

# Design Principles

Interfaces Over Implementations,  
Composition Over Inheritance

# Code to an Interface, Not an Implementation

## **Design Principle:**

### **Code to an Interface, Not an Implementation**

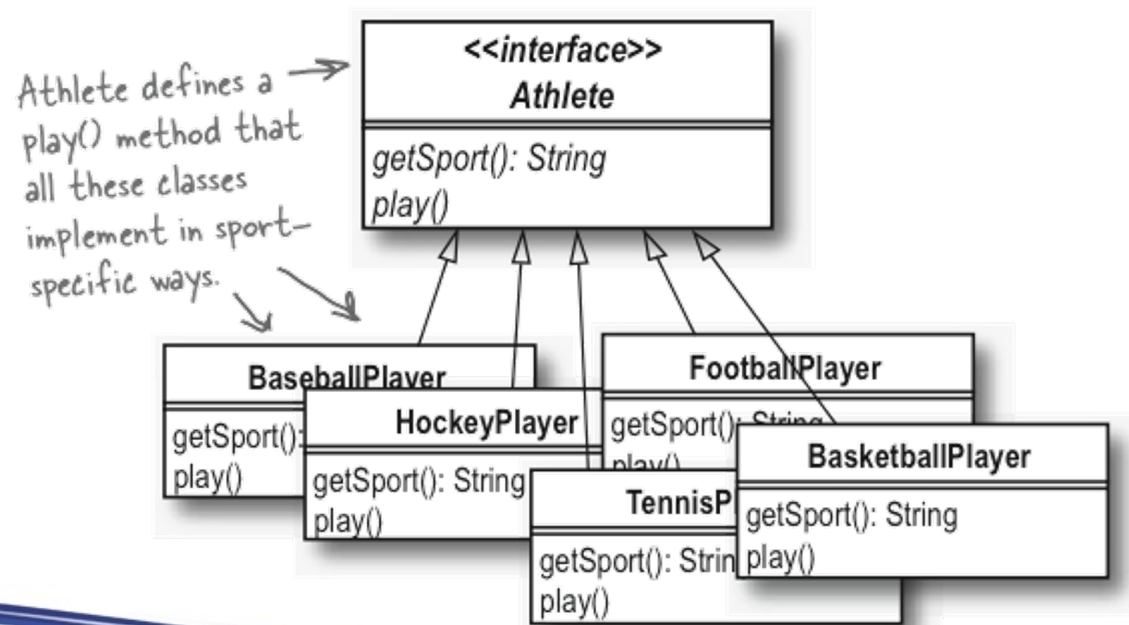
When faced with the choice between interacting with subclasses or interacting with a supertype, choose the supertype. Your code will be easier to extend and will work with all of the interface's subclasses – even those not yet created.

# 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

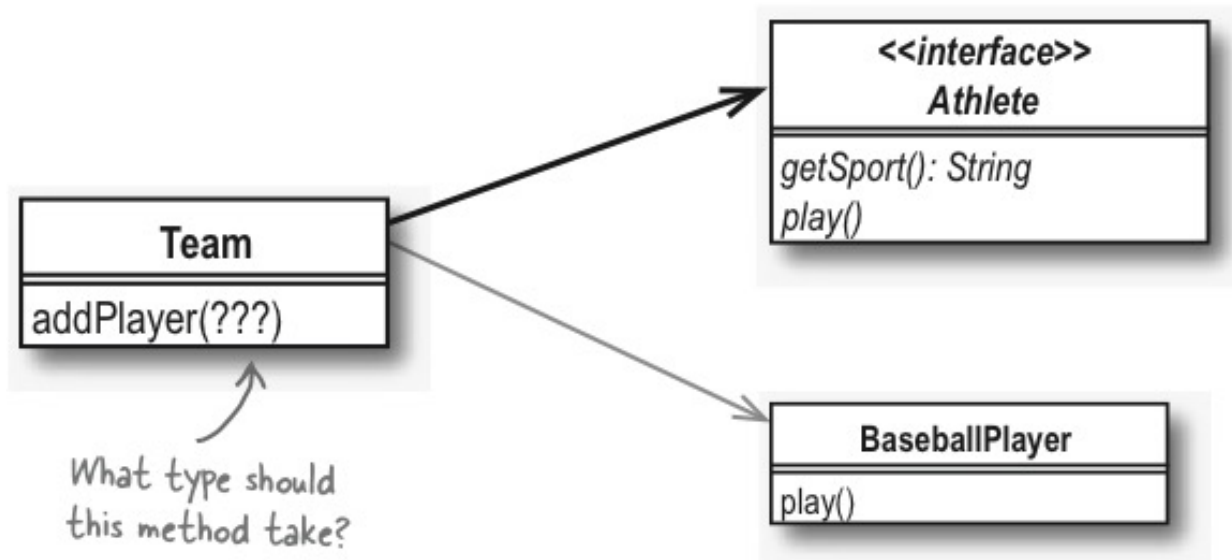
- We wish to model a team of athletes
- One option: create a class `Team` and then subclass that for each specific team type:
  - `BaseballTeam`
  - `FootballTeam`
  - `TennisTeam`
  - ...

