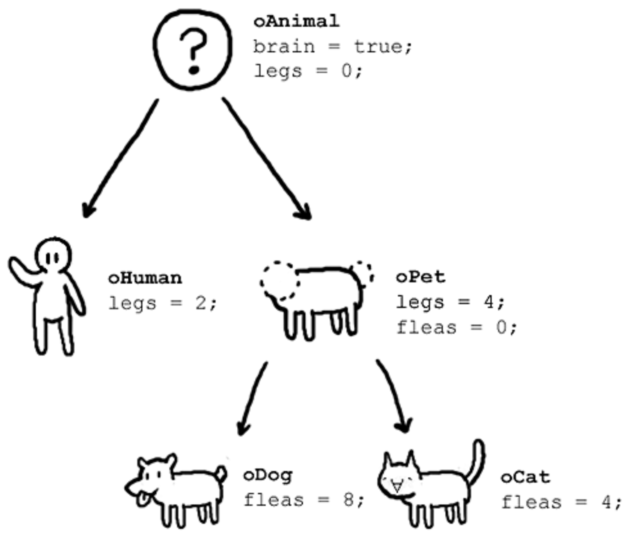
Topic 4

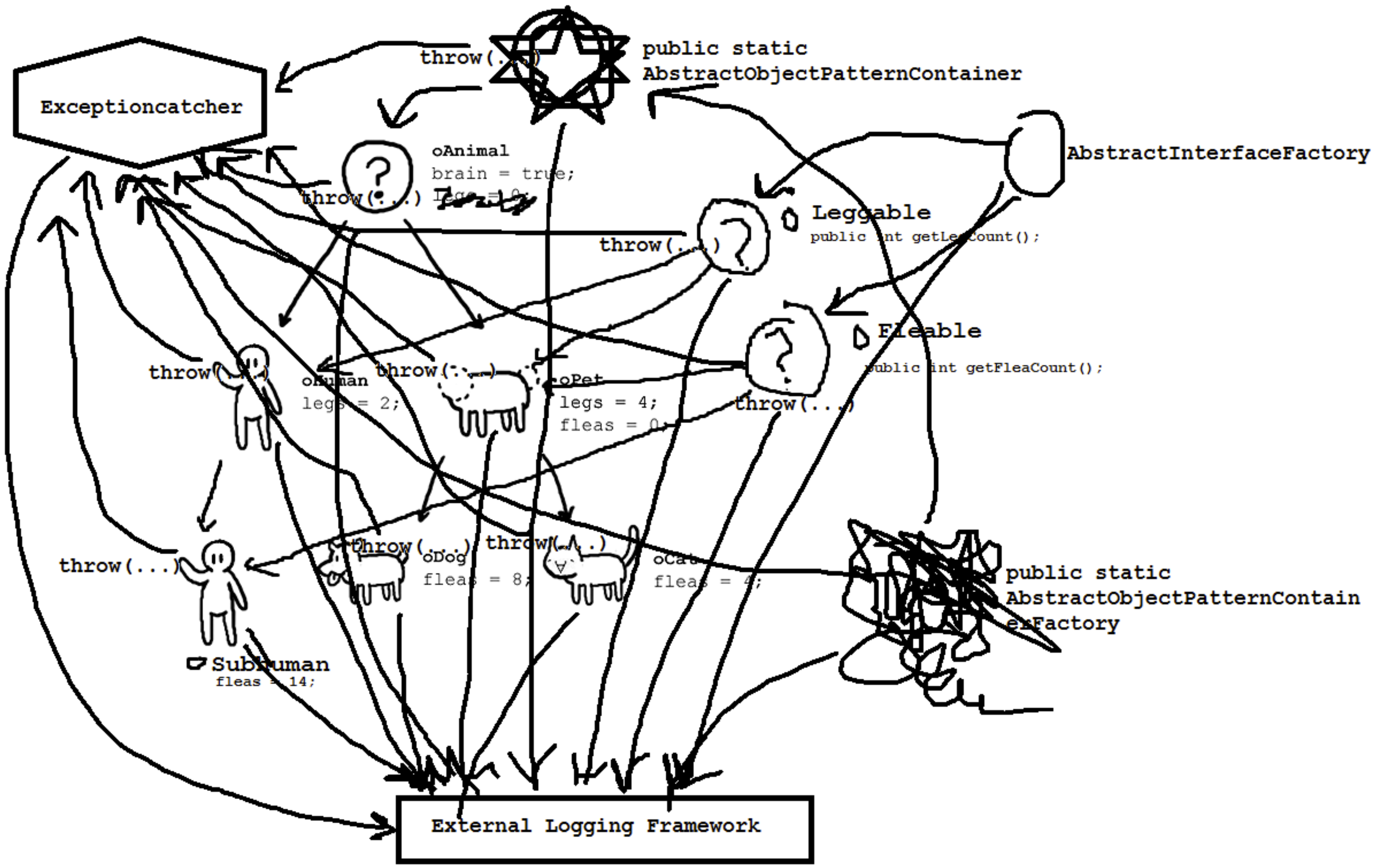
Software Design

Part 2

What OOP users claim



What actually happens



https://www.reddit.com/r/ProgrammerHumor/comments/418x95/theory_vs_reality/

Outline

- OO Design principles - Advanced
- OO Design patterns

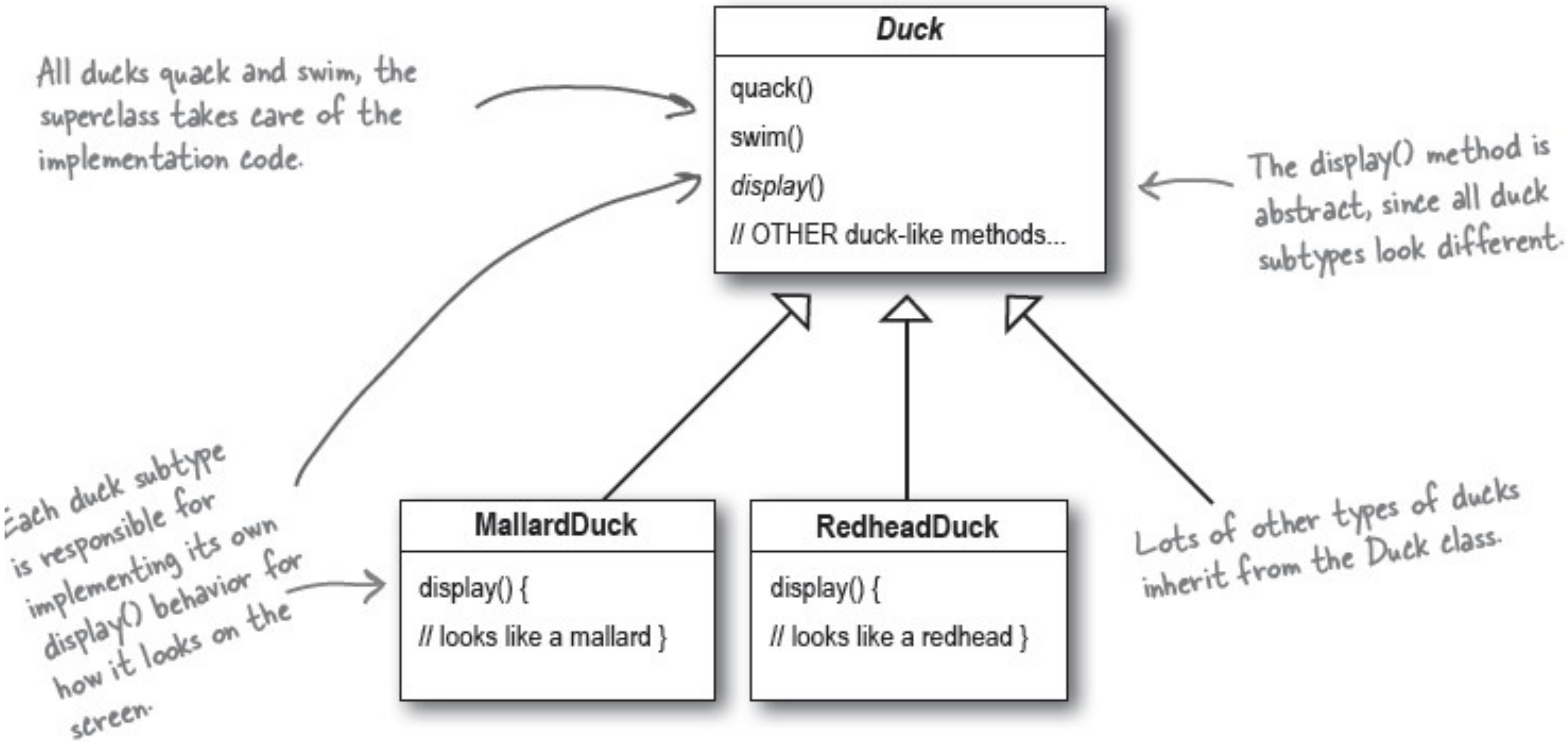
Design Principles - More Advanced

1. Encapsulate what varies
2. Code to an Interface, not an implementation
3. Favour composition over inheritance

