Design Patterns

Adapter

Code Smells: Couplers

COMP3607
Object Oriented Programming II

Week 9

Outline

- Design Patterns
  - Adapter
  - Code Smells:
    - Couplers

10

Adapter Design Pattern

https://www.youtube.com/watch?v=qG286LQM6BU

Adapter Pattern Video - Derek Banas

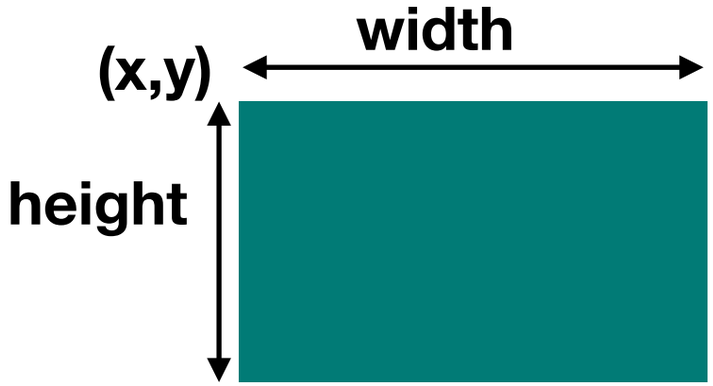
Intent

The Adapter design pattern lets classes work together that couldn't otherwise because of incompatible interfaces.

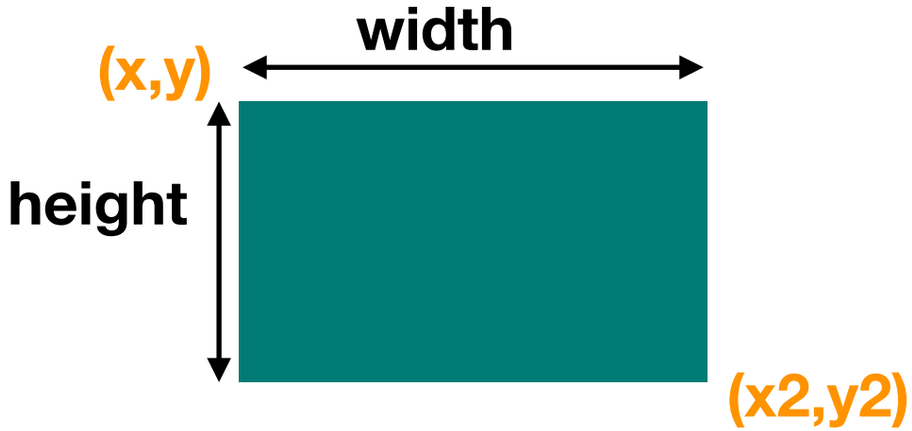
- Convert the interface of a class into another interface that clients expect.
- Wrap an existing class with a new interface.

Example

A legacy Rectangle component's `display()` method expects to receive "x, y, width, height" parameters.

