Computer Science 2212b

Topic 9 - Object-Oriented Design Principles

Jeff Shantz
Department of Computer Science

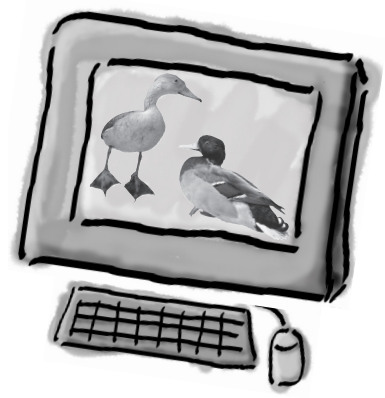
Based on the excellent books, *Head First Design Patterns* by Freeman and Freeman; and *Head First Object-Oriented Analysis and Design* by McLaughlin, Pollice, and West. Read these books.

Encapsulate What Varies

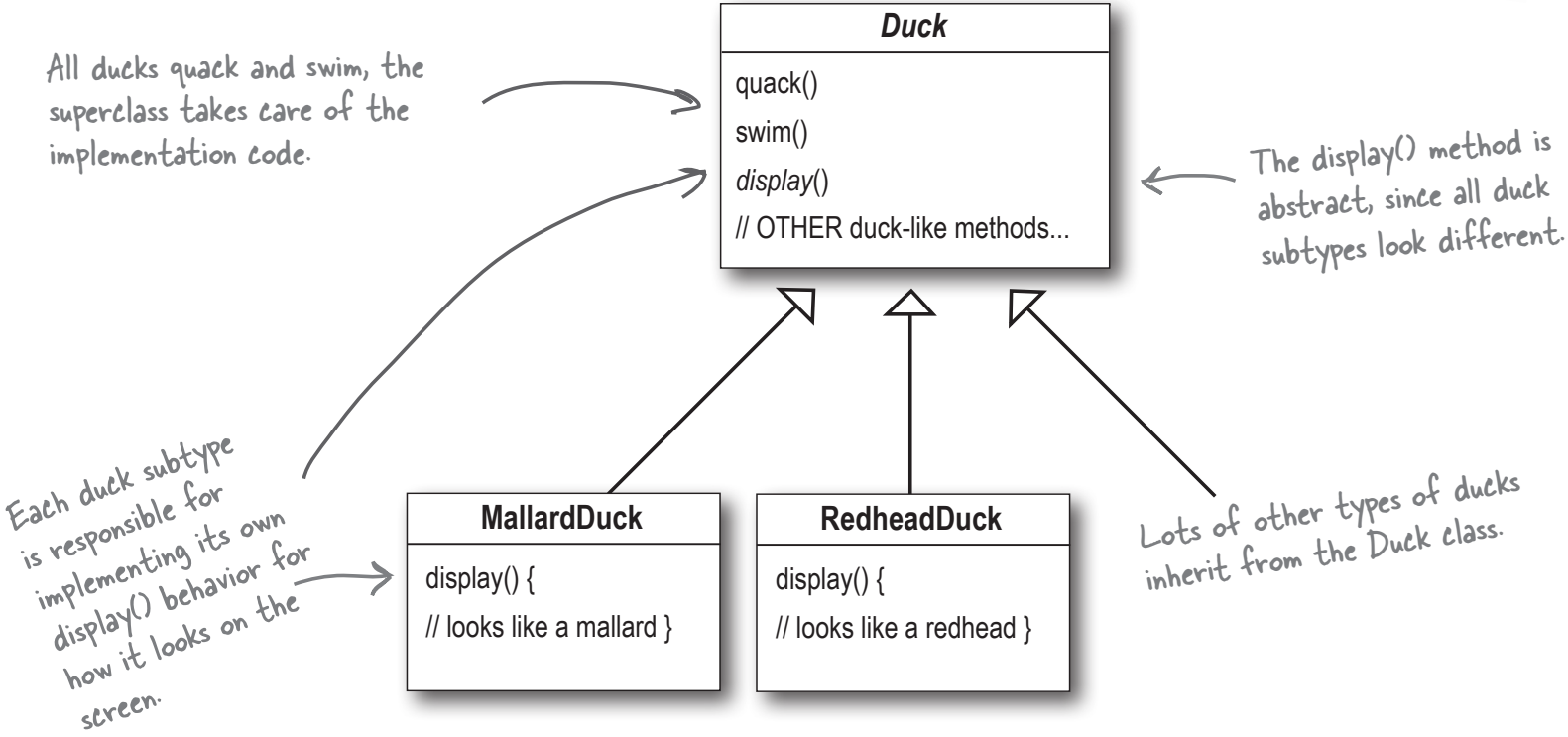
Design Principle 1:
Encapsulate what varies

Take the parts that vary and encapsulate them, so that later you can alter or extend the parts that vary without affecting those that don't.

Scenario: Duck Hunt Simulation Game

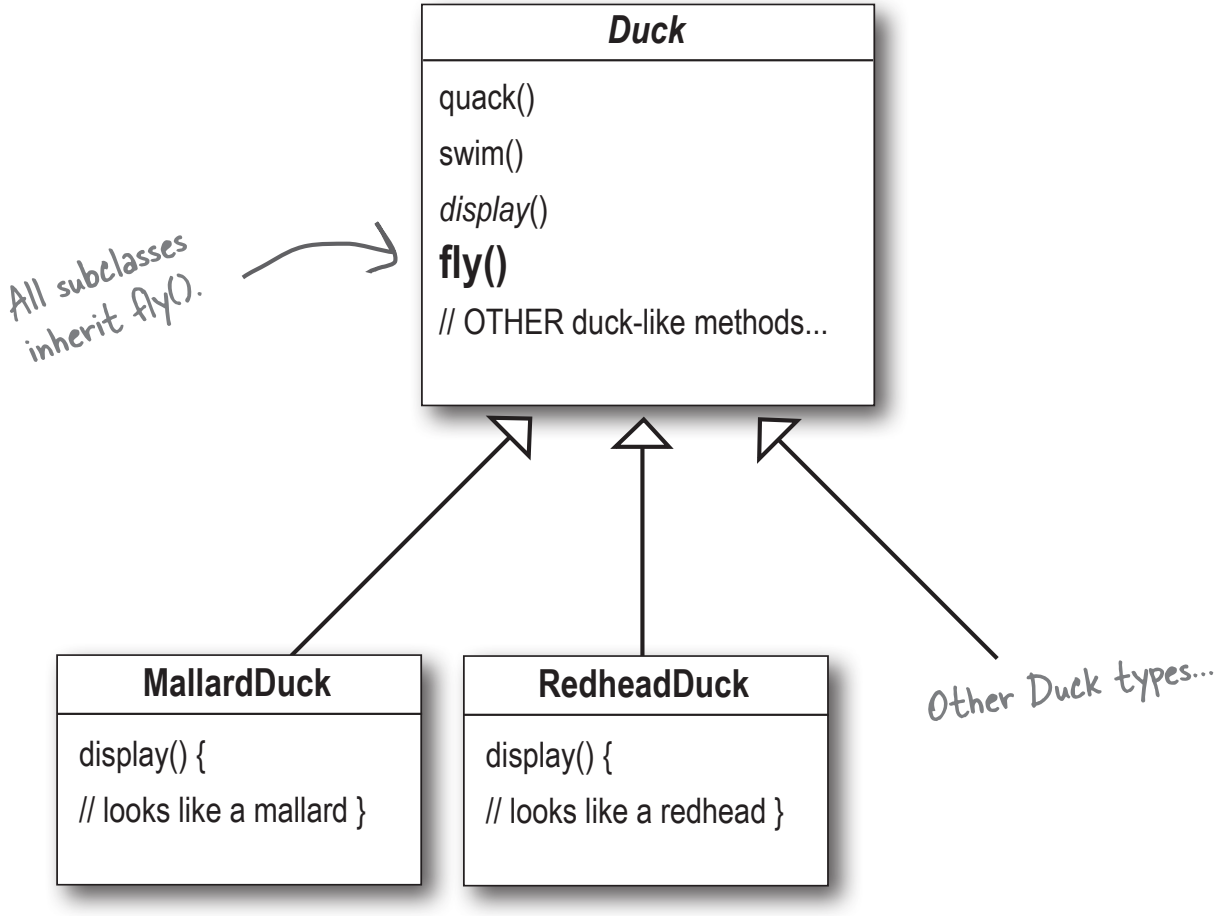


- Company building a duck hunt simulation game



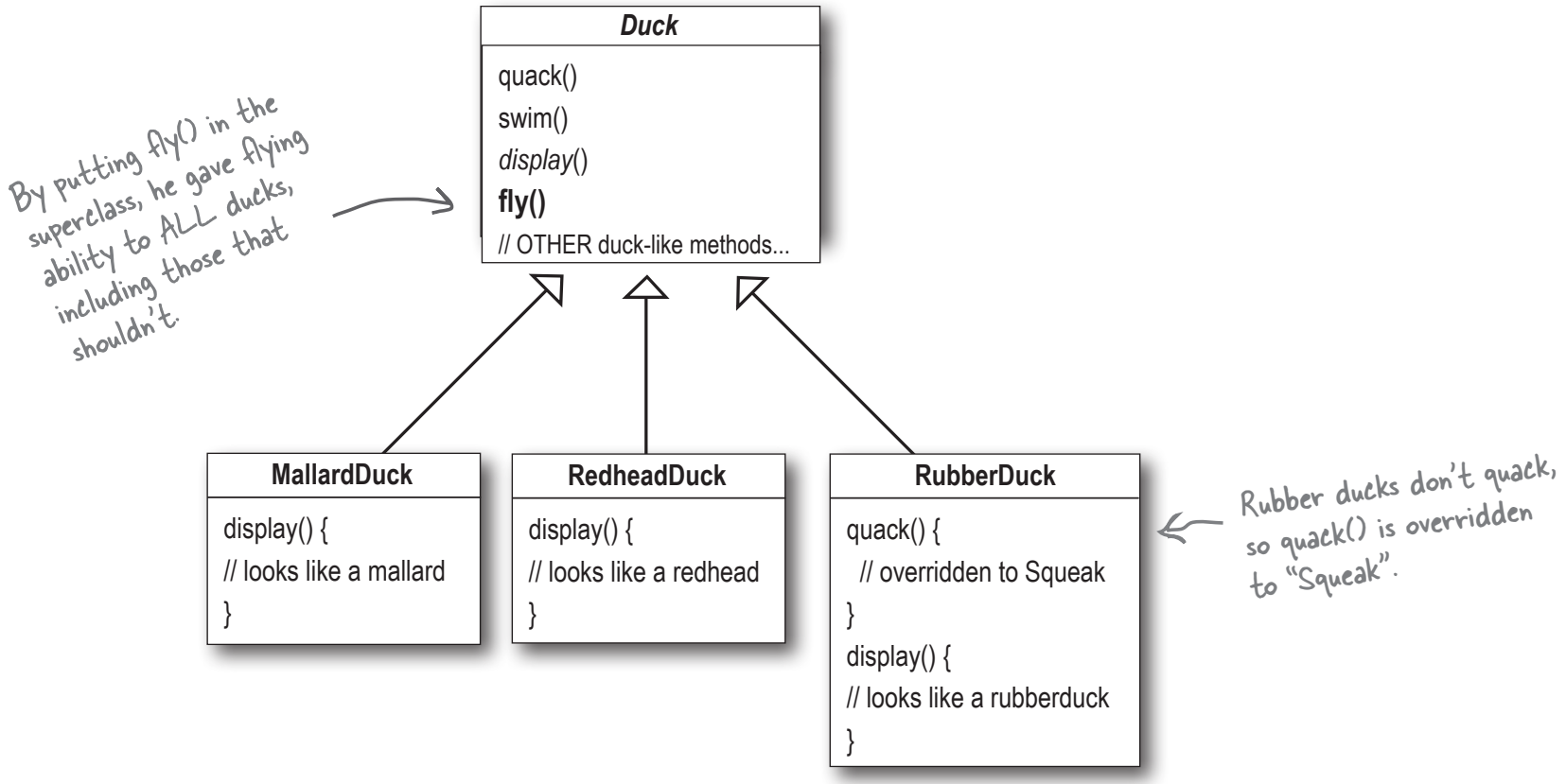
Making Ducks Fly

- Need to add code to allow ducks to fly in the simulation



Making Ducks Fly: Problem

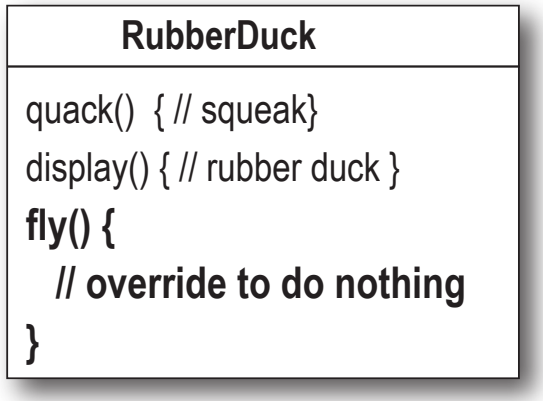
- The company decides to add a `RubberDuck` class to the game as an Easter egg



- Should all ducks be able to fly? Quack?