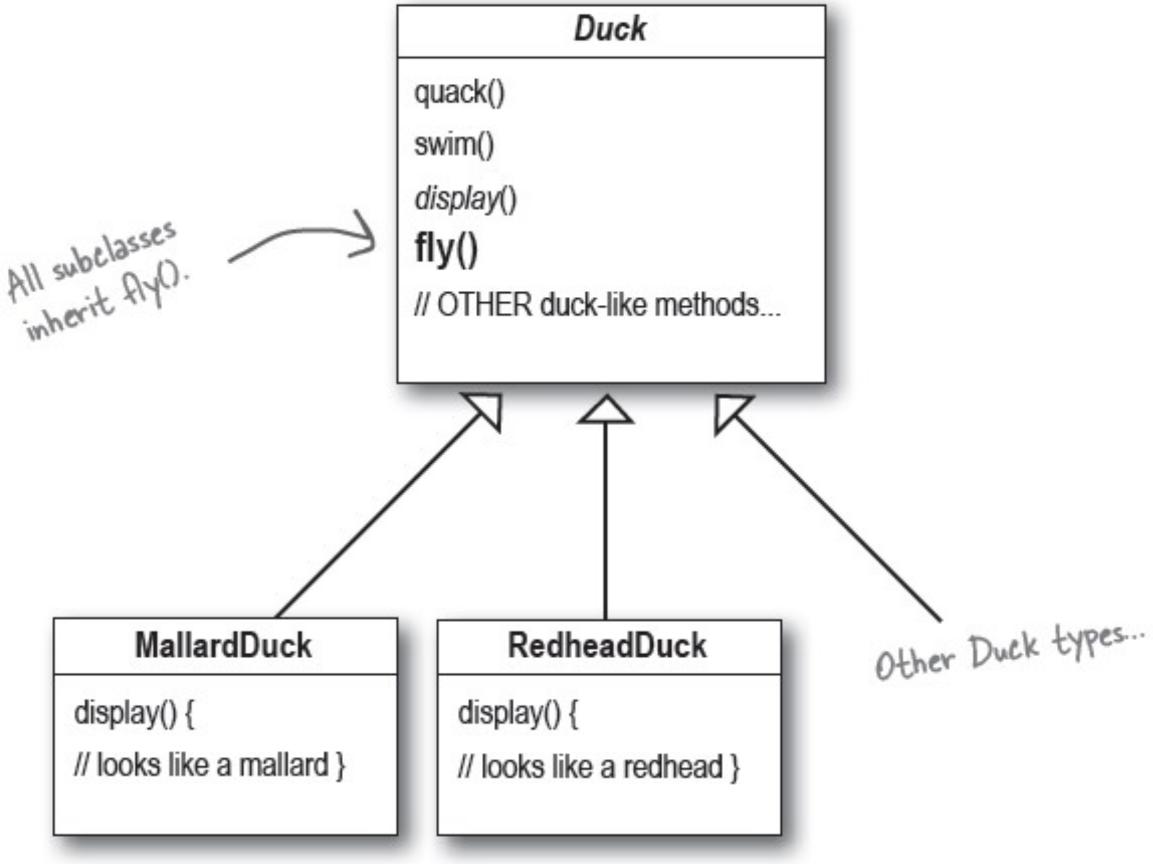
1. Encapsulate what varies

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

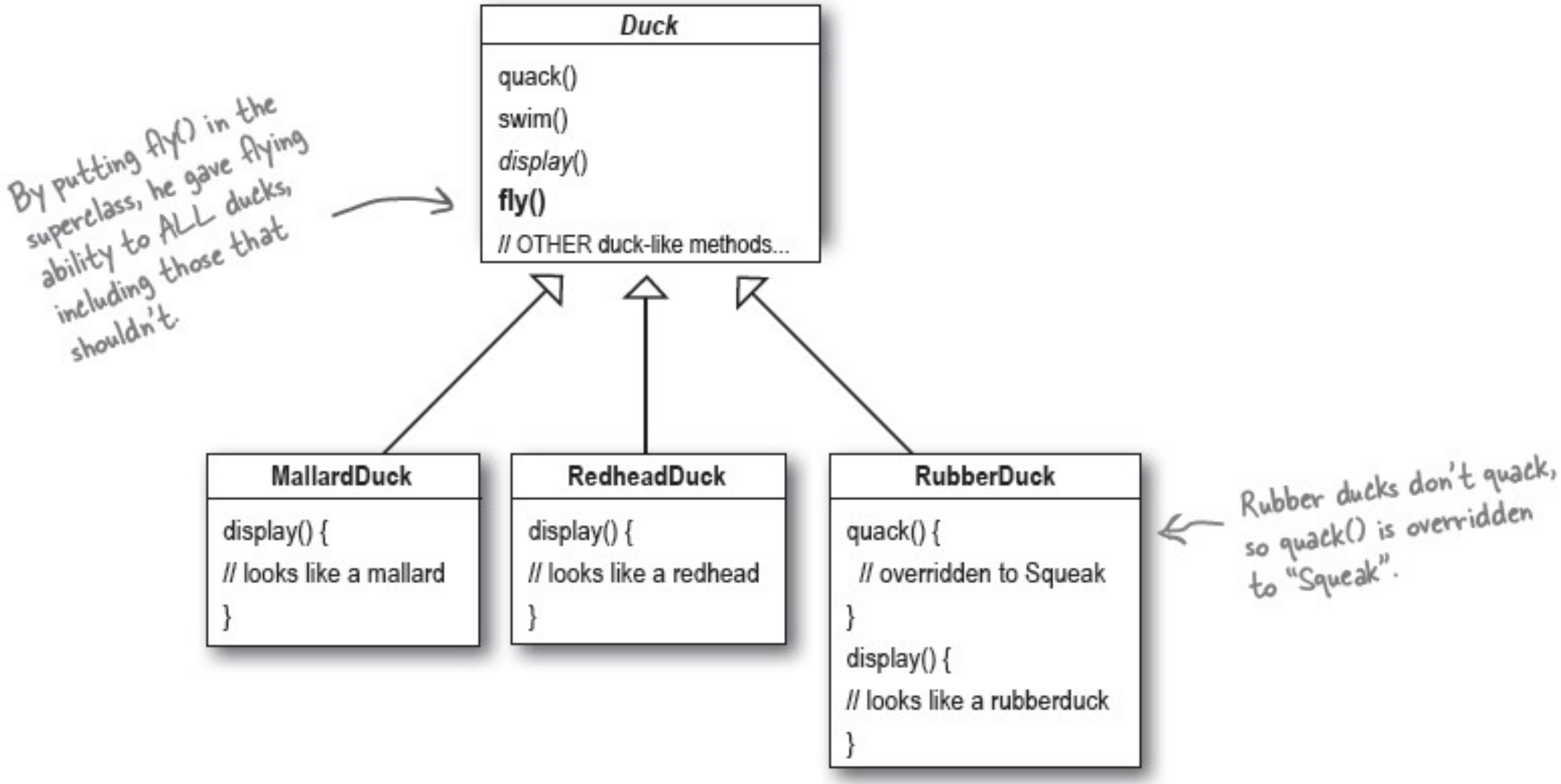
Ex. Duck Hunt Simulation Game



Making Ducks Fly



Making Ducks Fly: Issue

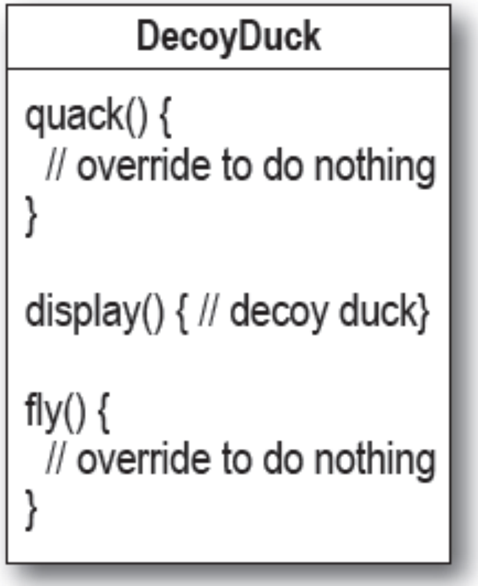


Making Ducks Fly: Issue

- Should all ducks be able to fly? quack?
- Solution?
  - Could just override the fly method in RubberDuck to do nothing

Making Ducks Fly: Possible Solution?

