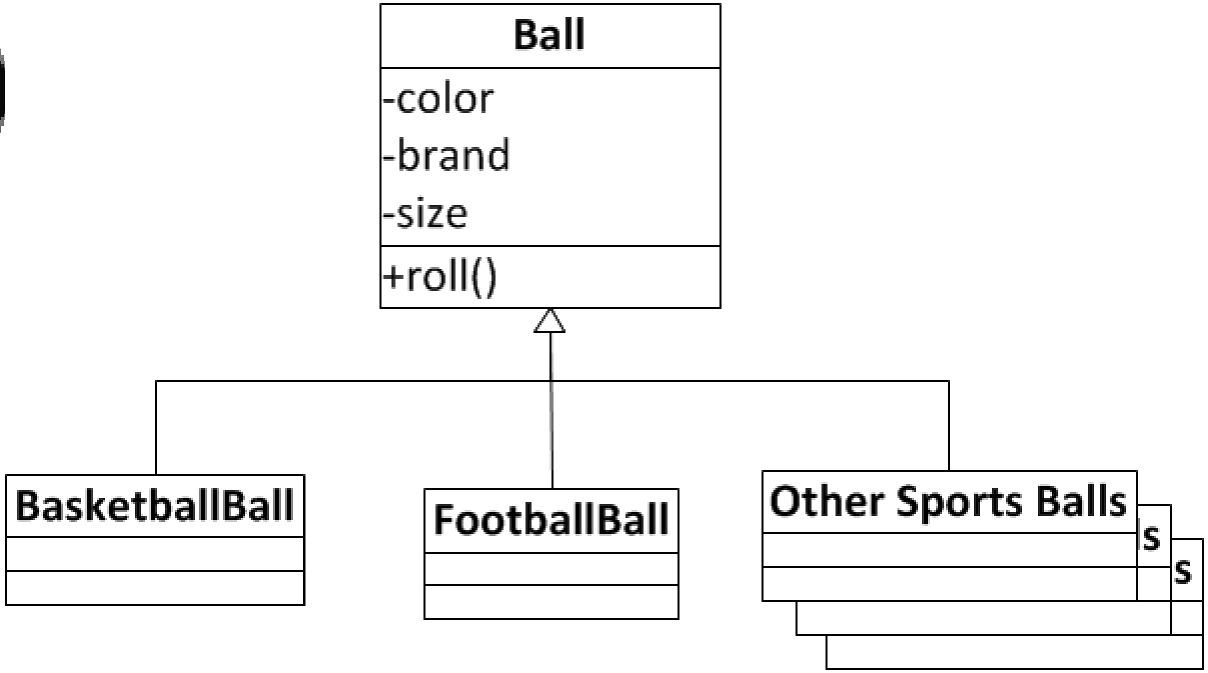
Strategy Pattern

Software Design

Jamaicon Sports app

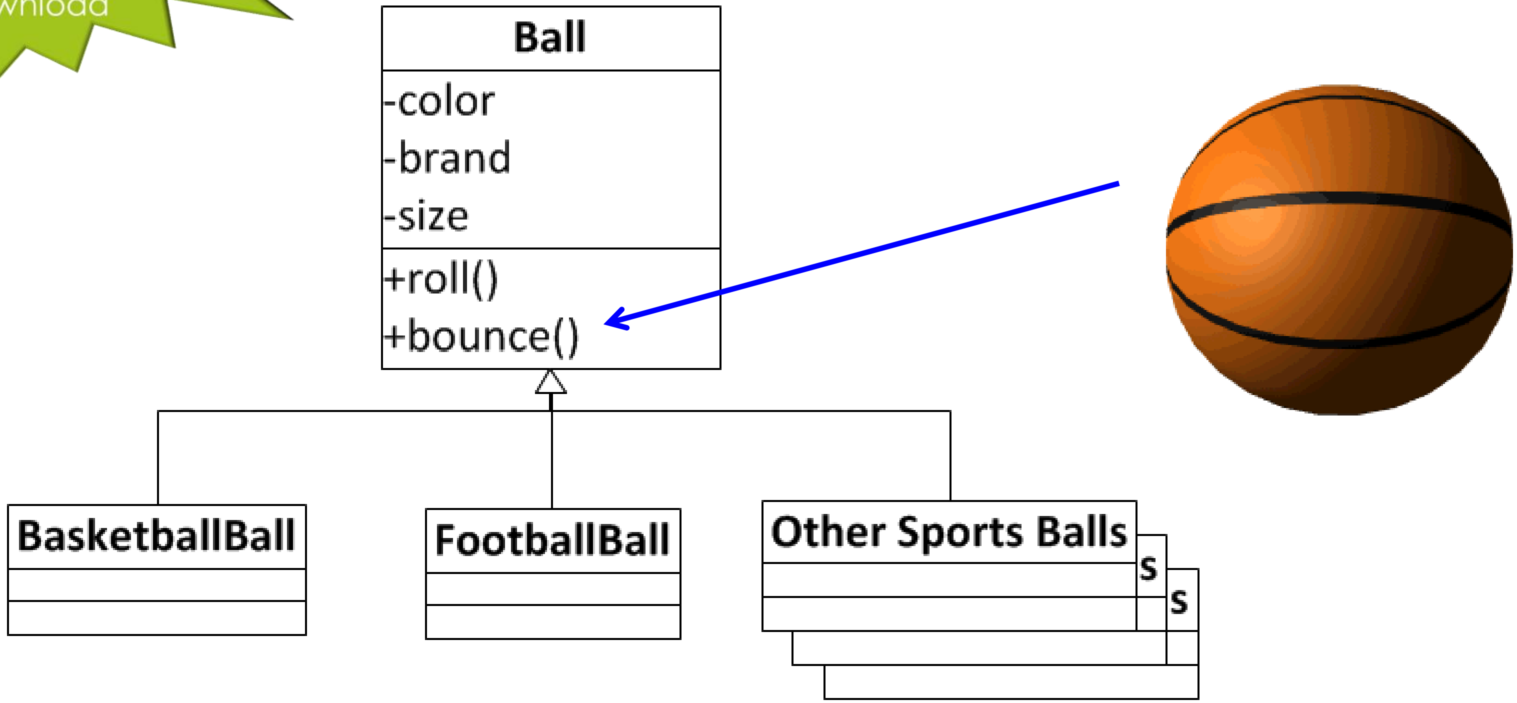
- Jamaicon sports is a store that has recently launched a mobile app for their online store.
- The application is working very well and online sales have increased in almost any department, except for balls.
- Marketing has proposed that the app should show the balls bouncing to get users attention.

Current design



But we need balls to bounce!

New app update! Available to download



Ball

-color
-brand
-size

+roll()
+bounce()

