MSO
Design Patterns: Bridge

Hans Philippi

September 19, 2018

The Bridge Pattern

Intent:

Decouple an abstraction from its implementation so the two can vary independently

The Bridge pattern

- The Bridge pattern is one of the more complicated patterns we will see – but once you understand the principles of object oriented design, it should make sense
- On the other hand, it's class diagram shows a remarkable similarity with the Strategy pattern

So remember:

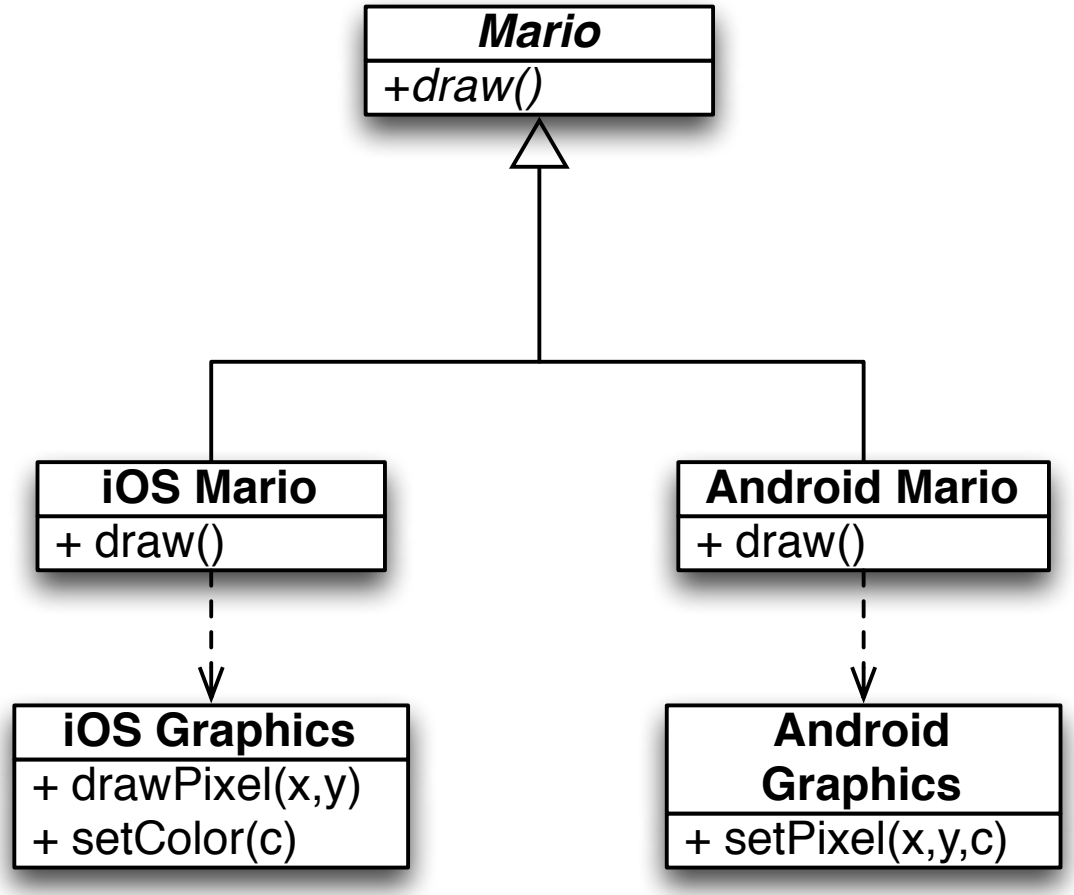
- Favour aggregation over inheritance
- Find what varies and encapsulate it