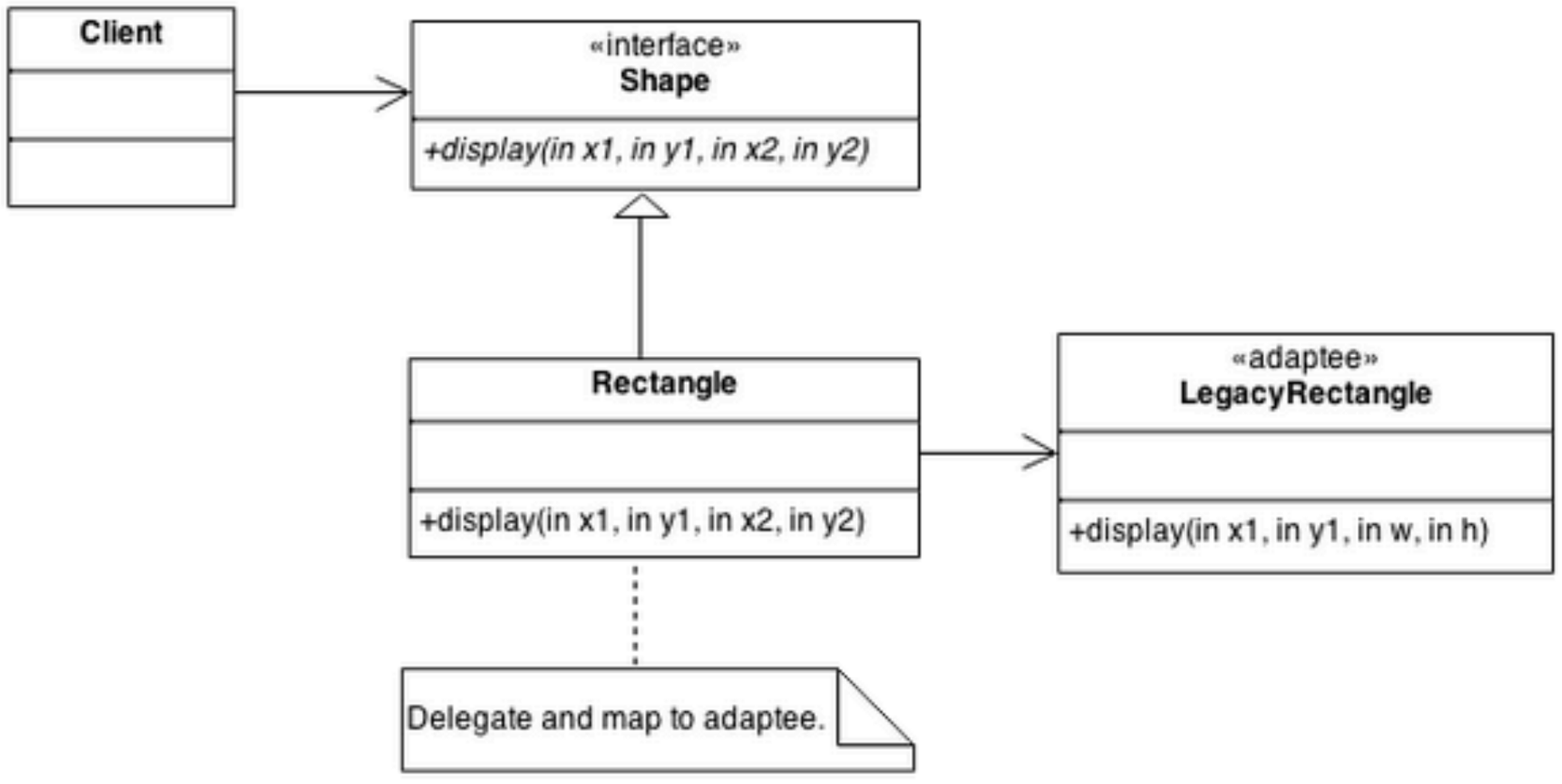


Example



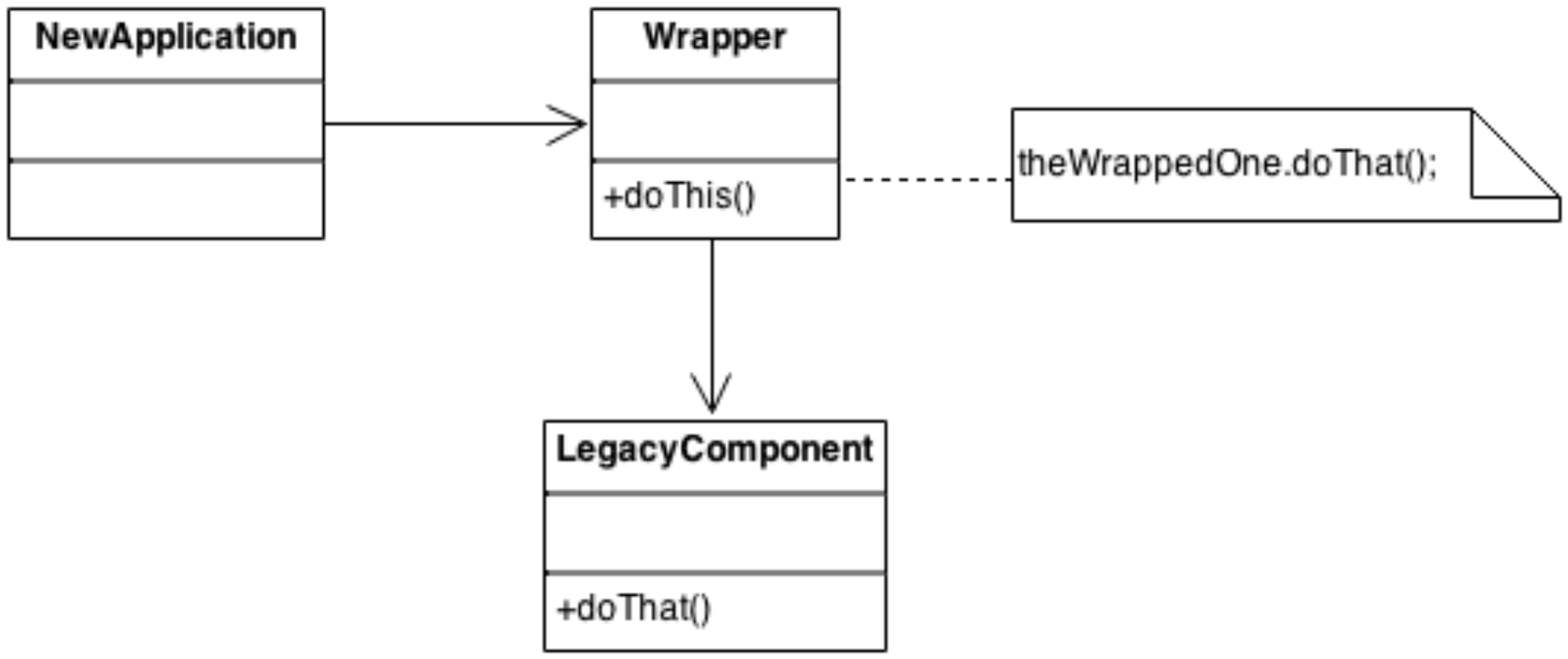
The client wants to pass "upper left x and y" and "lower right x and y". This incongruity can be reconciled by adding an additional level of indirection – i.e. an Adapter object.

Example


Client
«interface»
Shape
+display(in x1, in y1, in x2, in y2)
Rectangle
+display(in x1, in y1, in x2, in y2)
«adaptee»
LegacyRectangle
+display(in x1, in y1, in w, in h)
Delegate and map to adaptee.

Example

The Adapter could also be thought of as a "wrapper".



Example

```java
class Line {
    public void draw(int x1, int y1, int x2, int y2) {
        System.out.println("Line from point A(" + x1 + ";" + y1 + "),"
                           + "to point B(" + x2 + ";" + y2 + ")");
    }
}

class Rectangle {
    public void draw(int x, int y, int width, int height) {
        System.out.println("Rectangle with coordinate left-down point"
                           +"(" + x + ";" + y + "), width: " + width
                           + ", height: " + height);
    }
}

public class AdapterDemo {
    public static void main(String[] args) {
        Object[] shapes = {new Line(), new Rectangle()};
        int x1 = 10, y1 = 20;
        int x2 = 30, y2 = 60;
        int width = 40, height = 40;
        for (Object shape : shapes) {
            if (shape.getClass().getSimpleName().equals("Line")) {
                ((Line)shape).draw(x1, y1, x2, y2);
            } else if (shape.getClass().getSimpleName().equals("Rectangle")) {
                ((Rectangle)shape).draw(x2, y2, width, height);
            }
        }
    }
}
```