- Not horrible, but if we add DecoyDuck, for example, it neither quacks nor flies…

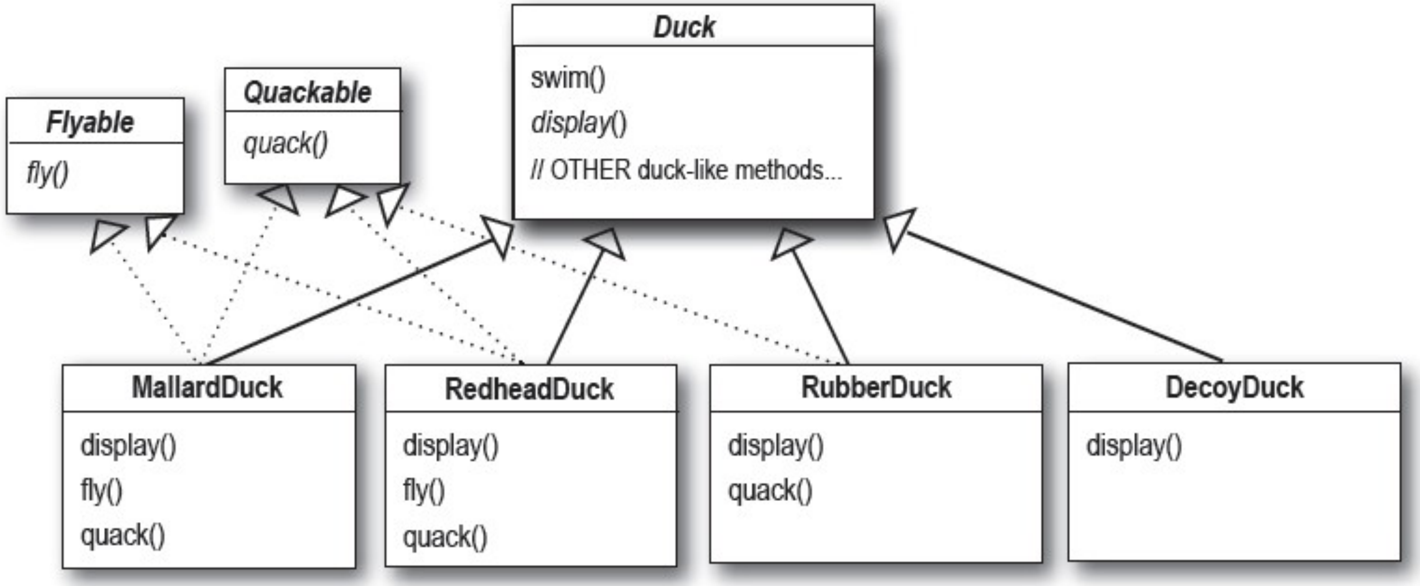


Making Ducks Fly

- Inheritance probably isn’t a great solution here
  - Each duck we add will have to examine and override fly and quack for each new class
  - We will likely have to duplicate code in several subclasses
    - How many ways can a duck really fly?

Making Ducks Fly

- Another option, use an interface

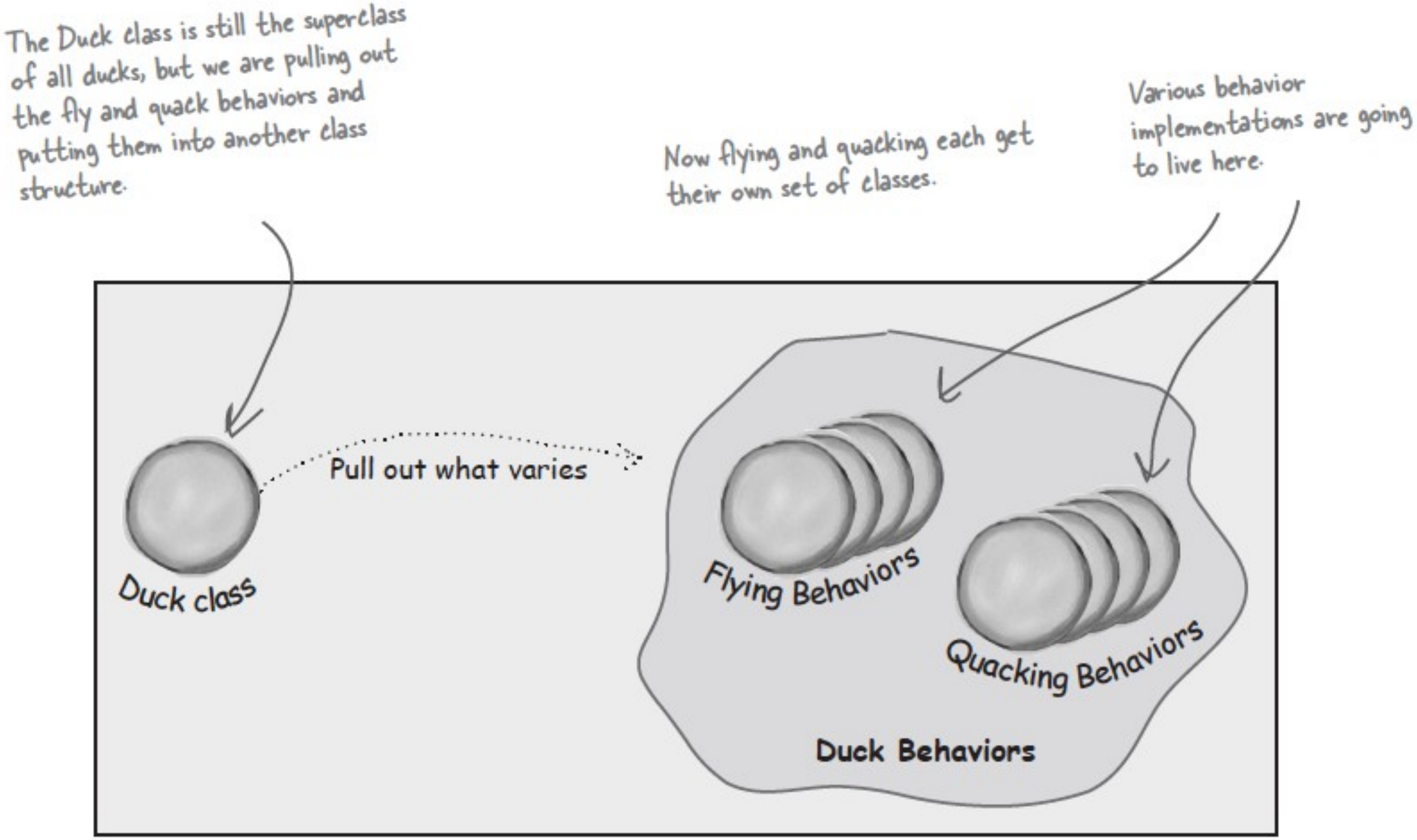


- Has this solved the issue?

