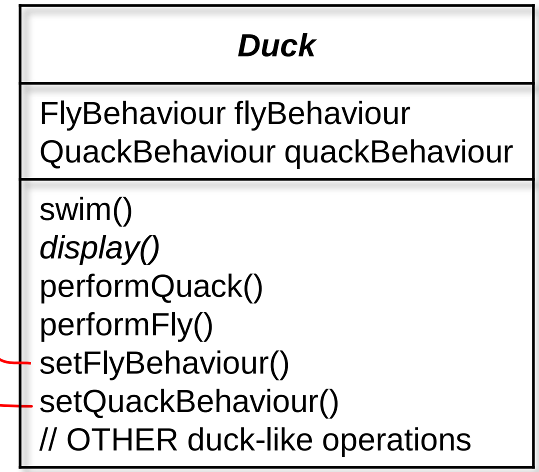
```
public class Duck {
    FlyBehaviour flyBehaviour;
    QuackBehaviour quackBehaviour;
    // more

    public void performFly() {
      flyBehaviour.fly();
    }

    public void performQuack() {
      quackBehaviour.quack();
    }
}

public class MallardDuck extends Duck {
    public MallardDuck() {
      quackBehaviour = new Quack();
      flyBehaviour = new FlyWithWings();
      // more
    }
}
```



# SIMUDUCK: SETTING BEHAVIOUR DYNAMICALLY

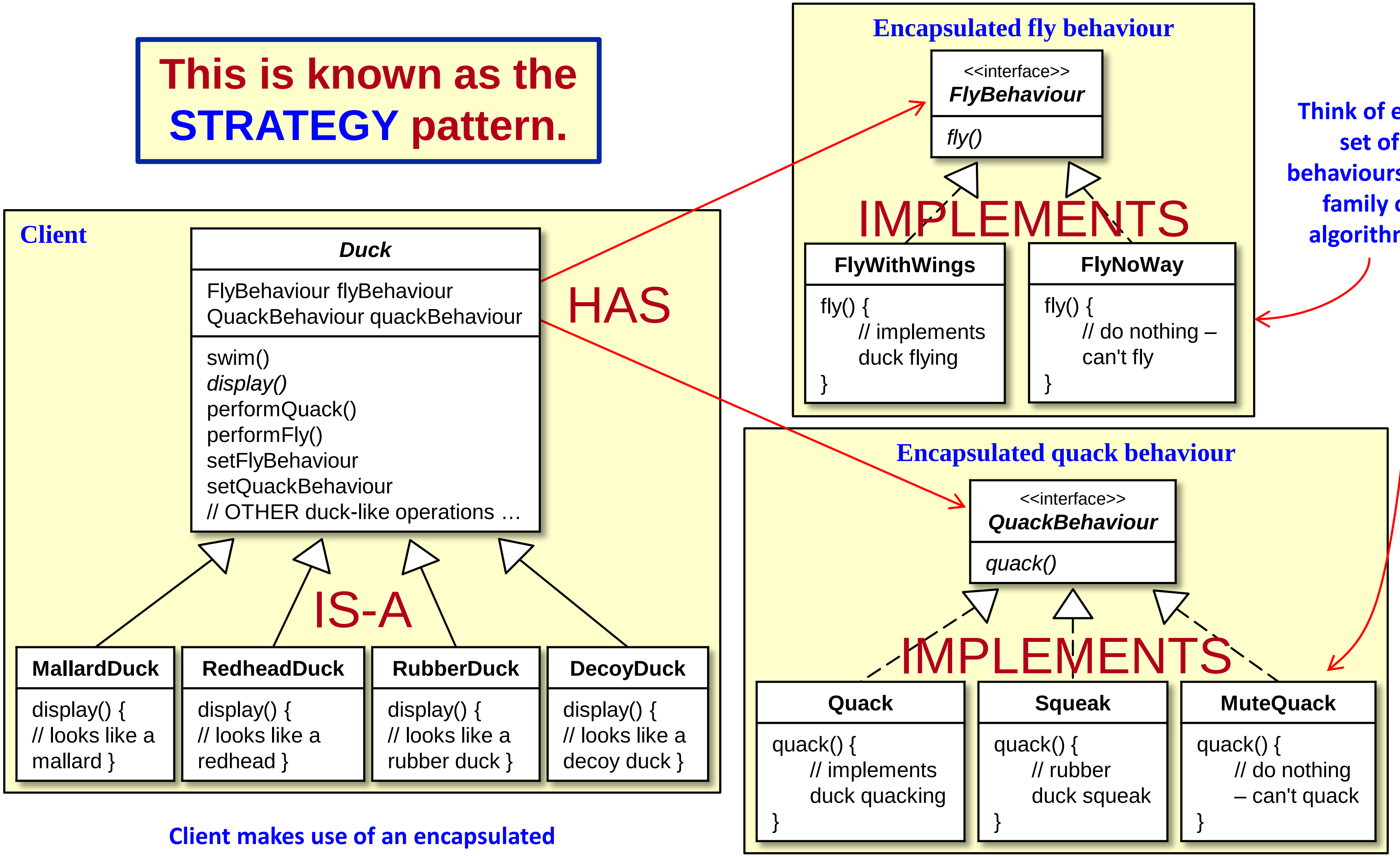
```
public void setFlyBehaviour(FlyBehaviour fb) {
    flyBehaviour = fb;
}


public void setQuackBehaviour(QuackBehaviour qb) {
    quackBehaviour = qb;
}
```