# 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

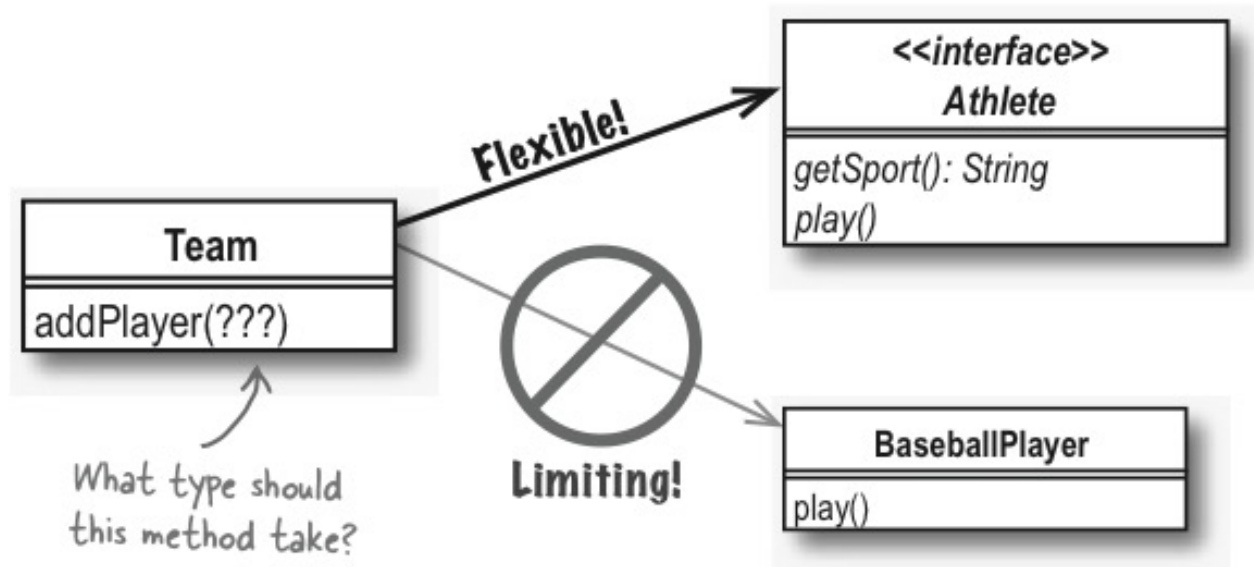
# Code to an Interface, Not an Implementation