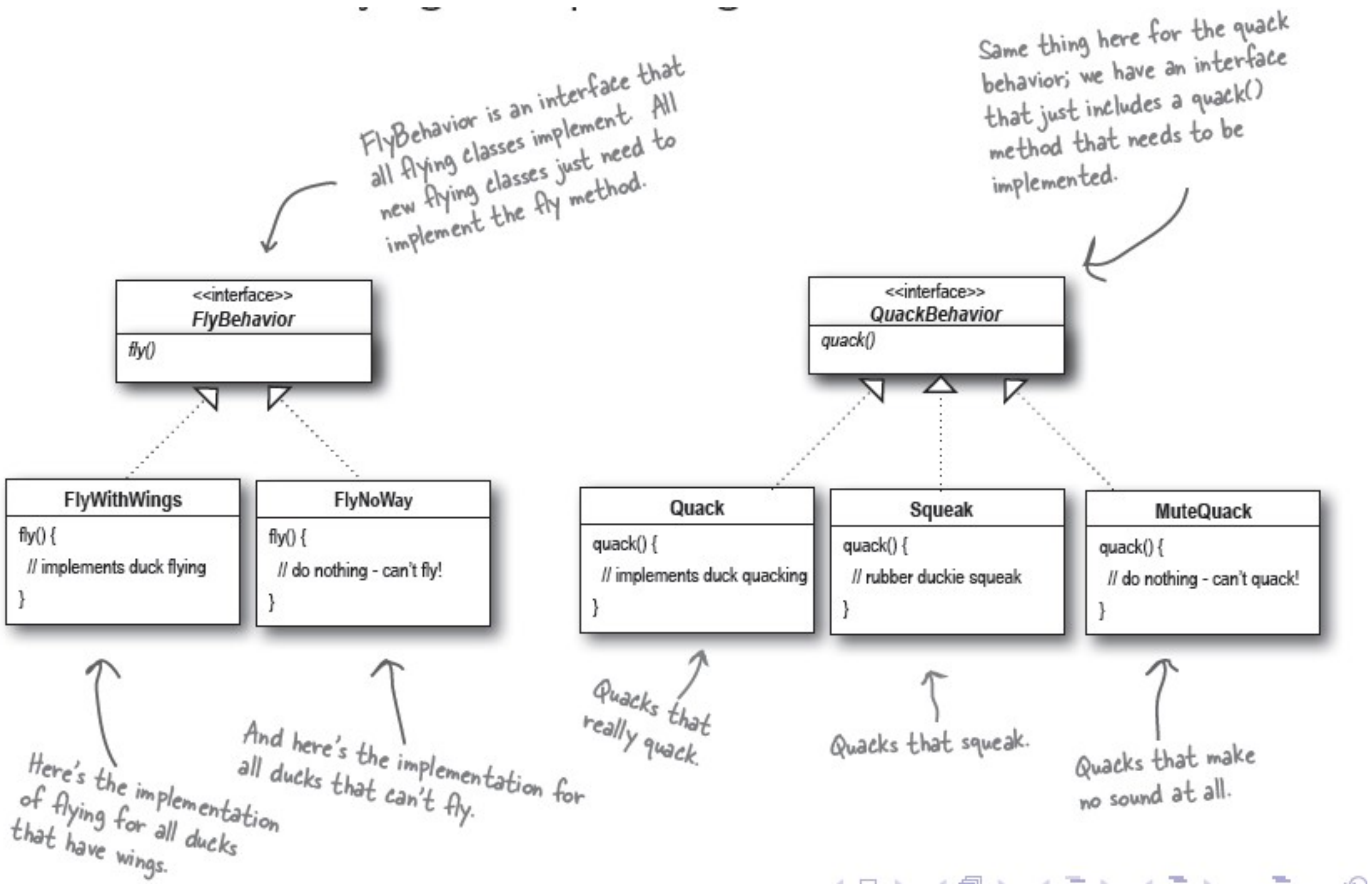
Making Ducks Fly: Solution

- New ducks will be added
- Duck behaviours differ from duck type to duck type
- Certain behaviours are not appropriate for all ducks

1. Encapsulate what varies

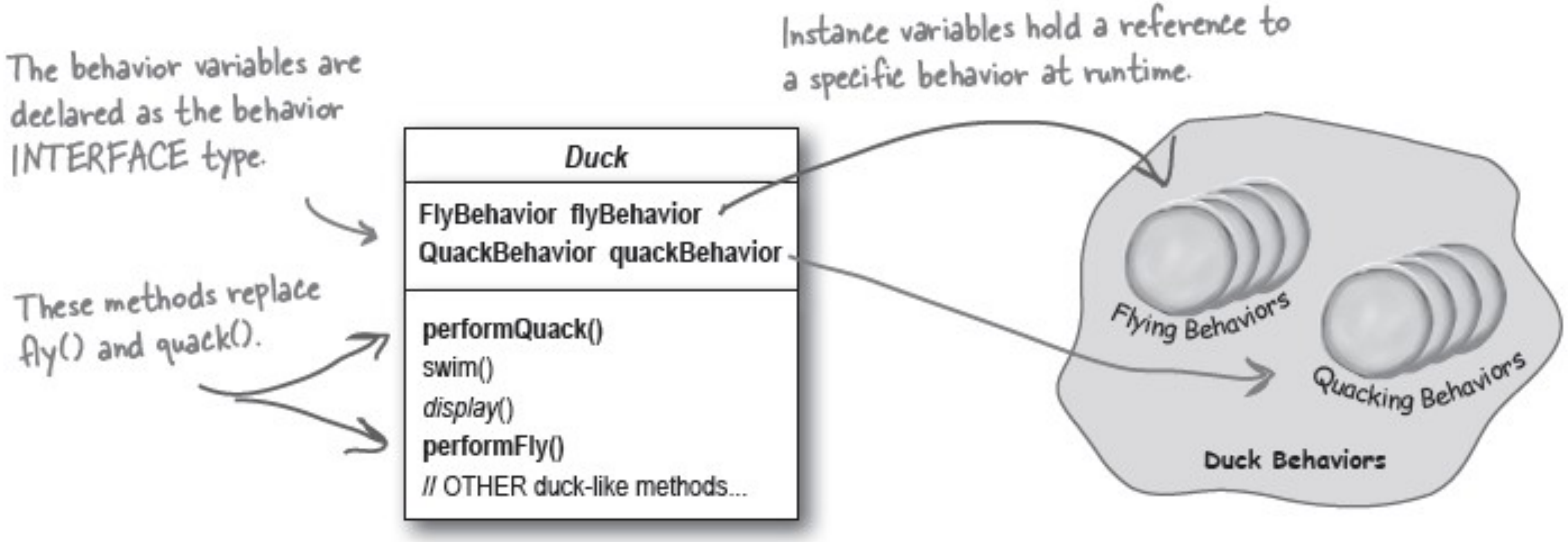


1. Encapsulate what varies



1. Encapsulate what varies

- Functionality that might vary between subclasses has been encapsulated in its own set of classes
- A duck will delegate its flying and quacking behaviours, rather than implementing them itself



1. Encapsulate what varies

- Gains
  - 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
    - Eg. could make a duck go mute
- This is also the Strategy design pattern

