Design Principles

Weeks of coding can save you hours of design.

Coupling

- Describes the interdependence of entities
- Bound together to be functional

Coupling

- High/strong coupling
  - One object uses the internal data of another directly
  - Returning a pointer to object data
  - Sharing global variables
  - C++ friends

Coupling

- Medium coupling
  - Controlling the flow of another entity
  - Passing entire data structures when only some data is needed
- Low coupling
  - Share only data, through parameters and returns
  - Events or messages passed

Coupling

- Low coupling leads to information hiding and encapsulation
  - Stable interfaces
  - Better maintainability
- High coupling leads to interdependent entities
  - Design is difficult to change and extend
  - Code is difficult to read

Cohesion

- Intra-relatedness or focus of an entity
- Relatedness of functionality and/or the data of an entity

Cohesion

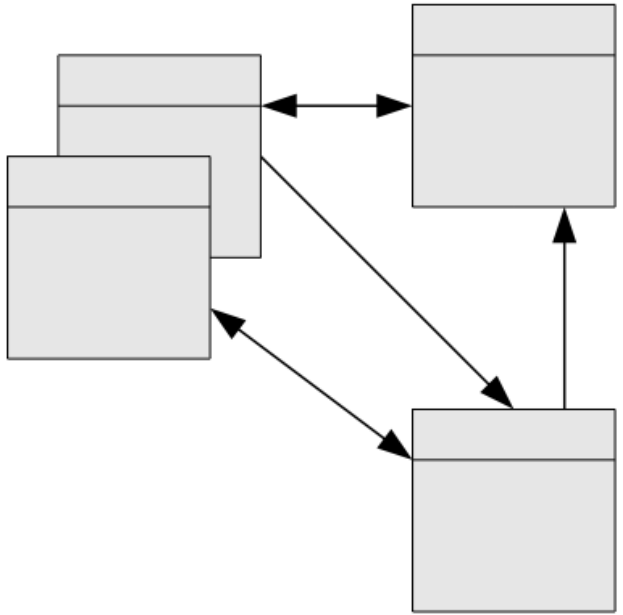
- Low cohesion
  - Related only because they are grouped together
  - Entity names often vague
  - Functions provided perform various activities
  - Groups of functions often act on distinct sets of data

Cohesion

- High cohesion
  - Entity's members contribute to a well defined purpose
  - Entity name often very specific
  - Functions act on the same set of data
  - Improved clarity of entities and their dependencies
  - Easier to maintain and extend
  - Leads to code reuse

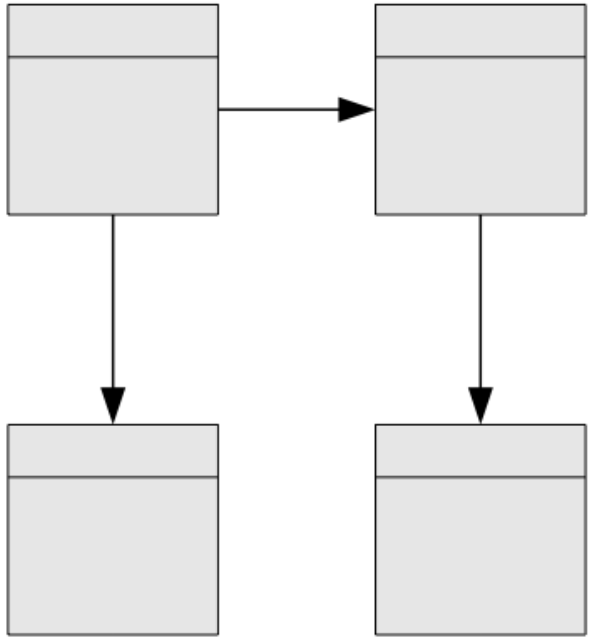
High Coupling / Low Cohesion

- Hard to read and understand
- Boundaries between entities are weak and few
- High interdependence causes unstable code



Low Coupling / High Cohesion

- Easier to read, maintain and extend
- Entities are responsible for specific functionality
- Entities function more independently



Encapsulate What Varies

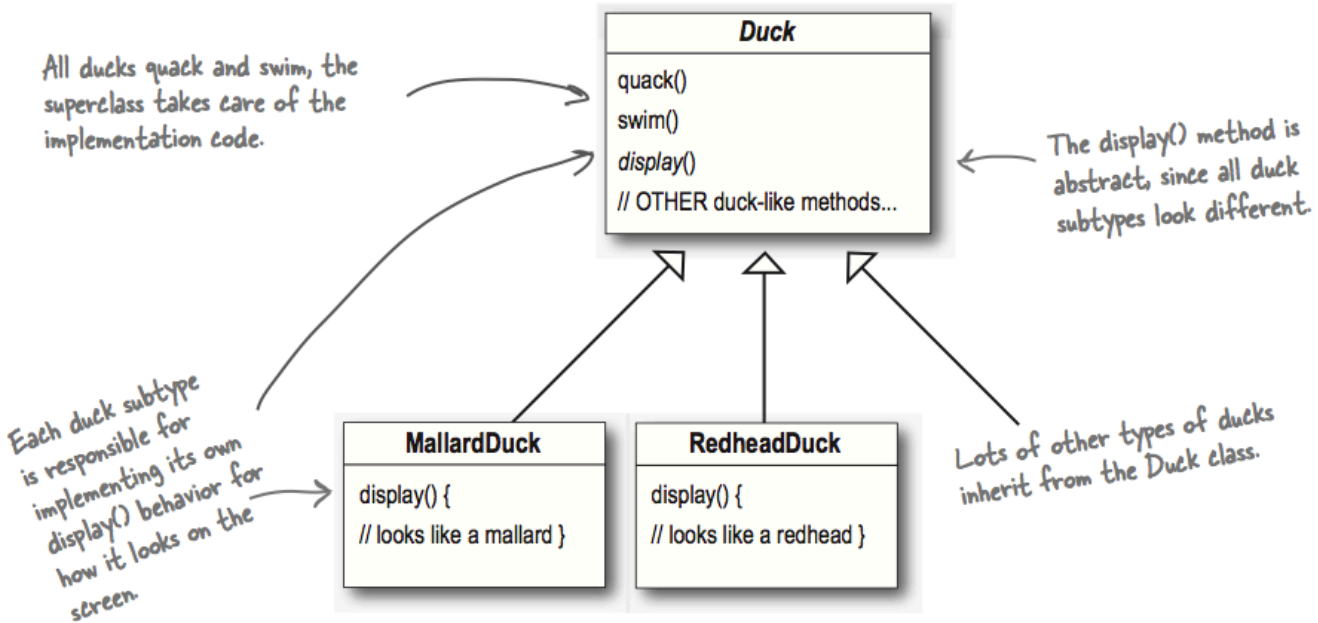
Design Principle:
Encapsulate what varies

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.

Case Study: Duck Hunt Simulation Game

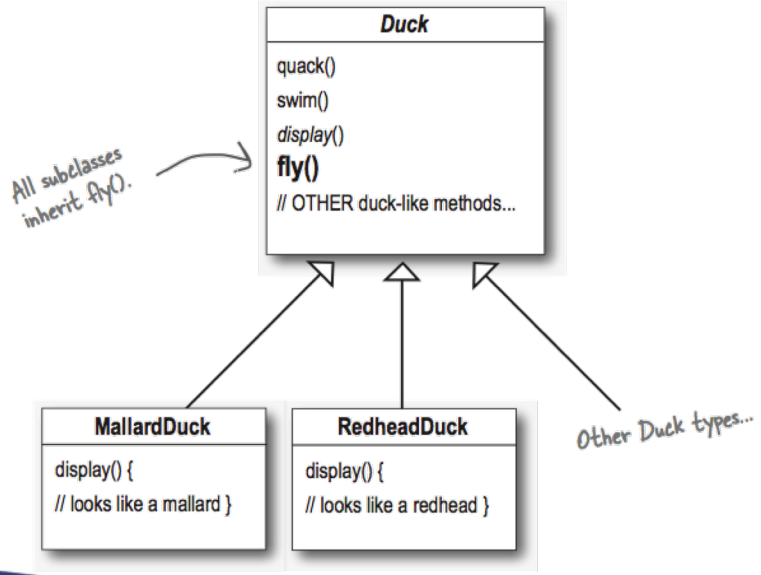
- Suppose a company is building a duck hunt simulation game …

Modeling Ducks



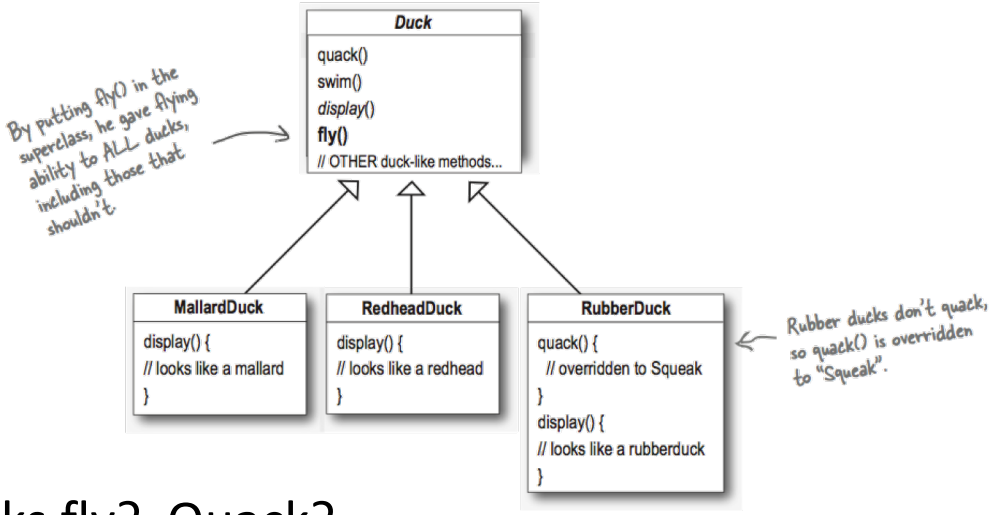
Making Ducks Fly

- Need to add code to make ducks fly in the simulation …



Making Ducks Fly: Problem

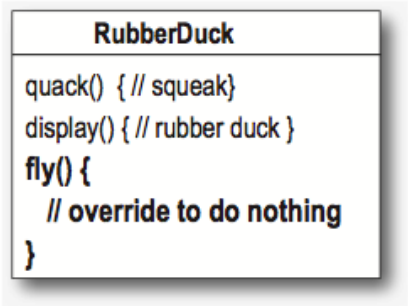
- The company decides to add a `RubberDuck` to the game as an Easter egg



- Should all ducks fly? Quack?