Output

```
Line from point A(10;20), to point B(30;60)
Rectangle with coordinate left-down point (30;60), width: 40, height: 40
```

Example


```java
interface Shape {
    void draw(int x, int y, int z, int j);
}

class Line {
    public void draw(int x1, int y1, int x2, int y2) {
        System.out.println("Line from point A(" + x1 + ";" + y1 + "), to point B(" + x2 + ";" + y2 + ")");
    }
}

class Rectangle {
    public void draw(int x, int y, int width, int height) {
        System.out.println("Rectangle with coordinate left-down point (" + x + ";" + y + "), width: " + width
                + ", height: " + height);
    }
}
```

Example

```java
class LineAdapter implements Shape {
    private Line adaptee;

    public LineAdapter(Line line) {
        this.adaptee = line;
    }

    @Override
    public void draw(int x1, int y1, int x2, int y2) {
        adaptee.draw(x1, y1, x2, y2);
    }
}

class RectangleAdapter implements Shape {
    private Rectangle adaptee;

    public RectangleAdapter(Rectangle rectangle) {
        this.adaptee = rectangle;
    }

    @Override
    public void draw(int x1, int y1, int x2, int y2) {
        int x = Math.min(x1, x2);
        int y = Math.min(y1, y2);
        int width = Math.abs(x2 - x1);
        int height = Math.abs(y2 - y1);
        adaptee.draw(x, y, width, height);
    }
}
```

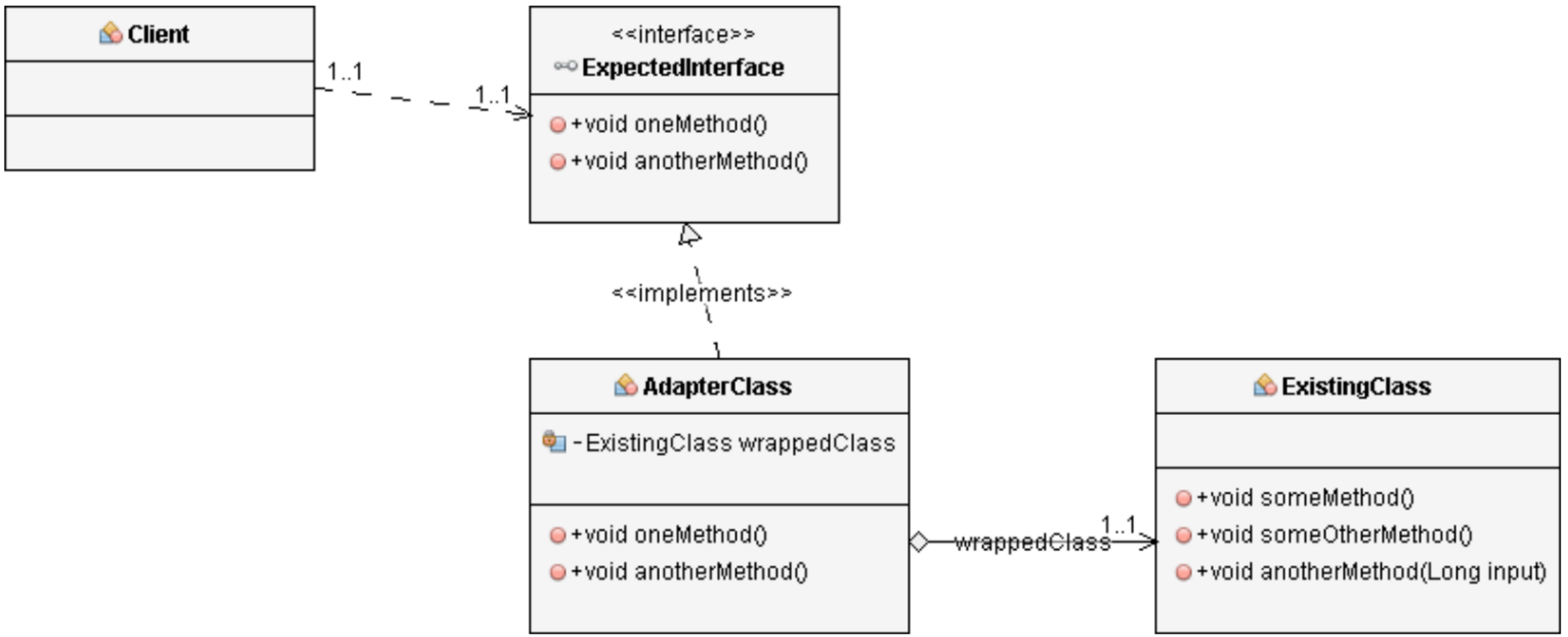
Example

```
31  class RectangleAdapter implements Shape {
32      private Rectangle adaptee;
33
34      public RectangleAdapter(Rectangle rectangle) {
35          this.adaptee = rectangle;
36      }
37
38      @Override
39      public void draw(int x1, int y1, int x2, int y2) {
40          int x = Math.min(x1, x2);
41          int y = Math.min(y1, y2);
42          int width = Math.abs(x2 - x1);
43          int height = Math.abs(y2 - y1);
44          adaptee.draw(x, y, width, height);
45      }
46  }
```

