# Exercise #1: Adding Code for Changing Shapes (Problem Statement)

- Imagine a legacy system with an existing subsystem for *drawing* shapes
- New requirement: Support also *changing* of shapes
  - Example: Change an oval into a rectangle 
- The manager says: "Do not touch the existing code!"

**Draw subsystem**

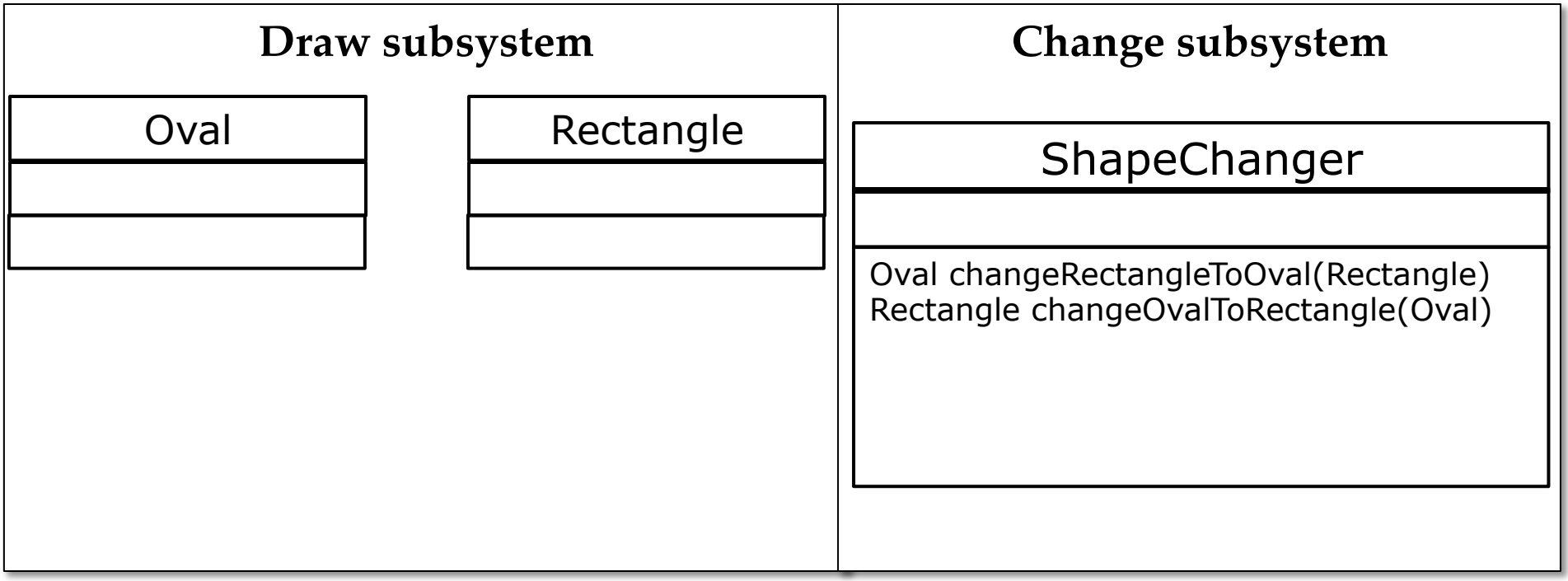
| Oval | Rectangle |
|---|---|
| id: UUID<br>width: int<br>height: int<br>x: int<br>y: int | id: UUID<br>width: int<br>height: int<br>x: int<br>y: int |
| Oval(int, int, int, int)<br>draw()<br>toString():String | Rectangle(int, int, int, int)<br>draw()<br>toString():String |

**Change subsystem**

| ShapeChanger |
|---|
| |
| Oval changeRectangleToOval(Rectangle)<br>Rectangle changeOvalToRectangle(Oval) |

