Reusable and modular code

Jean-Luc Delarbre

Some key rules for better code

- **Take care of object state**
  - Prefer immutable as much as possible
  - Use ‘final’ keyword
  - Assignments are never harmless
- **Every implementation should derive from an interface**
  - Find the right abstractions
- **Factories (or builder)**
  - Only them are allowed to create object instances (use new)
  - Inject dependencies to object
  - Do not use static instances

Some real case study

- **Let’s refactor some errors found in production code**
  - Using previous rules
  - Refacto1
    - Encapsulation breaking with unnecessary setters
    - Ensure correct object initialization with constructor
    - Communicate intent with Optional (avoid null for initialization - ambiguous)

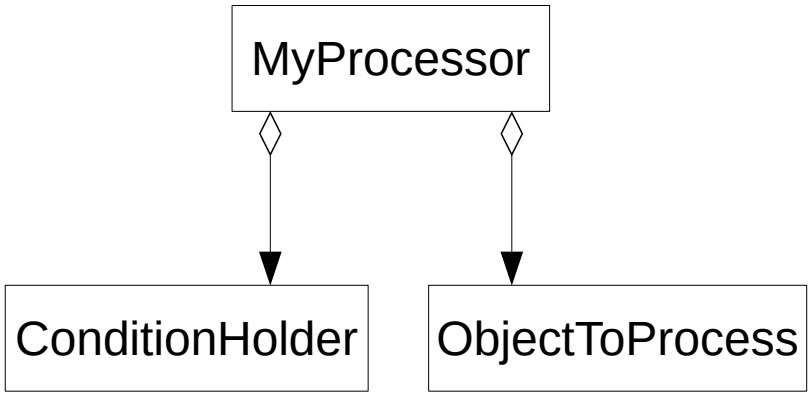
- Refacto2
  - Avoid in out parameters
  - Functions return value and do not modifiy object state
    - INPUT - PROCESS - OUTPUT
  - Procedure modify state and do not compute anything (information about mutation could be returned)
  - Assignment is a serious business

Reusable and modular code

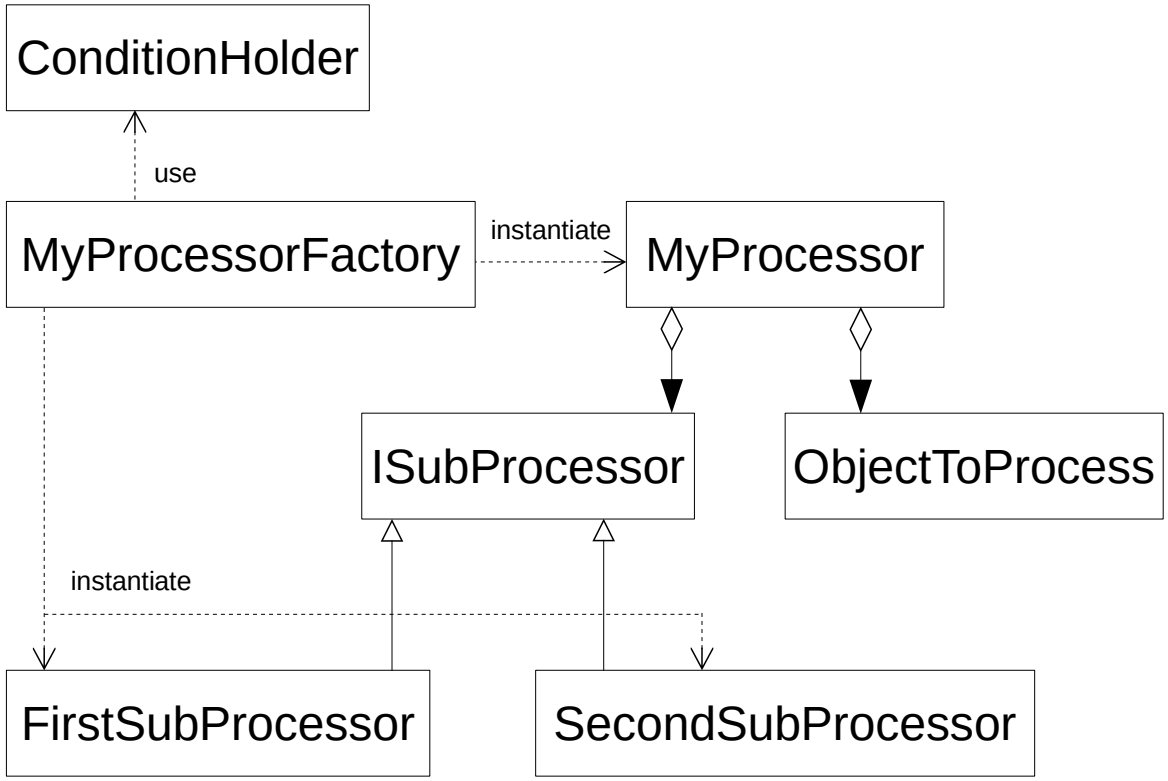
- **Code reuse**
  - is not copy-paste
  - is allowed by
    - definition of abstraction
    - dependency injection
  - See refacto3

Difficulties with modular architecture

Basic implementation



Modular architecture
Application wiring could be complex



Simplest case of wiring

- **Separation of interface and implementation**
- **Separation of responsibility**
  - Business logic
  - Application wiring

  - See refacto4

Implementation strategies

- **Beginning with a modular architecture ?**
- **Beginning with a basic implementation and evolve to a modular architecture when needed**

- **More on that later ?**

Next objective

- **Building testable code and test it**

Reference

- **SOLID principles**
  - Open/closed principle
  - Dependency inversion principle
  - (Download articles in ‘references’ section of Wikipedia article: SOLID)

Code organization (for java)

- **See organizations example**

	Dedicated implementation and factory classes	Factory as nested class in implementation class	Class that hold every factories in package	Factory on top with nested private implementation
Compacity	--	++	+	++
Implementation class hidden	++	--	++	++
Code change synchronization	+	++	-	++