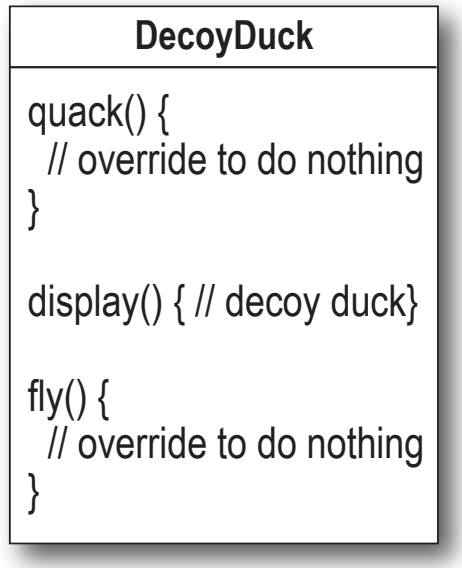
Making Ducks Fly: Solution

- Solutions?
  - Could override the `fly` method in `RubberDuck` to do nothing



Making Ducks Fly: Solution

- Not a *horrible* solution, but what about when we add a `DecoyDuck` class for hunters in the game?
  - A decoy should neither quack nor fly

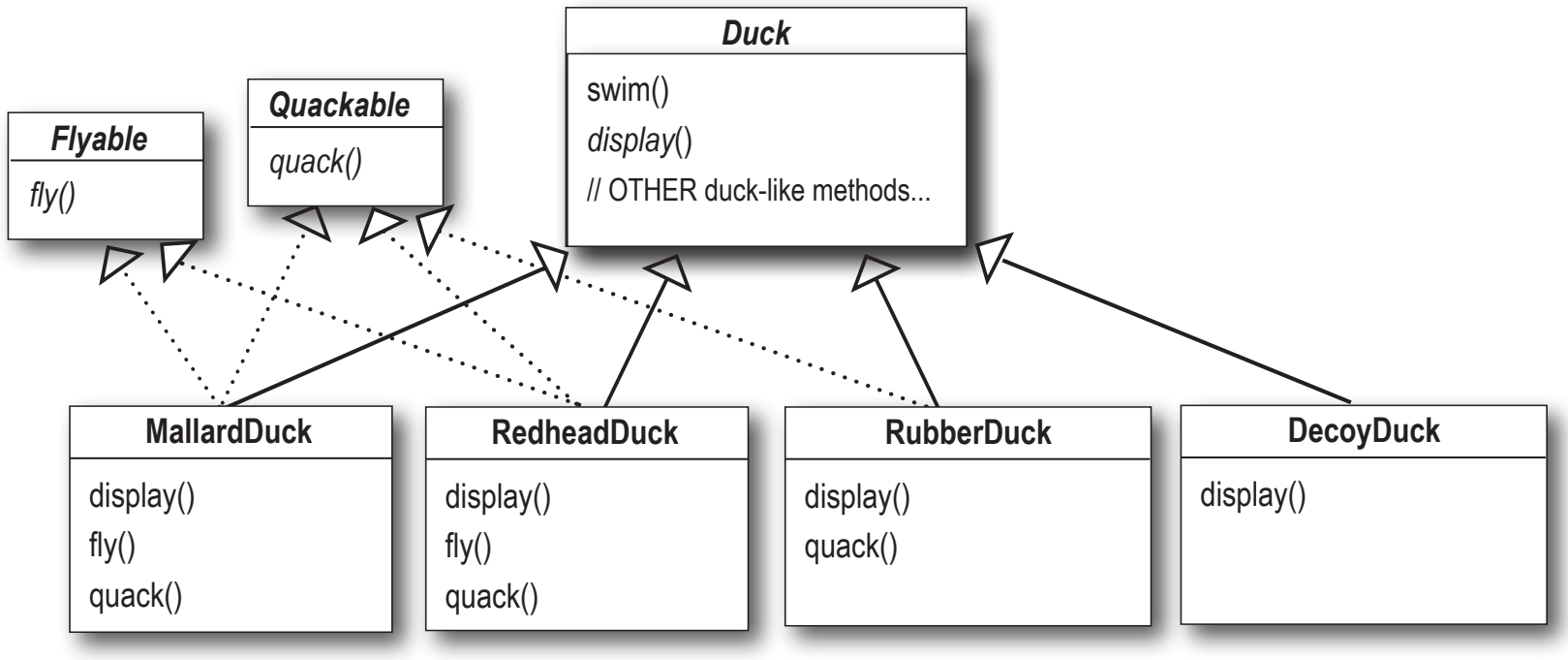


Making Ducks Fly: Solution

- Inheritance probably isn't the answer in this case
  - As we add new types of ducks, we will have to examine and possibly override `fly` and `quack` for each new class
  - Will likely have a lot of duplicate code in the subclasses
    - How many different ways can a duck really fly?

Making Ducks Fly: Solution

- Another option: use an abstract class (or, in Java, an `interface`)

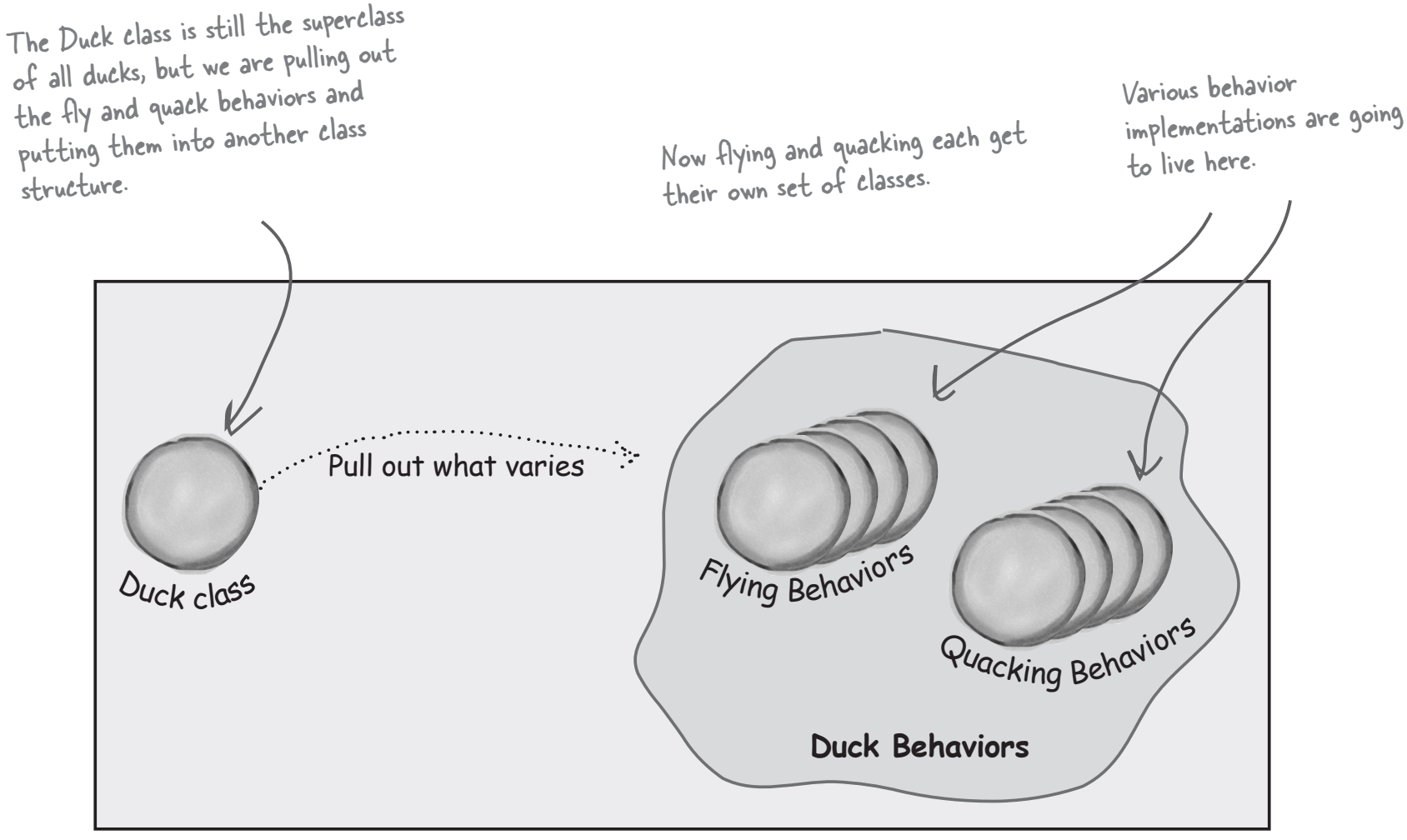


- Has this solved the problem?

Making Ducks Fly: Solution

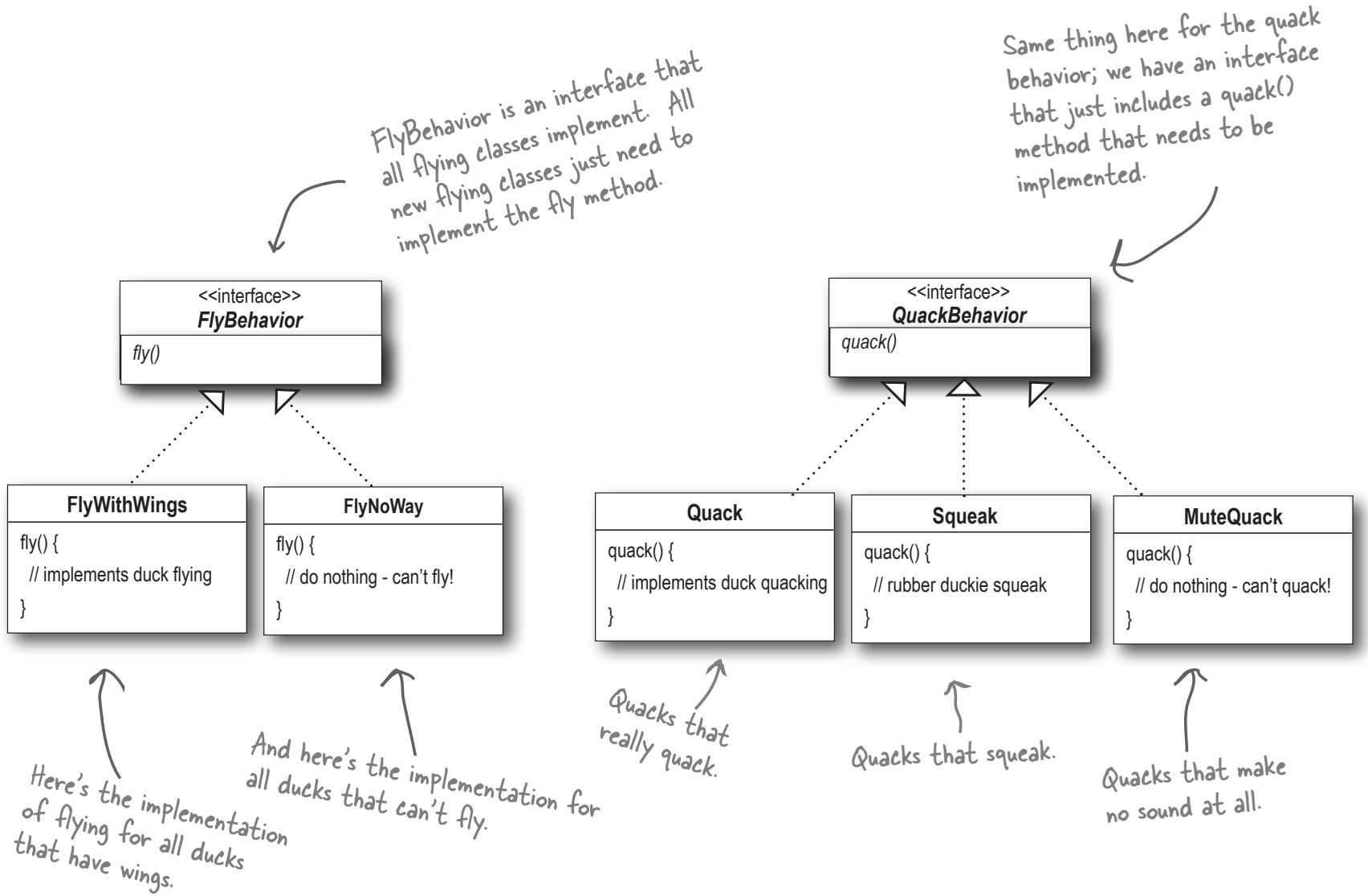
- We know:
  - New ducks will be added to the system as time passes
  - Duck behaviours differ from duck type to duck type
  - Certain behaviours are not appropriate for all ducks

Encapsulate What Varies



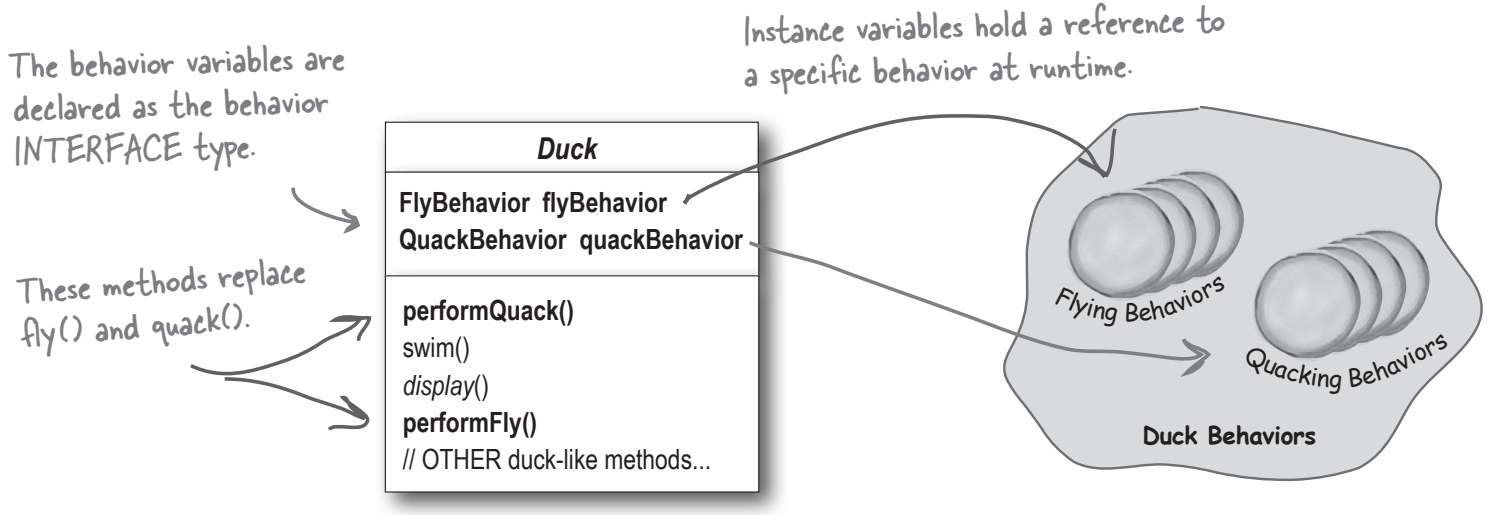
Encapsulate What Varies

- We will create an interface/abstract class for each behaviour: flying and quacking



Encapsulate What Varies

- Now the functionality that might change between subclasses has been encapsulated in its own set of classes
- We will store the flying and quacking behaviours of a duck as instance variables in the `Duck` class
- A duck will *delegate* its flying and quacking behaviours, rather than implementing them itself



Encapsulate What Varies

```
public interface FlyBehaviour {
  public void fly();
}

public class FlyWithWings implements FlyBehaviour {

  public void fly() {
    System.out.println("Flying with my wings!");
  }

}

public class FlyNoWay implements FlyBehaviour {

  public void fly() {
    System.out.println("Can't actually fly!");
  }

}
```

Encapsulate What Varies

```
public class Duck {

  protected FlyBehaviour flyBehaviour;
  protected QuackBehaviour quackBehaviour;

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

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

}
```

Encapsulate What Varies

```
public class MallardDuck extends Duck {

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

}

public class RubberDuck extends Duck {

  public RubberDuck() {
    this.quackBehavior = new Squeak();
    this.flyBehavior = new FlyNoWay();
  }

}
```

Encapsulate What Varies

- Benefits:
  - Eliminated code duplication
  - Other types of objects can reuse the fly and quack behaviours
  - Can easily add / modify existing behaviours without modifying our duck classes
  - Can dynamically change behaviours *at run-time*

Code to an Interface, Not an Implementation

Design Principle 2:
Code to an Interface, Not an Implementation

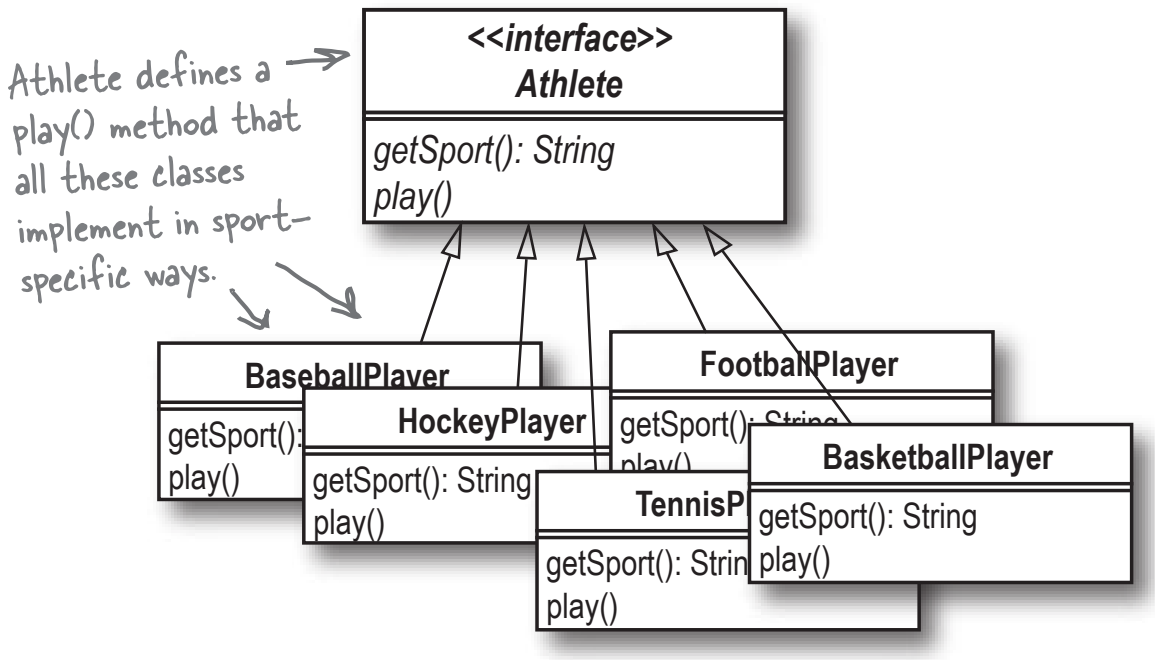
When faced with the choice between interacting with subclasses or interacting with a supertype, choose the supertype. Your code will be easier to extend and will work with all of the interface's subclasses – even those not yet created.