- 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

*always choose base type at first.*

# 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.

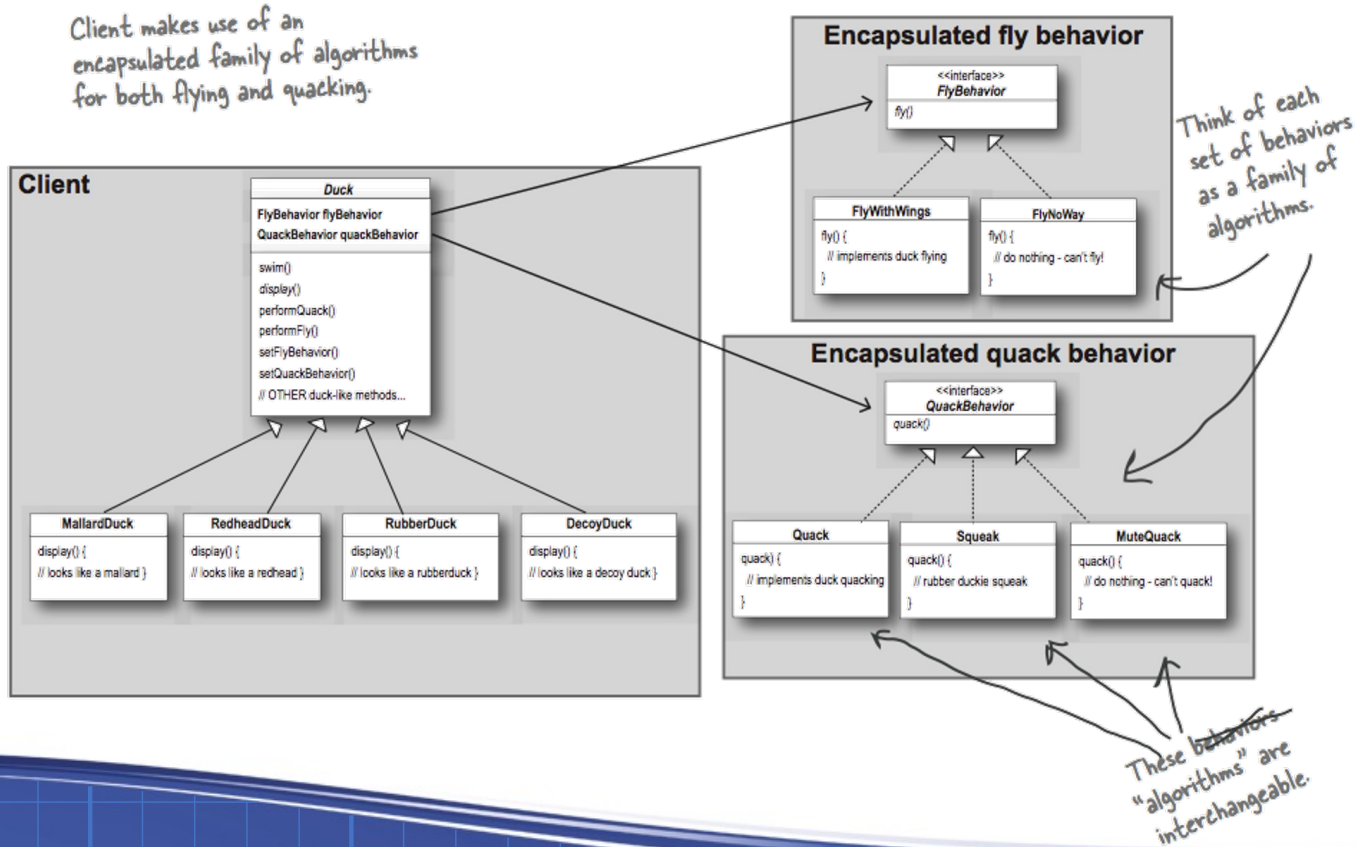
# Favour Composition Over Inheritance

- Inheritance establishes an IS-A relationship
- Composition/aggregation establish a HAS-A relationship – this can often be preferable