Towards the Bridge pattern

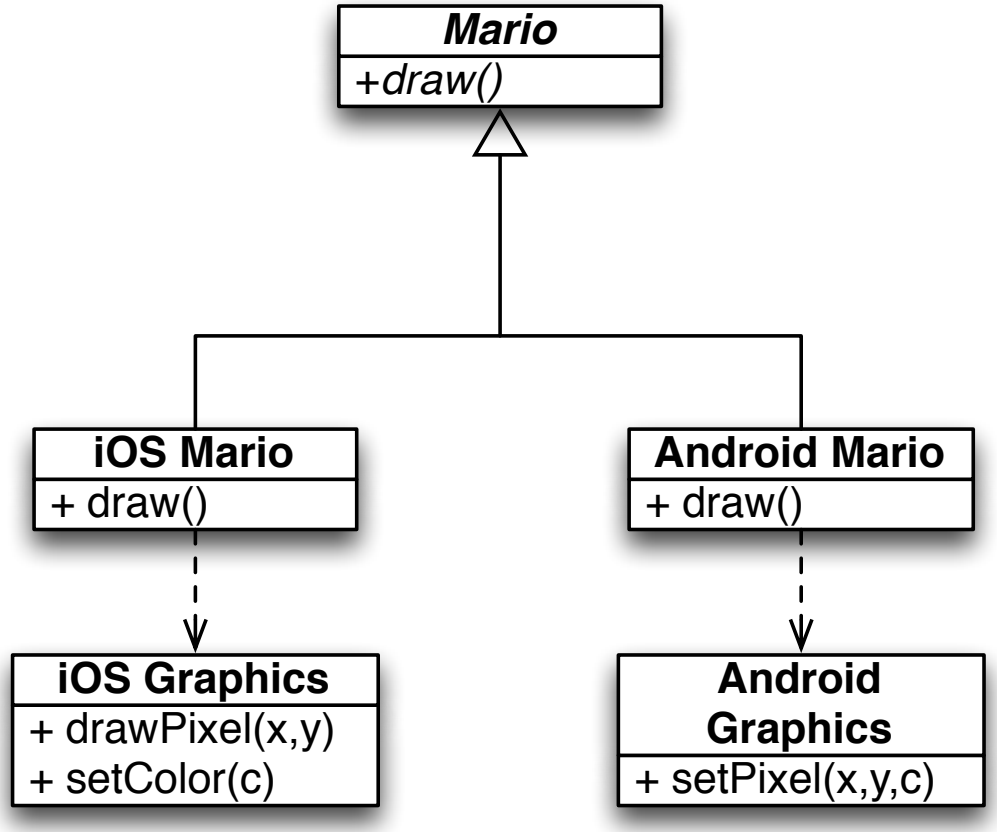
Case study:

- Suppose I have to write a game that works on different platforms
- Both these platforms have a similar graphics library, but there are subtle differences in the interface
- How do I encapsulate the variation?

Use inheritance!