BasketballBall

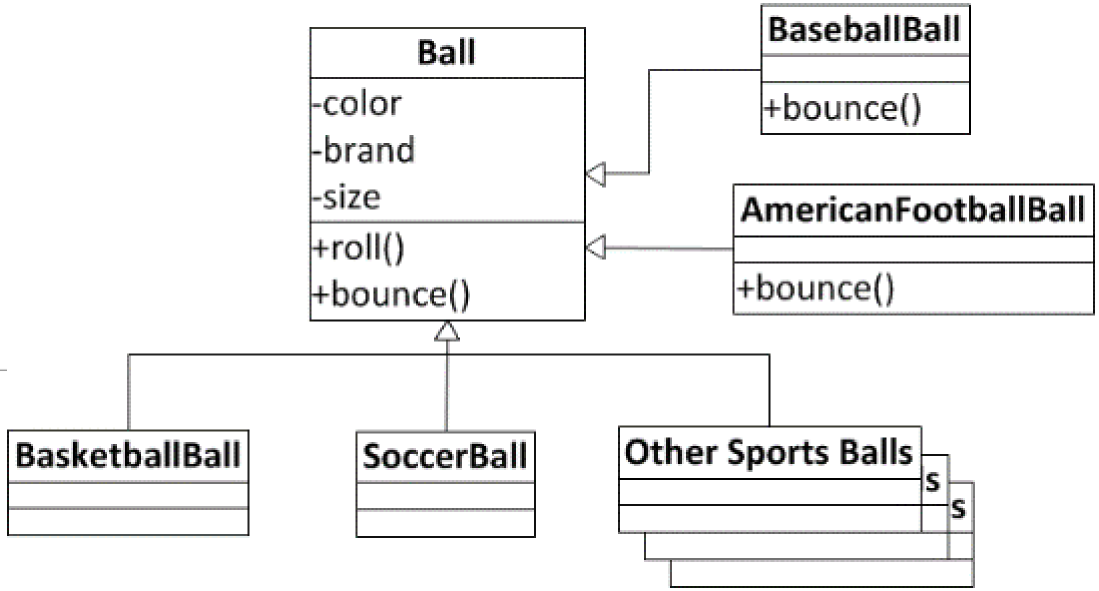
FootballBall

Other Sports Balls

Something went terribly wrong!

What would an OO expert do?

- Make Ball an abstract class and override bounce method to act differently on problematic balls



Ball

-color
-brand
-size

+roll()
+bounce()

BaseballBall

+bounce()

AmericanFootballBall

+bounce()

BasketballBall

SoccerBall

Other Sports Balls

s

s

But…

- What if there are hundreds of different types of balls and they all bounce differently?
- What if they want us to add an inflate/deflate animation?

Remember

- Change is the only constant in Software development
  - Customer wants something else
  - New technologies arrive
  - Managers bought a license for a different tool so they want to use it
  - Current database has being bought by a different company and they are ¨slightly¨ modifying the data model
  - …