Code to an Interface, Not an Implementation

- Note: we are talking about the *concept* of an interface here – not the actual `interface` construct in languages like Java
  - The *interface* we are talking about could be an `interface`, abstract class, or even a concrete superclass
  - This design principle really says:
    Code to a Supertype, Not a Subtype

Modelling Athletes and Teams

- Suppose we have the following hierarchy:



Modelling Athletes and Teams

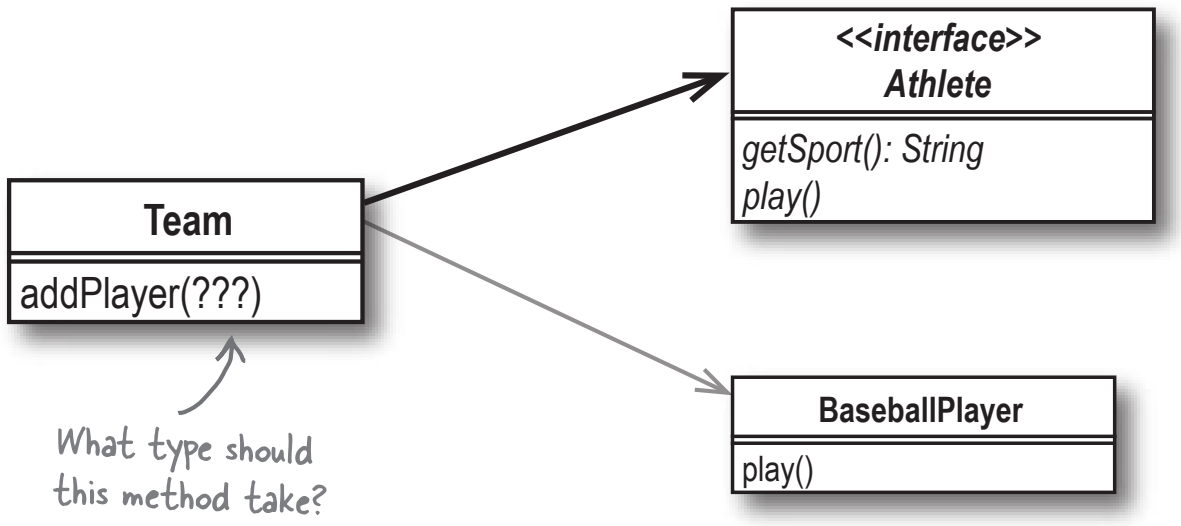
- We wish to model teams of athletes
- One option: create a class `Team` and then subclass that for each specific team type:
  - `BaseballTeam`
  - `FootballTeam`
  - `TennisTeam`
  - ...

Modelling Athletes and Teams

- Issues:
  - Creates a large inheritance hierarchy (KISS principle)
  - Results in extensive code duplication (DRY principle)
  - Have to add a new subclass for each new sport we wish to support

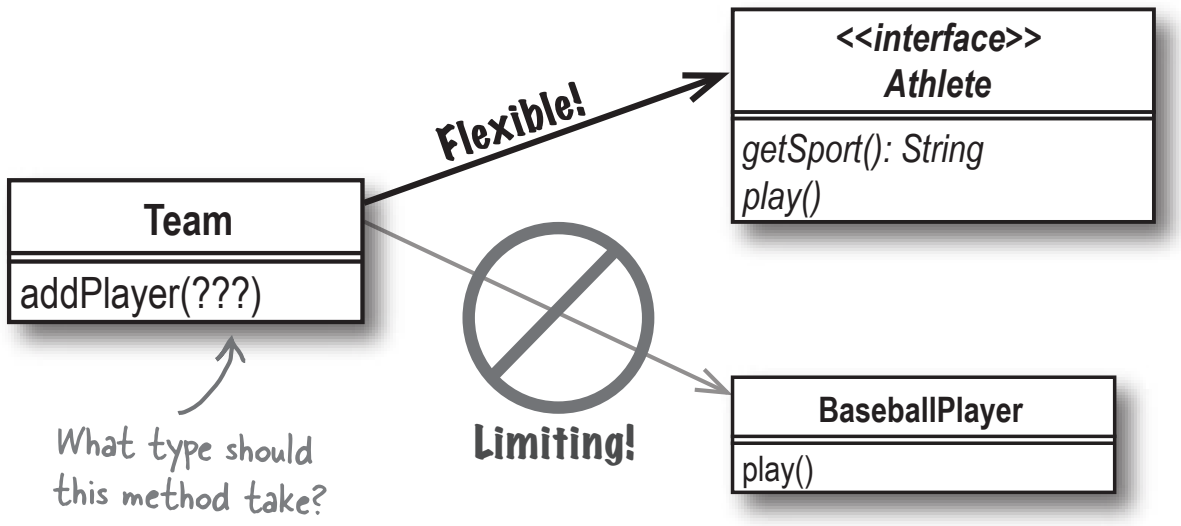
Code to an Interface, Not an Implementation

- A better way...



Code to an Interface, Not an Implementation

- A better way...



Code to an Interface, Not an Implementation

- Benefits
  - Adds flexibility
    - Code can now work with any type of `Athlete` – even those we haven't created yet
  - Simplified architecture
  - Reduced duplication
    - Having a hierarchy of teams (`BaseballTeam` , `FootballTeam` , ...) would result in extensive duplication of code.
    - `addPlayer` would be duplicated in each class

Favour Composition Over Inheritance

Design Principle 3:
Favour Composition Over Inheritance

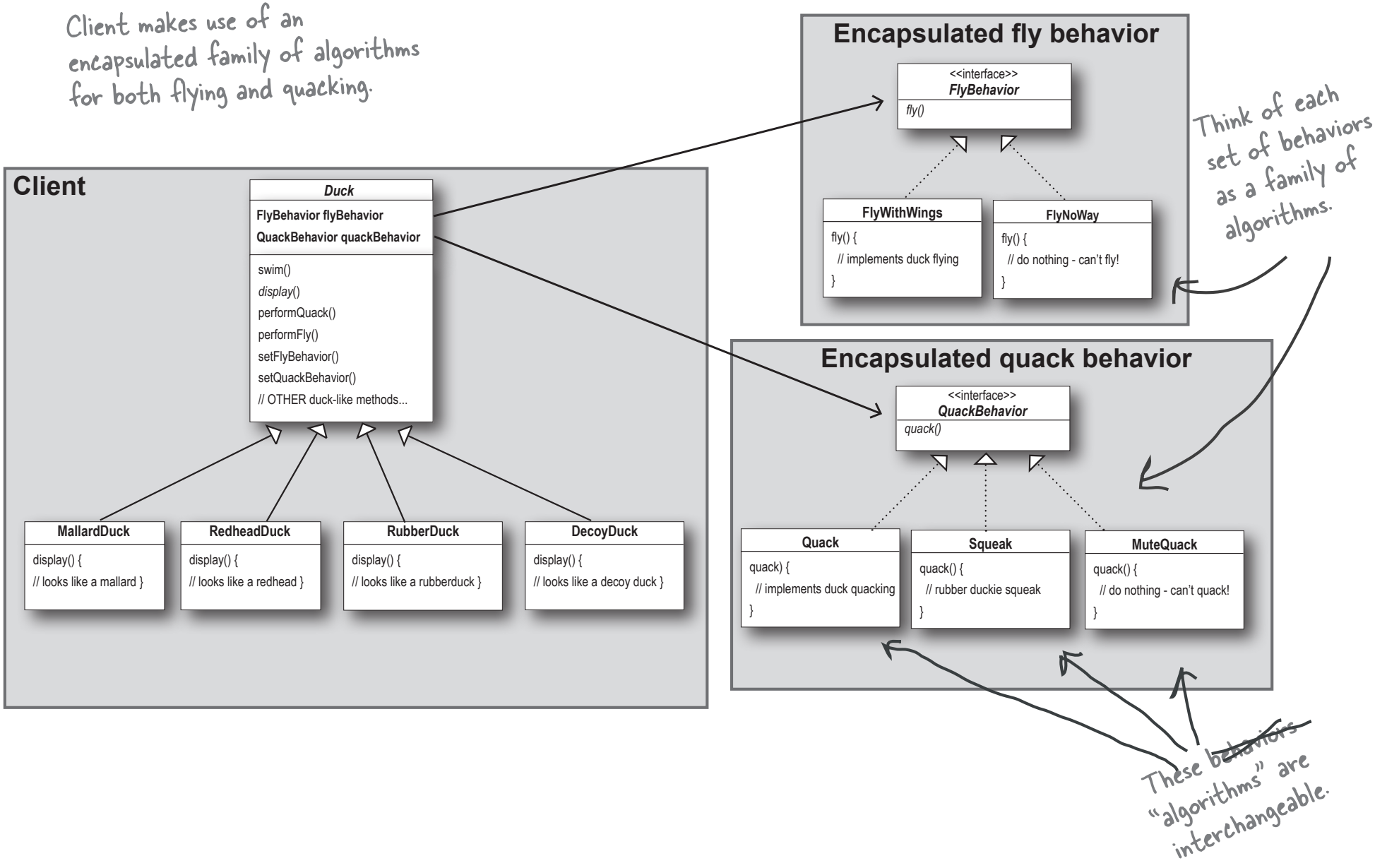
By favouring delegation, composition, and aggregation over inheritance, we can produce software than is more flexible, and easier to maintain, extend, and reuse.

Favour Composition Over Inheritance

- Inheritance establishes an **IS-A** relationship
- Composition / aggregation establish a **HAS-A** relationship – this can often be preferable
- We already saw an example of this in the duck simulation...

Favour Composition Over Inheritance

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



Favour Composition Over Inheritance

- Instead of inheriting their behaviour, the ducks get their behaviour by being *composed* with the right behaviour object
- Benefits:
  - Creating systems using composition gives us more flexibility
  - Encapsulate a family of algorithms into their own set of classes
  - Can easily extend the code with new behaviours
  - Can change behaviour at run-time
  - Reduce code duplication

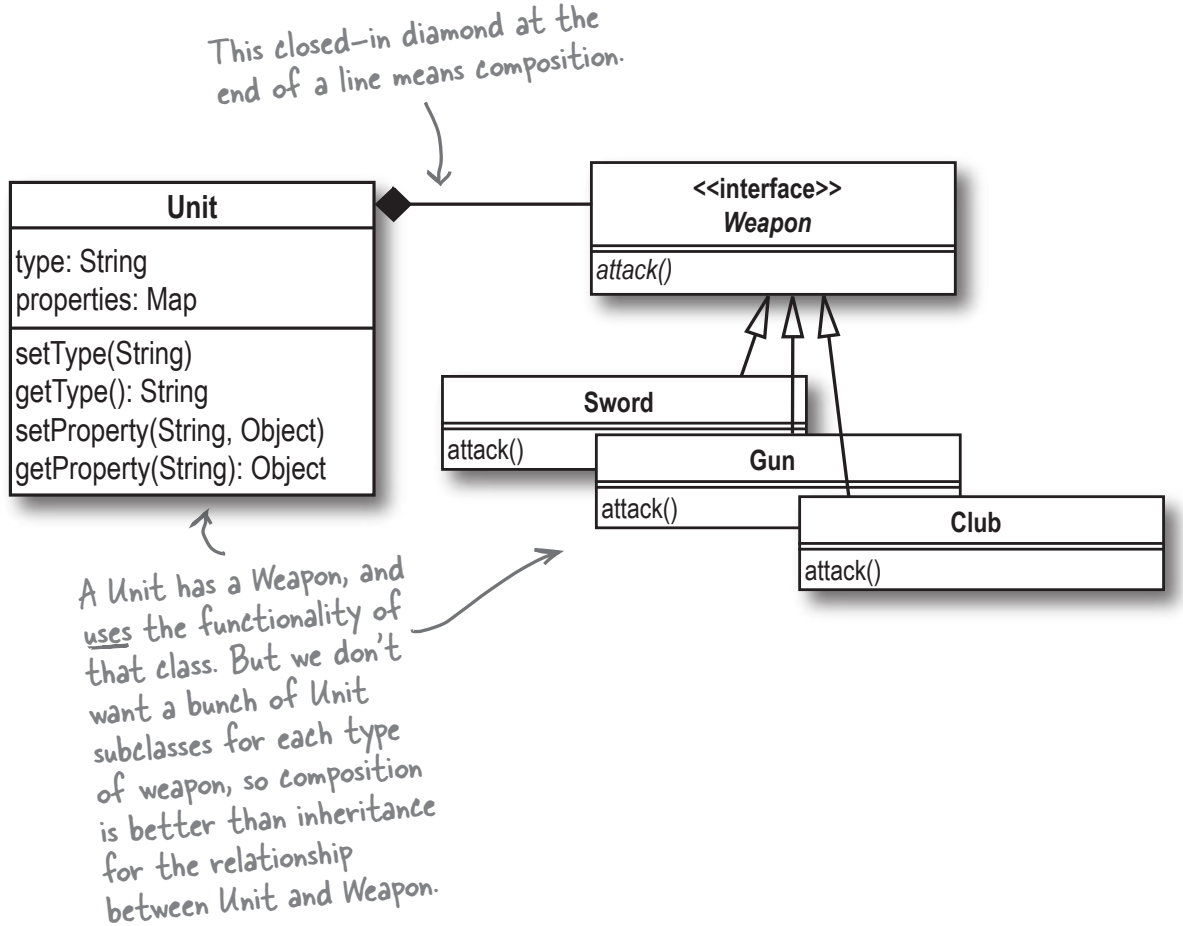
Favour Composition Over Inheritance

- Rule of thumb:

  If you need to use functionality in another class, but you don't need to *change* that functionality, consider using delegation instead of inheritance.