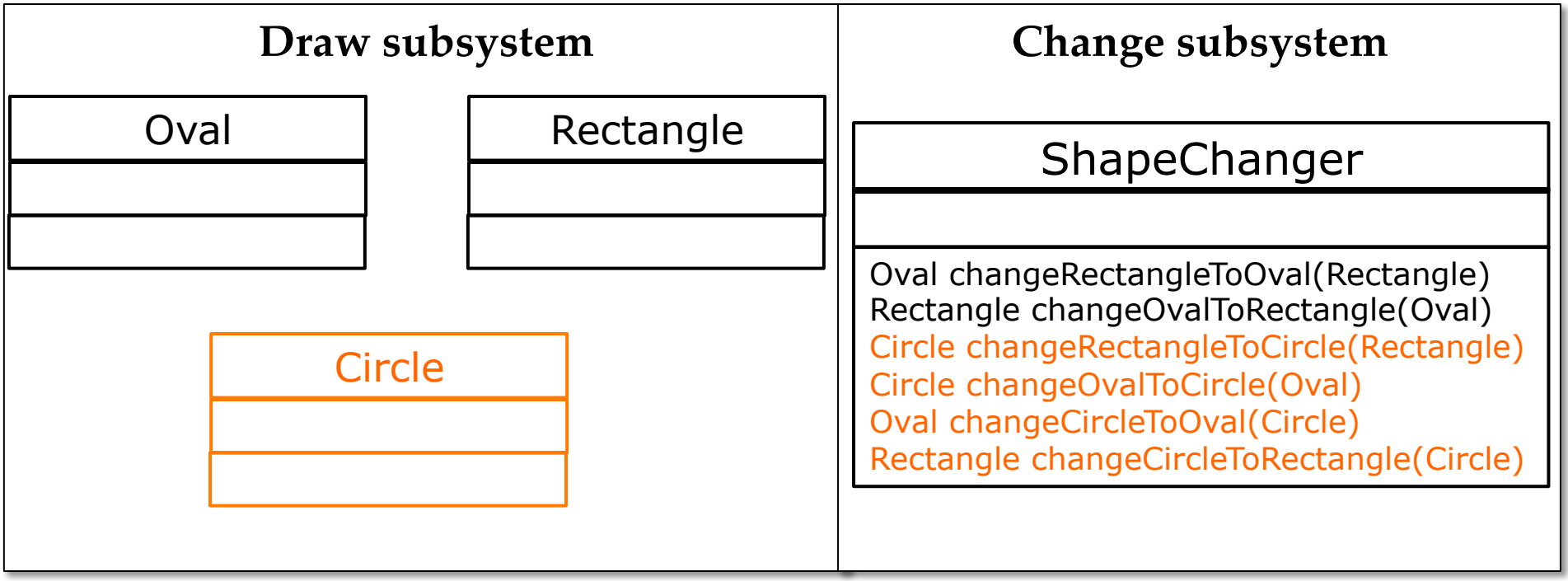
# Exercise #1: Adding Code for Changing Shapes (Problem Statement)

- The functional decomposition leads to the code implementing change functionality being in a separate subsystem
- For each combination of Shapes code implementing change functionality has to be written



# Exercise #1: Adding Code for Changing Shapes (Problem Statement)

- The functional decomposition leads to the code implementing change functionality being in a separate subsystem
- For each combination of Shapes code implementing change functionality has to be written
  - Especially adding new shapes becomes expensive