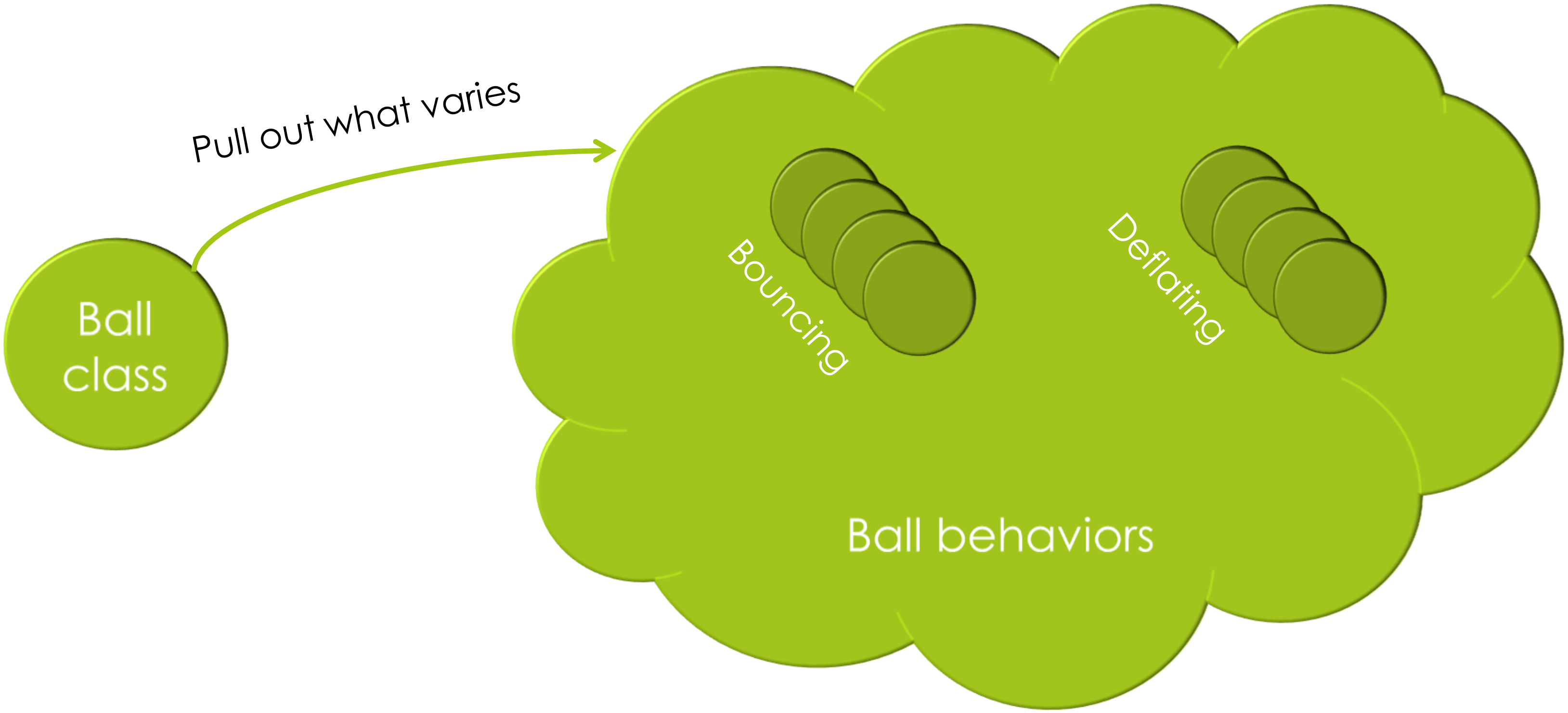
Design principle

- Identify the aspects of your applications that vary and separate them from what stays the same:
  - “Encapsulate” what varies so it won’t affect the rest of your code
  - You can easely alter or extend encapsulated parts
  - You don’t need to affect the rest of the code