Use inheritance

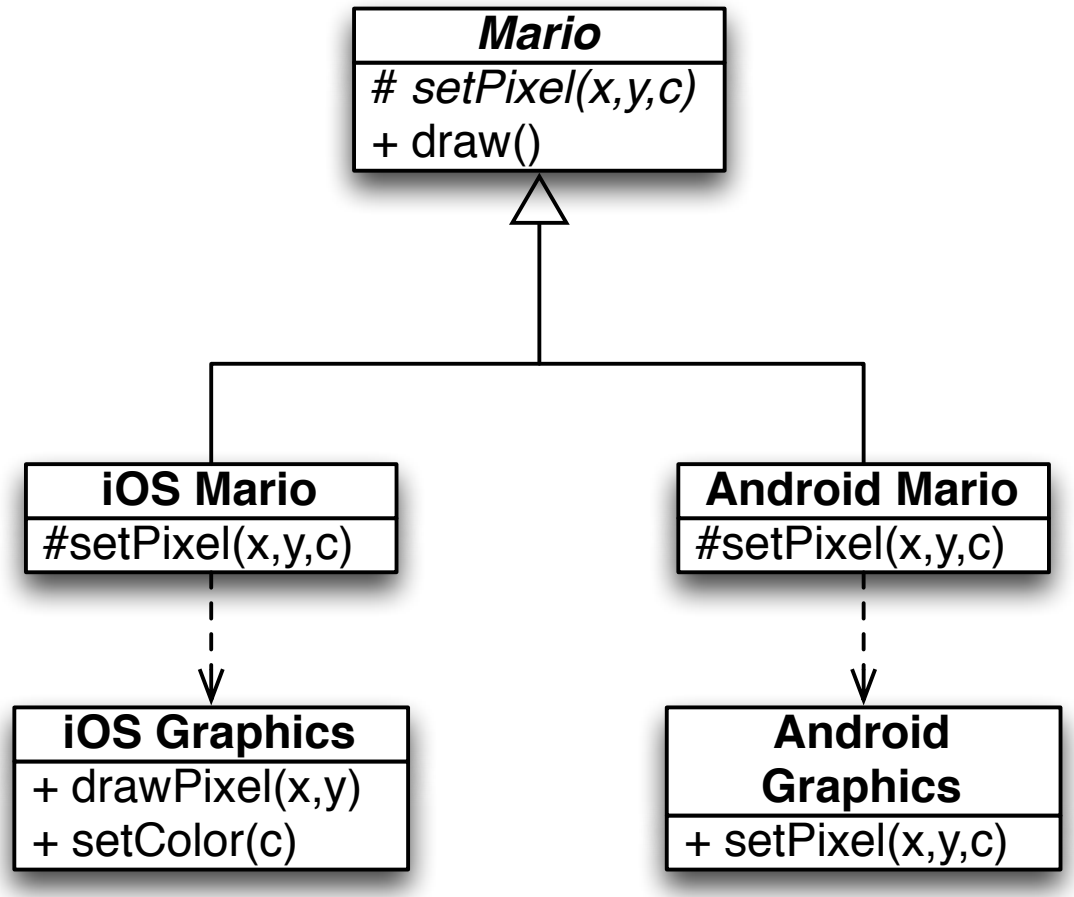


Question: Where might there be duplicate code? What happens when I need to ship to another platform?

Code duplication

- The two draw methods are probably very similar
- Duplicate code is a Very Bad Thing
- Can we avoid this duplication?

Inheritance and overloading

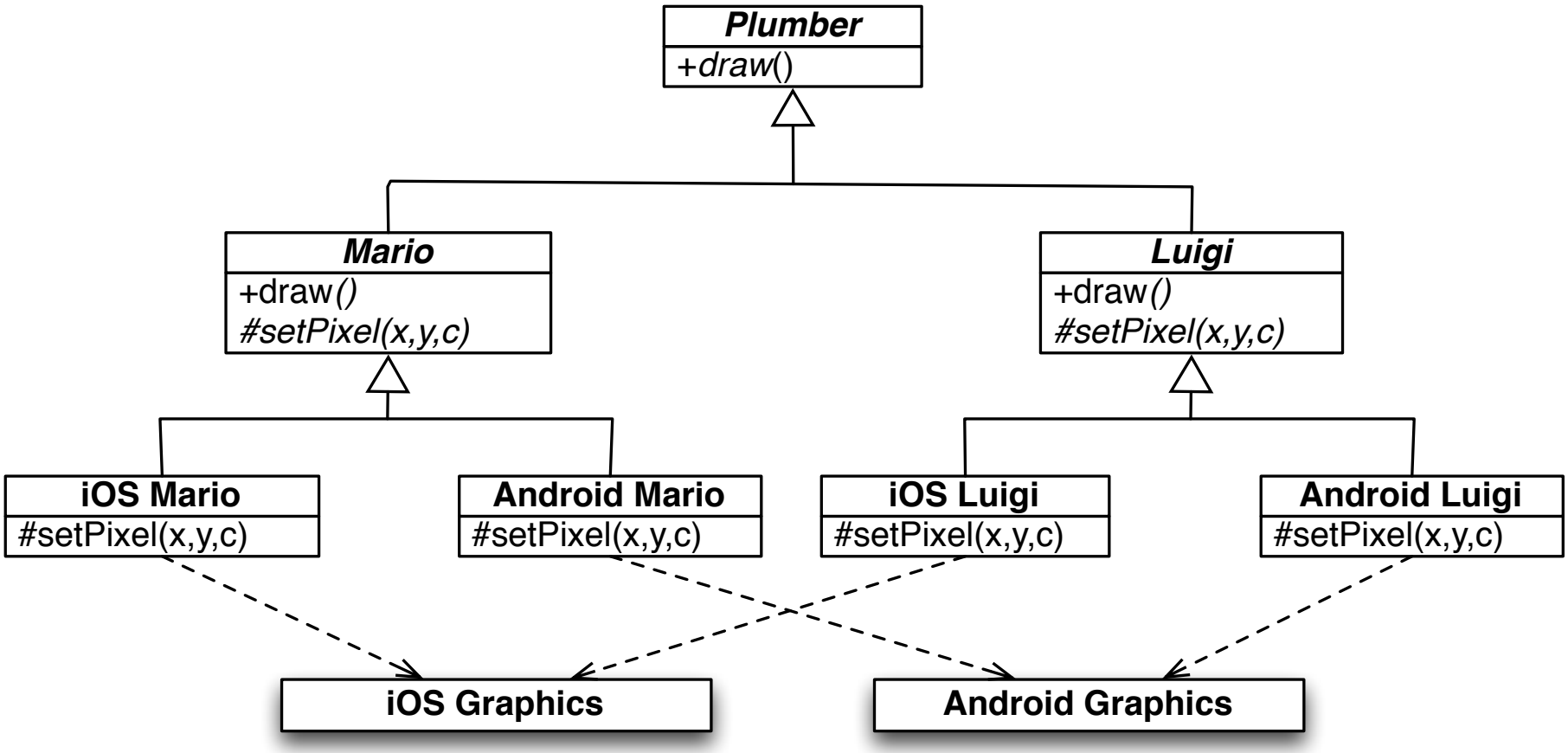


But then . . .