2. Code to an interface, not an implementation

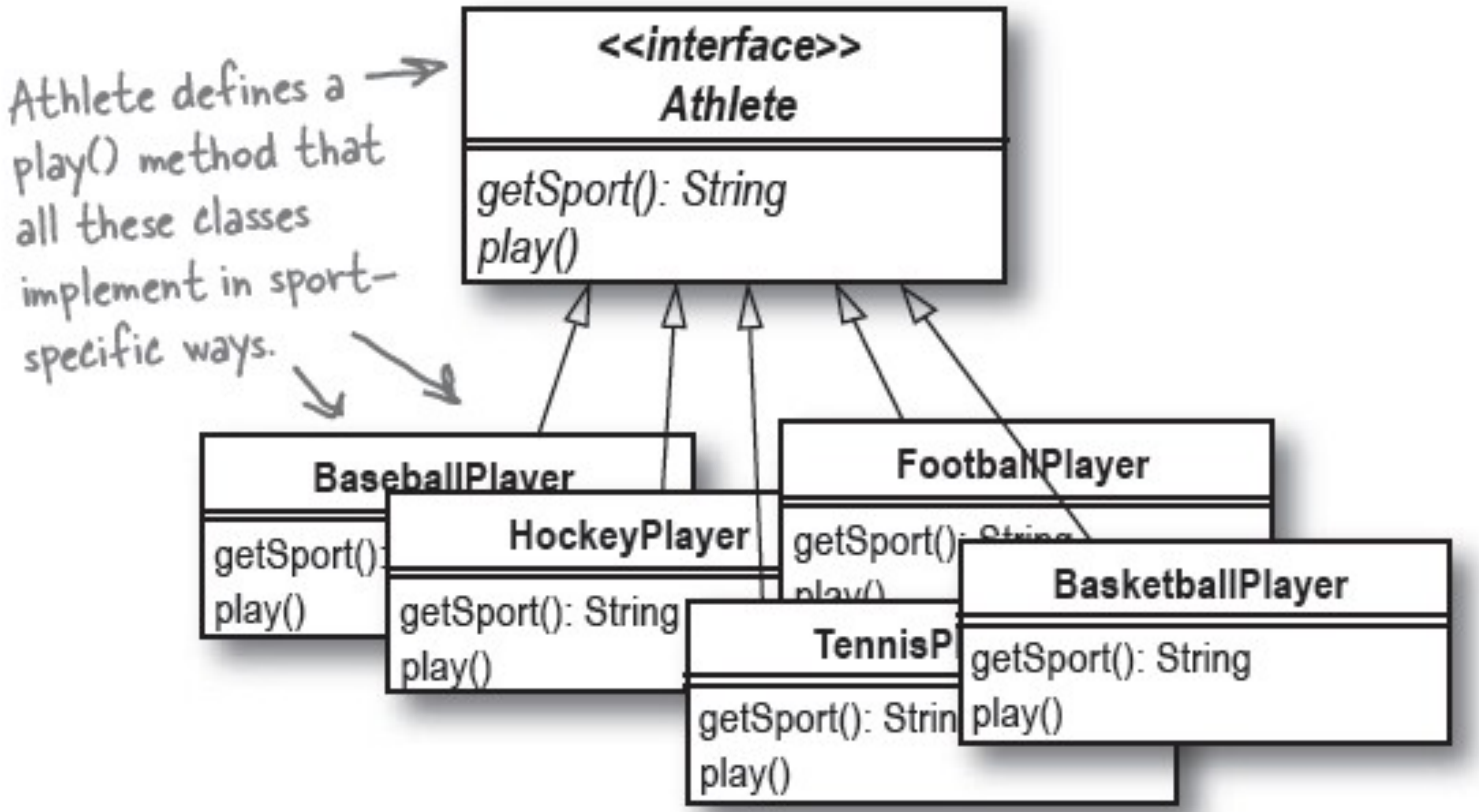
When faced with the choice between interacting with subclasses or interacting with a supertype, choose the supertype.

Your code will be easier to extend and will work with all of the interface's subclasses - even those not yet created.

2. Code to an interface

- Note: we are not talking just about the Java interface construct
  - could be an interface, abstract class, concrete superclass, etc.
  - favour interacting with generalizations

Modelling Athletes and Teams



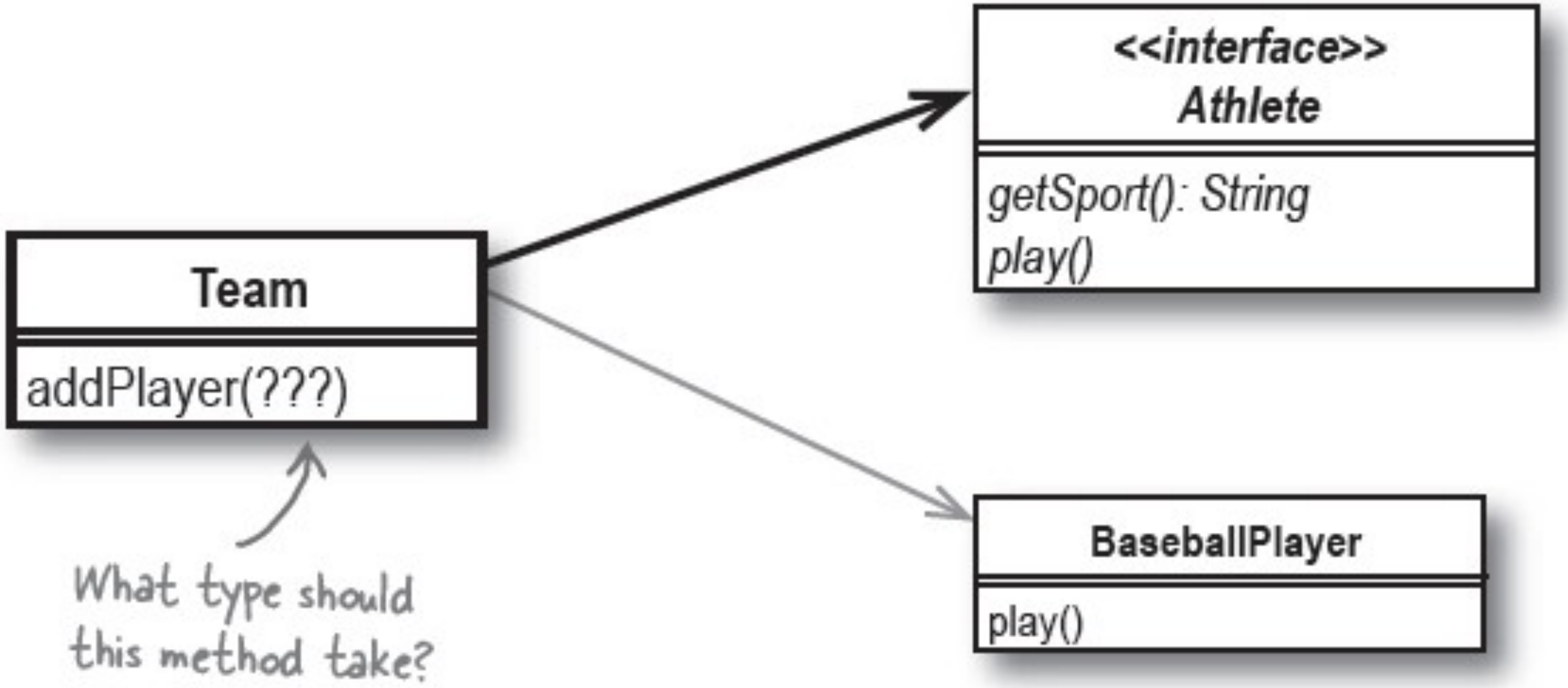
Modelling Athletes and Teams

- Option: create a Team class then subclass to specific team types
  - BaseballTeam
  - FootballTeam
  - TennisTeam
  - …