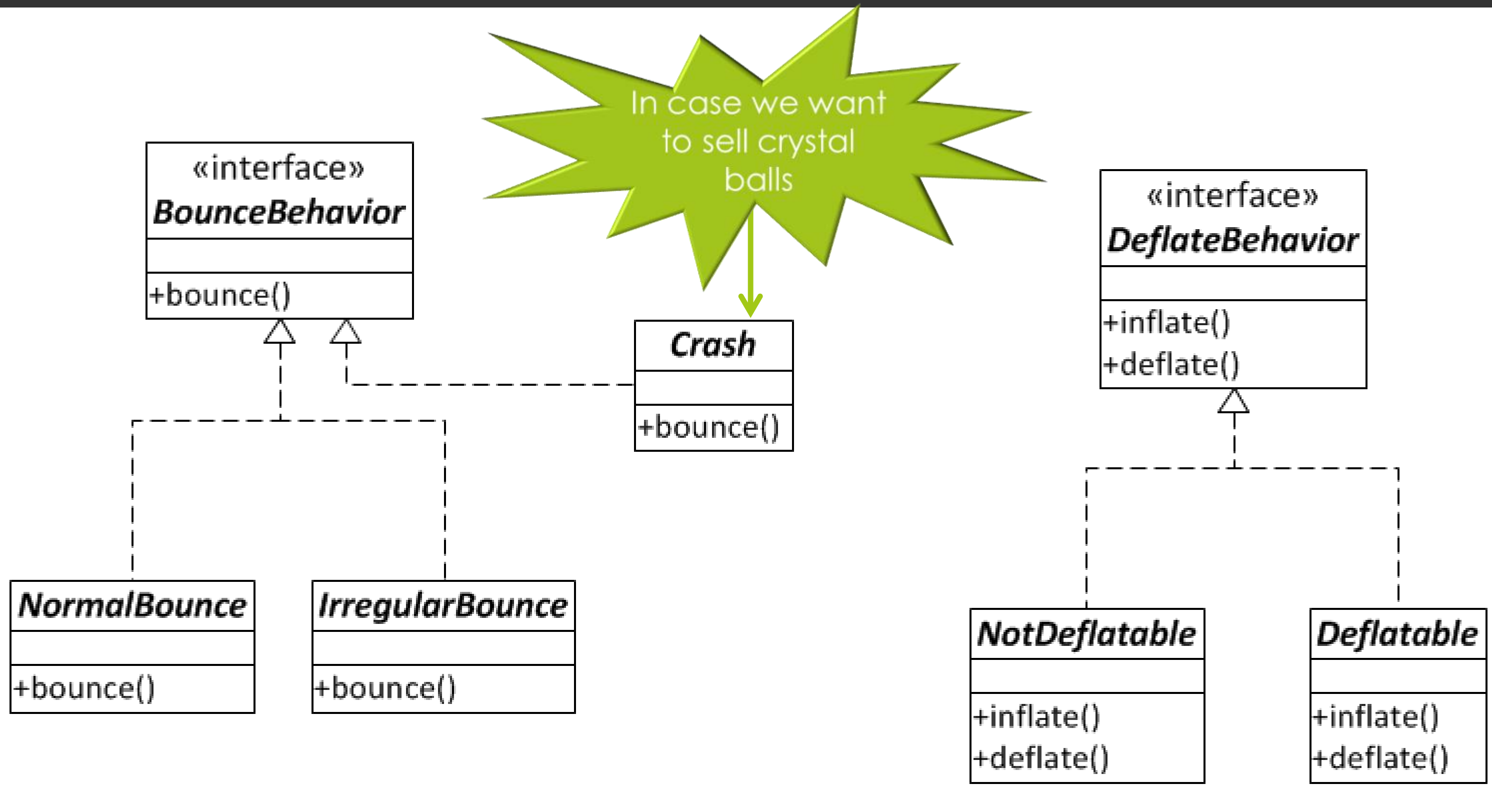
Encapsulate what varies

- Balls:
  - Baseball, American Football, Soccer, Golf, Basketball, Pool, Tennis, Hockey, Table Tennis…
- Mario Kart Characters:
  - Mario, Luigi, Toad, Bowser, Joshi, Princess, Donkey Kong…

Encapsulating on Jamaicon Sports app



Ball behaviors



