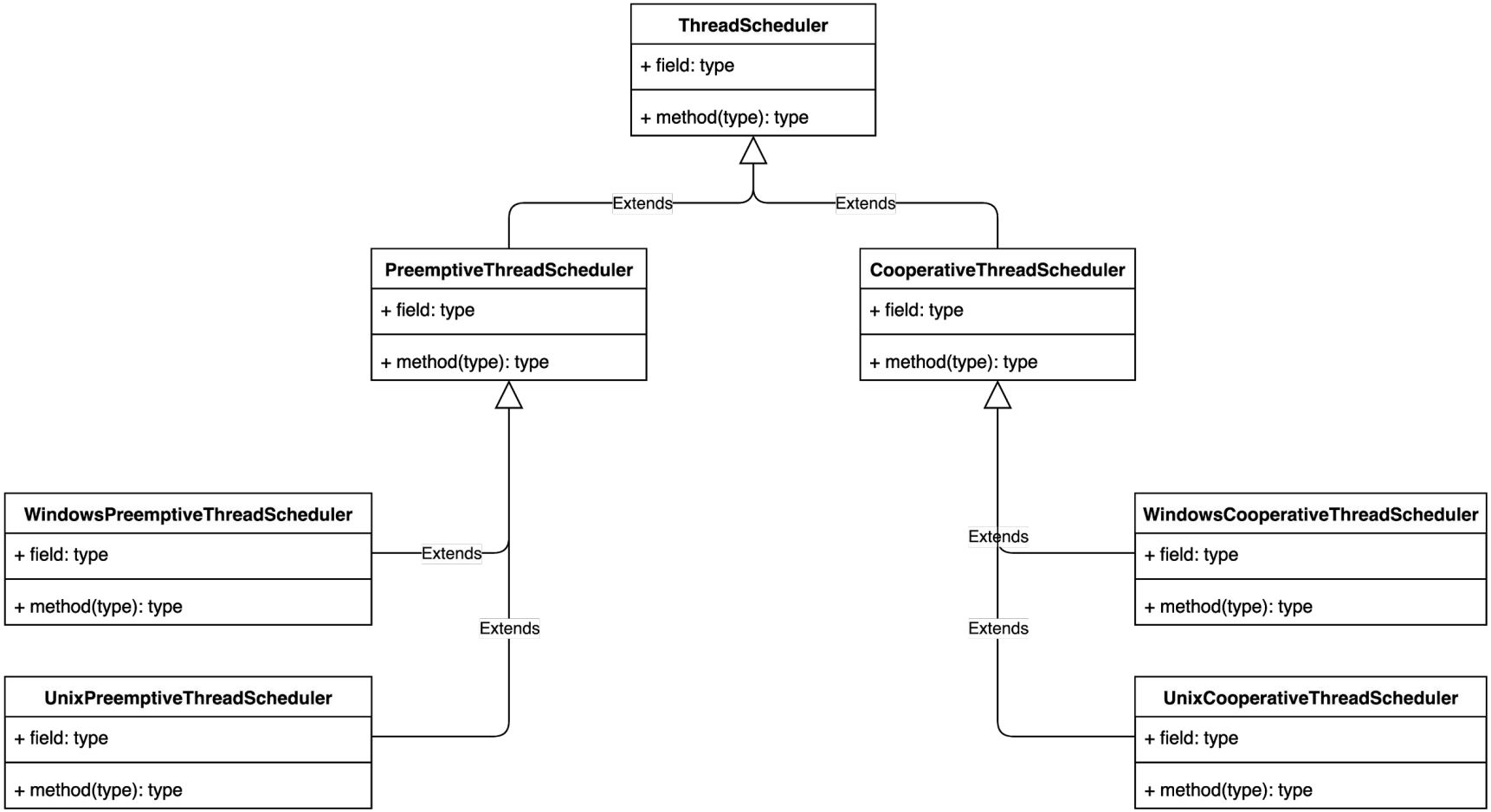
Bridge

By Alexander Tilkin

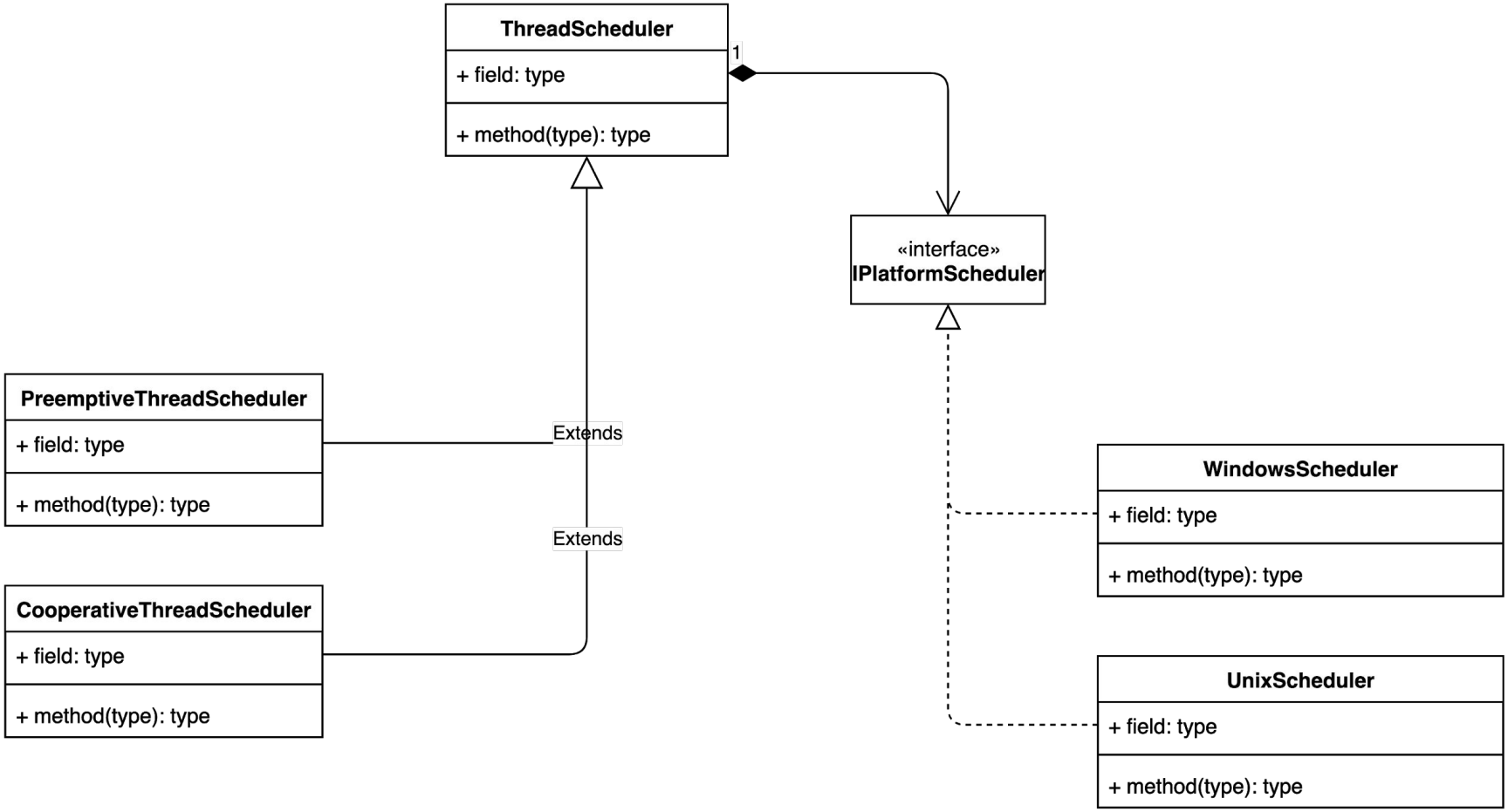
Motivation

- Connecting components together through abstraction
- Bridge prevents a “cartesian product” complexity explosion
- Example:
  - Base class ThreadScheduler
  - Can be preemptive or cooperative
  - Can run on WIndows or Unix
  - End up with 2x2 scenario: WindowsPts, UnixPts, WindowsCts, UnixCts
- The bridge pattern avoid the entity explosion