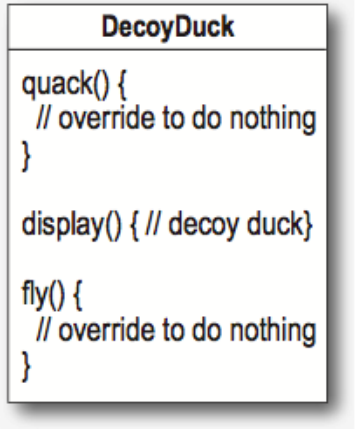
Making Ducks Fly: Solution

- Solutions?
  - We could override the `fly` method in `RubberDuck` to do nothing …



Making Ducks Fly: Solution

- This isn't a horrible solution, but what about when we add a `DecoyDuck` class for hunters in the game?
  - And decoys should neither quack nor fly …

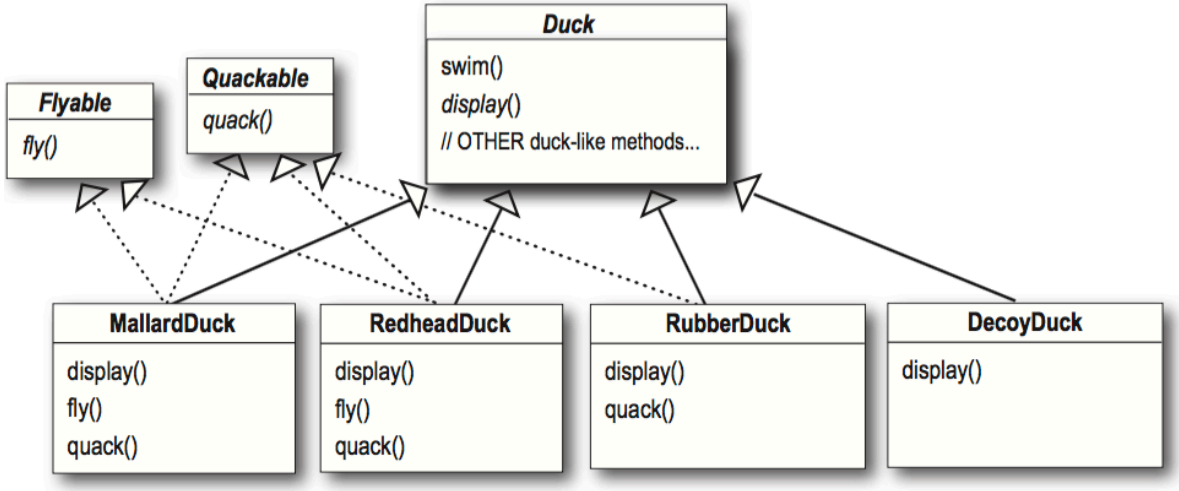


Making Ducks Fly: Solution

- Inheritance probably isn't the answer in this case
- As we add new types of ducks, we will have to examine and perhaps override fly and quack for each new class
- We 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)



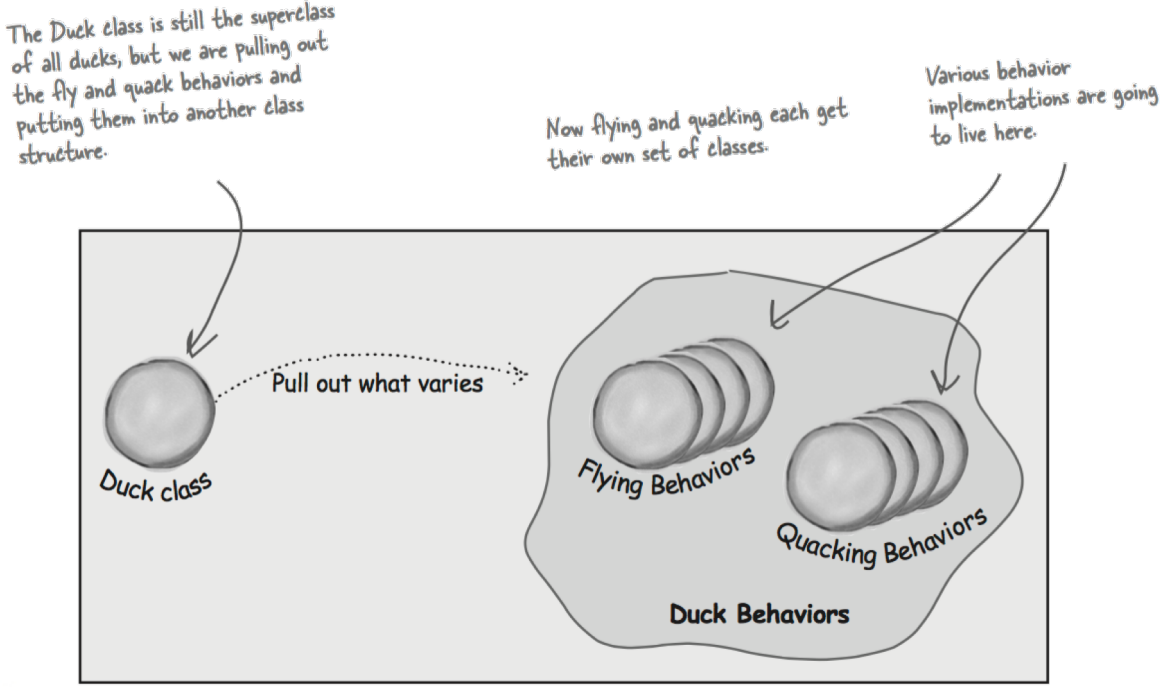
- Has this solved the problem?

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

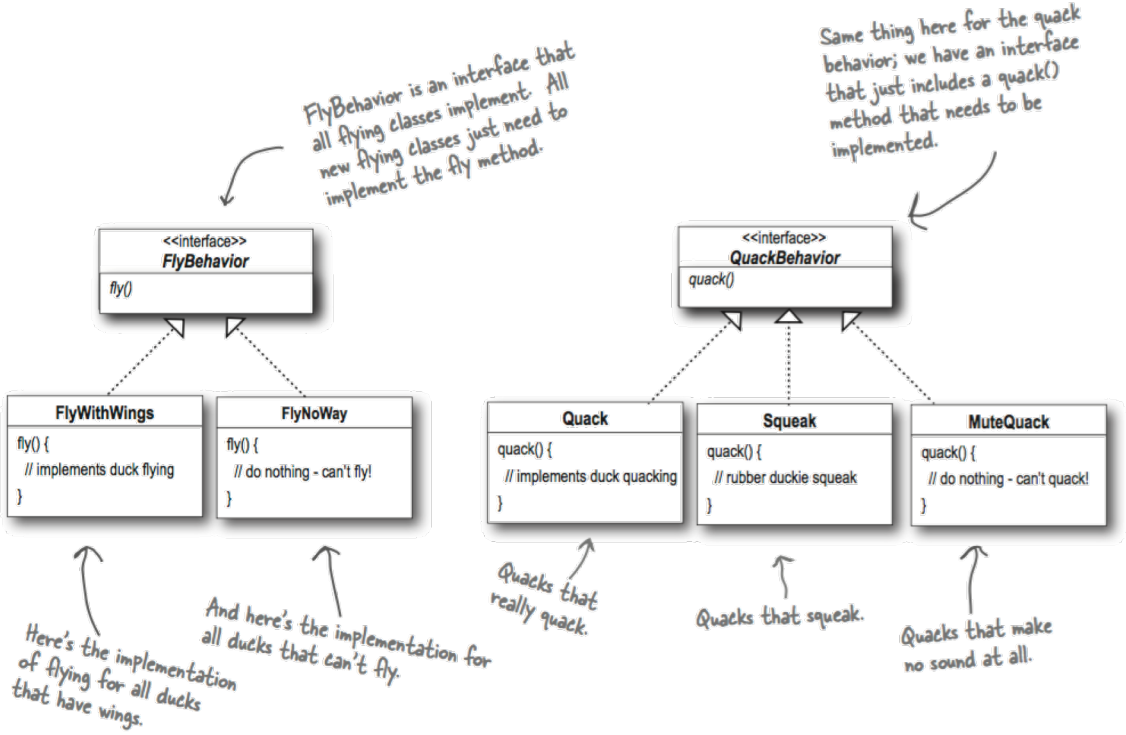
- We need to keep these considerations in mind when designing the system …