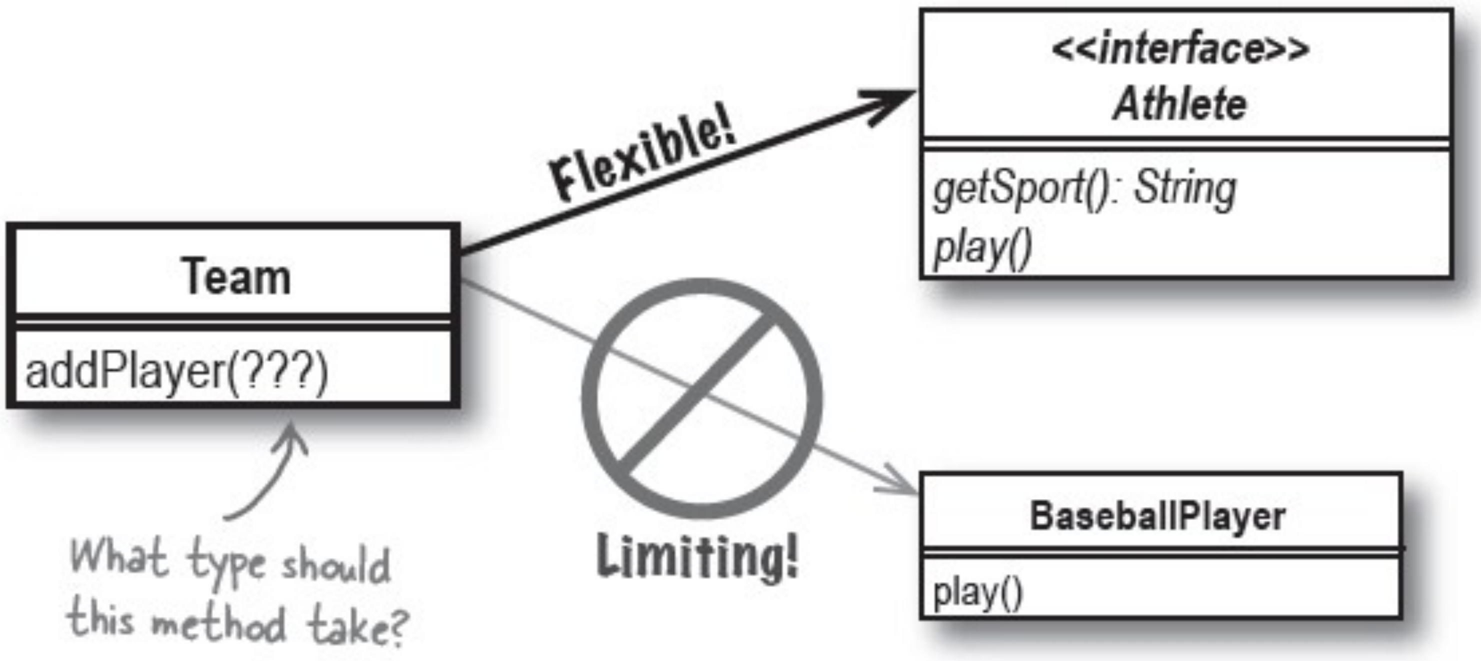
Modelling Athletes and Teams

- Issues:
  - Creates large inheritance hierarchy
    - Would mimic the Athlete hierarchy with the Team hierarchy
      - (BaseballPlayer for BaseballTeam, etc.)
  - Code duplication
  - Have to add new subclass for each new sport

2. Code to an interface



2. Code to an interface



2. Code to an interface

- Gains:
  - Adds flexibility
    - will work with any athlete - even for classes that are implemented in the future
  - Simpler architecture
  - Reduced duplication
    - In first approach, would have to have to implement addPlayer in each subclass

3. Favour composition over inheritence

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

3. Favour composition

- Inheritance establishes an IS-A relationship
- Composition / aggregation establish a HAS-A relationship
- Examples:
  - Is a baseball player an athlete?
  - Is a team composed of several athletes?
  - Is a team an athlete?

Example

```java
public interface Athlete {
    String getSport();
}
```

```java
public class BaseballPlayer implements Athlete {
    public String getSport(){
        return "Baseball";
    }
}
```

```java
public class HockeyPlayer implements Athlete {
    public String getSport(){
        return "Hockey";
    }
}
```

```java
public interface Team {
    public void addPlayer(Athlete a);
}
```

```java
public interface Team<A extends Athlete> {
    public void addPlayer(A player);
}
```

```java
public class HockeyTeam implements Team {

    List<Athlete> roster;

    public HockeyTeam() {
        this.roster = new ArrayList<>(30);
    }

    public void addPlayer(Athlete player){
        if(player instanceof HockeyPlayer){
            roster.add(player);
        }
        else{
            // player is not a hockey player and probably
            // shouldn't be on a hockey team...
        }
    }
}
```

```java
public class HockeyTeam implements Team<HockeyPlayer> {

    List<Athlete> roster;

    public HockeyTeam() {
        this.roster = new ArrayList<>(30);
    }

    public void addPlayer(HockeyPlayer player) {
        roster.add(player);
    }

}
```

What other changes could we make here?

```
Team hteam = new HockeyTeam();
hteam.addPlayer(new HockeyPlayer());

Team bteam = new BaseballTeam();
bteam.addPlayer(new HockeyPlayer());
```

Compiles, crashes

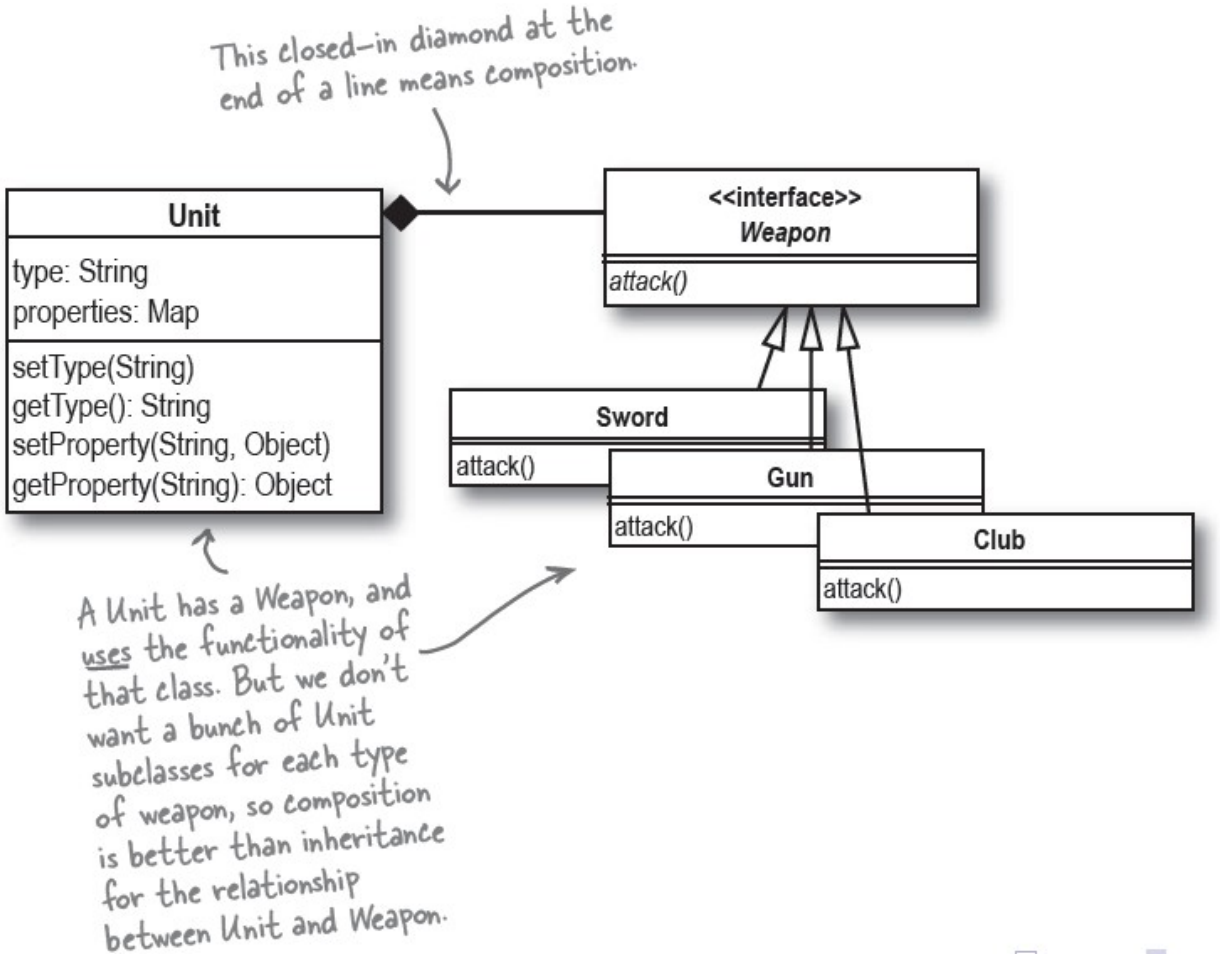
```
Team<BaseballPlayer> bteam = new BaseballTeam();
bteam.addPlayer(new HockeyPlayer());
```

Does not compile

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

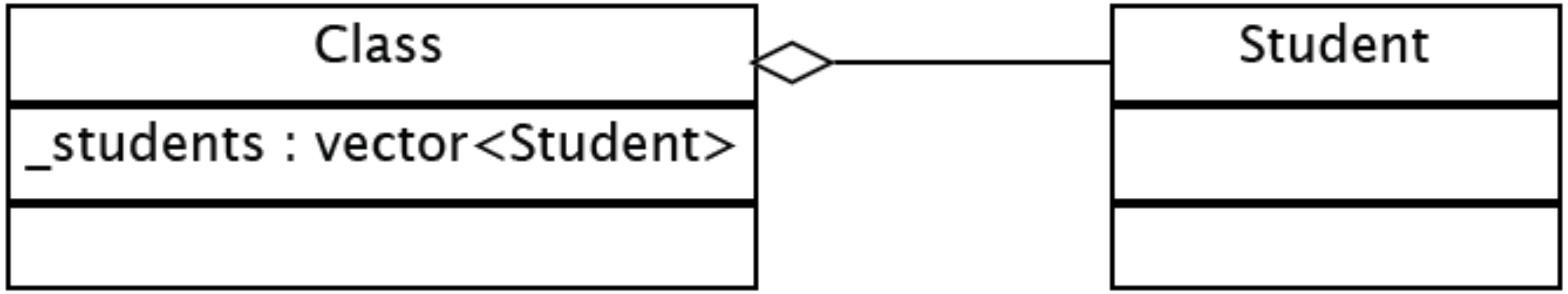


Composition

- Note: not a model of the real world
  - Weapon exists on its own in the real world
  - This may not be true in our model

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

- OWNS-A
- Object only makes sense as part of this class

- HAS-A
- Object may be needed outside this class

Players and Teams example:

Is this relationship OWNS-A or HAS-A?
How could we change this?