Of course, the requirements will change. Suppose we need to draw both Mario and Luigi with both drawing programs . . .

- Introduce an abstract class *Plumber*
- Introduce abstract subclasses, *Mario* and *Luigi*, of the *Plumber* class
- Introduce two concrete implementations for both drawing programs for both the *Mario* and the *Luigi* class

Another design



Evaluating this design

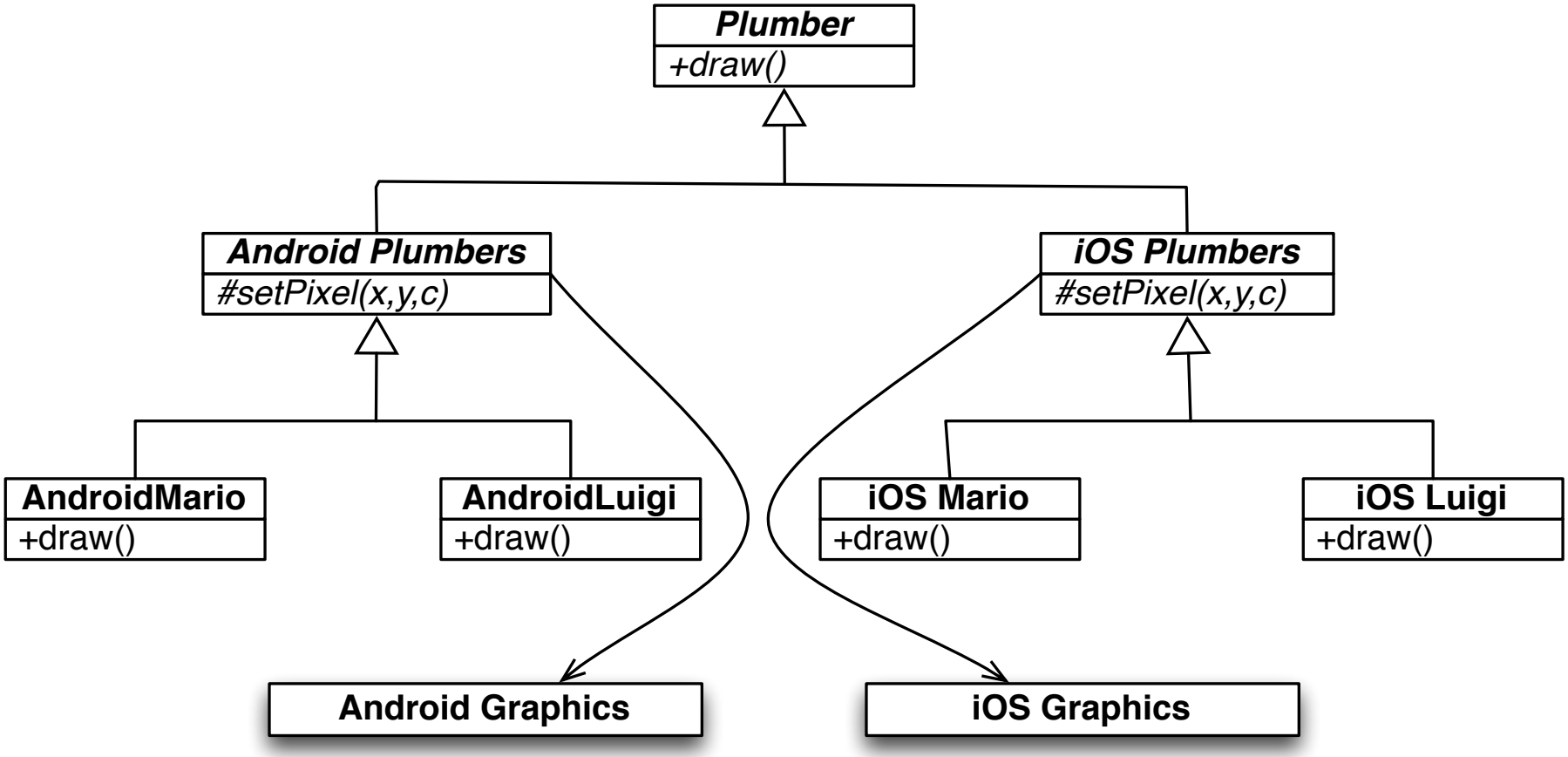
This solves our original problem but:

- Introducing a new Plumber requires the definition of three new classes (one abstract and one implementation for each drawing program)
- Introducing a new Graphics platform requires a new subclass for every Plumber
- The abstraction (the different kinds of Plumbers) and their implementation (the different kinds of Graphics classes) are *tightly coupled*
- There is *redundancy* – the different concrete Luigi or Mario classes may share a lot of code

Can we do better?