Example

```
48  public class AdapterDemo {
49      public static void main(String[] args) {
50          Shape[] shapes = {new RectangleAdapter(new Rectangle()),
51                            new LineAdapter(new Line())};
52          int x1 = 10, y1 = 20;
53          int x2 = 30, y2 = 60;
54          for (Shape shape : shapes) {
55              shape.draw(x1, y1, x2, y2);
56          }
57      }
58  }
59
```

Adapter Pattern (UML)



https://dzone.com/articles/design-patterns-explained-adapter-pattern-with-cod

Adapter Checklist:

- Identify the players: the component(s) that want to be accommodated (i.e. the client), and the component that needs to adapt (i.e. the adaptee).
- Identify the interface that the client requires.
- Design a "wrapper" class that can "impedance match" the adaptee to the client.
- The adapter/wrapper class "has a" instance of the adaptee class.
- The adapter/wrapper class "maps" the client interface to the adaptee interface.
- The client uses (is coupled to) the new interface

Consequences:

Objects Adapters - Based on Delegation

- Objects Adapters are the classical example of the adapter pattern. It uses composition so the Adapter delegates the calls to Adaptee
- This behaviour gives us a few advantages over the class adapters (however the class adapters can be implemented in languages allowing multiple inheritance).
- The main advantage is that the Adapter adapts not only the Adaptee but all its subclasses with one "small" restriction:
  - all the subclasses that don't add new methods (because the used mechanism is delegation).
  - So for any new method the Adapter must be changed or extended to expose the new methods as well.
  - The main disadvantage is that programmers are required to write all the code for delegating all the necessary requests to the Adaptee.