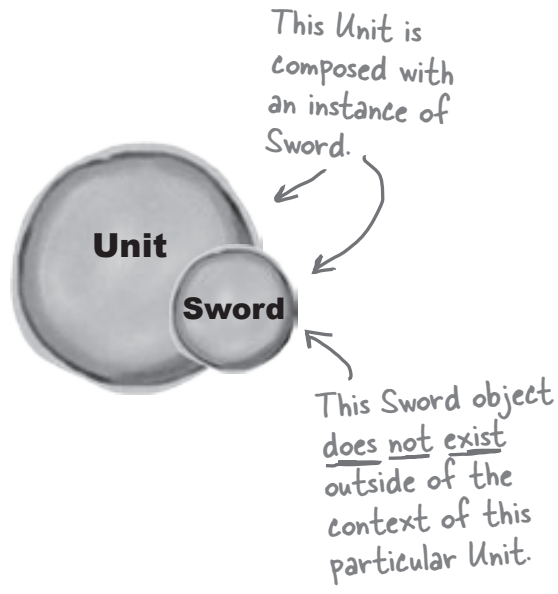
Composition

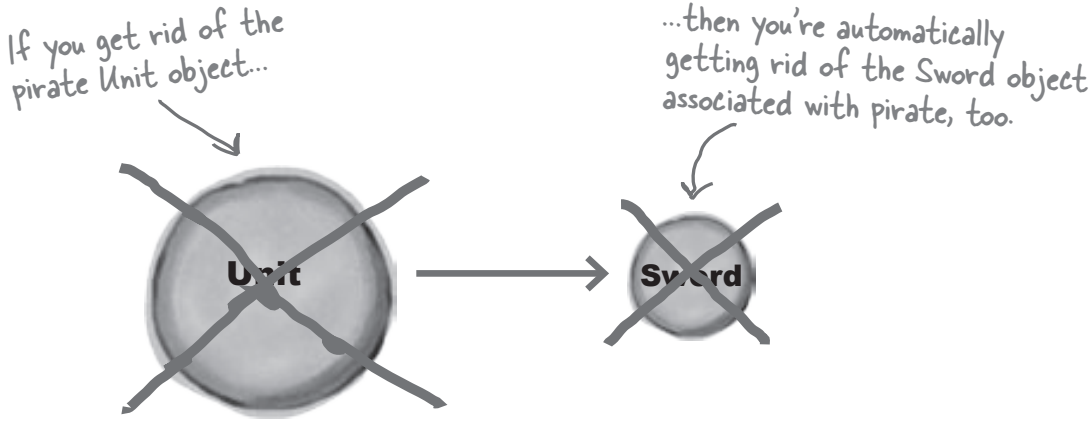
- An object composed of other objects *owns* those objects
- When the object is destroyed, so are all the objects of which it is composed



Composition

```
Unit pirate = new Unit();
pirate.setProperty("weapon", new Sword());
```



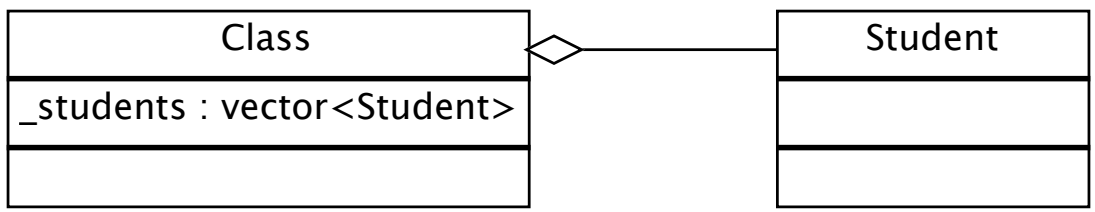


Composition

- Note: we are not necessarily seeking to model the real world exactly here:
  - Yes, a weapon could exist on its own in the real world
  - The question you should be asking yourself is whether or not *in your model*, your need to track a weapon outside of a unit
  - If not, use **composition** (*OWNS-A*)
  - If so, use **aggregation** (*HAS-A*)

Aggregation

- An object composed of other objects *uses* those objects
- Those objects exist outside of the object
- When the object is destroyed, the objects that compose it remain



Composition vs. Aggregation

- When deciding which to use, simply ask:

  Do I need this object outside of the class?

  - If no, use *composition* (black diamond of death)
  - If yes, use *aggregation* (white diamond of life)

SOLID

Single Responsibility Principle

Open/Closed Principle

Liskov Substitution Principle

I nterface Segregation Principle

Dependency Inversion Principle

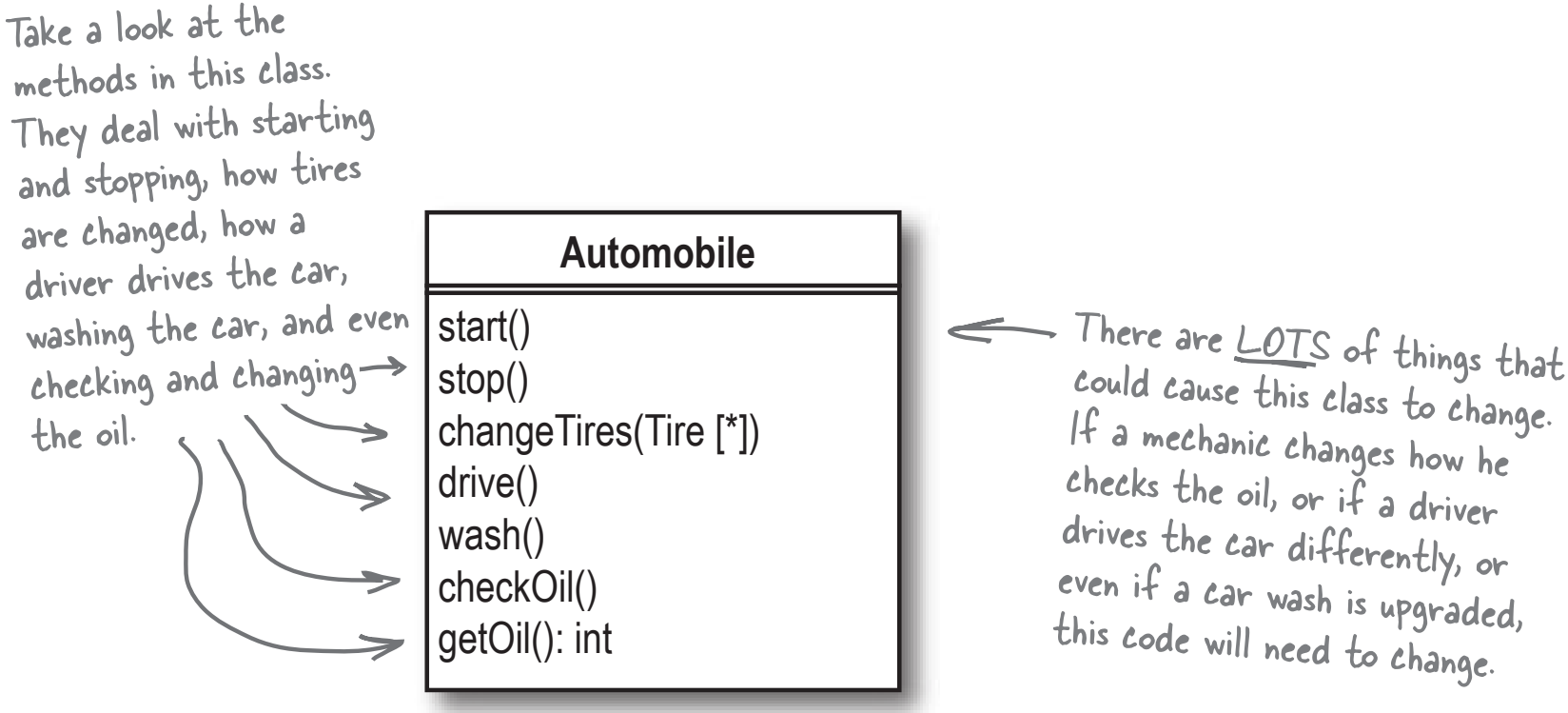
SOLID: Single Responsibility Principle

Design Principle 4:
Single Responsibility Principle

Every object in a system should have a single responsibility, and all the object's services should be focused on carrying out that single responsibility.

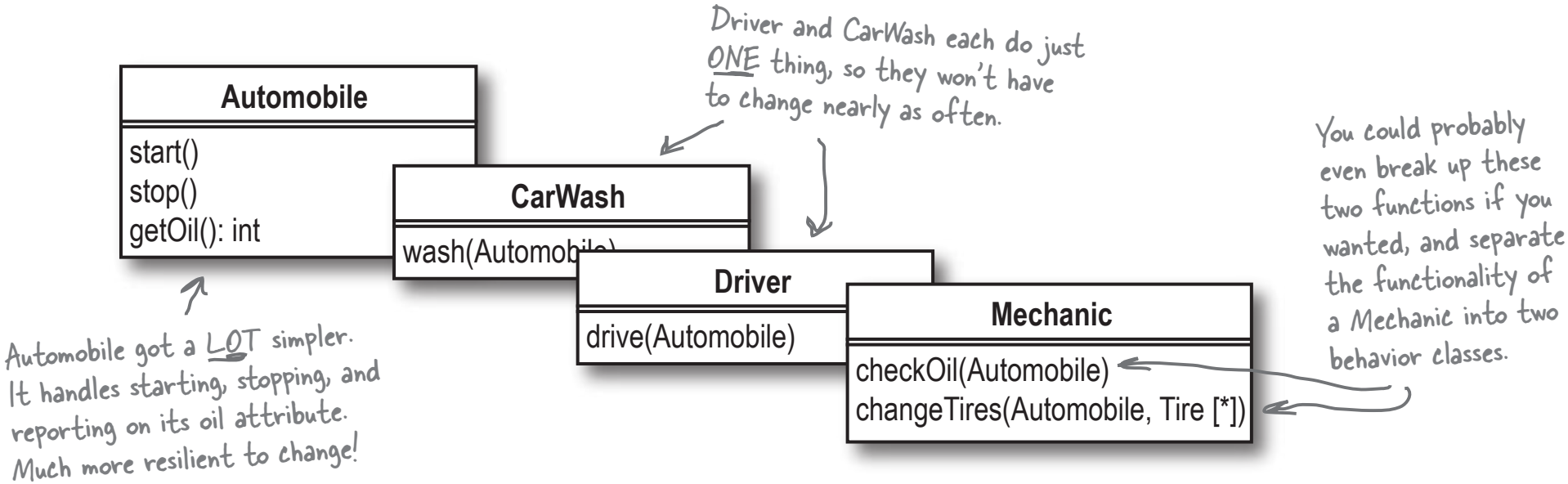
SOLID: Single Responsibility Principle

- Every object in a system should have a single responsibility
- Responsibility: a *reason to change*



SOLID: Single Responsibility Principle

- When a class has more than one reason to change, it is likely trying to do too much
- We should break up the class into multiple classes where **each individual class has a single responsibility** and thus has only **one reason to change**



SOLID: Single Responsibility Principle

- Benefits:
  - Minimize the chance that a class will need to be changed by reducing the number of things in the class that *can* change
  - Results in high cohesion:
    - **Cohesion**: the extent to which the elements of a module (e.g. methods of a class) belong together
    - High cohesion generally increases reusability, robustness, understandability, etc.

S**O**LID: Open/Closed Principle

Design Principle 5:
Open/Closed Principle

Classes should be open for extension, and closed for modification.

S**O**LID: Open/Closed Principle

- Classes should be:
  - **Closed for modification**: The source code of our classes is to be treated as immutable. No one should be allowed to modify it
    - Changing existing code can introduce new bugs
    - If we need to change a behaviour, we will *extend* the class
  - **Open for extension**: Behavioural changes that may be required should be accomplished through subclassing or other means (e.g. the Observer pattern) – we should not touch our existing, well-tested code

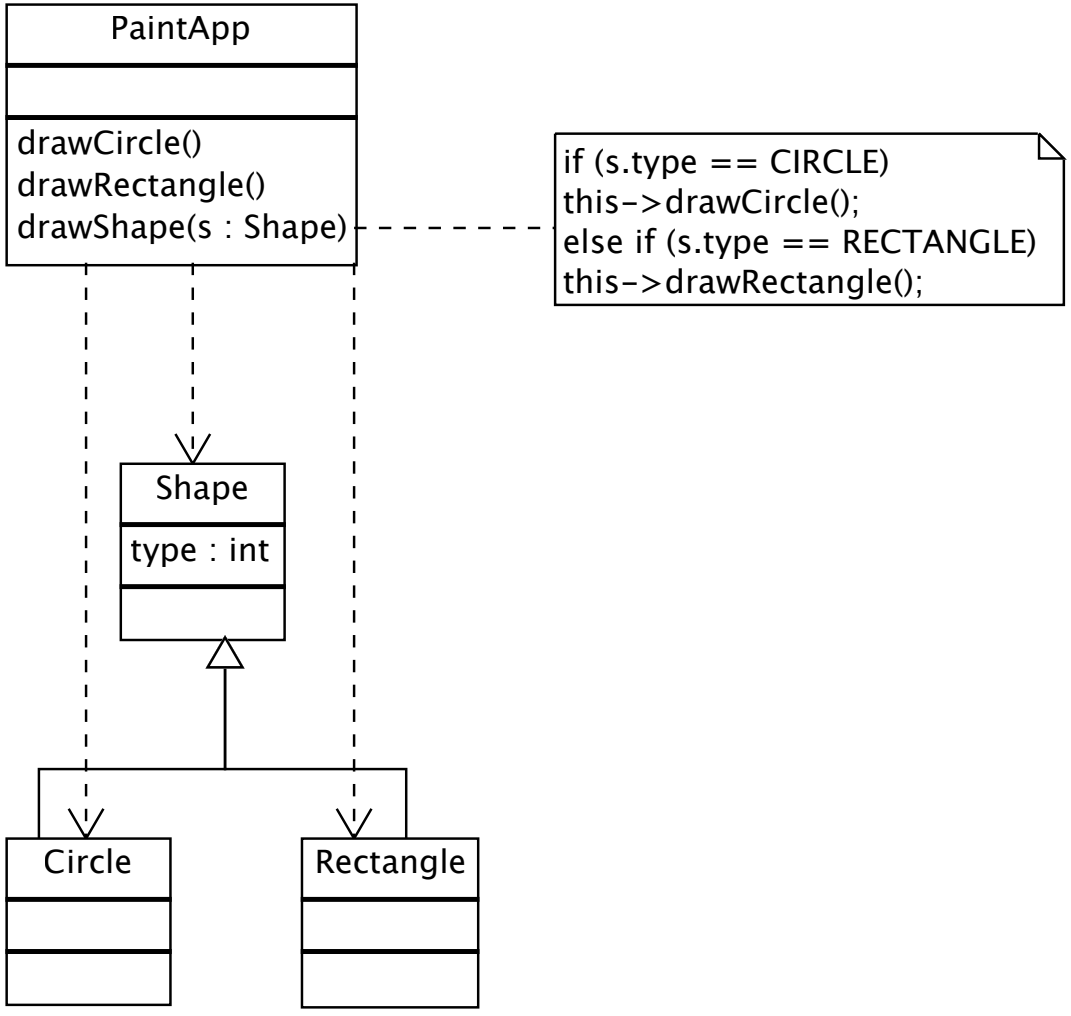
SOLID: Open/Closed Principle

You open classes by allowing them to be subclassed and extended.