Design Patterns

A solution to an abstract software problem
within a particular context

- Abstract
  - Not specific to a language
- Context
  - Common situations in code design
- Problem
  - Goals vs. constraints in some context
- Solution
  - Design (template) for cooperating entities resolving the goals and constraints

Model-View-Controller

- Architectural design pattern for user interaction software
- Encapsulate components of interactive applications

- Model
  - The data classes
- View
  - The user interface components
- Controller
  - Application logic / functionality

Model

- Class(es) which capture data
  - Persistent - stored between program executions
  - Dynamic - data required during execution
- Controls
  - Persistent data ↔Dynamic data
- handle computation(s) on the data

View

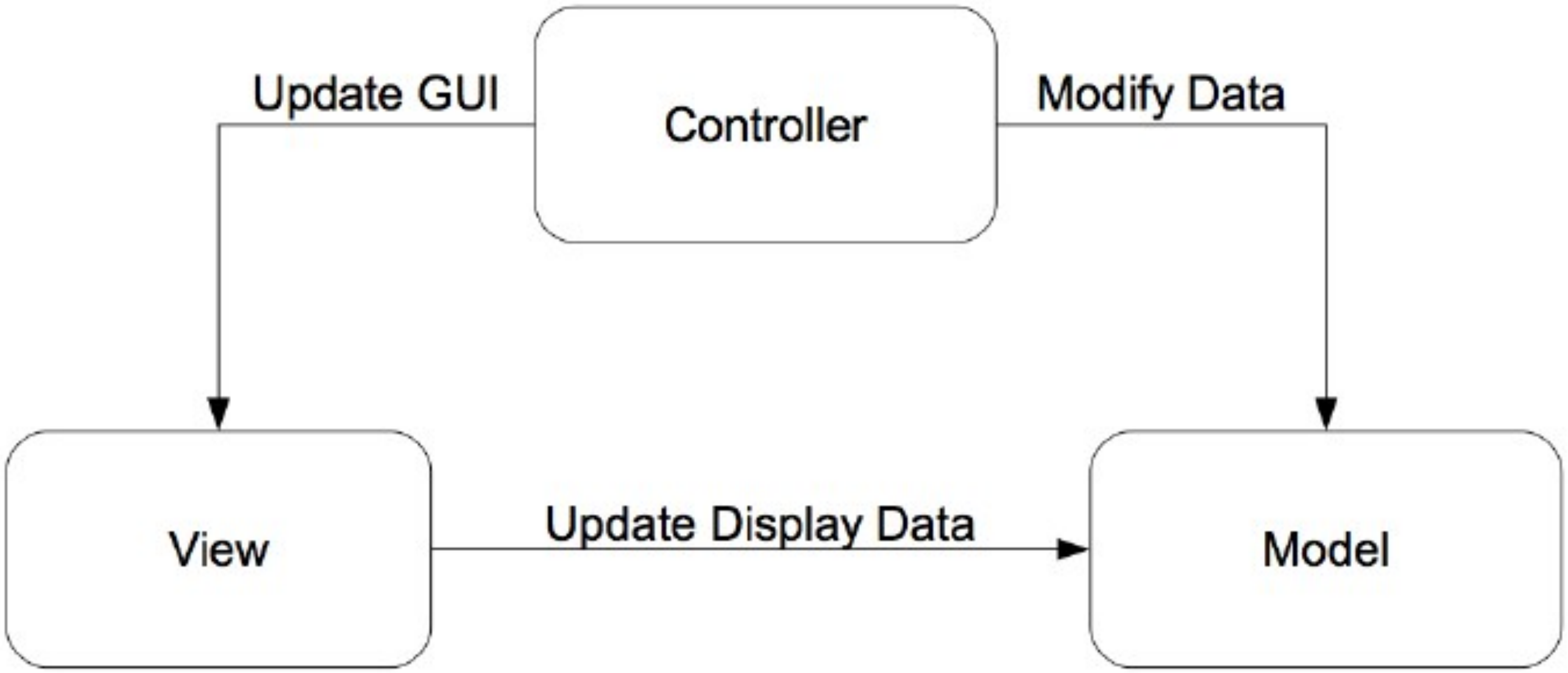
- Class(es) which present the interface to the user
- Typically GUI elements

Controller

- Coordinates Model and View classes
  - May change the view
  - Can modify and retrieve data from the model