**These operations can be called anytime to set the behaviour of a duck on-the-fly.**

# SIMUDUCK: THE BIG PICTURE

**This is known as the STRATEGY pattern.**




**Think of each set of behaviours as a family of algorithms.**

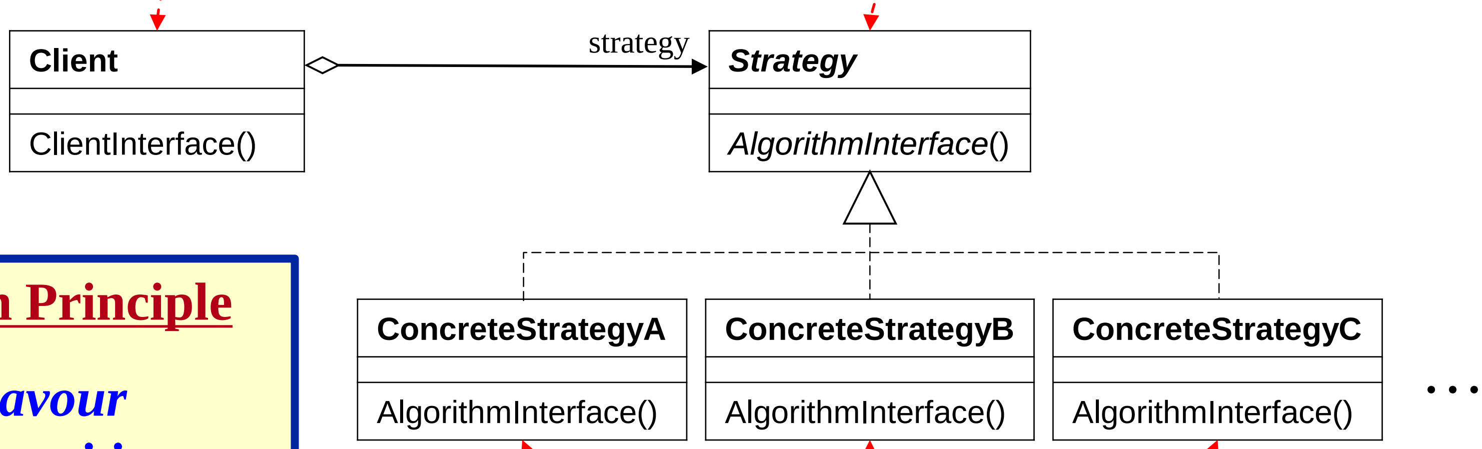
**Client makes use of an encapsulated family of algorithms for both flying and quacking.**

**These ~~behaviours~~ "algorithms" are interchangeable.**

# STRATEGY PATTERN: CLASS DIAGRAM

The *Client* class is configured with a *ConcreteStrategy* object, maintains a reference to a *Strategy* object and may define an interface that lets *Strategy* access its data.

The *Strategy* class declares an interface common to all supported algorithms. *Client* uses this interface to call the algorithm defined by a *ConcreteStrategy*.



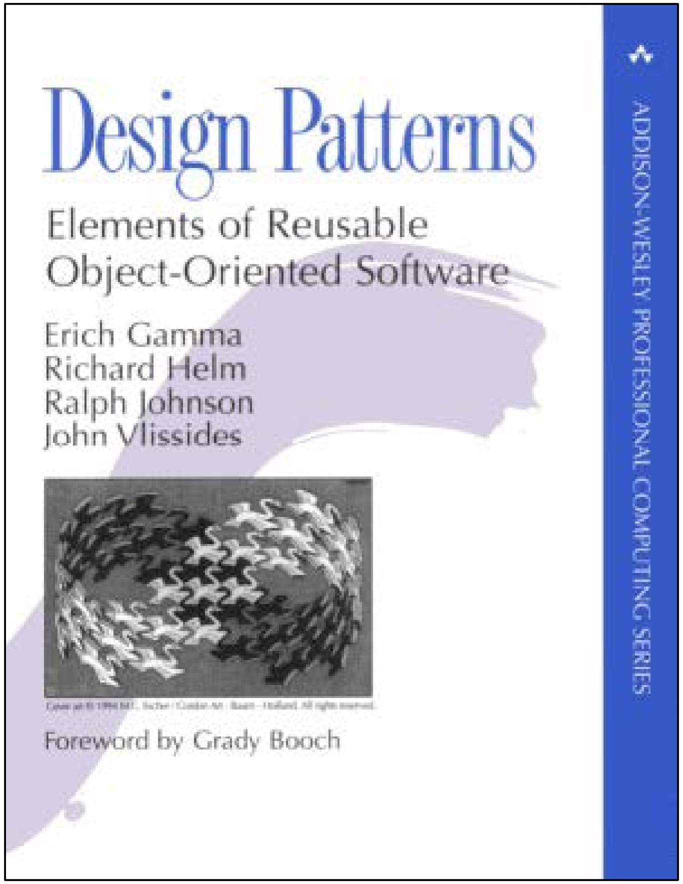

Each *ConcreteStrategy* implements the algorithm using the *Strategy* interface.

**Design Principle**

***Favour composition (HAS-A) over inheritance (IS-A).***

# DESIGN PATTERNS SPACE

- **Creational patterns**
  - Deal with initializing and configuring classes and objects
- **Structural patterns**
  - Deal with decoupling interface and implementation of classes and objects
- **Behavioural patterns**
  - Deal with dynamic interactions among societies of classes and objects
- **Concurrency patterns**
  - Deal with multi-threaded programming