You close classes by not allowing anyone to touch your working code.

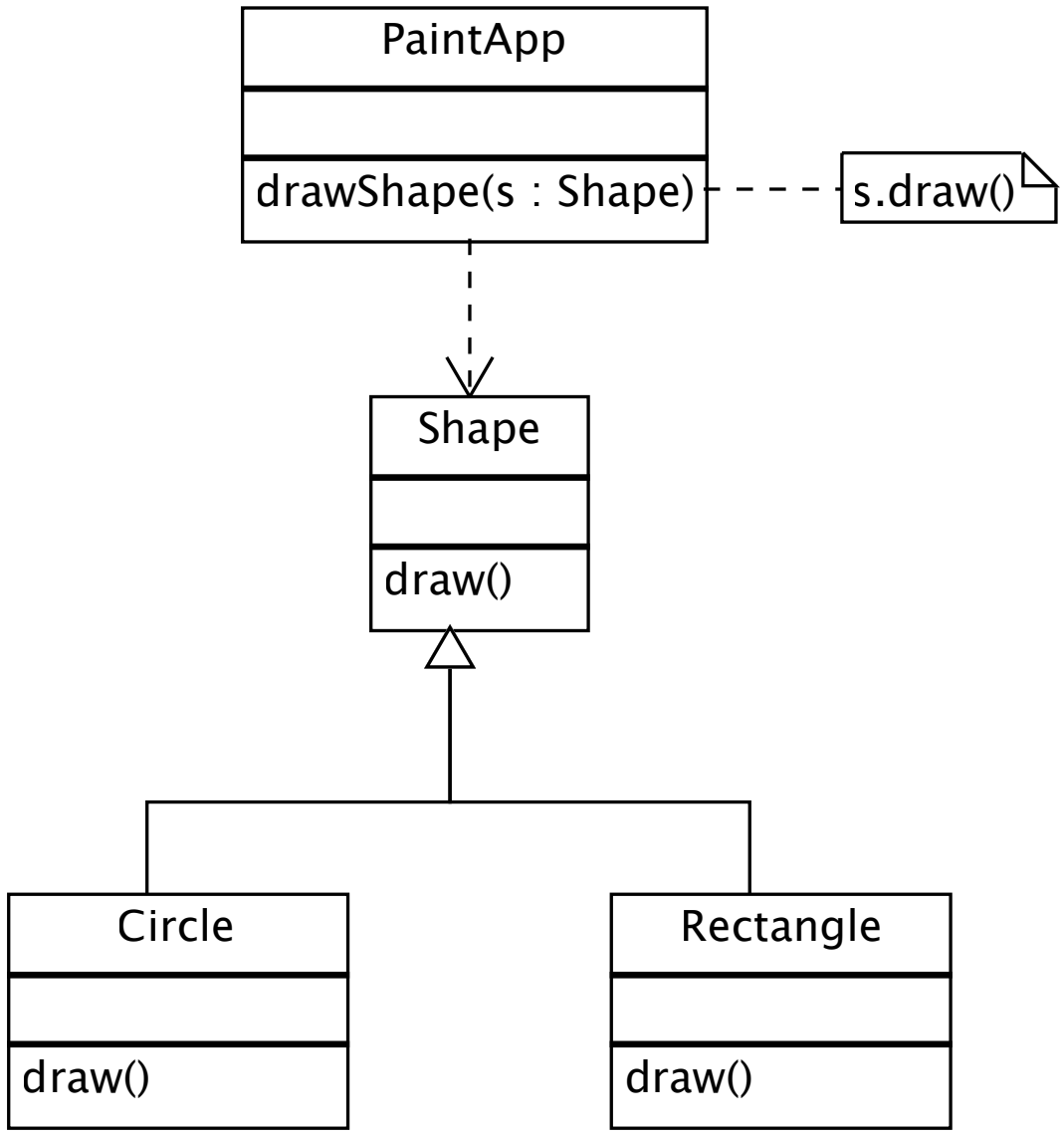
S**O**LID: Open/Closed Principle

- The following violates the OCP
  - What happens when we need to add a new shape?



S**O**LID: Open/Closed Principle

- Better:



S**O**LID: Open/Closed Principle

- Benefits:
    - Allow the behaviour of a class to be modified without touching its source code
        - Don't risk breaking well-tested code
        - We only modify existing code to fix errors – new/modified features require extension
- The most obvious way to extend a class is through inheritance, but it is not the only option. You will learn about the Decorator and Observer patterns in CS 3307 that can be used to extend the behaviour of a class without modifying its source

SOLID: Liskov Substitution Principle

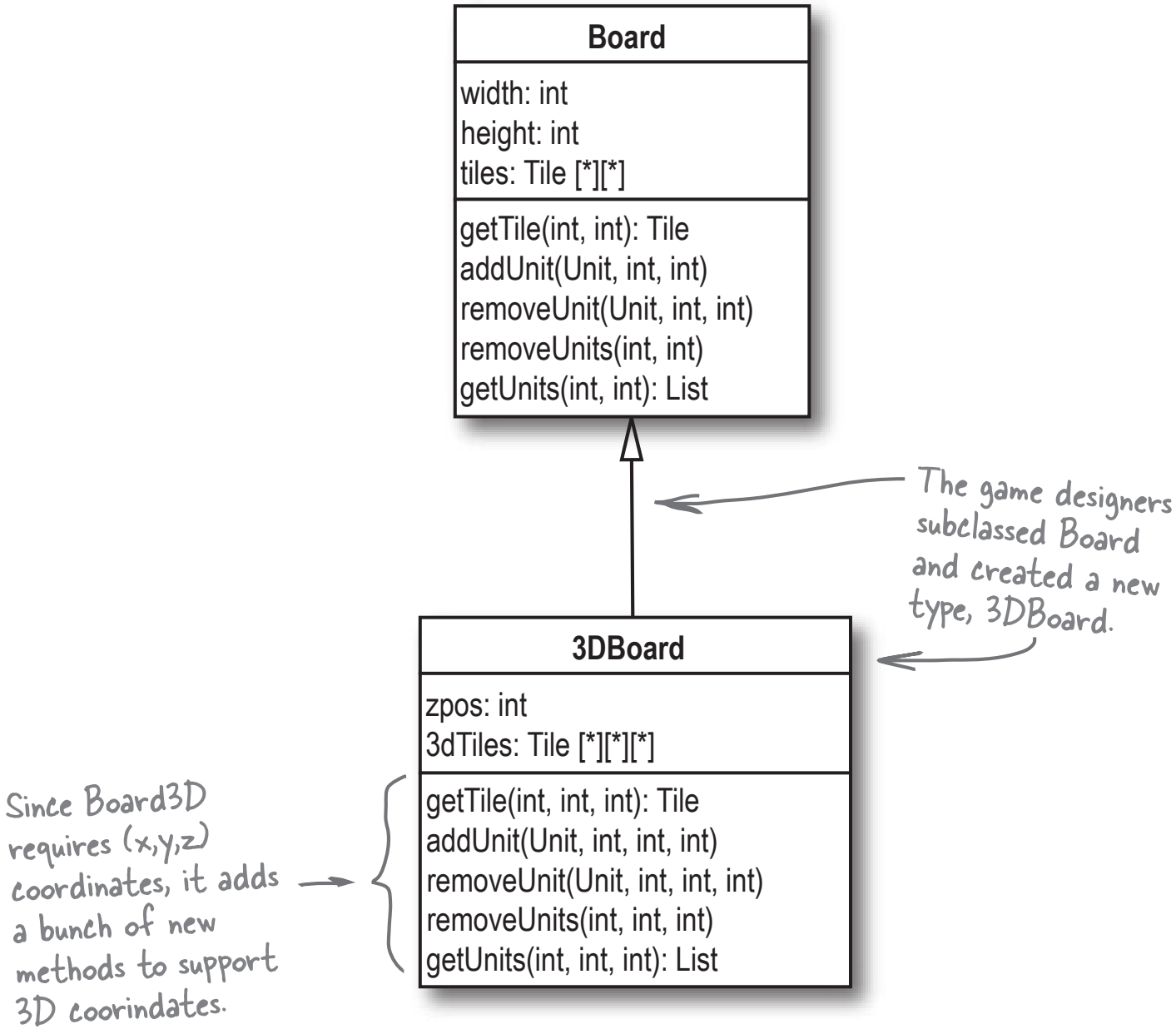
Design Principle 6:
Liskov Substitution Principle

Subtypes must be substitutable for their base types.

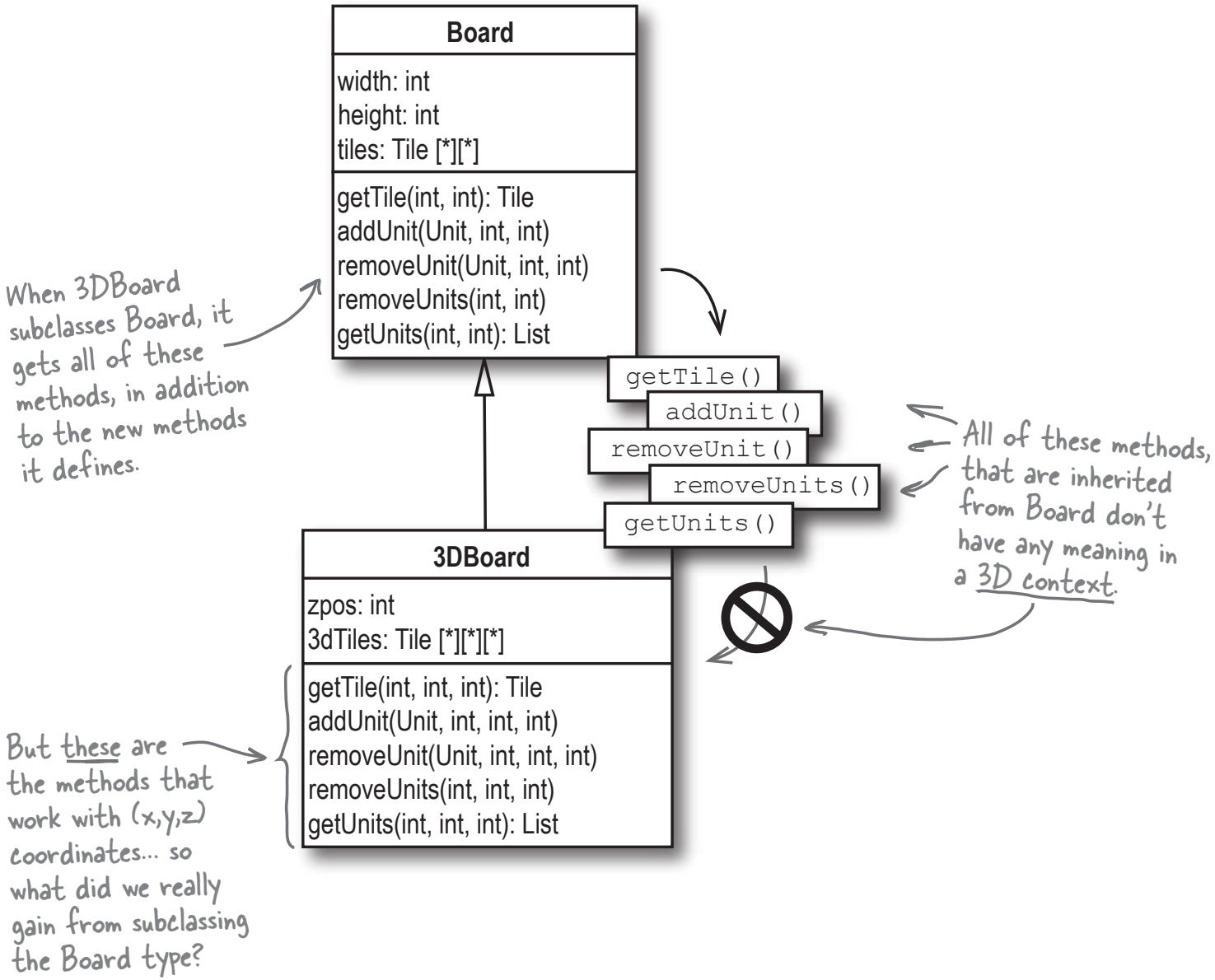
SO**L**ID: Liskov Substitution Principle

- The LSP is all about **well-designed inheritance**
  - When you inherit from a base class, you must be able to substitute your subclass for that base class without affecting program correctness
  - If not, you are misusing inheritance

SOLID: Liskov Substitution Principle



SOLID: Liskov Substitution Principle



SOLID: Liskov Substitution Principle

- The compiler will allow us to substitute `3DBoard` for `Board` just fine:

```
Board board = new 3DBoard();
```

- But `3DBoard` cannot really stand in for `Board` without affecting program correctness

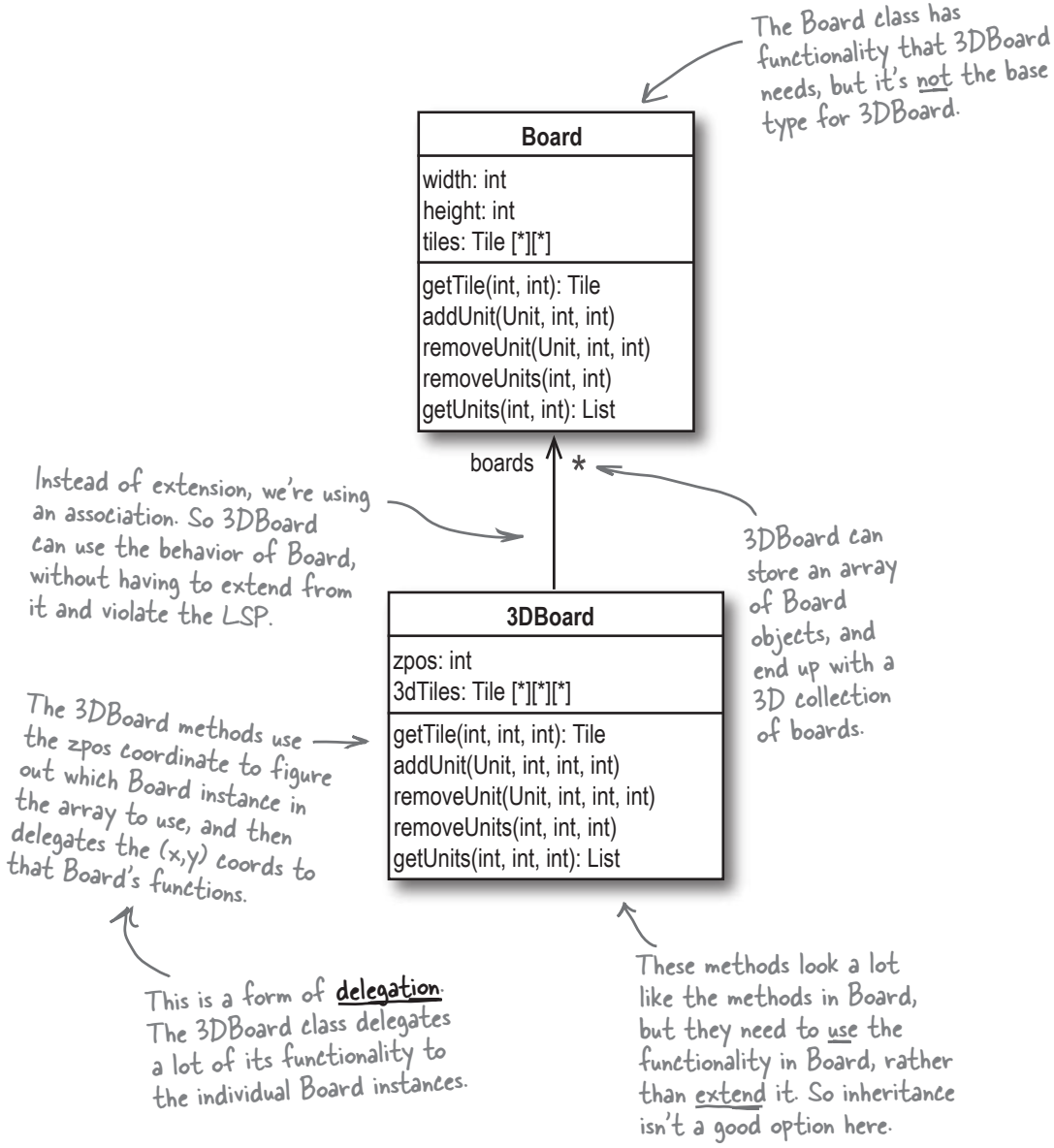
```
List<Unit> units = board.getUnits(8, 4);
```

- What does this method mean on `3DBoard`?