Encapsulate What Varies



Encapsulate What Varies

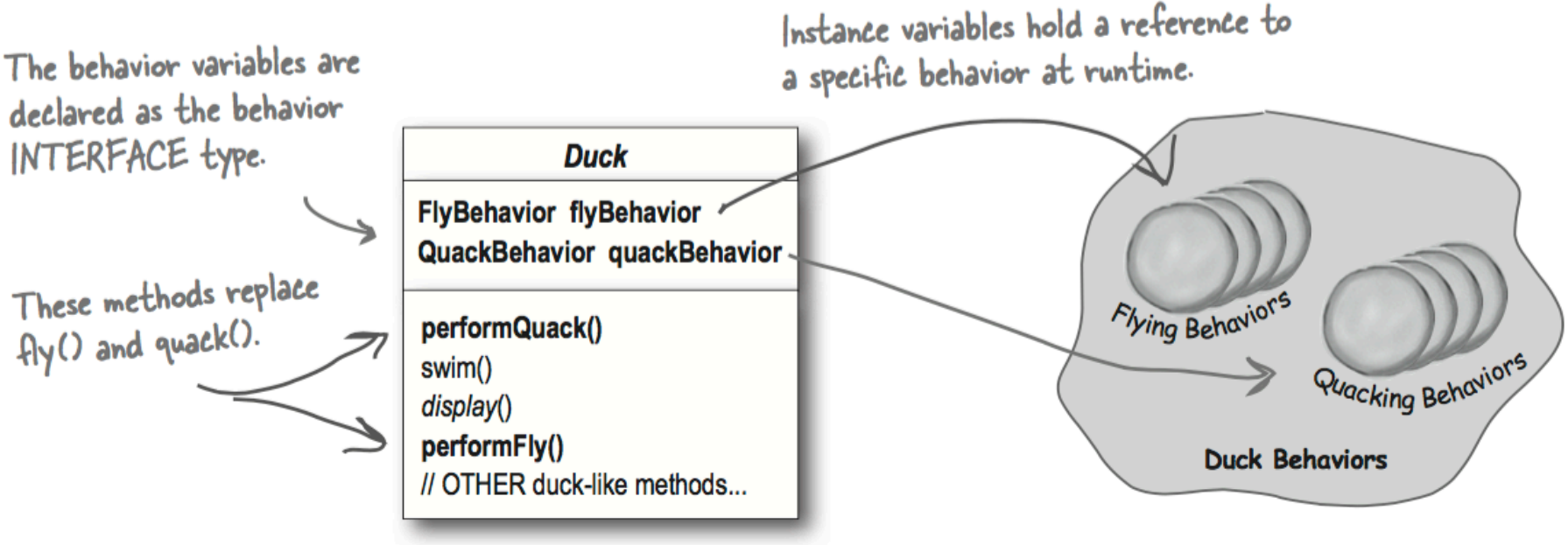
- We create an interface/abstract class for flying and quacking behaviour



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



Encapsulate What Varies

```
class FlyBehaviour
{
   public:
      virtual void fly() = 0;
};
class FlyWithWings : public FlyBehaviour
{
   public:
      virtual void fly() {
        cout << "I'm flying with my wings!" << endl;
      }
};
class FlyNoWay : public FlyBehaviour
{
   public:
      virtual void fly() {
         cout << "I can't actually fly!" << endl;
      }
};
```

Encapsulate What Varies

```
class Duck {
   public:
      Duck() { }
      ~Duck() {
         delete this->_flyBehaviour;
         delete this->_quackBehaviour;
      }
      void performFly() {
         this->_flyBehaviour->fly();
      }
      void performQuack() {
         this->_quackBehaviour->quack();
      }

   protected:
      FlyBehaviour* _flyBehaviour;
      QuackBehaviour* _quackBehaviour;
};
```

Encapsulate What Varies

```
class MallardDuck : public Duck
{
   public:
      MallardDuck() {
         this->_quackBehavior = new Quack();
         this->_flyBehavior = new FlyWithWings();
      }
};
```

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 necessarily (or heavily) modifying our duck classes
  - Can dynamically change behaviours at run-time

Code to an Interface, Not an Implementation

Design Principle:
Code to an Interface, Not an Implementation

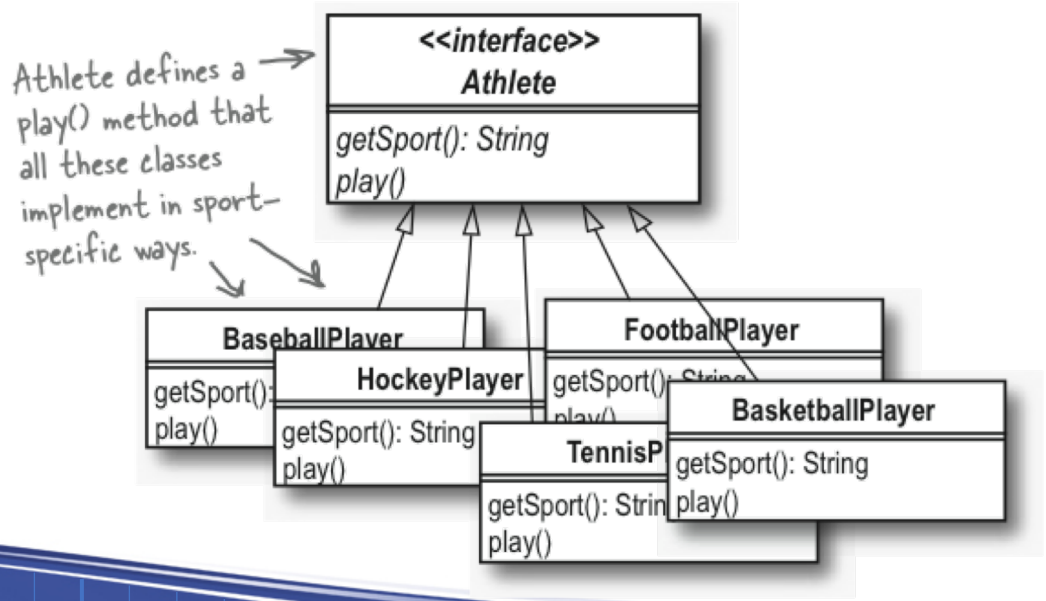
When faced with the choice between interacting with subclasses or interacting with a supertype, choose the supertype. Your code will be easier to extend and will work with all of the interface's subclasses – even those not yet created.

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 a team of athletes

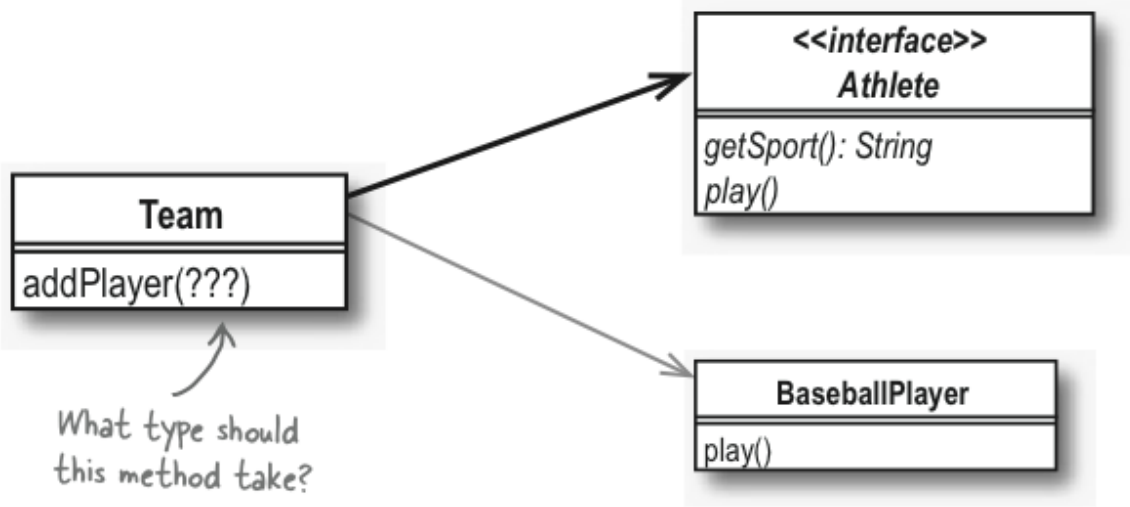
- One option: create a class `Team` and then subclass that for each specific team type:
  - `BaseballTeam`
  - `FootballTeam`
  - `TennisTeam`
  - …

Modelling Athletes and Teams

- Issues
  - Creates a large inheritance hierarchy
    (KISS principle – Keep It Simple, Stupid)
  - Results in extensive code duplication
    (DRY principle – Don’t Repeat Yourself)
  - Have to add a new subclass for each new sport we wish to support