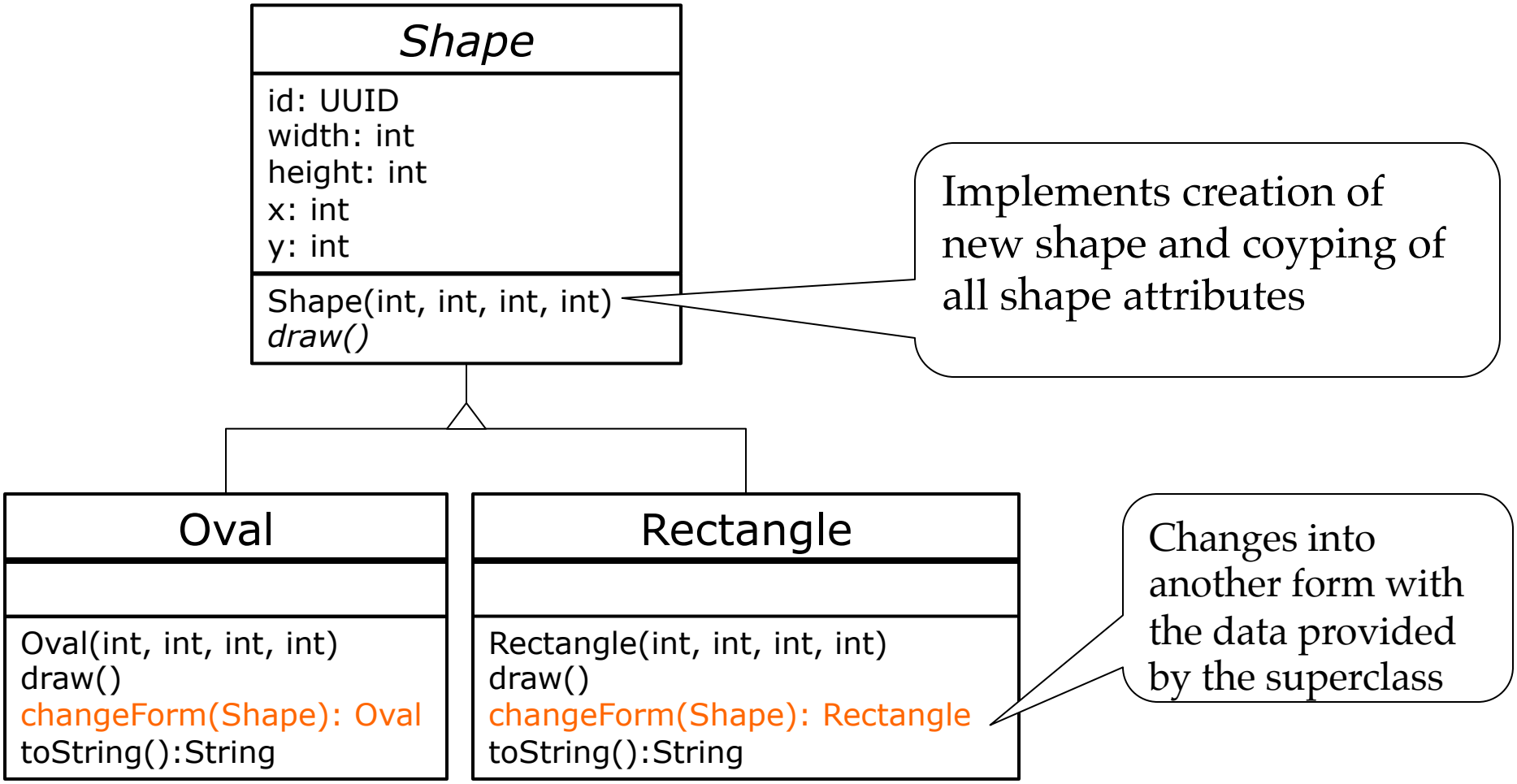


# Task #1: Adding Code for Changing Shapes

15 min

- Re-engineer the legacy code to produce a refactored solution
  - Step 1: Introduce an abstract Shape class
  - Step 2: Introduce a method changeForm(Shape) that transforms one shape into another
  - Step 3: Delete ShapeChanger
    - The new solution does not use the ShapeChanger class anymore
- Benefit
  - Shape instance can be changed at runtime
  - After you are done, adding a new shape subclass should be possible without changing the code for the existing shapes.

# Task #1: Adding Code for Changing Shapes



# Task #1: Adding Code for Changing Shapes