- The LSP states that **any** method on `Board` should be usable on `3DBoard`, i.e. `3DBoard` can stand in for `Board` without affecting program correctness

- `3DBoard` is not substitutable for `Board`: none of the methods on `Board` work correctly in a 3D environment

SOLID: Liskov Substitution Principle



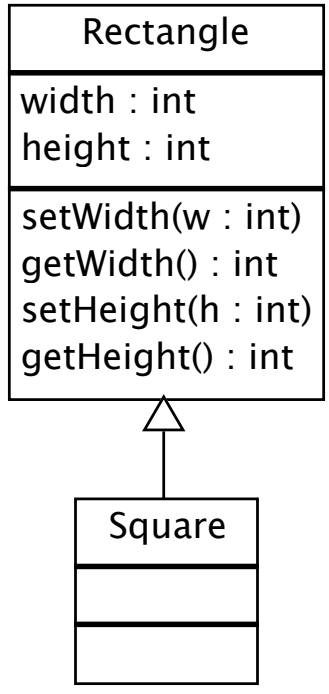
SOLID: Liskov Substitution Principle

- Suppose we have a class `Rectangle` in our system

```
public class Rectangle {

  private int width;
  private int height;

  public void setWidth(int w) {
    this.width = w;
  }

  public void setHeight(int h) {
    this.height = h;
  }

  // ...

}
```

SOLID: Liskov Substitution Principle

- A few months later, we realize that we need to add a `Square`
  - Does the following hierarchy violate the LSP?
  - A square **IS-A** rectangle



SOLID: Liskov Substitution Principle

- This one is more subtle. Excellent discussion:
  - http://www.objectmentor.com/resources/articles/lsp.pdf
- First clue that something is wrong:
  - A `Square` does not need both the `width` and `height` members, but will inherit them nonetheless
  - No bother, we've got lots of RAM...

SOLID: Liskov Substitution Principle

- Second clue:
  - The `setWidth` and `setHeight` methods inherited from `Rectangle` will not be appropriate for `Square`
  - No bother, we will override them...

```
public class Square extends Rectangle {

  public void setWidth(int w) {
    this.width = this.height = w;
  }

  public void setHeight(int h) {
    this.width = this.height = h;
  }

}
```

SOLID: Liskov Substitution Principle

- Third clue:
    - Our function `f` works for `Rectangle`, but not for `Square`

```
public void f(Rectangle r)
{
  r.setWidth(5);
  r.setHeight(4);
  assert(r.getWidth() * r.getHeight() == 20);
}
```

- The LSP says that anywhere we can use the base type, we should be able to use the subclass type without affecting program correctness
- Does this hold true here?

SOLID: Liskov Substitution Principle

- What gives? In real life, a square **IS A** rectangle
- A `Square` object, though, is **not** a `Rectangle` object
  - The behaviour of the `Square` is not consistent with the behaviour of the `Rectangle`
  - Behaviour is what software is all about
- Conclusion: **IS A** does not tell the whole story

> Use inheritance when one object *behaves* like another, rather than just when the **IS A** relationship applies.

SOLID: Interface Segregation Principle

Design Principle 7:
Interface Segregation Principle

Many client-specific interfaces are better than one general purpose interface. Clients should not be forced to depend upon interfaces that they do not use.

SOLID: Interface Segregation Principle

- **Fat interface:** class whose interface is not cohesive
  - Many responsibilities – unfocused, hard to understand/modify
- The ISP seeks to avoid fat interfaces
  - Some objects may require non-cohesive interfaces
  - Clients *should not* know about them as a single class
  - Clients *should* know about abstract base classes with cohesive interfaces

SOLID: Interface Segregation Principle

- Suppose we are implementing a security system
- We start with an interface `Door`:

```
public interface Door {
  public void lock();
  public void unlock();
  public boolean isDoorOpen();
}
```

SOLID: Interface Segregation Principle

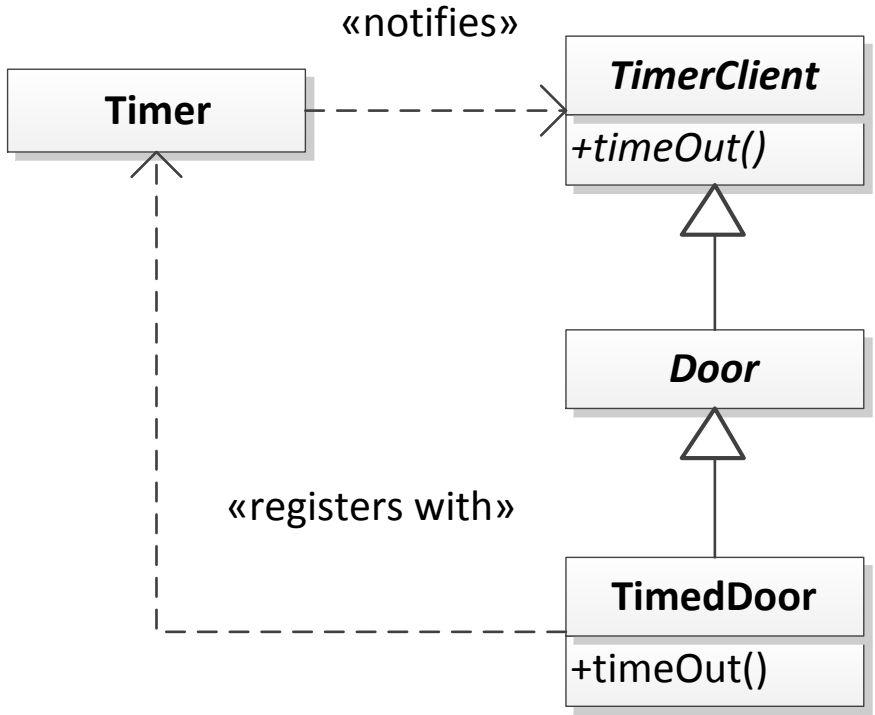
- We wish to have a class `TimedDoor` that will sound an alarm if left open for too long
- First, we will create a class `Timer` which `TimerClient`s can register with to receive notifications about timeouts

```
public class Timer {
  public void subscribe(int timeout, TimerClient client);
}

public interface TimerClient {
  public void timeOut();
}
```

SOLID: Interface Segregation Principle

- We want `TimedDoor` to be able to register itself with `Timer` so that it can receive notifications when the door has been open for too long

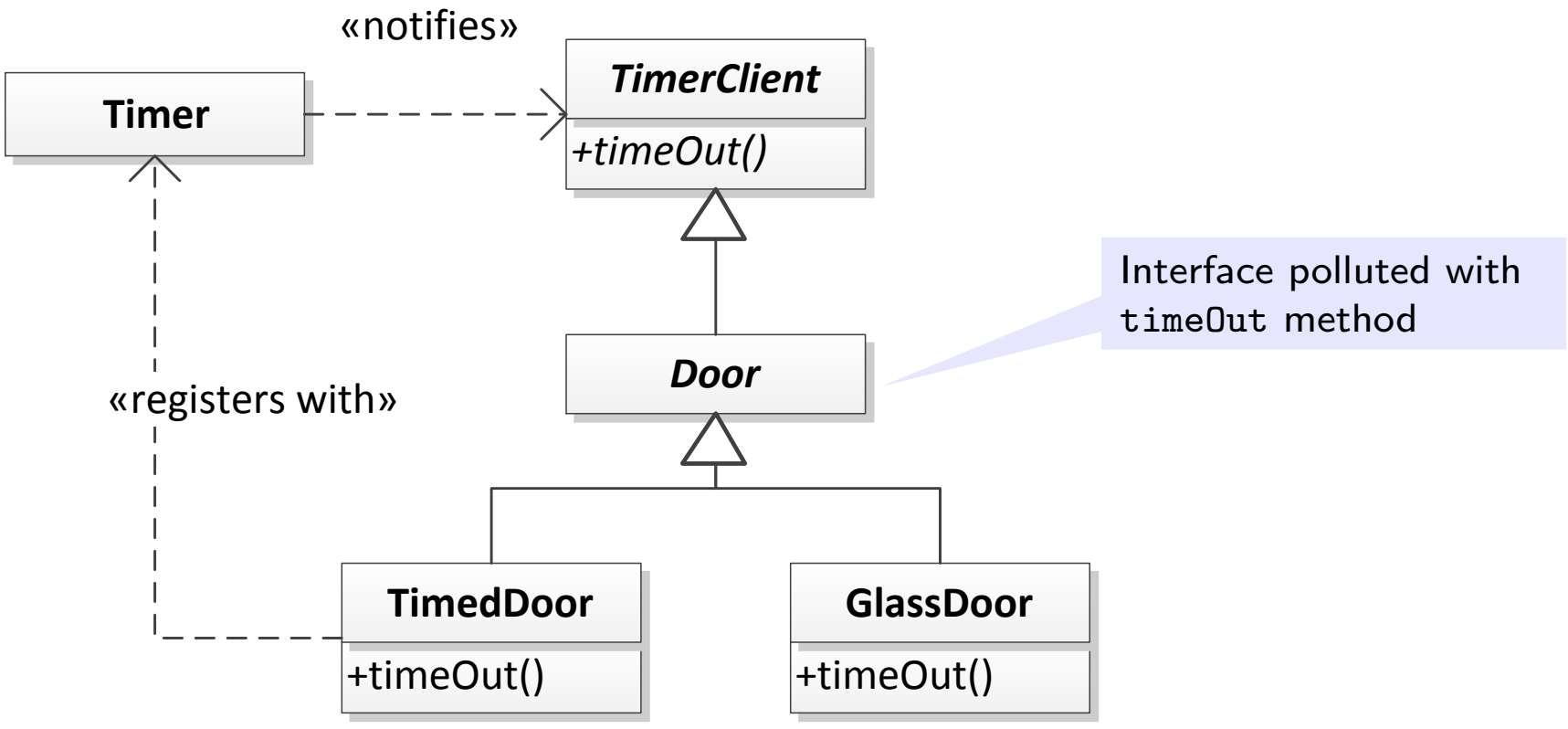
- We choose to have `Door` implement `TimerClient`, so that `TimedDoor` will be able to register itself with `Timer`



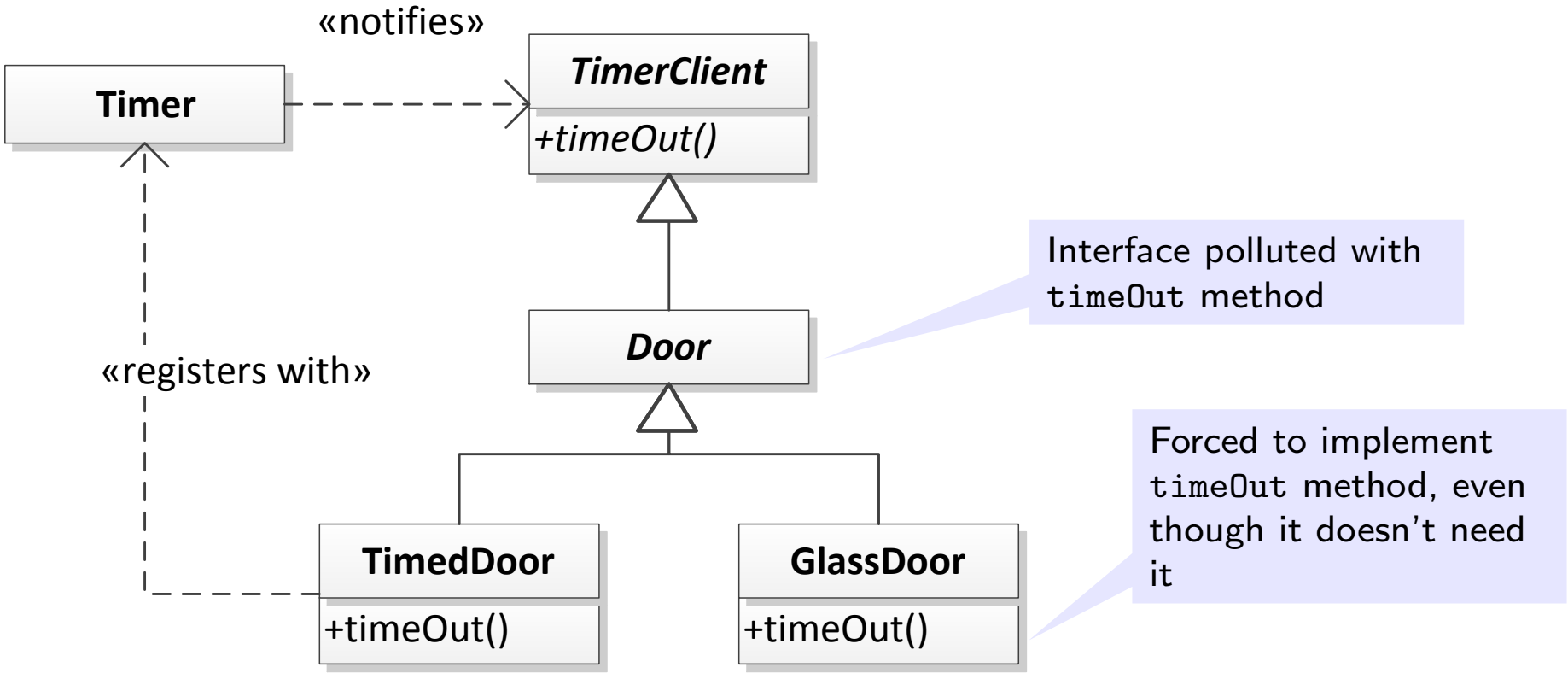
SOLID: Interface Segregation Principle

- Problems:
  - The interface of `Door` has been polluted with an interface it does not require
    - `Door` is now dependent on `TimerClient`, but not all doors need timing
    - Those that don't need timing will have to override the `timeOut` method to do nothing
    - When clients `import` those timing-free doors, they will import the definition of the `TimerClient` interface, even though it won't be used