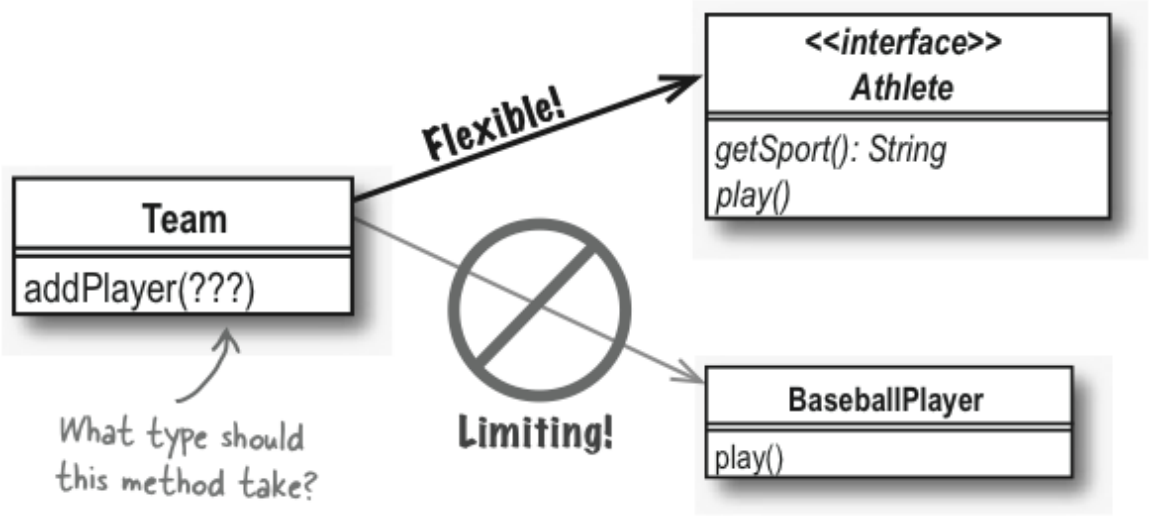
Code to an Interface, Not an Implementation

- 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 duplicate in each class

Favour Composition Over Inheritance

Design Principle:

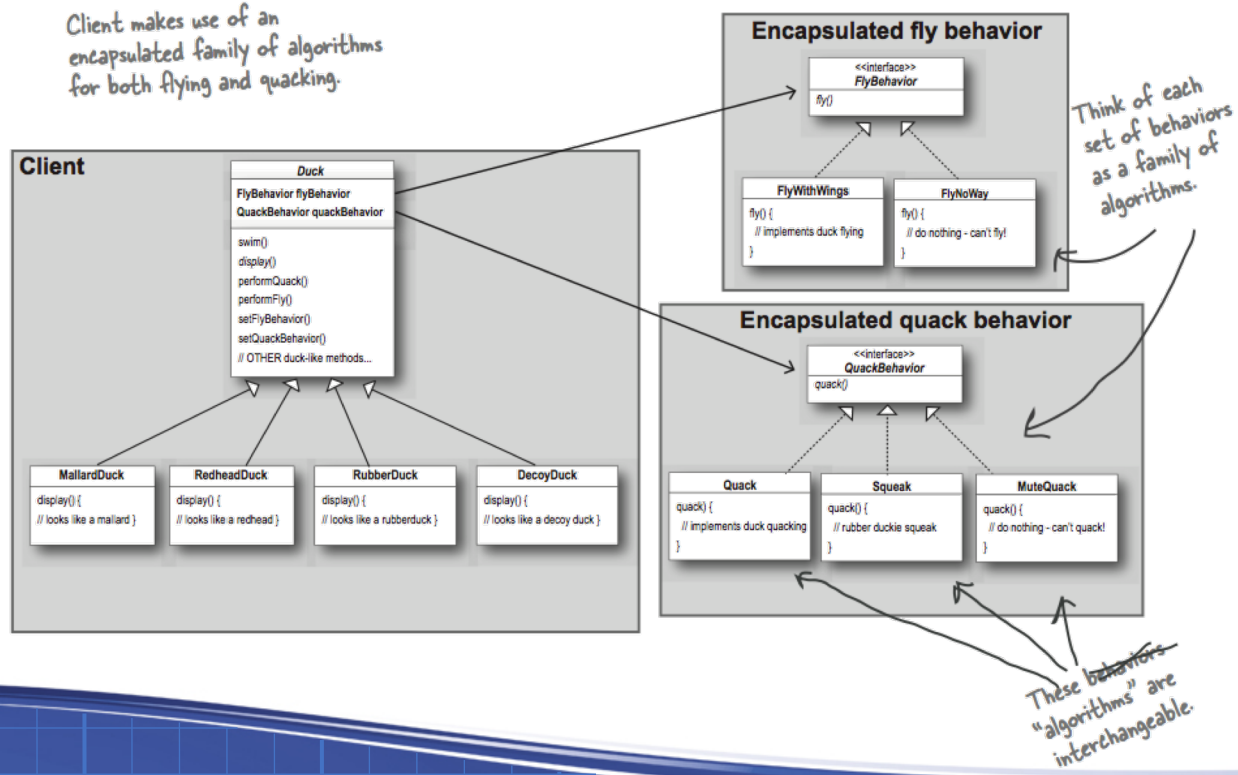
Favour Composition Over Inheritance

By favouring delegation, composition, and aggregation over inheritance, we can produce software that 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



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

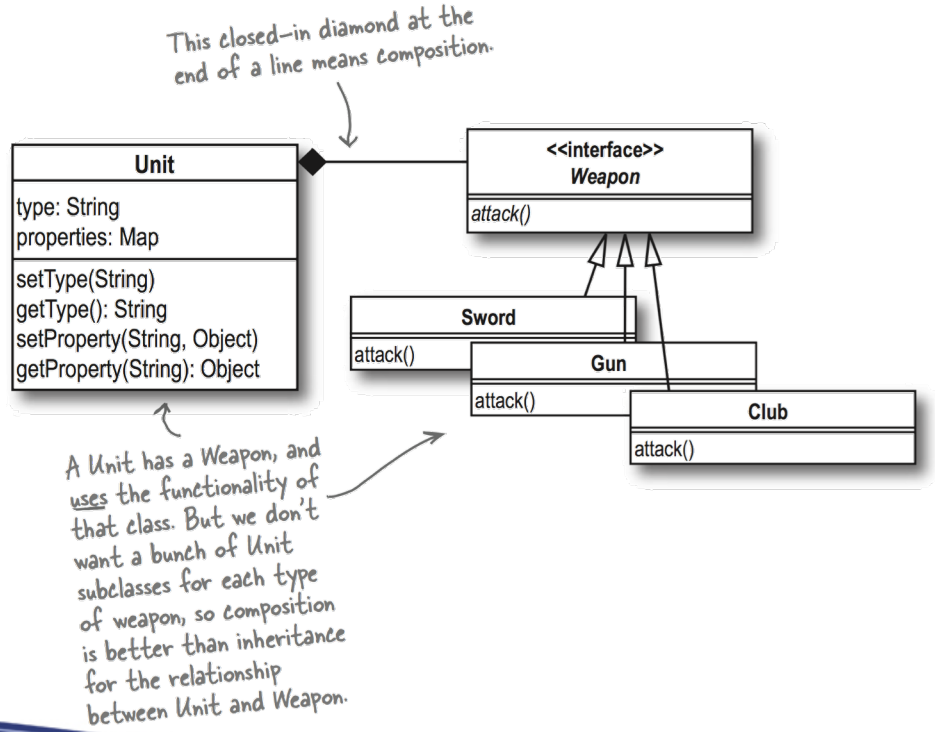
Rule of thumb:

- 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

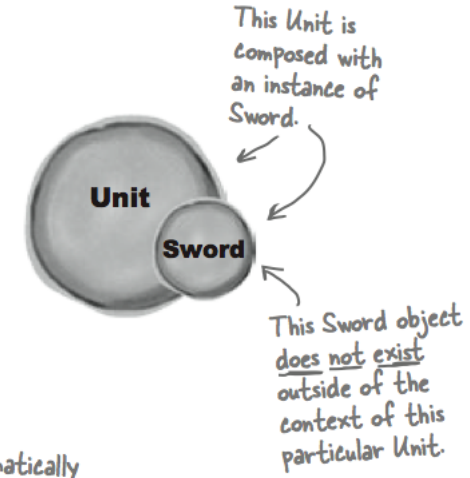
- An object composed of other objects *owns* those objects
- When the object is destroyed, so are all the objects of which it is composed

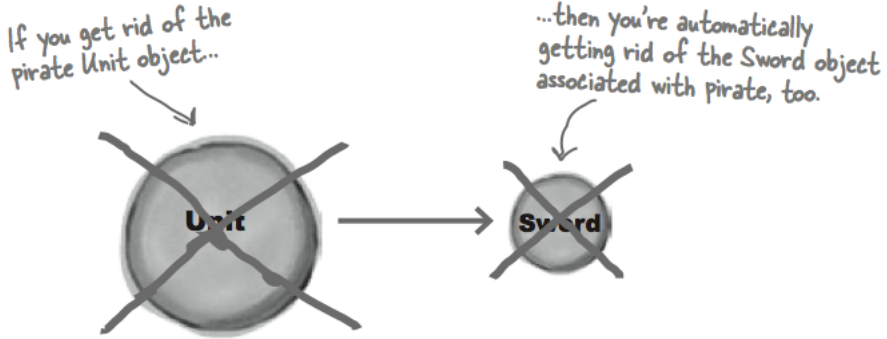
Composition



Composition

```
Unit* pirate = new Unit();
pirate->SetProperty("weapon", new Sword());
```



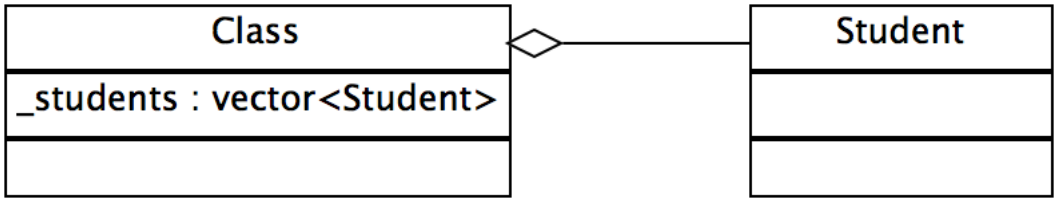


Composition

- 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, you need to track a weapon outside of a unit
  - If not, use **composition** (*OWNS-A*)
  - If so, use **aggregation** (*HAS-A*)