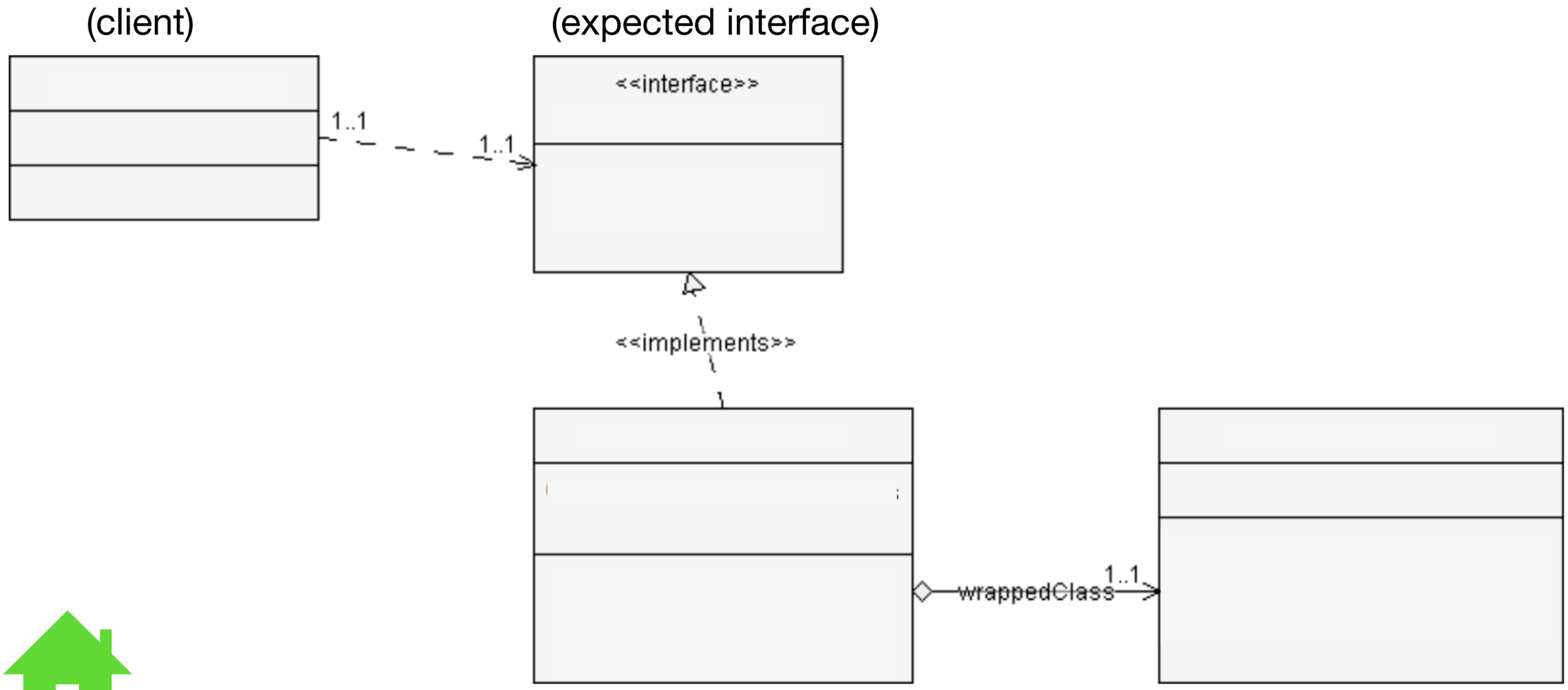
Consequences:

Class Adapters - Based on (Multiple) Inheritance

- Class adapters can be implemented in languages supporting multiple inheritance. Java, C# or PHP do not support multiple inheritance. Thus, such adapters can not be easy implemented in these languages.
- Class adapters use inheritance instead of composition. This means that instead of delegating the calls to the Adaptee, it subclasses it. In conclusion it must subclass both the Target and the Adaptee. There are advantages and disadvantages:
- It adapts the specific Adaptee class. The class it extends. If that one is subclassed it can not be adapted by the existing adapter.
- It doesn't require all the code required for delegation, which must be written for an Object Adapter.
- If the Target is represented by an interface instead of a class then we can talk about "class" adapters, because we can implement as many interfaces as we want.

Robot Example Exercise

The following diagram shows the general design of the Adapter pattern.
Insert the classes from the robot example described in the video by Derek Banas.



Bonus Exercise

Code Smells

“A surface indication that usually corresponds to a deeper problem in the software system” - Martin Fowler

Couplers

All the smells in this group contribute to excessive coupling between classes or show what happens if coupling is replaced by excessive delegation.