- So maybe this was not the right design choice
- Perhaps we should introduce two abstract Plumber classes, one for each platform program ...

Can we do better?



Question: Problem solved?

Evaluating this design

This design does split things up differently, but it still suffers from many of the same disadvantages as our previous approach:

- Duplicate code in the Android Mario and iOS Mario classes (and also for Luigi of course)
- Introducing new platforms requires a new abstract class, together with a new concrete implementation for each different Plumber
- Introducing a new Plumber requires a new concrete implementation for each different Graphics platform
- The implementations and abstractions are still too tightly coupled

The Bridge pattern

Intent:

- Decouple an abstraction from its implementation so the two can vary independently
- Here we have precisely this situation: we have different Plumbers (*abstractions*) that we want to draw with different Graphics platforms (*implementations*)
- How can we decouple the abstractions and implementations?

The Bridge pattern

Let's try to *derive* the Bridge pattern ourselves, by simply applying some of the principles of good object oriented design:

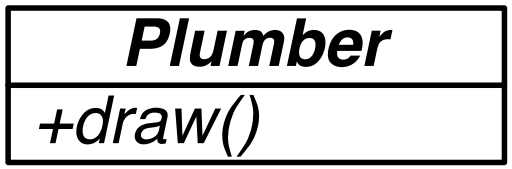
- Find what varies and encapsulate it
- Favour aggregation over inheritance
- Program to an interface, not an implementation

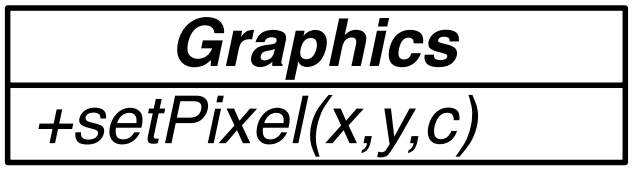
Commonality-variability analysis

We want to support different kinds of plumbers and different kinds of Graphics platforms:

- We want to be able to draw a Plumber (although Mario and Luigi get drawn differently)
- Our drawing programs support the drawing of individual pixels (even if they do so differently)

Starting the design...





This tries to capture the *commonality* between the different Plumbers and the different Graphics platforms

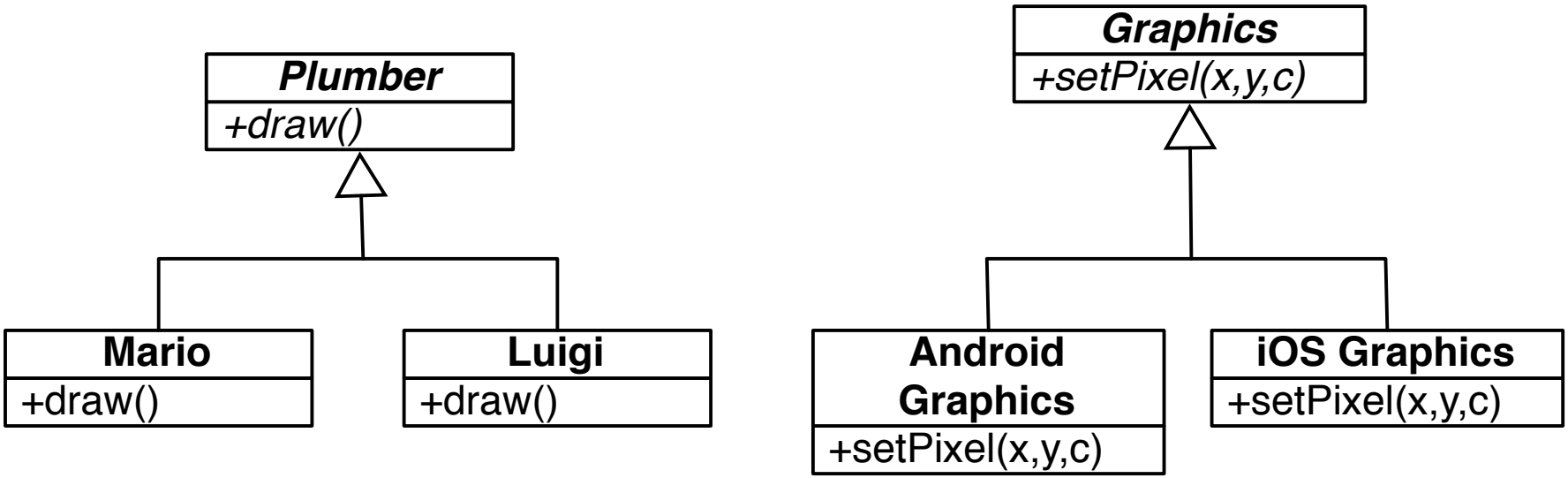
Introducing variation

Next we want to introduce some variation:

- We have two different kinds of Plumber
- We have two different kinds of Graphics

Let's try adding some subclasses ...