Aggregation

- An object comprised of other objects *uses* those objects
- Those objects exist outside of the object
- When the object is destroyed, the objects that comprise 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 Design Principles

- **S**ingle Responsibility Principle
- **O**pen/Closed Principle
- **L**iskov Substitution Principle
- **I**nterface Segregation Principle
- **D**ependency Inversion Principle

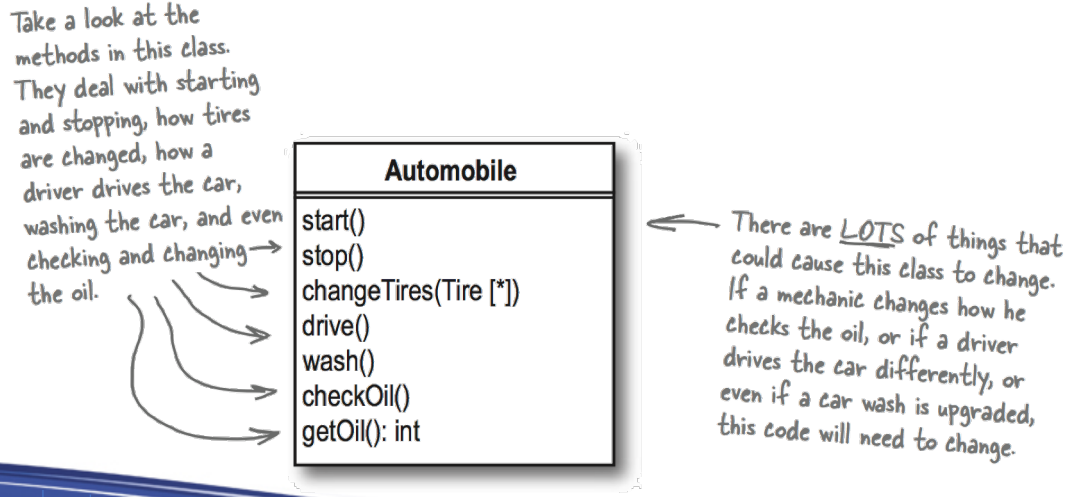
SOLID: Single Responsibility Principle

Design Principle:
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

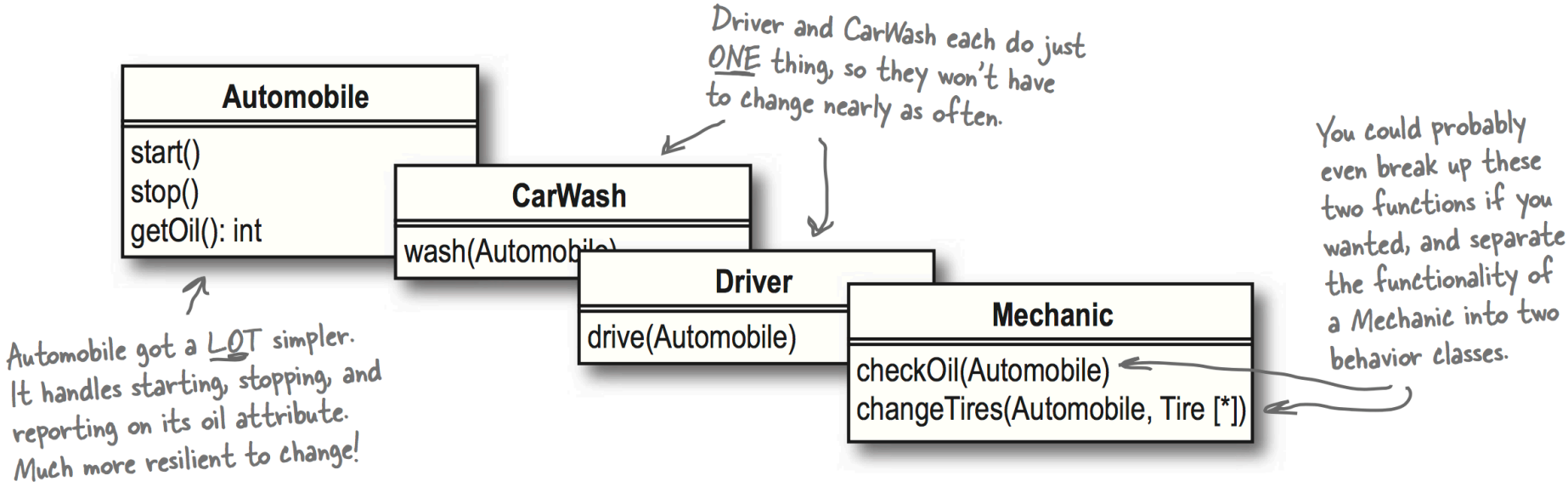
- Every object in a system should have a single responsibility
- Another way to think about a responsibility is as a reason to change



SOLID: Single Responsibility Principle

- When a class has more than one reason to change, it might be trying to do too much
- In such a case, we should try to break 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



SOLID: Single Responsibility Principle

- Benefits:
  - Doing this minimizes the chance that a class will need to be changed by reducing the number of things in the class that can change
  - Generally, this also results in high cohesion as the elements of the class belong together in a stronger way
  - Achieving higher cohesion generally increases reusability, robustness, understandability, and so on

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

Design Principle:
Open/Closed Principle

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

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

- **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 a different behaviour, we should extend the class

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

- **Open for extension:** Behavioural changes that may be required should be accomplished through inheritance or other means (e.g. the Observer pattern … more on this later)

- We should not touch our existing, well-tested code!

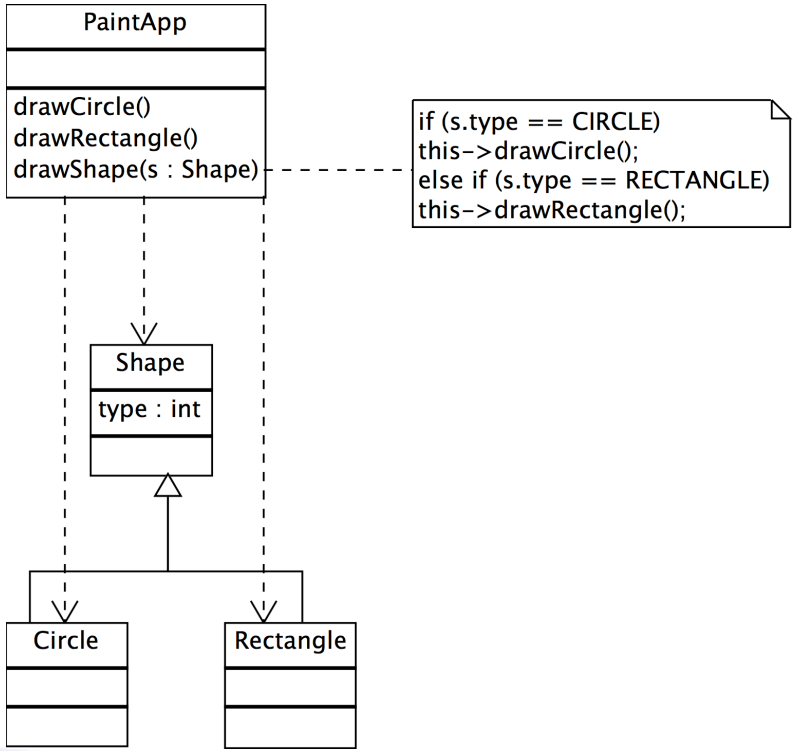
SOLID: Open/Closed Principle

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

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

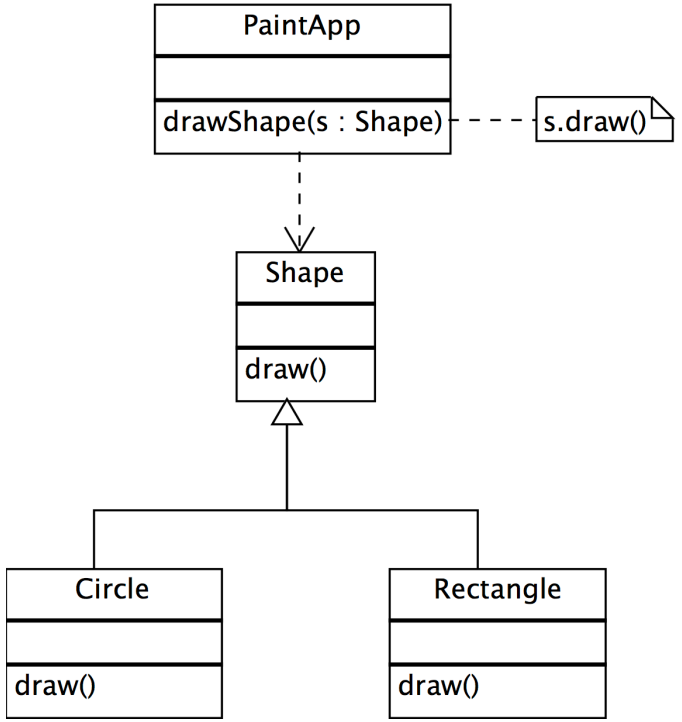
SOLID: Open/Closed Principle

- This example violates the open/closed principle
- What happens when we need to add a new `Shape` subclass?
- We would need to change `PaintApp` to accommodate the new shape



SOLID: Open/Closed Principle

- A better approach is to refactor the code and take advantage of the inheritance hierarchy that is in place
- We can now create new `Shape` subclasses without requiring changes to `PaintApp`