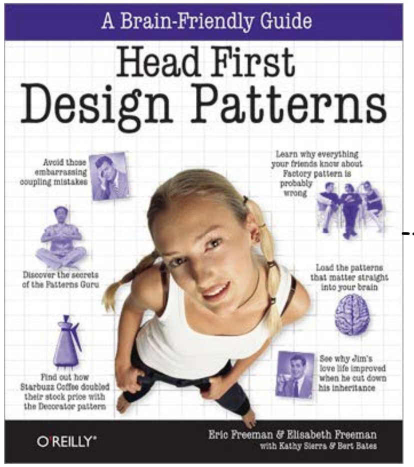
# DESIGN PATTERNS: DESCRIPTION

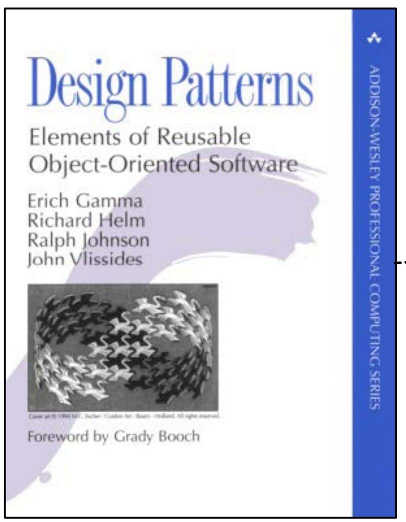
Design patterns are described using the following main parts:

1. Name and intent (what does the pattern do).
2. Problem (the difficulty being addressed).
3. Context (the general situation in which the pattern applies).
4. Concern(s) (issues that need to be considered when solving the problem).
5. Solution (an abstract description of the structure and collaborations in the solution).
6. Positive and negative consequence(s) of use.
7. Implementation guidelines and sample code.
8. Known uses and related patterns.

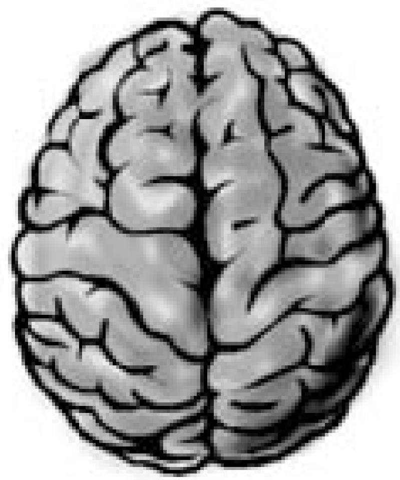
**Design pattern descriptions are usually independent of programming language or implementation details.**

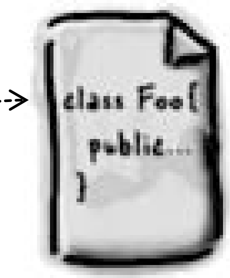
# DESIGN PATTERNS: HOW TO USE




Your BRAIN






Your code, now
new and improved
with design patterns!

# DESIGN PATTERNS SPACE

Behavioural patterns are concerned with how classes and objects interact and distribute responsibility.

Creational patterns involve object instantiation and all provide a way to decouple a client from the objects it needs to instantiate.

## Creational

- Singleton
- Builder
- Prototype
- Abstract Factory
- Factory

## Behavioural

- Mediator
- Command
- Template Method
- Memento
- Observer
- Visitor
- Iterator
- Interpreter
- Chain of responsibility
- State
- Strategy

## Concurrency

- Active object
- Balking
- Read write lock
- Monitor object
- Reactor
- Thread pool
- Scheduler

## Structural

- Proxy
- Decorator
- Façade
- Bridge
- Composite
- Flyweight
- Adapter

Concurrency patterns manage concurrent interactions among multiple objects.

Structural patterns compose classes or objects into larger structures.

# OBSERVER PATTERN: EXAMPLE

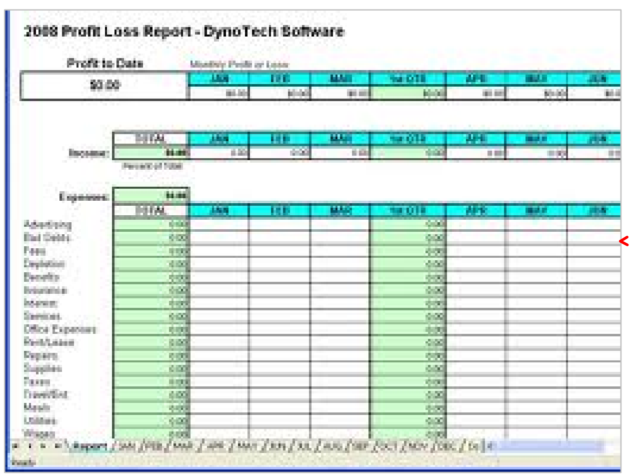
Consider a spreadsheet:

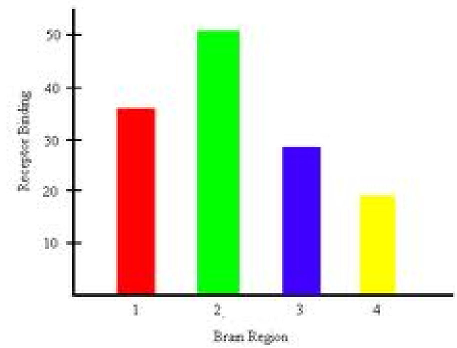
- You enter data into a cell.
- As as you do so, any related formulas and charts change.
- New formulas/charts can be created and are updated as the data changes.