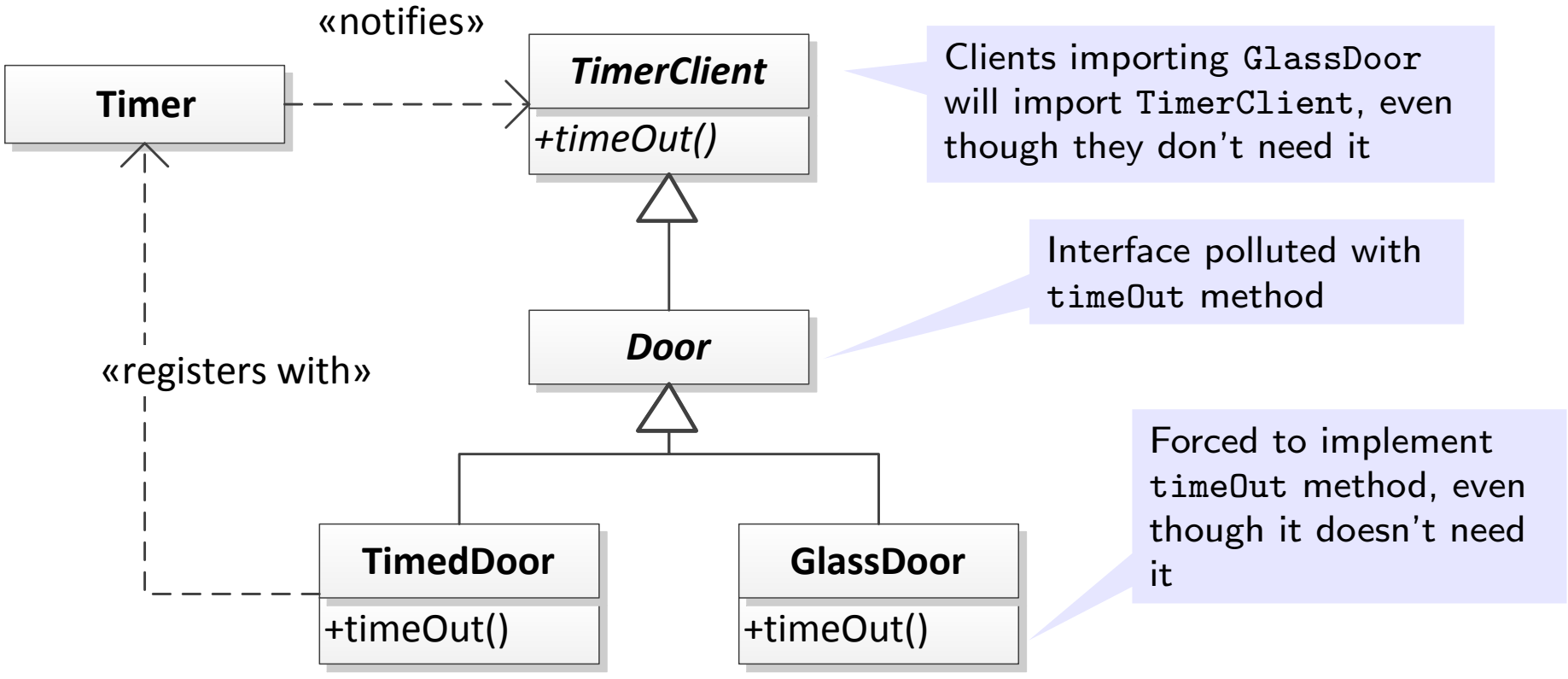
SOLID: Interface Segregation Principle



SOLID: Interface Segregation Principle



SOLID: Interface Segregation Principle



SOLID: Interface Segregation Principle

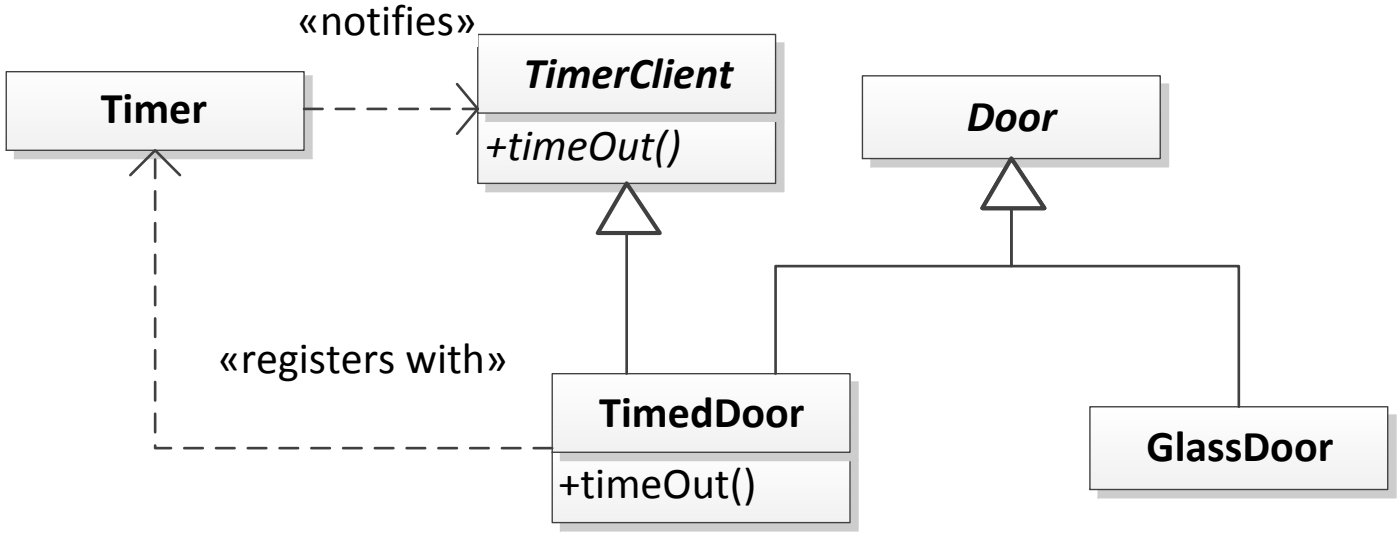
- If we continue this practice, then each time we need a new interface, we'll add it to the base class, further polluting its interface
    - Will have to go back and implement the new interface methods in every subclass – violation of the OCP

SOLID: Interface Segregation Principle

- `Door` and `TimerClient` provide interfaces used by completely different clients:
  - `Timer` uses `TimerClient`
  - Classes that manipulate doors use `Door`
  - If the clients are separate, then so, too, should the interfaces be separate
- Bottom line:
  - Don't add new methods appropriate to only one or a few implementation classes
  - Instead, divide the bloated interface into multiple smaller, more cohesive interfaces
  - New classes can then implement only the ones they need

SOLID: Interface Segregation Principle

- Solution using an interface:



- The Adapter design pattern can also be used to solve this problem – more on this pattern in CS 3307.

SOLI**D**: Dependency Inversion Principle

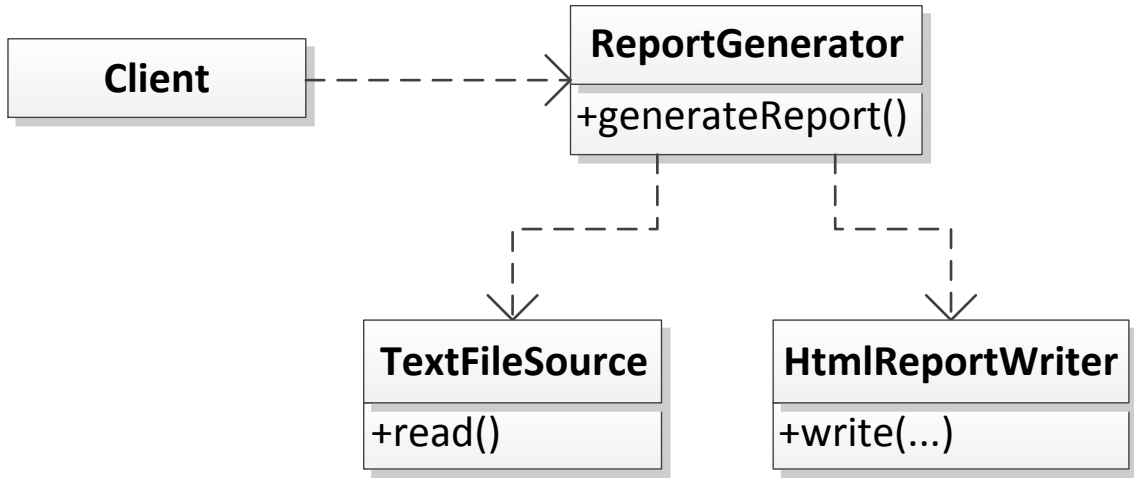
Design Principle 8:
Dependency Inversion Principle

High-level modules should not depend upon low-level modules. Both should depend upon abstractions.

Abstractions should not depend upon details. Details should depend upon abstractions.

SOLI**D**: Dependency Inversion Principle

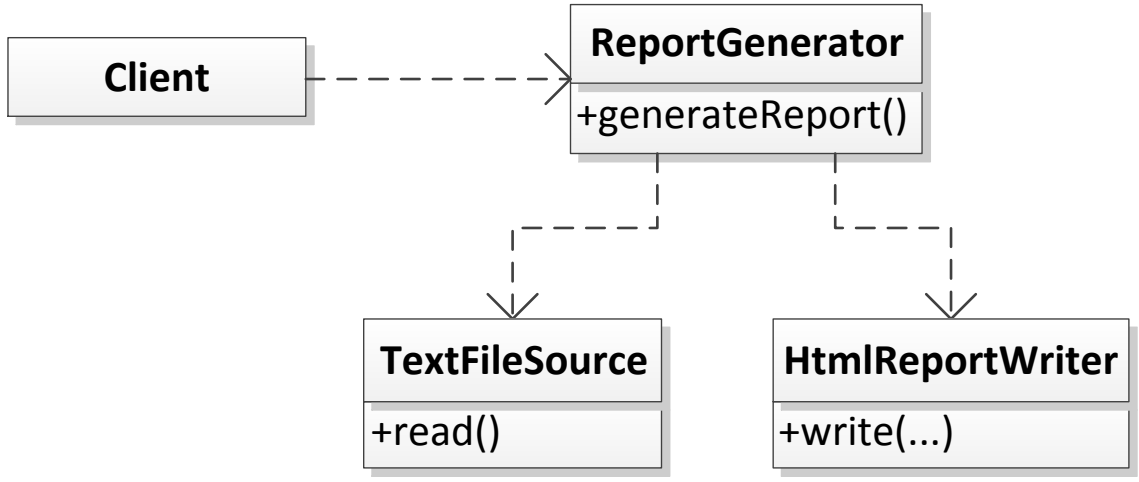
- Suppose we want to take data stored in text files and generate reports in HTML format...



SOLI**D**: Dependency Inversion Principle

```
public class ReportGenerator {
  .
  .
  .

  public void generateReport() {
    TextFileSource src = new TextFileSource(this.inFile);
    HtmlReportWriter dest = new HtmlReportWriter(this.outFile);

    String line;

    while ((line = src.readNextLine()) != null) {
      // Compile report
    }

    // Write report in HTML format
    dest.write(...);

  }

}
```

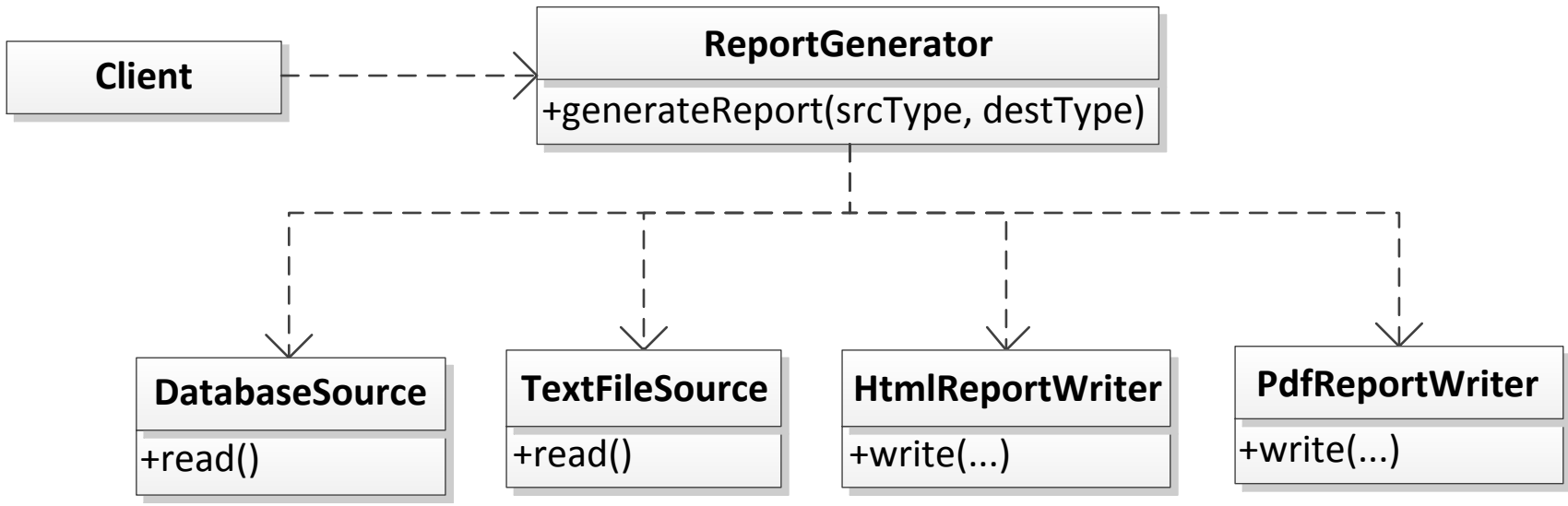
SOLID: Dependency Inversion Principle



- `TextFileSource` and `HtmlReportWriter` are certainly reusable
- Cannot reuse `ReportGenerator` unless we want to read from text files and write to HTML files
- Suppose we write a new program that needs to read from a database and write to PDF files – would be nice to reuse `ReportGenerator`
  - `ReportGenerator` is dependent on `TextFileSource` and `HtmlReportWriter`, so this is not possible

SOLI**D**: Dependency Inversion Principle

- Could modify `generateReport` to accept the type of source and destination to use...