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

- Benefits:
  - This effectively allows the behaviour of a class to be modified without touching its source code
  - In doing so, we don’t risk breaking well-tested code
  - Instead, we only modify existing code to fix errors; new/modified features require extension

SOLID: Liskov Substitution Principle

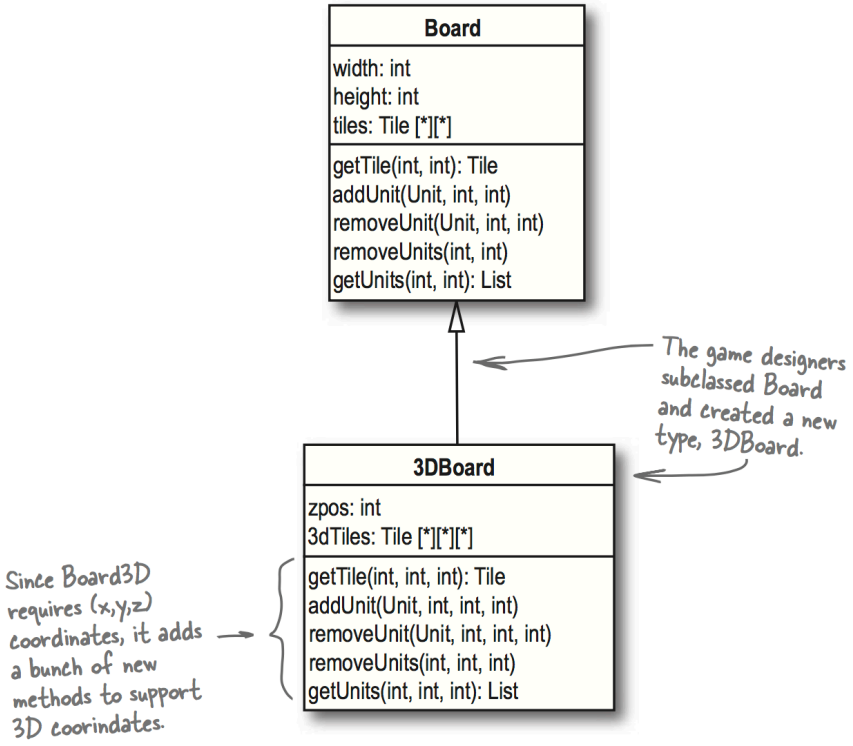
Design Principle:
Liskov Substitution Principle

Subtypes must be substitutable for their base types.

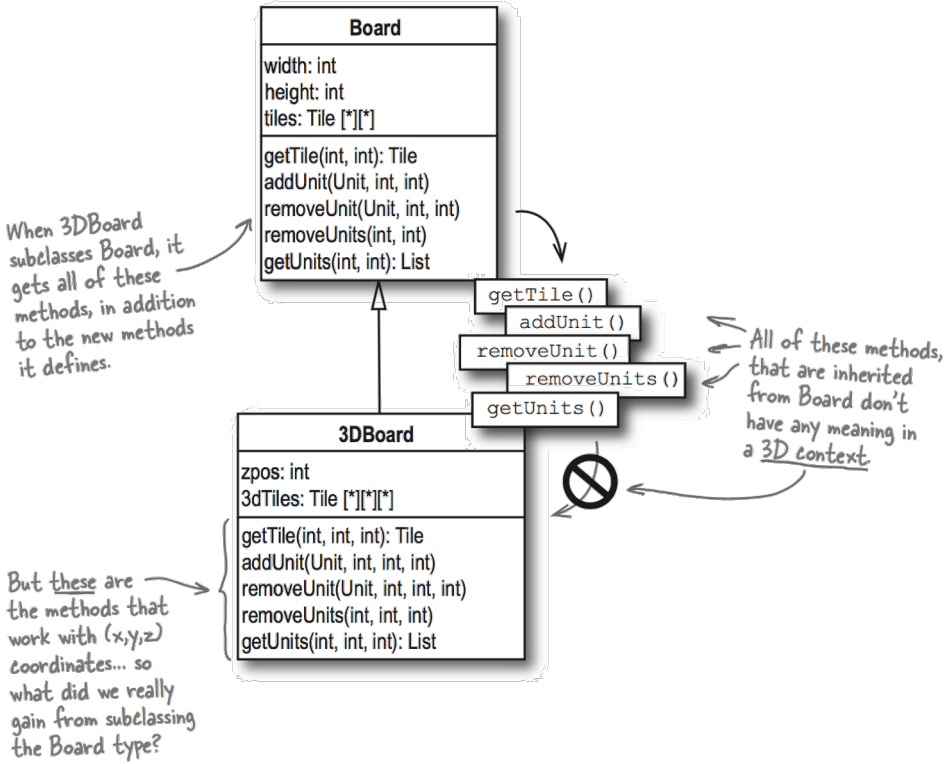
SOLID: Liskov Substitution Principle

- The Liskov Substitution Principle 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



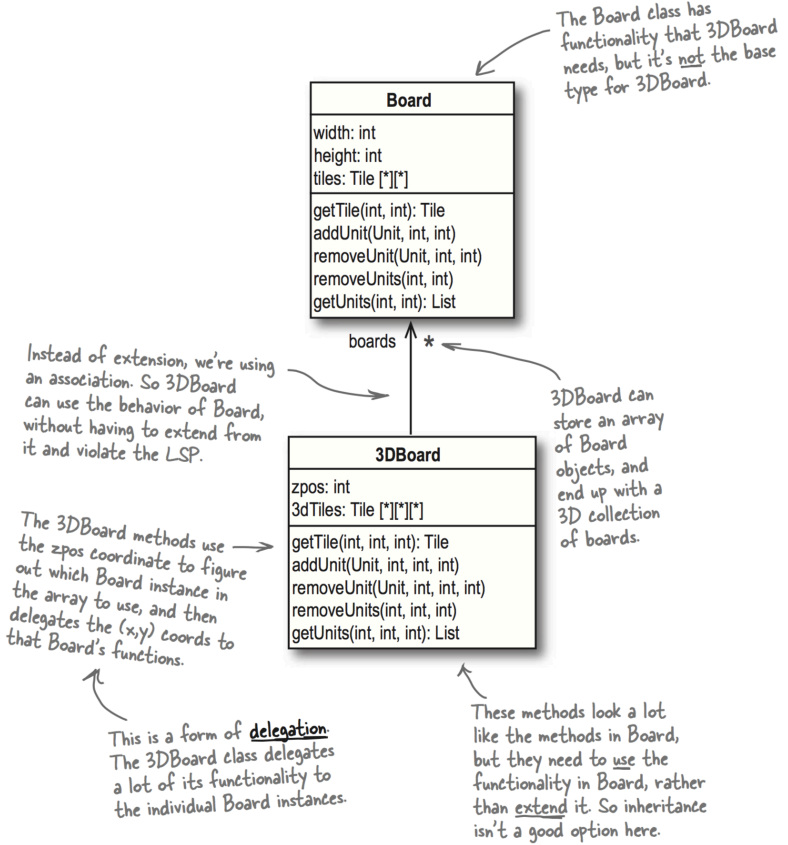
SOLID: Liskov Substitution Principle



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 units = board->getUnits(8, 4);
  ```
- What does this method mean on `3DBoard`?
  - The Liskov Substitution Principle states that any method on `Board` should be usable on `3DBoard` without affecting correctness
  - `3DBoard` really is not substitutable for `Board`: none of the methods on `Board` will work in a 3D environment

SOLID: Liskov Substitution Principle



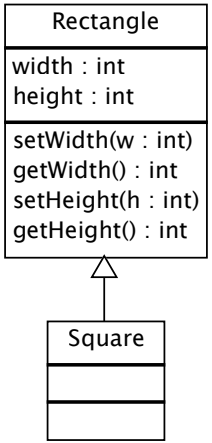
SOLID: Liskov Substitution Principle

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

```
class Rectangle {
public:
  void setWidth(int w) {
     this->width = w;
  }
  void setHeight(int h) {
     this->height = h;
  }

// ...
protected:
  int width;
  int height;
};
```

SOLID: Liskov Substitution Principle

- A few months later we realize we need to add a `Square`
  - Does the following hierarchy violate the Liskov Substitution Principle?
  - After all a square IS-A rectangle …



SOLID: Liskov Substitution Principle

- This one is more subtle and requires more consideration
- The first clue that something is wrong:
  - A `Square` does not need both `width` and `height` members, but it inherits them anyways
  - Really, a `Square` only needs a single side length
  - It's not a big deal though … memory is cheap these days, we have lots of RAM, and we won't be making tons of `Square` objects anyways

SOLID: Liskov Substitution Principle

- The second clue:
  - The `setWidth` and `setHeight` methods inherited from `Rectangle` will not be appropriate for `Square`
  - That's okay; we can override them …

```
class Square : public Rectangle {
public:
   void setWidth(int w) {
      this->width = this->height = w;
   }
   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`

```
void f(Rectangle& r)
{
   r.setWidth(32);  // calls Rectangle::setWidth
}
```

SOLID: Liskov Substitution Principle

- No worries … we can fix this too by making `setWidth` and `setHeight` virtual in the `Rectangle` base class
- The fact that we have to violate our Open/Closed Principle to make this work is another hint that something is wrong …

```
class Rectangle {
public:
  virtual void setWidth(int w) {
     this->width = w;
  }
  virtual void setHeight(int h) {
     this->height = h;
  }
};
```