Design principle

- Program to an interface not to an implementation

  - Programming to an implementation:

    ```
    Cat c = new Cat();
    c.meow();
    ```

  - Programming to an interface:

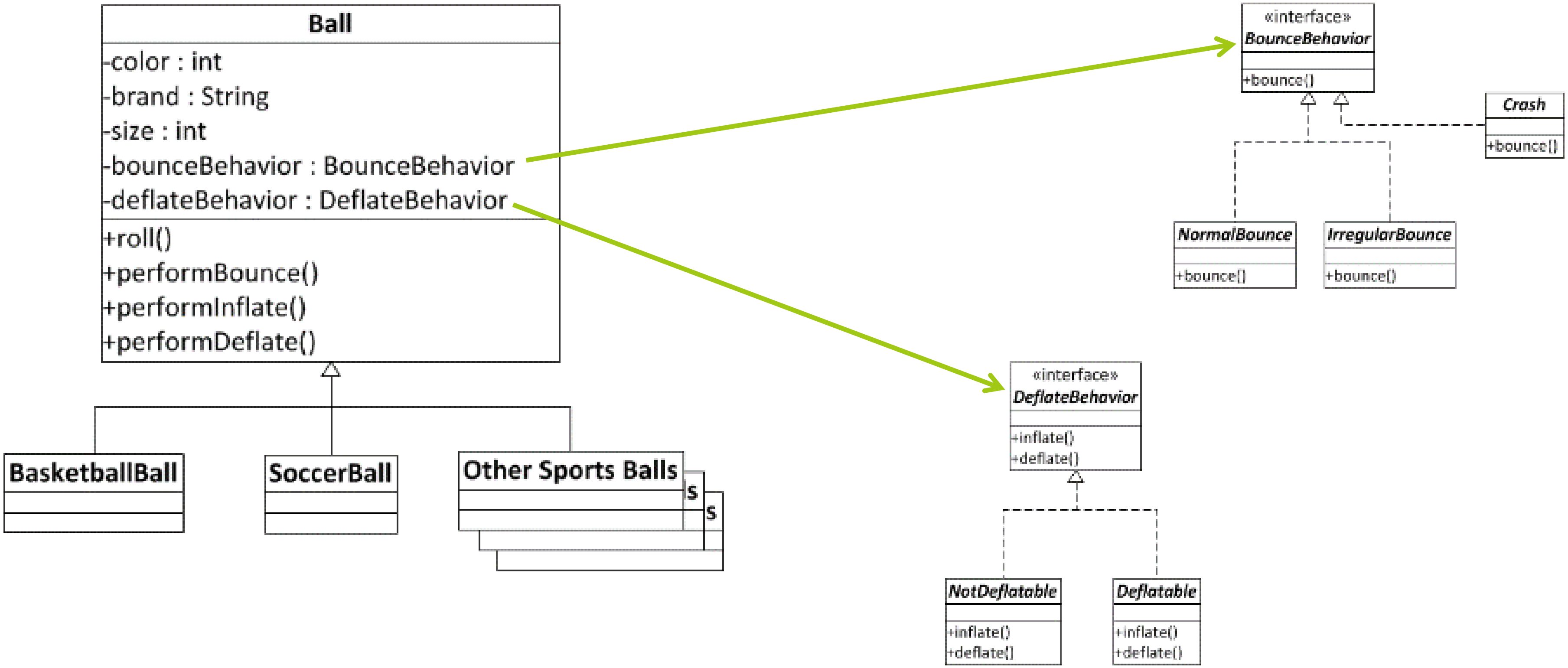
    ```
    Animal animal = new Cat();
    animal.makeSound();
    ```