*something HAS A .... behavior, that is, adding feature to an object.*

- 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

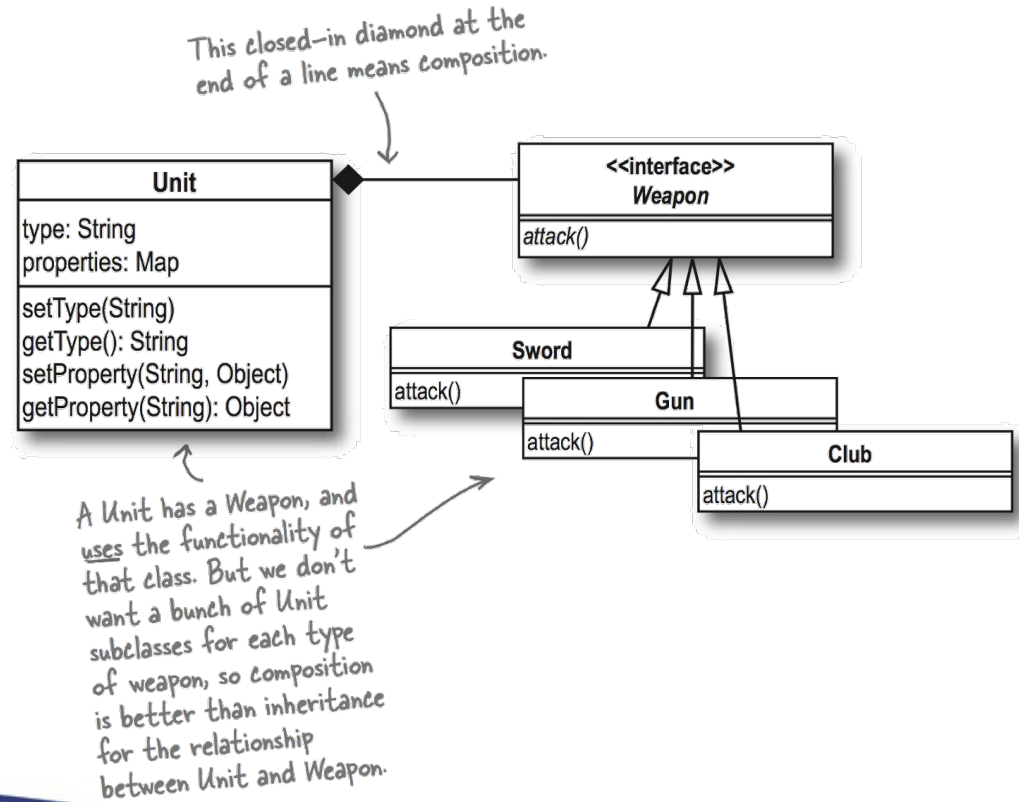
Rule of thumb:

- 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

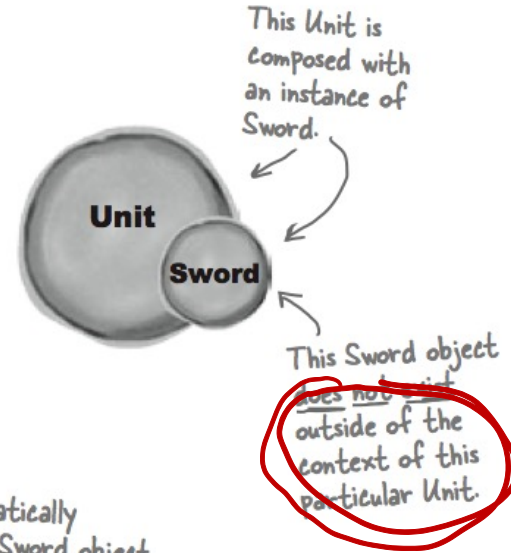
# Composition





# Composition

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



If you get rid of the pirate Unit object...



...then you're automatically getting rid of the Sword object associated with pirate, too.

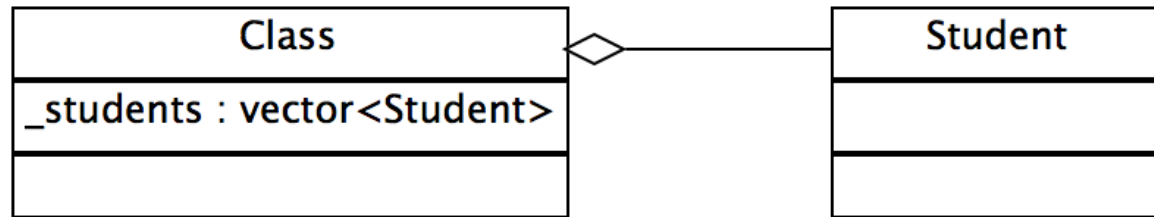


# Composition

- 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)  
~~composition~~
  - If yes, use *aggregation* (white diamond of life)  
aggregation