SUBJECT OBSERVER

~~Data~~ + ~~Charts~~

= Observer Pattern




# OBSERVER PATTERN

**Problem:** A large monolithic design does not scale well as new requirements arise.

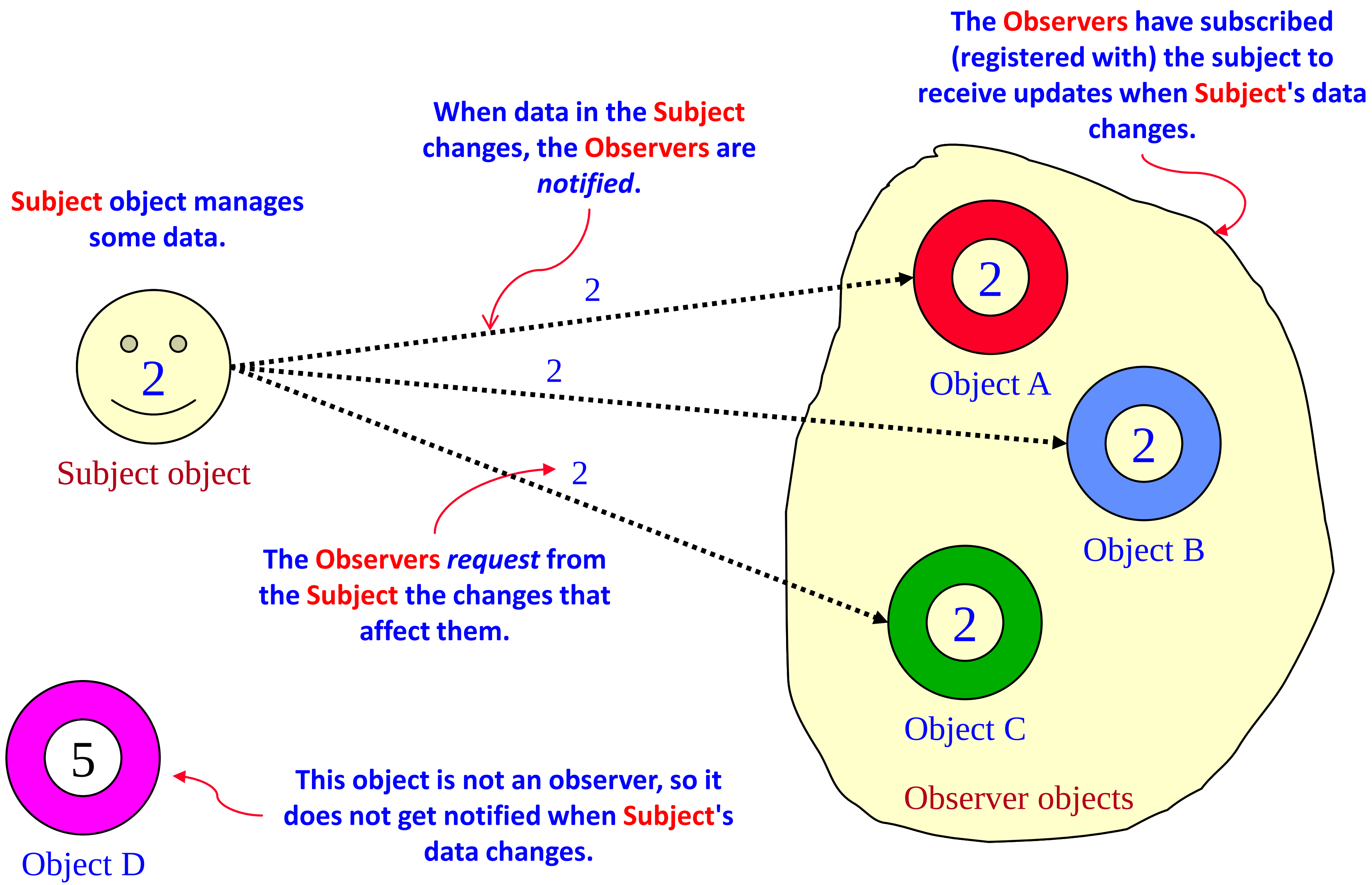
**Solution:** Define an object that is the "keeper" of the data model and/or business logic (the Subject).

Delegate all "view" functionality to decoupled and distinct Observer objects.

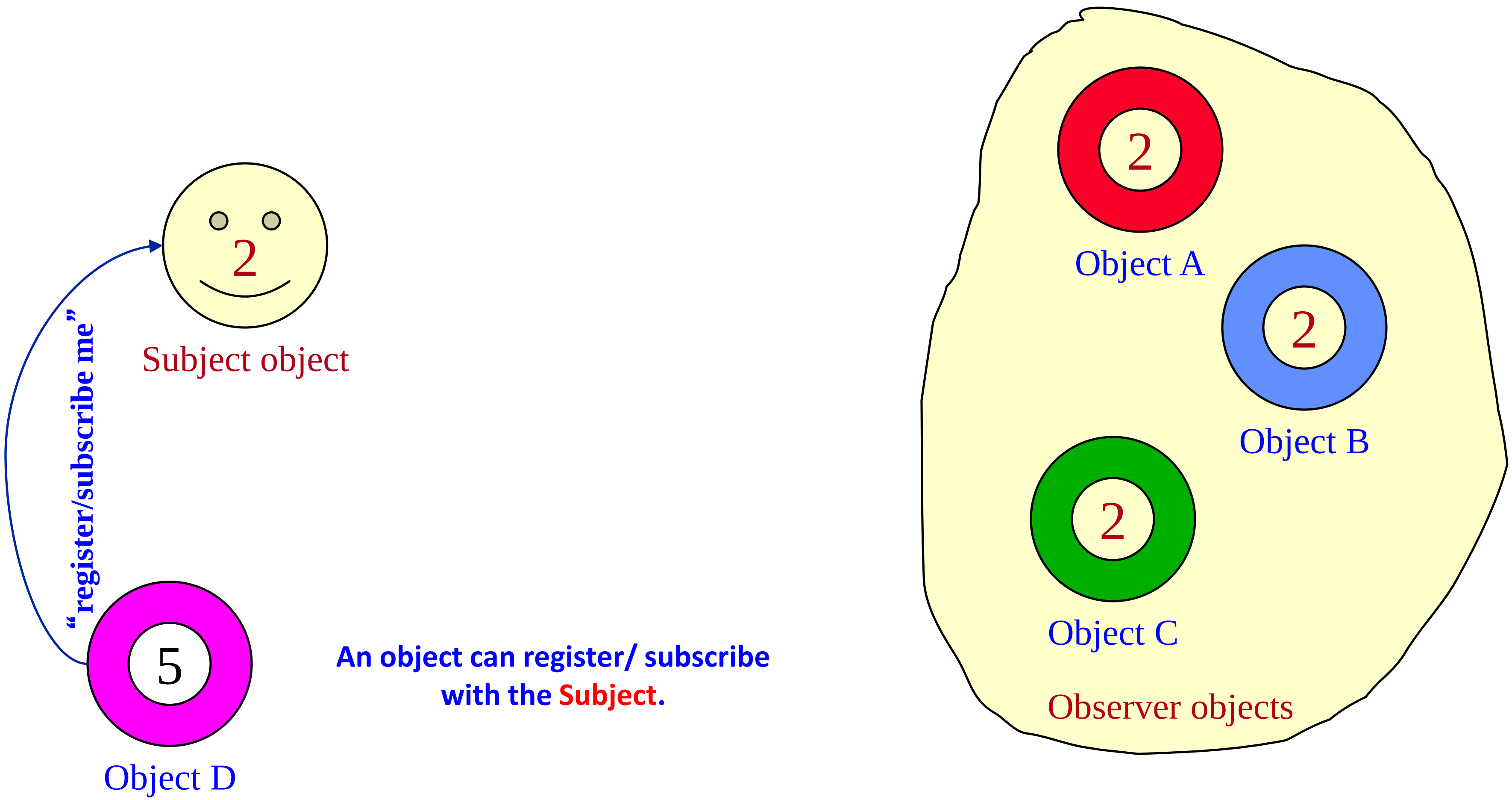
Observers register themselves with the Subject as they are created.