Introducing variation



Find what varies and encapsulate it

How are the two related?

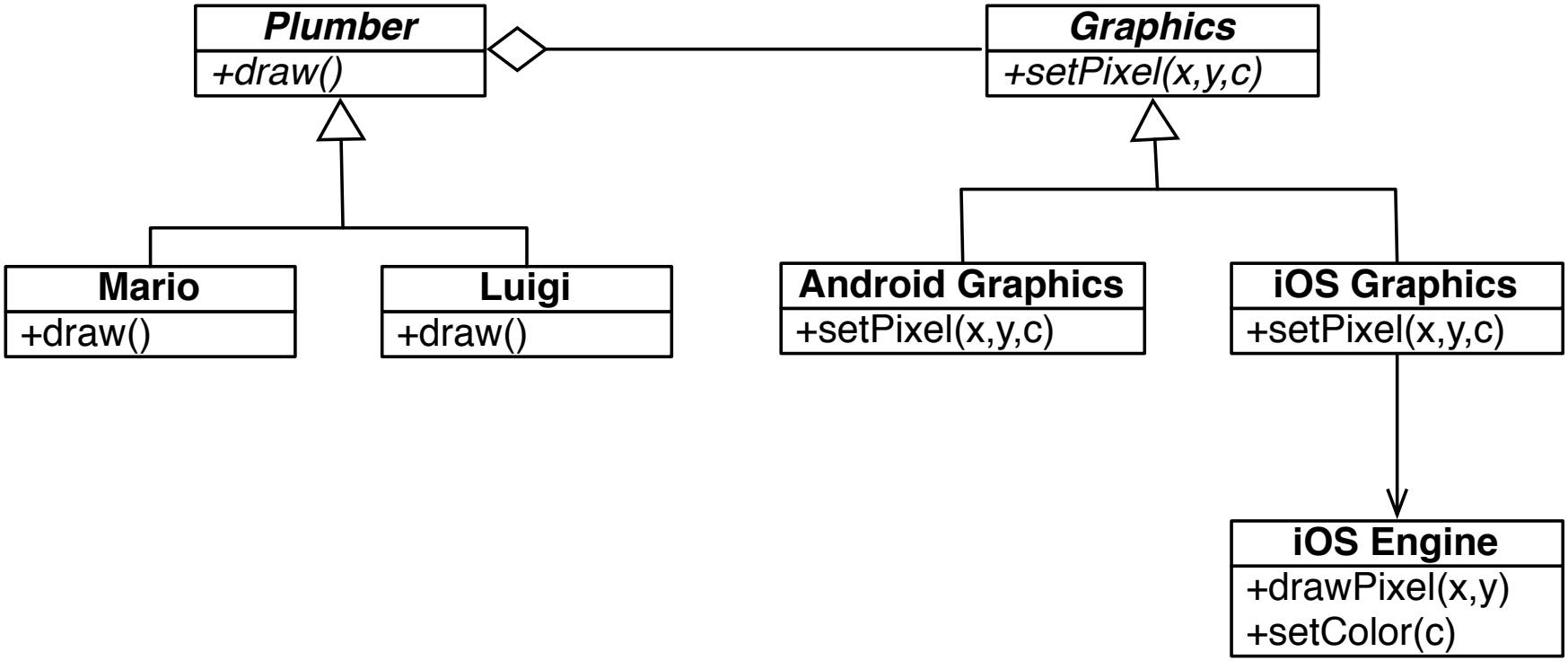
Favour aggregation over inheritance – let's not try to introduce any further subtypes for the moment.
Should Graphics use a Plumber object?

- Probably not. Then the Graphics would need to inspect what *kind* of Plumber they have to figure out how to draw them.

Should Plumber use Graphics?

- Yes! When a Plumber needs to be drawn, we will use functions from Graphics . . .
- . . . and we do not want to worry about the platform (Android/iOS)
- We are free to use different drawing programs, because the Plumber class *programs to an interface, not an implementation*.