Before


ThreadScheduler
+ field: type
+ method(type): type
Extends
Extends
PreemptiveThreadScheduler
+ field: type
+ method(type): type
CooperativeThreadScheduler
+ field: type
+ method(type): type
WindowsPreemptiveThreadScheduler
+ field: type
+ method(type): type
Extends
Extends
UnixPreemptiveThreadScheduler
+ field: type
+ method(type): type
Extends
Extends
WindowsCooperativeThreadScheduler
+ field: type
+ method(type): type
UnixCooperativeThreadScheduler
+ field: type
+ method(type): type

After


ThreadScheduler
+ field: type
+ method(type): type
1
«interface»
IPlatformScheduler
PreemptiveThreadScheduler
+ field: type
+ method(type): type
Extends
Extends
CooperativeThreadScheduler
+ field: type
+ method(type): type
WindowsScheduler
+ field: type
+ method(type): type
UnixScheduler
+ field: type
+ method(type): type

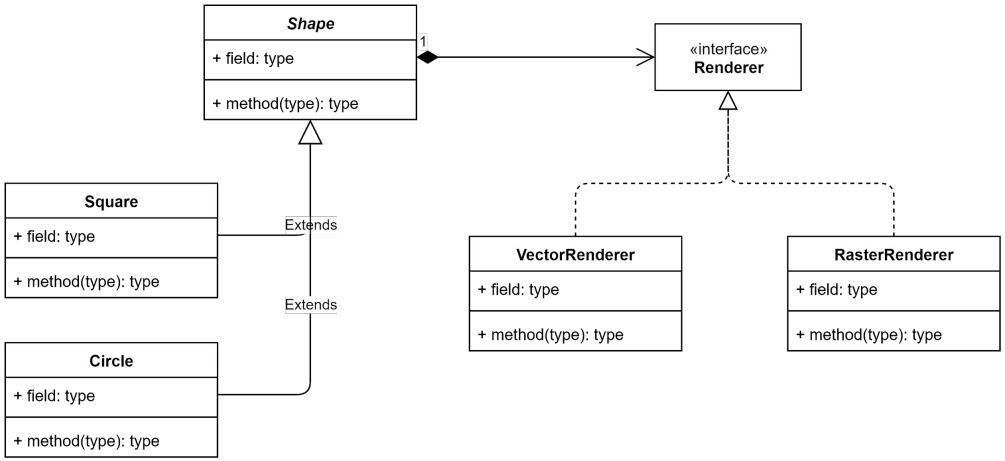
Summary

- Decouples abstraction from implementation
- Both can exist as hierarchies
- A stronger form of encapsulation

units.unit_07_bridge.BridgeExample.java

Bridge Example

- In this example we implement an application that draws a shape using different rendering techniques
- Renderers: Vector, Raster
- Shapes: Circle, Square
- Using traditional object oriented hierarchy technique we will end up with four classes: CircleVectorRenderer, CircleRasterRenderer, SquareVectorRenderer, SquareRasterRenderer
- If we will add another render we will have six classes and so on
- Instead we will create an interface Render with rendering methods; Renderer classes that implements the interface; abstract class that composites the interface; and classes that implements the abstract class
- You can read about Google Guice here



units.unit_07_bridge.BridgeExercise.java

Exercise

- You are given an example of an inheritance hierarchy results in Cartesian-product duplication
- Refactor this hierarchy, giving the base class Shape a constructor that takes an interface called Renderer defined as the code in the slide
- As well as **VectorRenderer** and **RasterRenderer** classes. Each implementer of the **Shape** interface should have a constructor that takes an **Renderer** such that, subsequently, each constructed object's toString() operates correctly and produces a message similar to the following code in the link above

```
interface Renderer{
        String whatToRenderAs();
}
```

Summary

A mechanism that decouples an interface (hierarchy) from the implementation (hierarchy)