Program to interface

- Complete the code:

```
public class BaseballBall extends Ball {
    public BaseballBall() {
        _____________  bounceBehavior =  new _______________();
        _____________  deflateBehavior = new ______________();
    }
}
```

The big picture

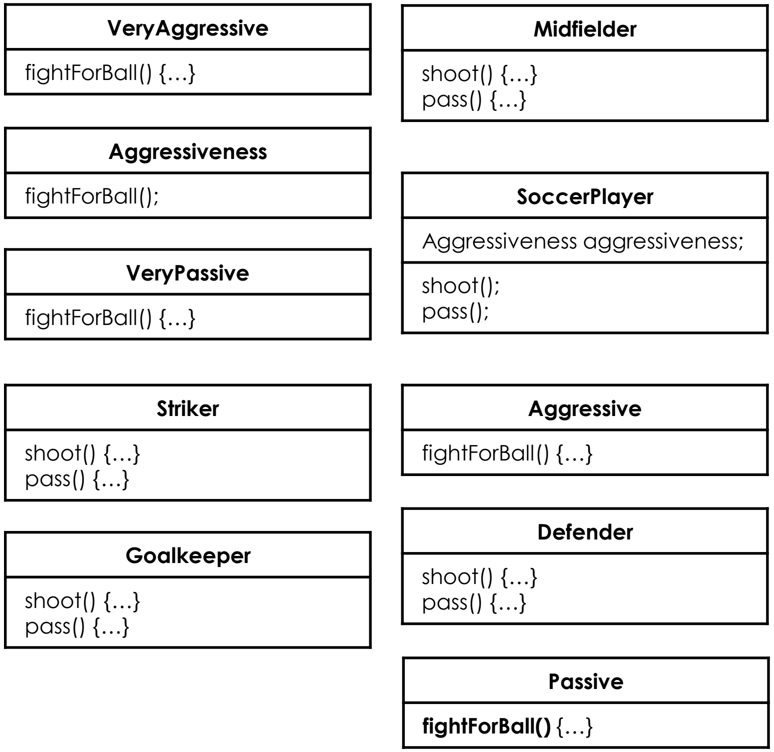


The implementation

Exercise

- Arrange the classes
- Identify them as abstract, interface or class
- Draw arrows between classes (inheritance, composition or interface)
- Put the method in the correct class

```
setAggresiveness( Agressiveness a) {
        this.agressiveness = a;
}
```



VeryAggressive
fightForBall() {…}

Aggressiveness
fightForBall();

VeryPassive
fightForBall() {…}

Striker
shoot() {…}
pass() {…}

Goalkeeper
shoot() {…}
pass() {…}

Midfielder
shoot() {…}
pass() {…}

SoccerPlayer
Aggressiveness aggressiveness;
shoot();
pass();

Aggressive
fightForBall() {…}

Defender
shoot() {…}
pass() {…}

Passive
fightForBall() {…}

Strategy pattern

- Strategy pattern defines a family of algorithms (behaviors), encapsulate each one, and makes them interchangeable
- Strategy lets the algorithm vary independently from the clients that use it

Design Principle

- Favor composition over inheritance:
  - HAS-A can be better than IS-A
  - Composition gives you more flexibility (things can change at runtime)
  - Composition is used in many design patterns

- Question: What if Jamaicon Sports also sells lifesavers that can be inflated/deflated? can we use the DeflateBehavior interface we made for Balls?