SOLI**D**: Dependency Inversion Principle

```
public class ReportGenerator {
  .
  .
  .

  public void generateReport(String srcType, String destType) {
    if (srcType.equals("text") && destType.equals("html"))
      generateHtmlReportFromText();
    else if (srcType.equals("text") && destType.equals("pdf"))
      generatePdfReportFromText();
    else if (srcType.equals("db") && destType.equals("html"))
      generateHtmlReportFromDb();
    else if (srcType.equals("db") && destType.equals("pdf"))
      generatePdfReportFromDb();
    else
       // throw exception

    }

}
```

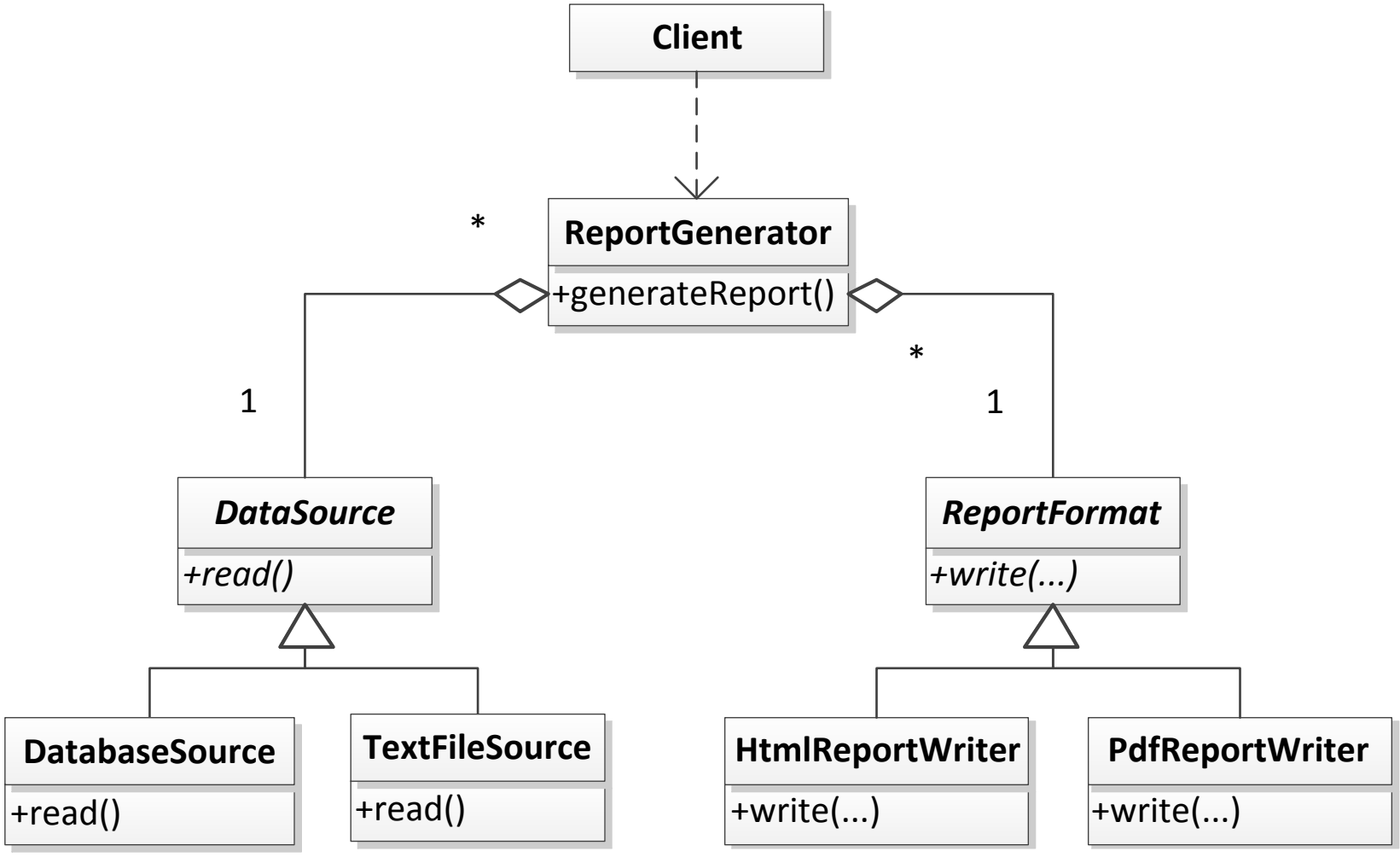
SOLI**D**: Dependency Inversion Principle

- This drastically increases coupling in the system
  - Over time, more source and destination types will be added to `generateReport`
  - The `ReportGenerator` class will be littered with `if-else` statements and dependent upon many lower-level modules
  - **Rigid**: system will become hard to change since every change will affect too many parts of the system
  - **Fragility**: when changes are made to the system, unexpected parts will break due to the changes

SOLI**D**: Dependency Inversion Principle

- Better solution:
  - Make `ReportGenerator` (the higher-level class) independent of the lower-level classes it controls
  - Can then reuse it freely
  - This is called *dependency inversion*

SOLI**D**: Dependency Inversion Principle



SOLI**D**: Dependency Inversion Principle

```
public class ReportGenerator {

  private DataSource src;
  private ReportFormat dest;

  .
  .
  .

  public void generateReport() {
String line;

    while ((line = this.src.read()) != null) {
      // Compile report
    }

    // Write report
    this.dest.write(...);

  }

}
```

SOLID

- Summing up SOLID, courtesy of globalnerdy.com...

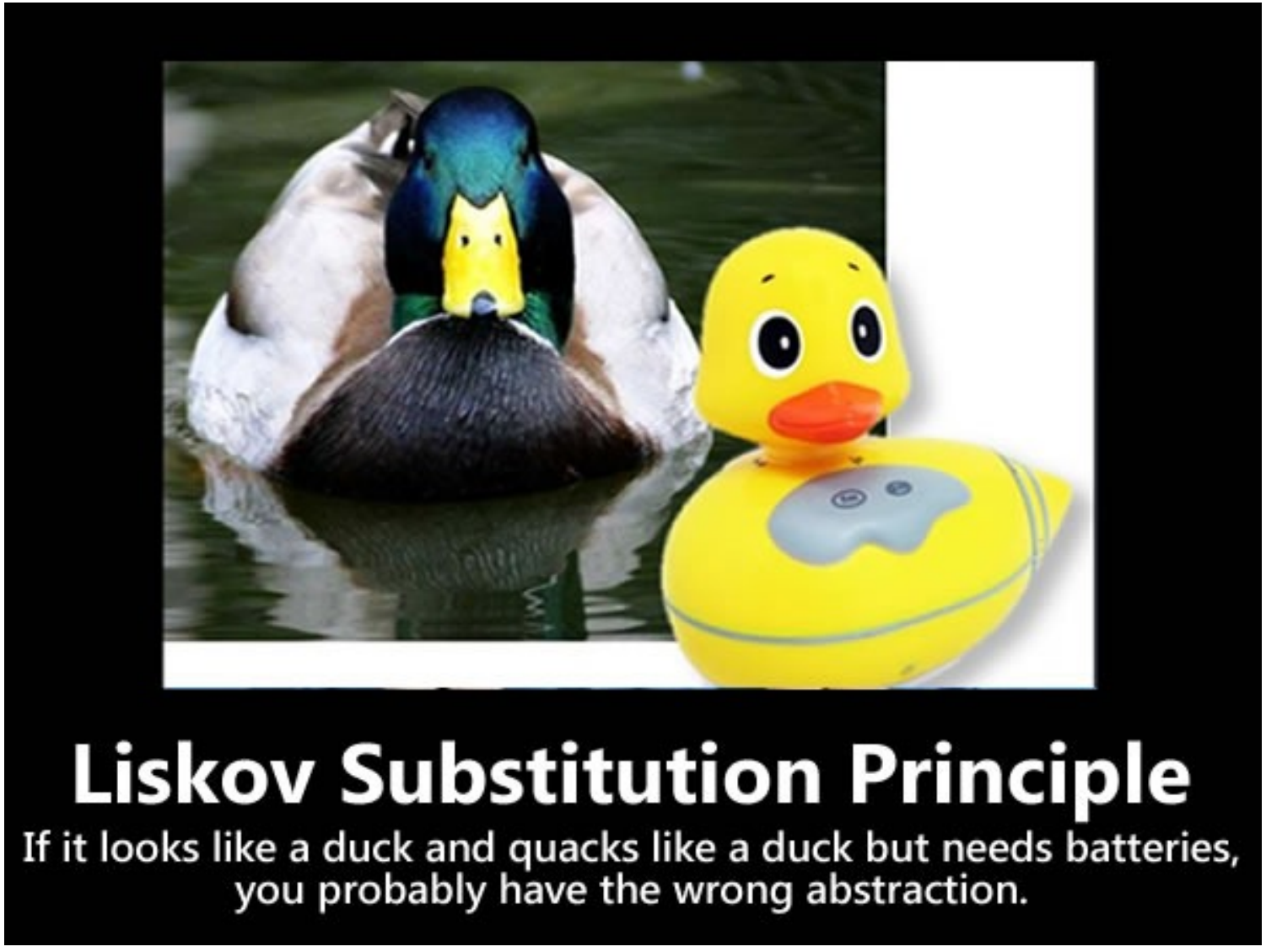
SOLID


Single Responsibility Principle
Just because you *can* doesn't mean you *should*.

SOLID


Open-Closed Principle
Open-chest surgery isn't needed when putting on a coat.

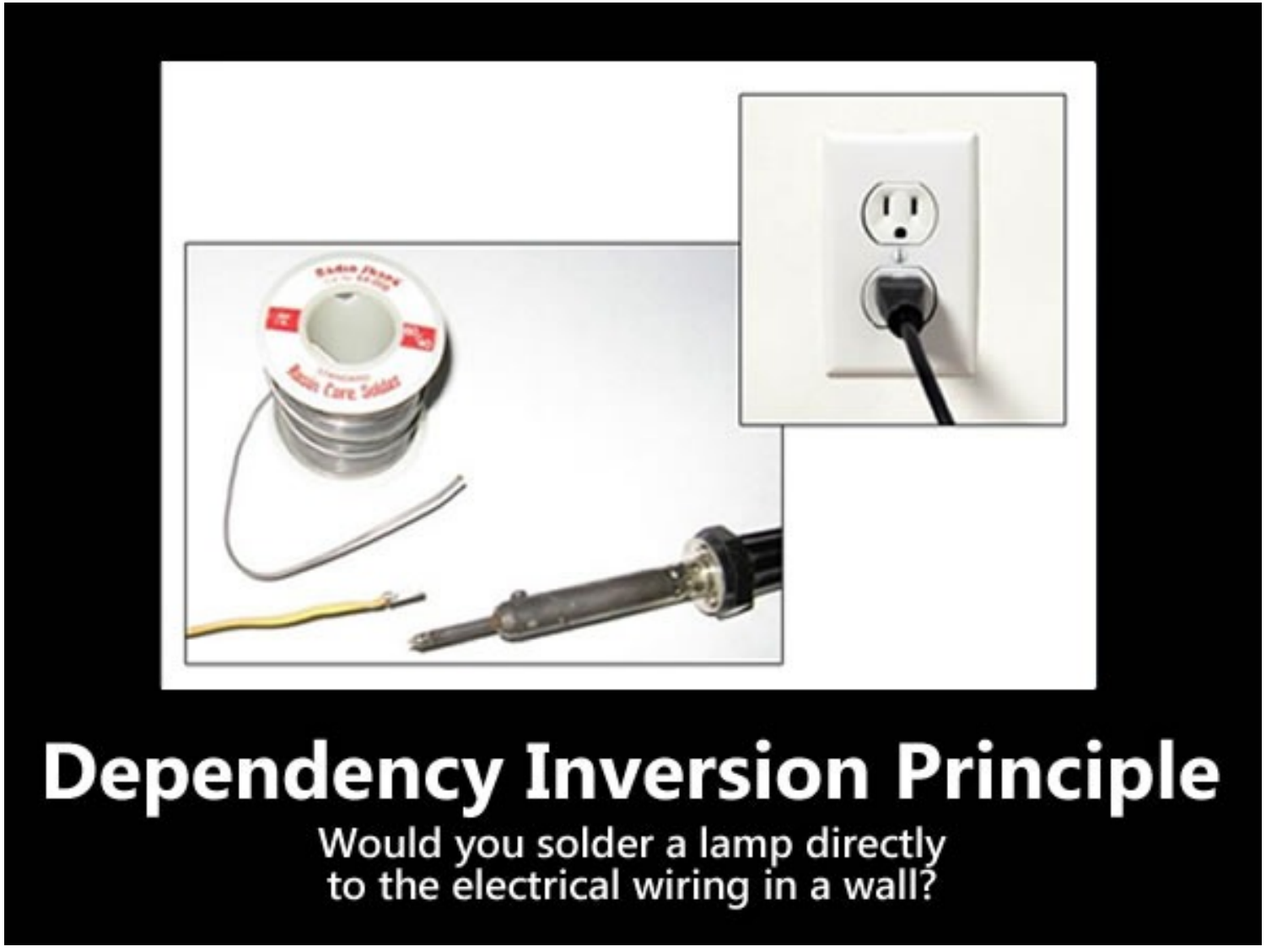
SOLID


Liskov Substitution Principle
If it looks like a duck and quacks like a duck but needs batteries,
you probably have the wrong abstraction.

SOLID


Interface Segregation Principle
You want me to plug this in *where?*

SOLID


Dependency Inversion Principle
Would you solder a lamp directly
to the electrical wiring in a wall?

- A few more principles...

Principle of Least Knowledge

Design Principle 9:
Principle of Least Knowledge

Talk only to your immediate friends.

Also known as the Law of Demeter

Principle of Least Knowledge

- Suppose we are writing an application to graph the temperature data from a car's computer system..

```
public void plotTemperature(Sensor sensor) {
  double temp = sensor.getSensorData().getOilData().getTemp();
  .
  .
  .
}
```

- How many classes is this code coupled with?

Principle of Least Knowledge

- May work for awhile until we make a change to `Sensor`, `SensorData`, or `OilData`
- Due to the strong coupling, a change to any of these classes could cause us to have to refactor `plotTemperature`
- Better...

```
public void plotTemperature(Sensor sensor) {
  double temp = sensor.getOilTemp();
  .
  .
  .
}
```

Principle of Least Knowledge

- Ideally, for any method in an object, we should only invoke methods that belong to:
    - The object itself
    - Objects passed in as a parameter to the method
    - Any object the method creates or instantiates
    - Any components of the object
        - i.e. methods on instance variables of the object

DRY: Don't Repeat Yourself

Design Principle 10:
DRY Principle

Every piece of knowledge must have a single, unambiguous, authoritative representation within a system.

DRY: Don't Repeat Yourself

- Modifying an element of a system should not require a change in other logically unrelated elements
- Most understand DRY to mean: *don't duplicate code*
- DRY includes *copy and modify programming*, but goes beyond it:
  - Code
  - Architecture
  - Requirements documents
  - User documentation
- Duplication of knowledge causes maintenance nightmares

DRY: Don't Repeat Yourself

- Obvious *copy and modify* example:

```
public class CustomerOrder {

  private Date paymentDate;

  public void sendEmail() {

    if (this.paymentDate != null)
      this.sendThankYouEmail();
    else
      this.sendInvoiceEmail();
  }

  public boolean shouldShip() {
    return (this.paymentDate != null);
  }

}
```

DRY: Don't Repeat Yourself

- Less obvious DRY violation...

```
public class Customer {

  private boolean validatePostalCode() {
    return this.postalCode.matches("[a-z][0-9][a-z] [0-9][a-z][0-9]")
        ;
  }

}

public class Shipping {

  private bool isValidPostCode() {
    return (
      this.pCode.length() == 7 &&
      Character.isLetter(pcode.charAt(0)) &&
      Character.isDigit(pcode.charAt(1)) &&
      Character.isLetter(pcode.charAt(2)) &&
      Character.isDigit(pcode.charAt(4)) &&
      Character.isLetter(pcode.charAt(5)) &&
      Character.isDigit(pcode.charAt(6)) &&
    );

  }

}
```