For example

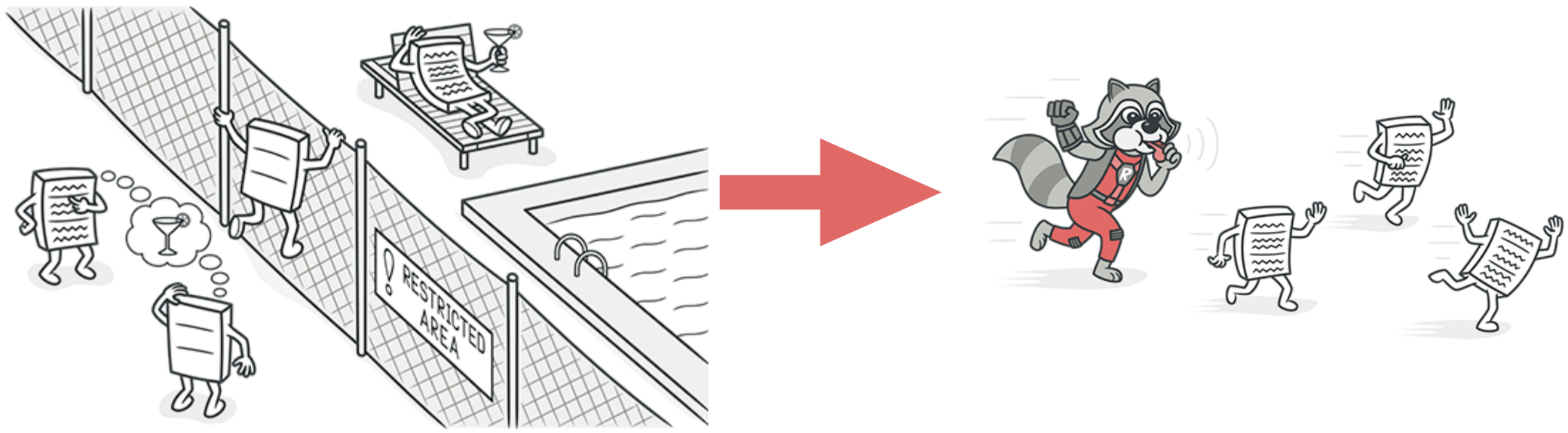
- Feature Envy
- Inappropriate Intimacy
- Message Chains
- Middle Man

https://refactoring.guru/refactoring/smells/couplers

Feature Envy

It is when a method does not leverage data or methods from the class it belongs to.

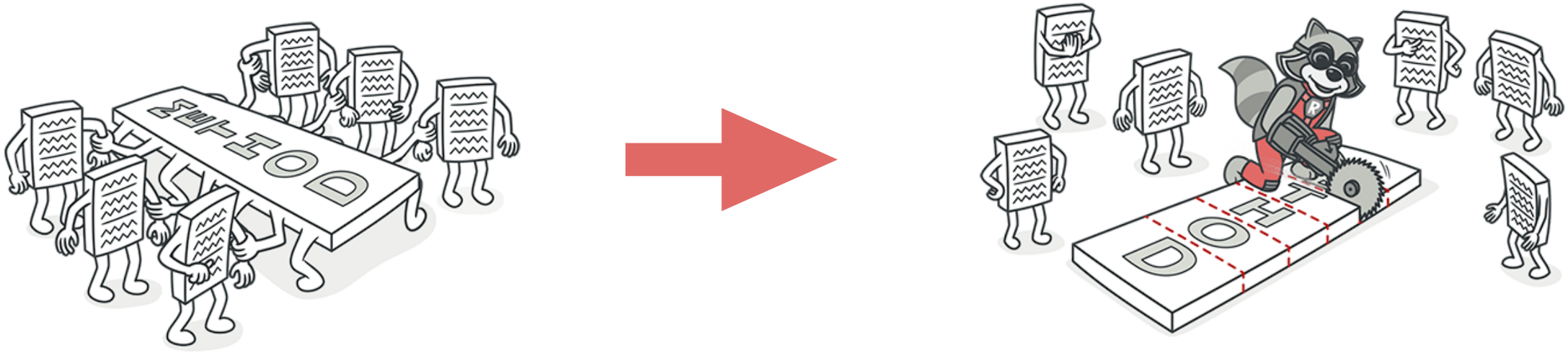
Instead, it requires lots of data or methods from a different class