Whenever the Subject changes, it broadcasts to all registered Observers that it has changed, and each Observer queries the Subject for that subset of the Subject's state that it is responsible for monitoring.

# OBSERVER PATTERN: HOW IT WORKS

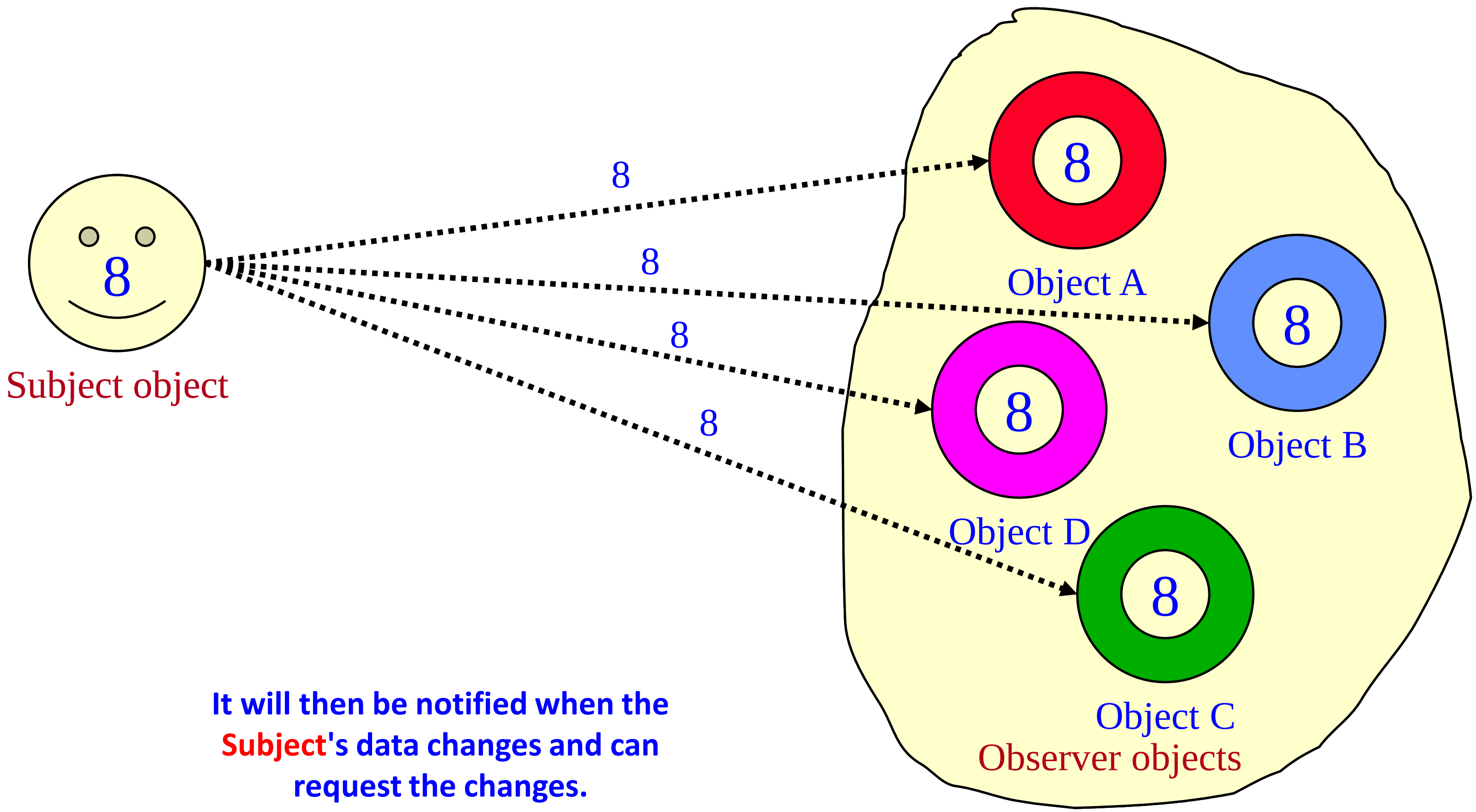


# OBSERVER PATTERN: HOW IT WORKS



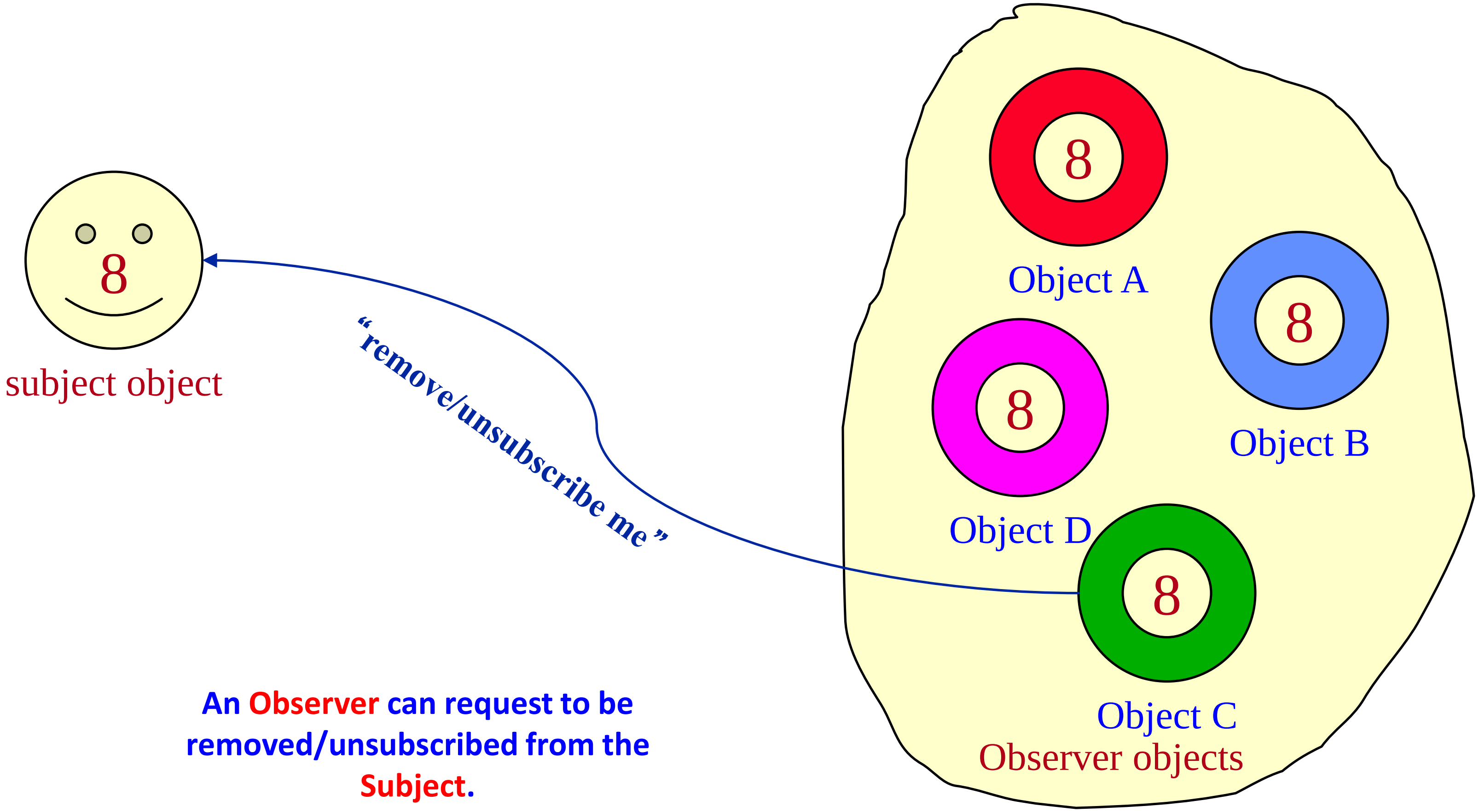

**An object can register/ subscribe with the Subject.**

# OBSERVER PATTERN: HOW IT WORKS



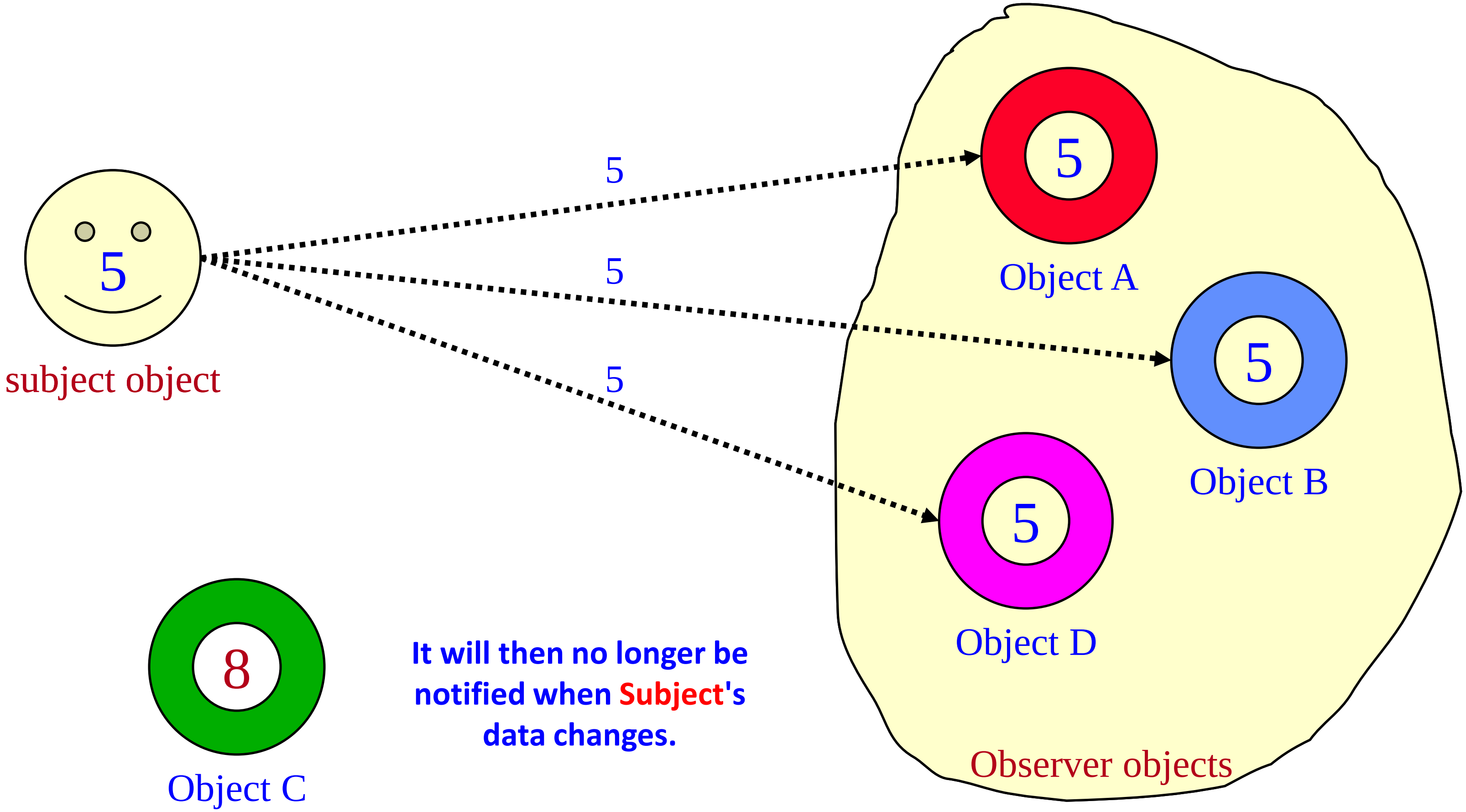

**It will then be notified when the Subject's data changes and can request the changes.