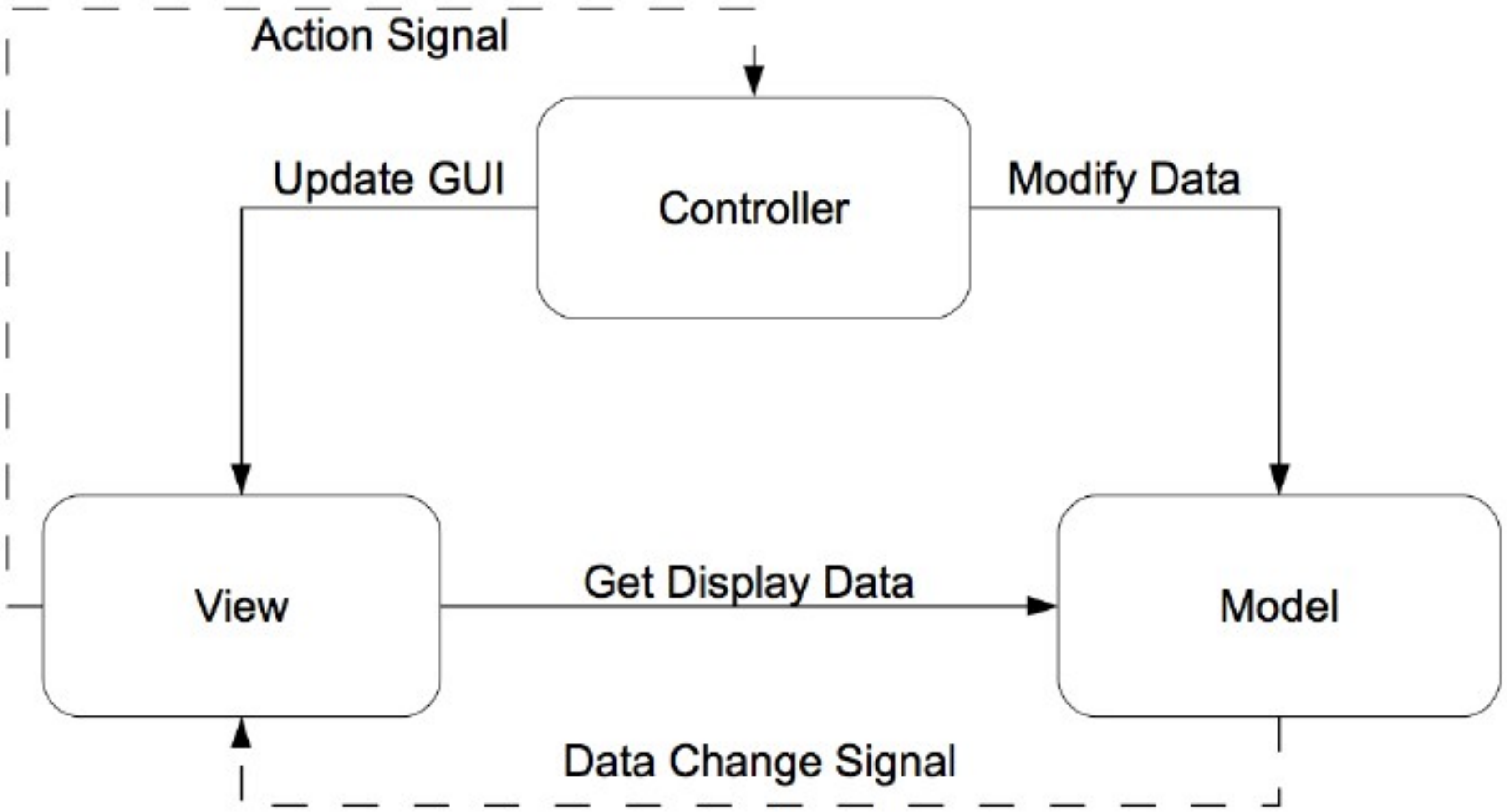
- Application logic/flow control

MVC

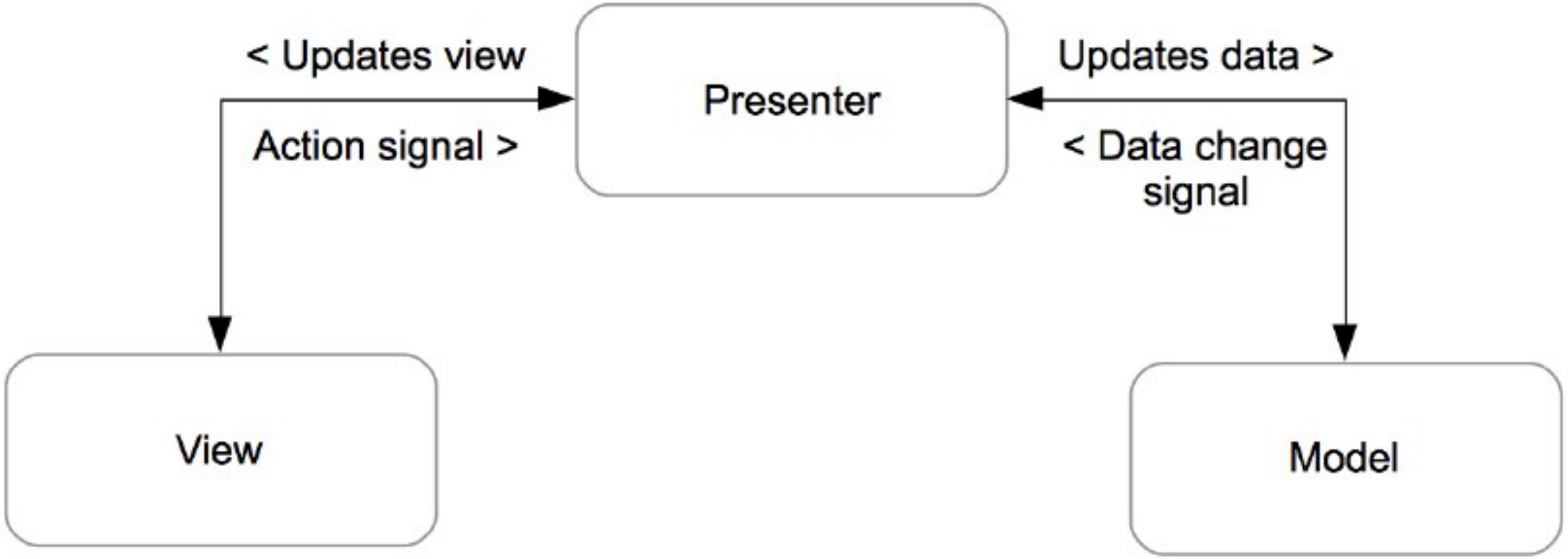


Controller
Update GUI
Modify Data
View
Update Display Data
Model

MVC


Action Signal
Update GUI
Controller
Modify Data
View
Get Display Data
Model
Data Change Signal

Model-View-Presenter



Singleton

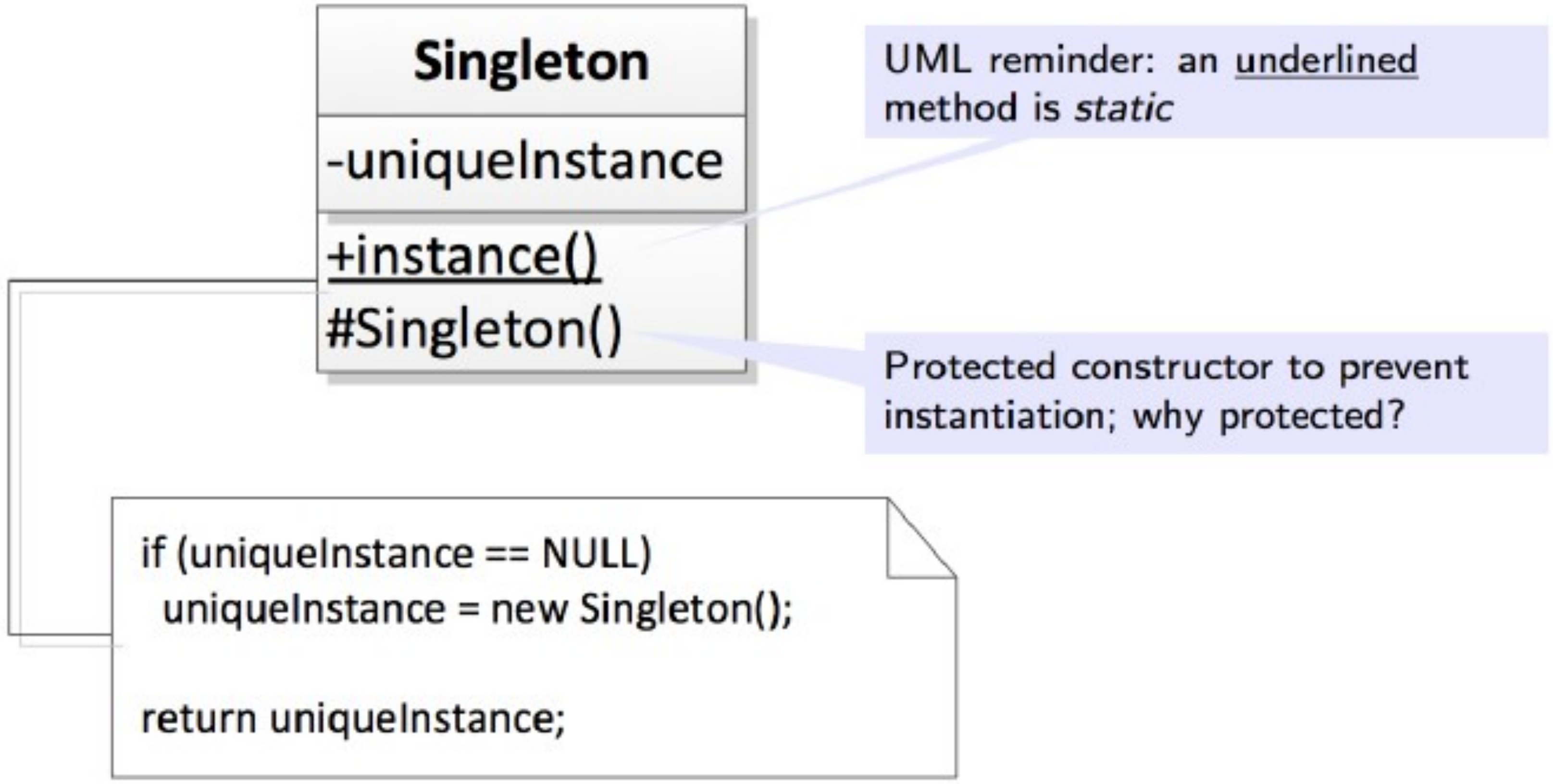
- Creational design pattern

Ensure a class had onlyone instance,
and provide a global point of access to it

Singleton

- One and only one:
  - hard to assign ownership (another class that should be responsible for it)
  - lazy initialization is desirable
  - need a global access point
  - may want to subclass

Singleton Pattern



Key implementation features

- public static accessor function
  - returns a reference to the instance
- protected or private constructors
  - will be called when accessor is first used
- private static attribute
  - stores the reference to the instance (or null)

Warning

- Not often required
  - Can easily become a wrapper for all the caveats of global variables
  - Can be difficult to delete
    - once made, will exist until exit
  - When possible, opt to pass a reference to an object resource only to entities needing them, rather than making the object available to every entity