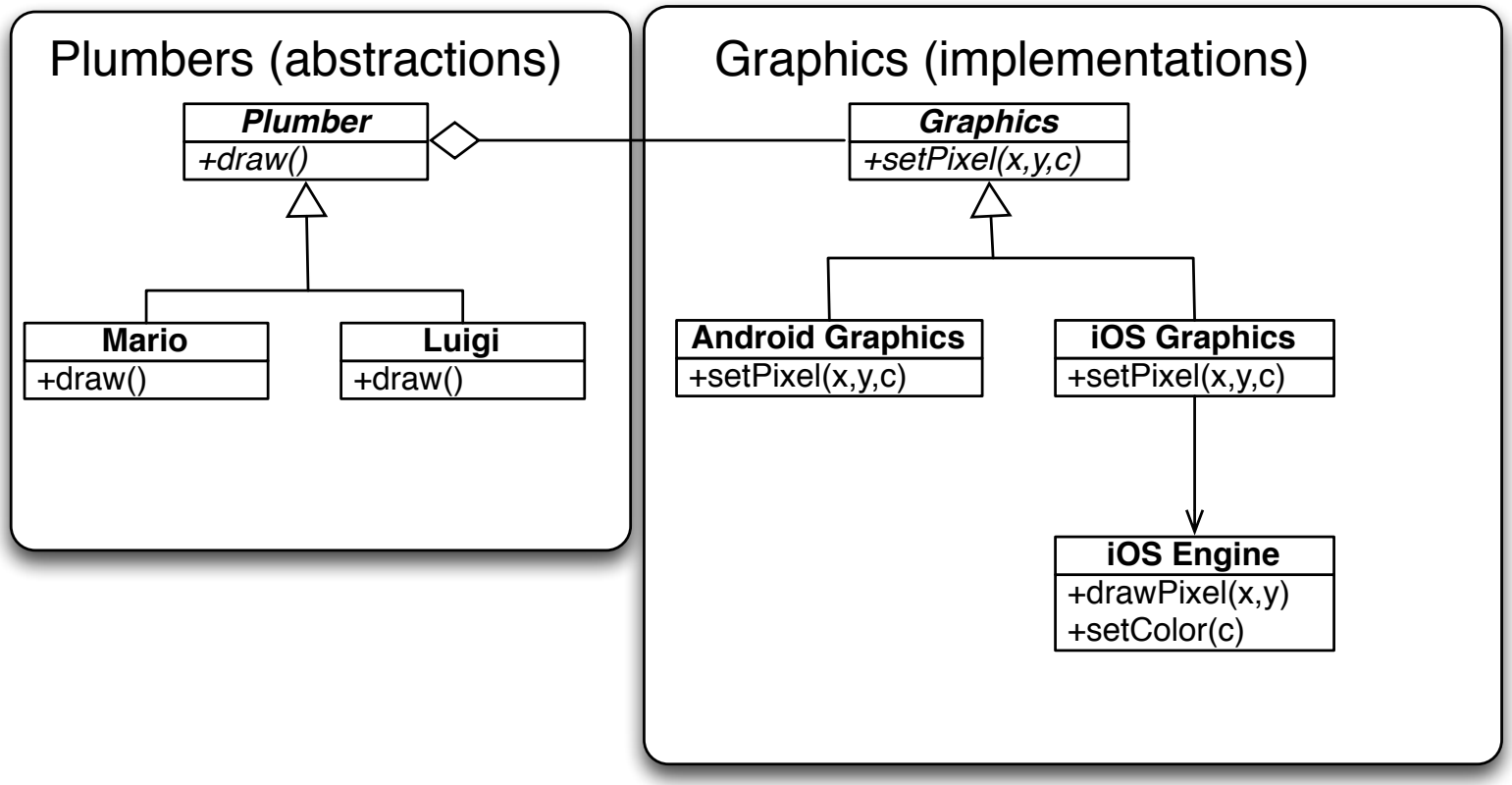
The Bridge pattern



Evaluating the Bridge

How well does this separate implementations (Graphics) from the abstractions (Plumbers)?