**

# OBSERVER PATTERN: HOW IT WORKS




**An Observer can request to be removed/unsubscribed from the Subject.**

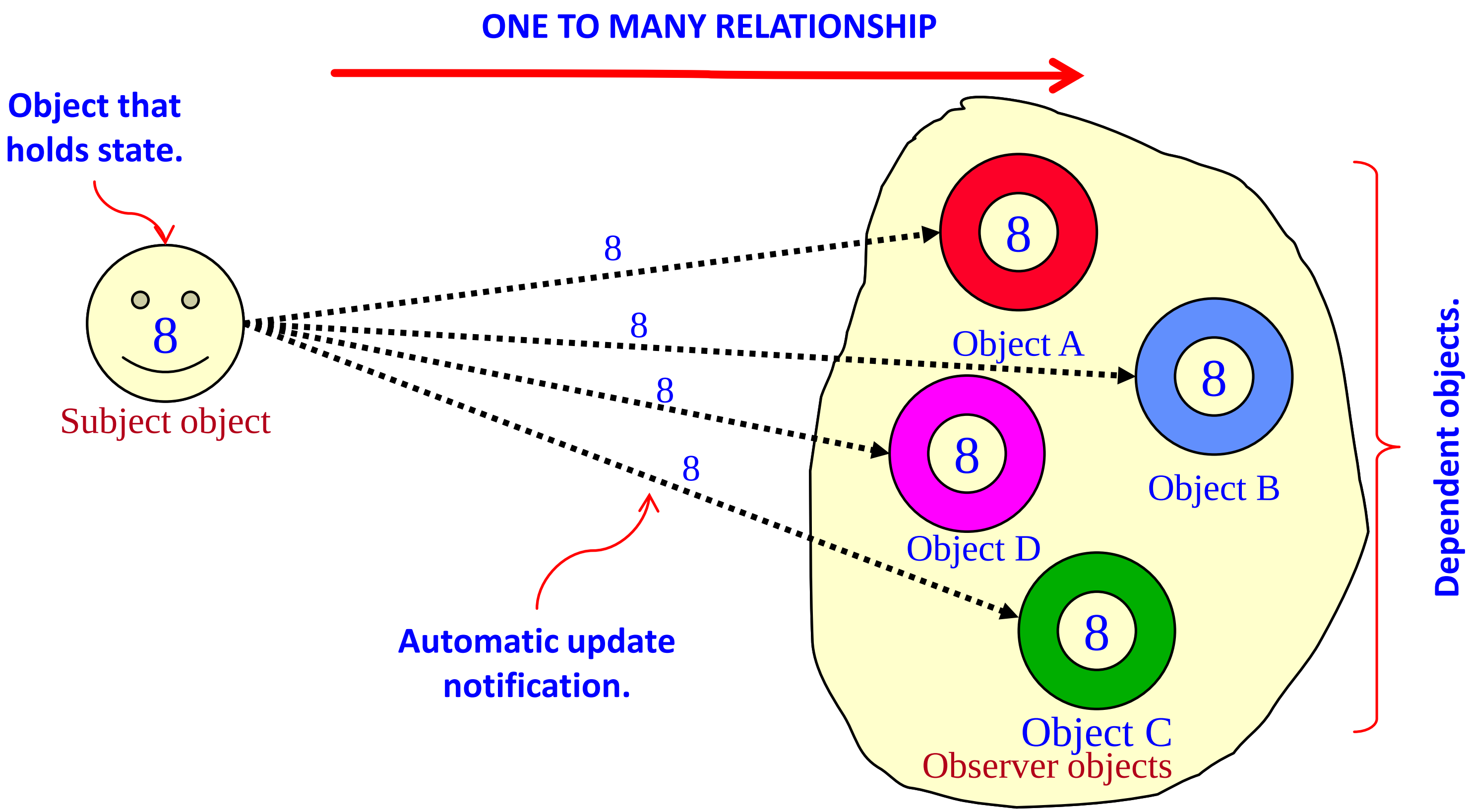
# OBSERVER PATTERN: HOW IT WORKS



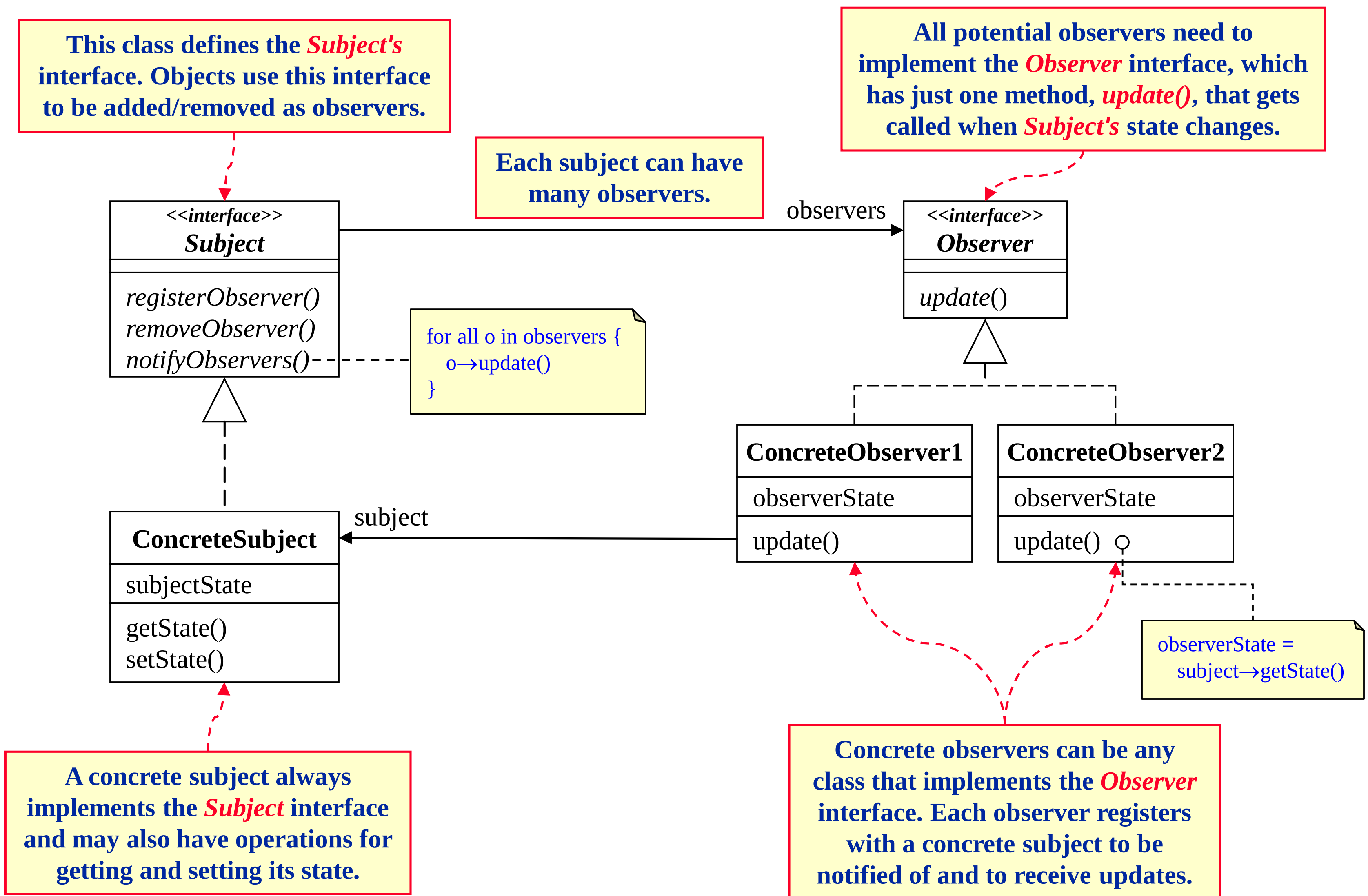

**It will then no longer be notified when Subject's data changes.**

# OBSERVER PATTERN: DEFINITION

The **OBSERVER** pattern defines a **one-to-many dependency** between objects so that when one object changes state, all of its dependents are notified and updated automatically.



# OBSERVER PATTERN: CLASS DIAGRAM



# OBSERVER PATTERN:
# THE POWER OF LOOSE COUPLING

- When two objects are loosely coupled, they can interact, but have very little knowledge of each other.

- The **Observer Pattern** provides an object design where subjects and observers are loosely coupled.

**WHY?**

**DESIGN PRINCIPLE**

***Strive for loosely coupled designs between objects that interact.***

**Loosely coupled designs minimize the interdependency between objects.**

# OBSERVER PATTERN: OBSERVATIONS AND ISSUES

- **Support for broadcast communication is needed so that**
  - No need for ConcreteSubject to communicate directly with every observing object.
  - Observing objects can more easily change at runtime.

- **Event-notification protocol is needed**
  - What events should the subject announce?
  - Should every event be announced to every observer?
  - Should the subject define different kinds of events and allow the observers to subscribe selectively?

# EXERCISE 2: DESIGN PRINCIPLE CHALLENGE

**How does the observer pattern use the following principles?**

**Design Principle**

***Identify the aspects of your application that vary and separate them from what stays the same.***

**Design Principle**

***Program to an interface, not an implementation.***

**Design Principle**

***Favour composition over inheritance.***