SOLID: Liskov Substitution Principle

- Fourth clue:
  - Our function `g` works for `Rectangle` but not for `Square`

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

- The Liskov Substitution Principle 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
  - In the end, the behaviour of a `Square` is not consistent with the behaviour of a `Rectangle`
  - Behaviour is what software is all about
- Conclusion: IS-A does not tell the whole story
- We should use inheritance when one object behaves like another, rather than just when the IS-A relationship applies

SOLID: Interface Segregation Principle

Design Principle:
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

- A “fat interface” is supplied by a class whose interface is not cohesive
  - It has many responsibilities and is unfocused and hard to understand/modify
- The Interface Segregation Principle 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 abstract class `Door`:

```
class Door {
   public:
      virtual void lock()  = 0;
      virtual void unlock() = 0;
      virtual bool isDoorOpen() = 0;
};
```

SOLID: Interface Segregation Principle

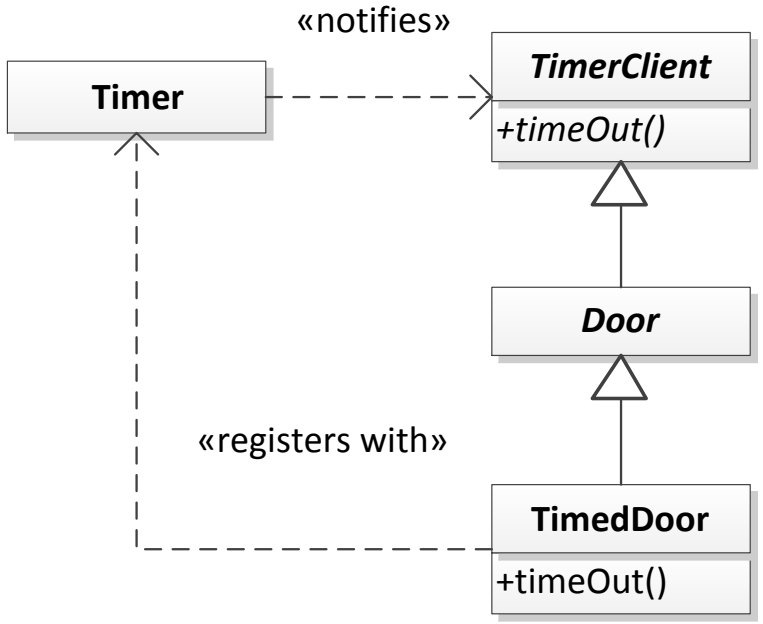
- We wish to have a class `TimedDoor` that will sound an alarm if left open for too long
- First, we will create a class `Timer` which `TimerClients` can register with to receive notifications about timeouts

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

class TimerClient {
   public:
      virtual void timeOut() = 0;
};
```

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` extend `TimerClient` , so that a new derived class `TimedDoor` will be able to register itself with `Timer`

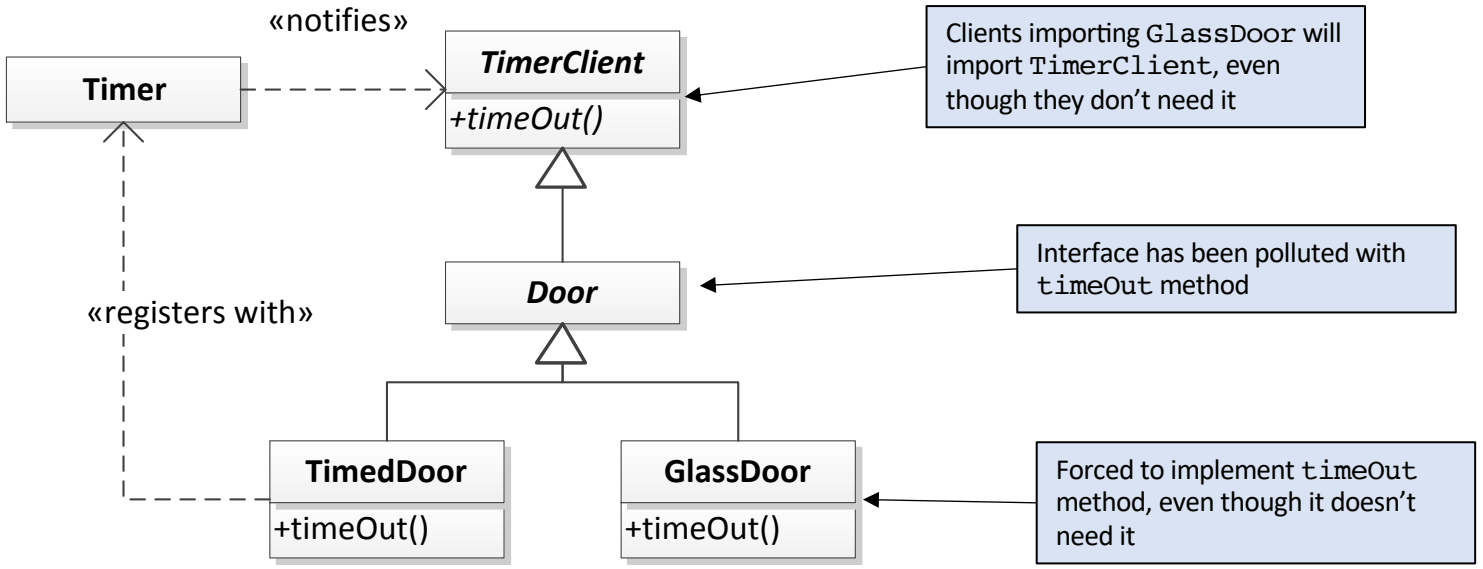
SOLID: Interface Segregation Principle



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 `#include` those timing-free doors, they will include the definition of the `TimerClient` class even though it won't be used

SOLID: Interface Segregation Principle



SOLID: Interface Segregation Principle

- If we continue this practice, then each time we need a new interface, we will have to add it to the base class, further polluting its interface
- We will have to go back and implement the new interface methods in every subclass, violating the Open/Closed Principle

SOLID: Interface Segregation Principle

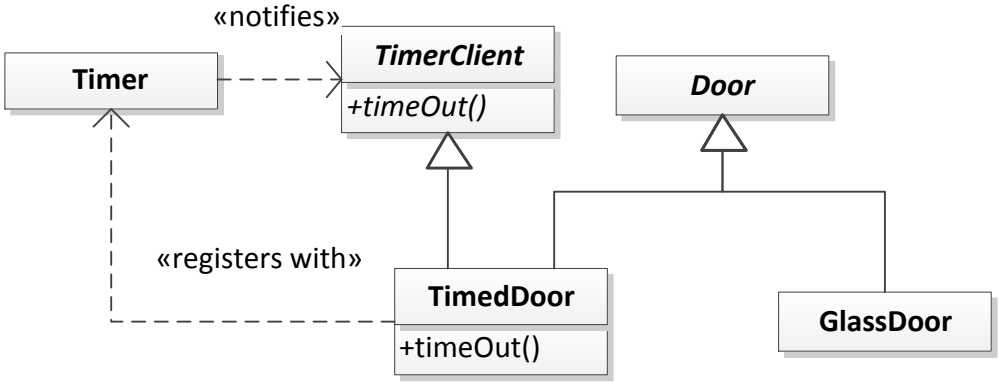
- `Door` and `TimerClient` provide interfaces used by completely different clients:
  - `Timer` uses `TimerClient`
  - Classes that manipulate doors use `Door`
  - If the clients are separate, then so, too, should the interfaces be separate

SOLID: Interface Segregation Principle

- 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 multiple inheritance:



- The Adapter design pattern can also be used to solve this sort of problem – more on this pattern later

SOLID: Dependency Inversion Principle

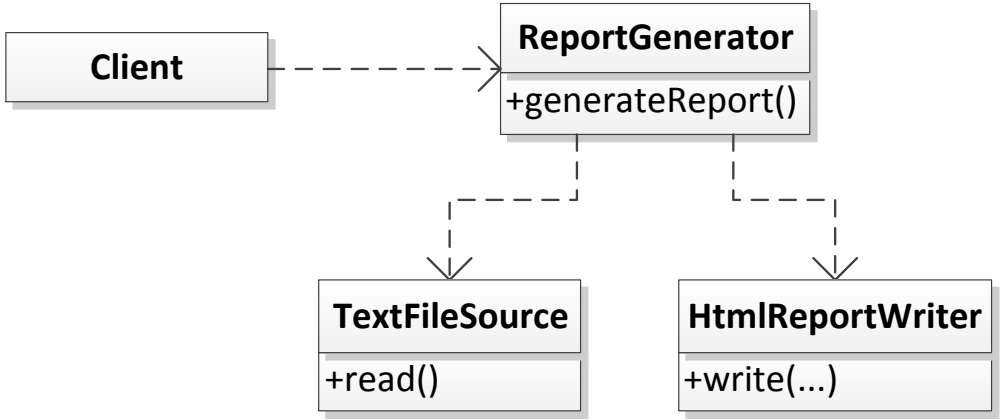
Design Principle:
Dependency Inversion Principle

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.

SOLID: Dependency Inversion Principle

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



SOLID: Dependency Inversion Principle

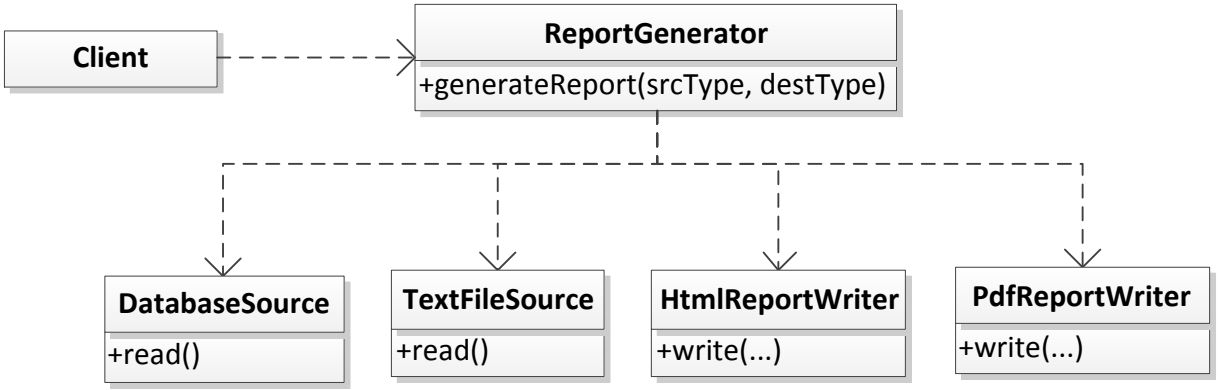
```
class ReportGenerator {
   public:

   ...

   void generateReport() {
      TextFileSource* src = new TextFileSource(this->_inFile);
      HtmlReportWriter* dest = new HtmlReportWriter(this->_outFile);

      string line;
      while (line = src->read()) {
         // Compile report
      }

      // Write report in HTML format
      dest->write(...);
   }
};
```

SOLID: Dependency Inversion Principle

- `TextFileSource` and `HtmlReportWriter` are certainly reusable
- But, we 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 – it would be nice to reuse `ReportGenerator`
- `ReportGenerator` is dependent on `TextFileSource` and `HtmlReportWriter`, so this is not possible

SOLID: Dependency Inversion Principle

- We could modify `generateReport` to accept the type of source and destination to use …



SOLID: Dependency Inversion Principle

```
class ReportGenerator {
   public:

   ...

      void generateReport(string srcType, string destType) {
         if ((srcType == "text") && (destType == "html")
            generateHtmlReportFromText();
         else if ((srcType == "text") && (destType == "pdf"))
            generatePdfReportFromText();
         else if ((srcType == "db") && (destType == "html"))
            generateHtmlReportFromDb();
         else if ((srcType == "db") && (destType == "pdf"))
            generatePdfReportFromDb();
         else
                  // throw exception
      }
};
```

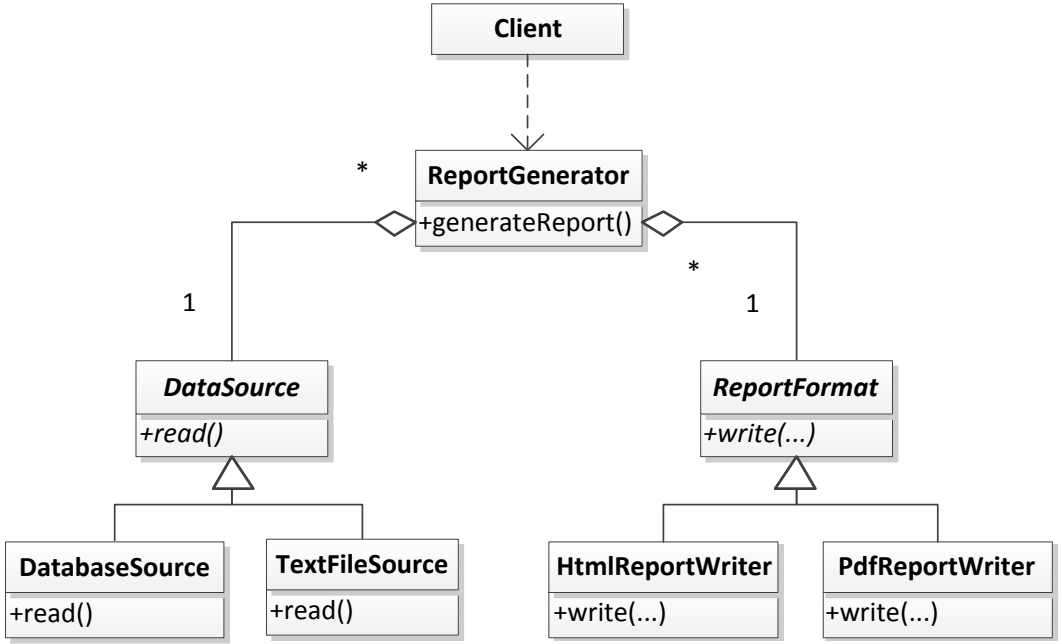
SOLID: Dependency Inversion Principle

- 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
- This also results in a rigid and fragile system
  - **Rigid**: the 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

SOLID: Dependency Inversion Principle

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

SOLID: Dependency Inversion Principle



SOLID: Dependency Inversion Principle

```
class ReportGenerator {
   public:

   ...

   void generateReport() {
      string line;
      while (line = this->_src->read()) {
         // Compile report
      }
      // Write report
      this->_dest->write(...);
   }

   private:
      DataSource* _src;
      ReportFormat* _dest;
};
```

Summing up SOLID, courtesy of globalnerdy.com

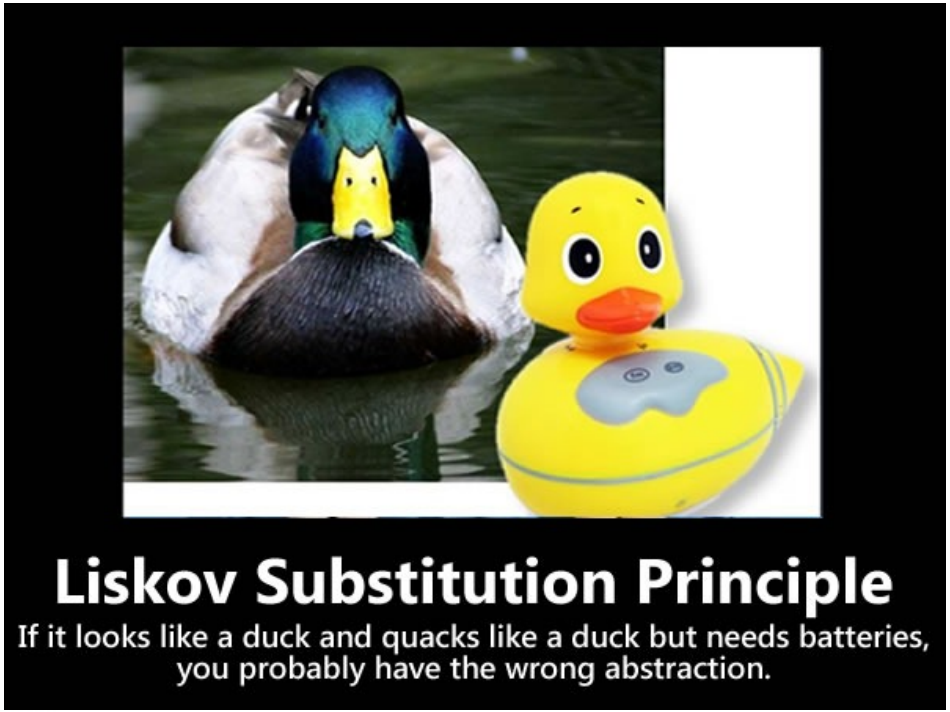
SOLID


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.

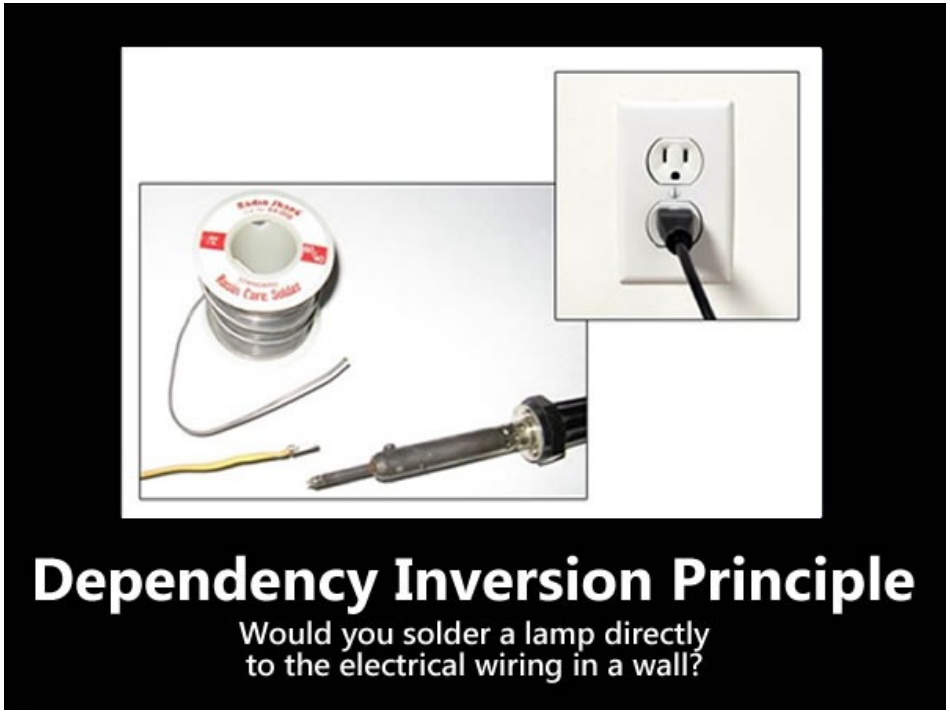
SOLID


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?