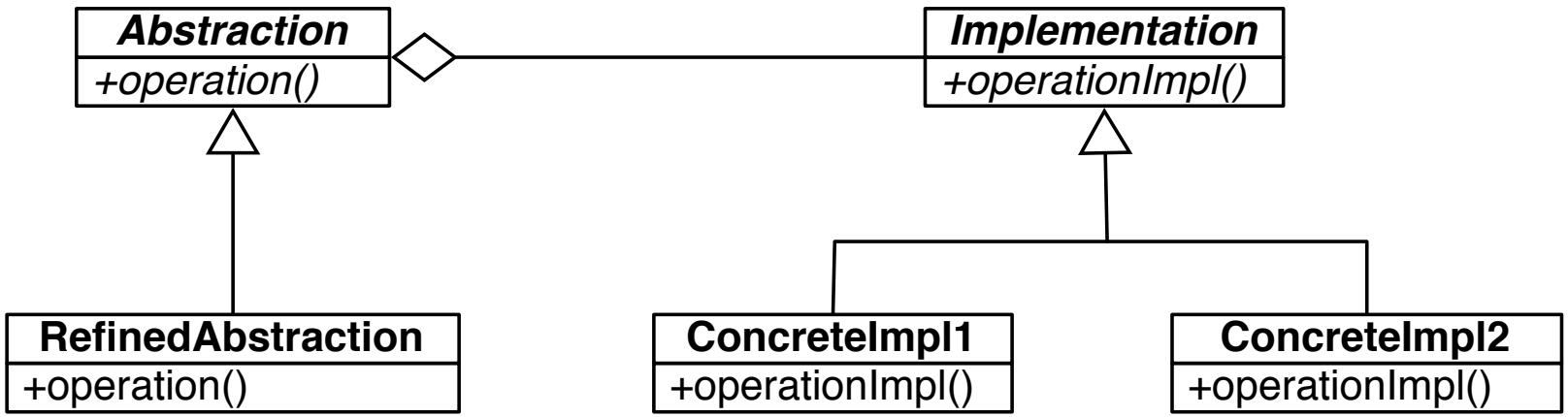
Evaluating the Bridge



The Bridge Pattern

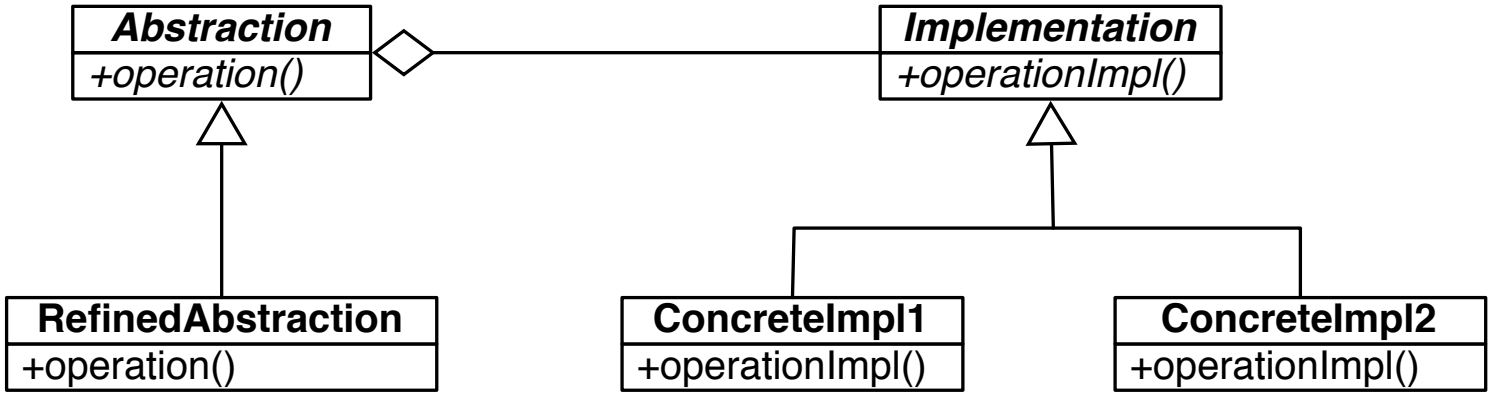
Intent: Decouple a set of implementations from the objects using them.
Problem: The derivations of an abstract class must use multiple implementations, without causing an explosion in the number of classes;
Solution: Define an interface for all implementations to use and have the derivations of the abstract class use that
Consequences: The decoupling of the implementations from the objects that use them increases extensibility: client objects are not aware of implementation issues

Bridge Pattern: Implementation



Bridge & Strategy: class diagrams

- Compare the generic Bridge diagram with the diagram of Strategy (slide 24)
- Do you see similarities?
- Try to explain the differences



One rule, one place

Design principle: If you have a rule for how to do something, only implement it *once*

- Maintainability: if the rule changes, there is only piece of code to update
- Cohesion: the responsibility for this computation is in a single place

Material covered

- Design Patterns explained: chapter 9 – 10