https://refactoring.guru/smells/feature-envy

Inappropriate Intimacy

One class uses the internal fields and methods of another class. Good classes should know as little about each other as possible. Such classes are easier to maintain and reuse.

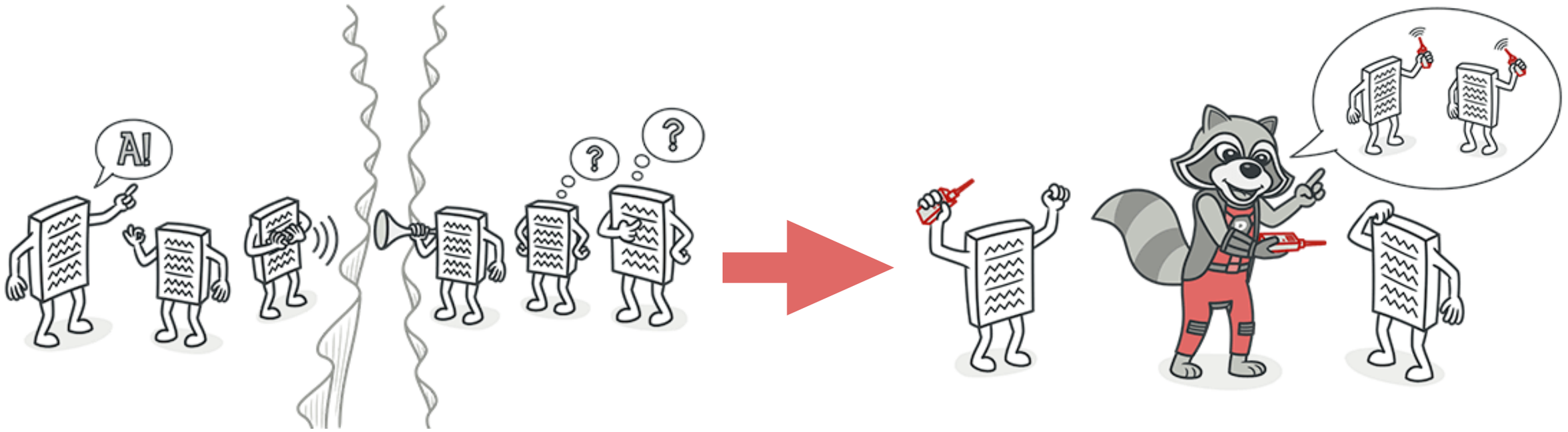


https://refactoring.guru/smells/inappropriate-intimacy

Message Chains

In code you see a series of calls resembling

```
$a->b()->c()->d()
```



https://refactoring.guru/smells/message-chains

Middle Man

When a class exists just to delegate to another, a developer should ask themselves what is its real purpose.

Sometimes this is the result of a refactoring task, where logic has been moved out of a class gradually, leaving an almost empty shell

```
class Pet {
  private Animal animal;

  public String getName() {
    return animal.getName();
  }

  public String getBreed() {
    return animal.getBreed();
  }

  public Owner getOwner() {
    return animal.getOwner();
  }
}
```

https://refactoring.guru/smells/middle-man

References

Design Patterns: online reading resources

- Adapter
  - https://sourcemaking.com/design_patterns/adapter
  - https://www.oodesign.com/adapter-pattern.html
- Code Smells
  - https://refactoring.guru/refactoring/smells/couplers