OO design

Are really objects just nouns and methods verbs?

Objects

encapsulation

polymorphism

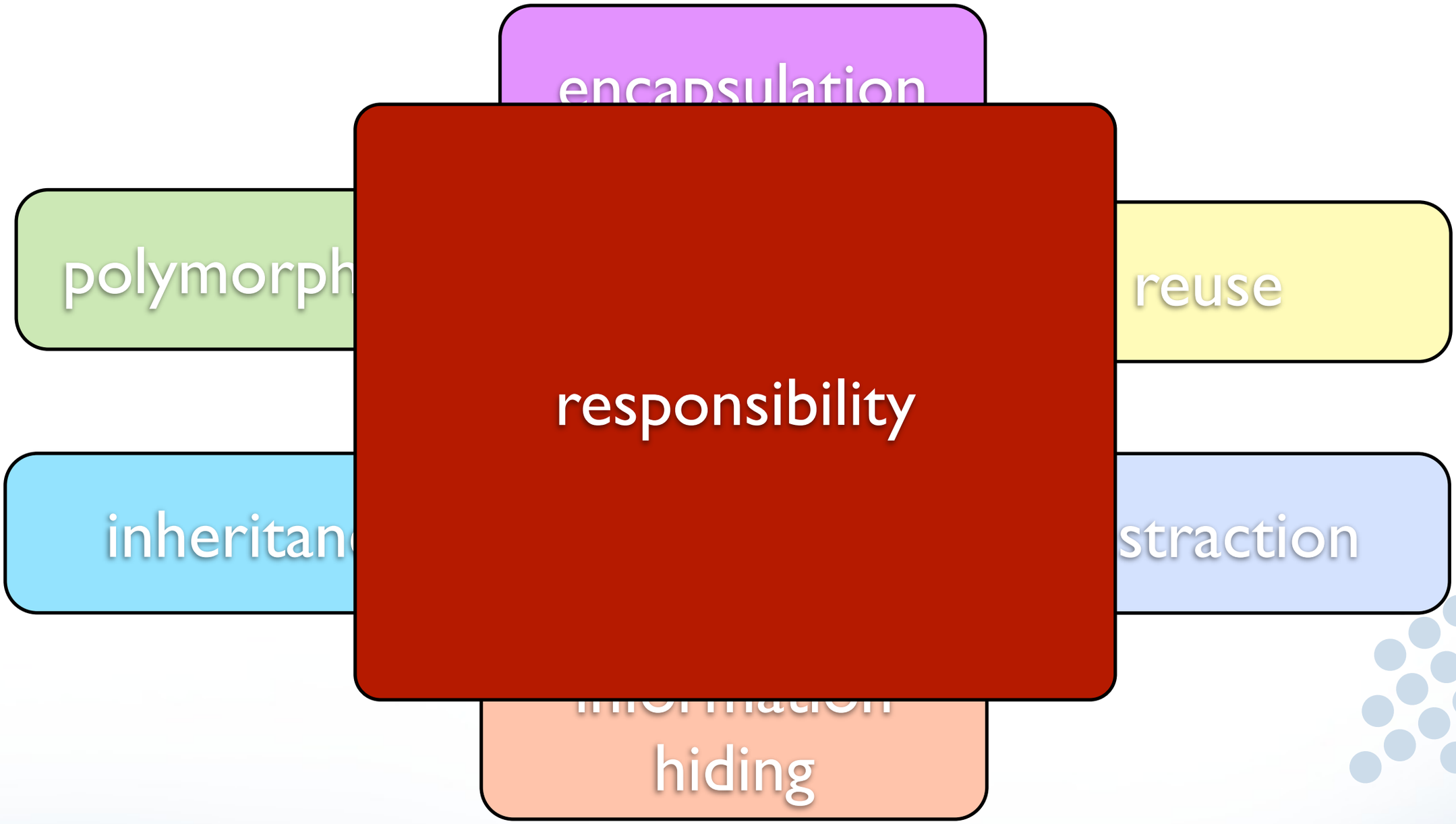
reuse

inheritance

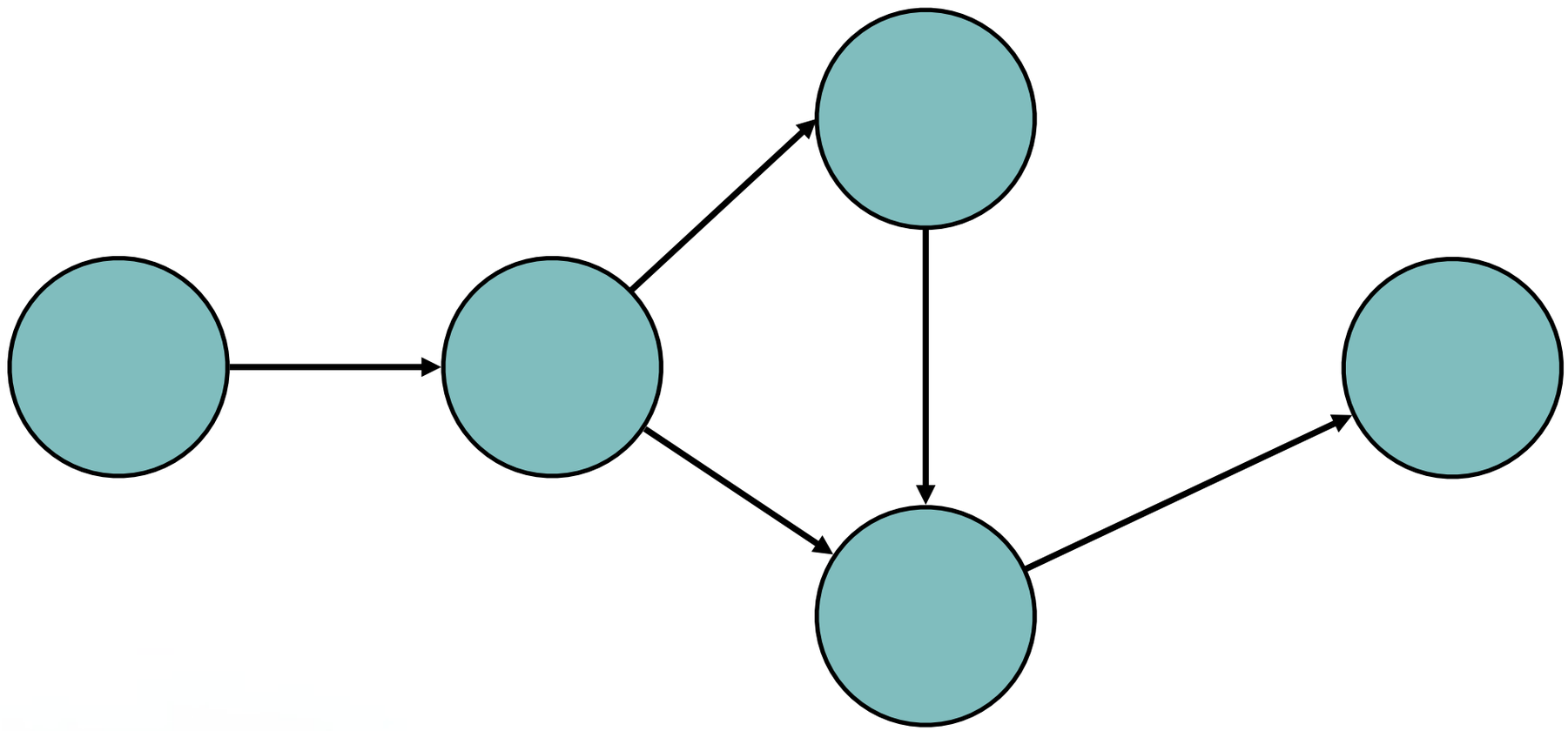
abstraction

information hiding

Objects


encapsulation
polymorph
reuse
responsibility
inheritan
straction
hiding

A net of collaborating objects



Cohesion

Class

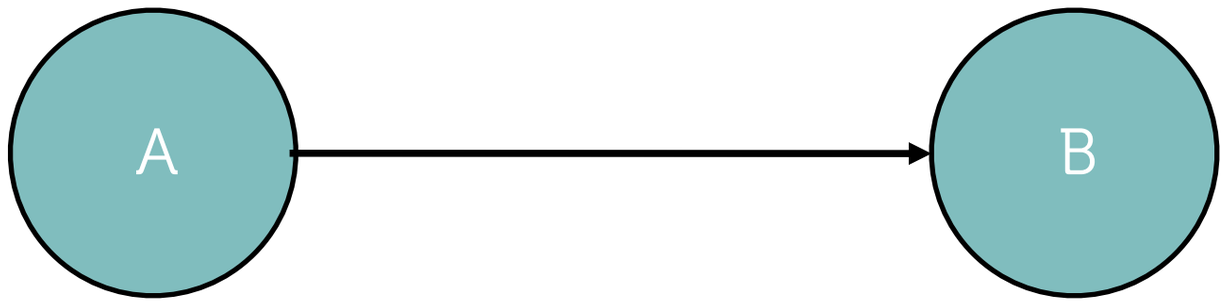
- methods relate to fields and other methods of the same class

Method

- instructions do what the name suggests
- one level of abstraction

Hi cohesion of classes and methods indicates good design.

Coupling



Strength is dependent on:

- number
- type (interface/concrete class)

Low level of coupling improves testability and eases maintenance.

Design heuristics

- based on experience
- help understand what makes code easy / hard to understand and maintain
- explain why I don't like some code and how to improve it
- SOLID, GRASP

What characterises bad design?

What characterises bad design?

- Impossible to change, each change influences many elements of the system (rigidness)
- Changes result in bugs in unexpected places (fragility)
- Code cannot be reused - it cannot be detached from the current component (immobility)

What is a good design?

- Highly modular structure composed of components of high cohesion and low coupling.
- Components have meaningful names, so that it’s easy to understand what they do and why they exist.

SOLID

Software Development is not a Jenga game

SOLID Principles

- S ingle Responsibility Principle
- O pen-Closed Principle
- L iskov Substitution Principle
- I nterface Segregation Principle
- D ependency Inversion Principle

slides from: http://lostechies.com/derickbailey/2009/02/11/solid-development-principles-in-motivational-pictures/

SINGLE RESPONSIBILITY PRINCIPLE

Just Because You Can, Doesn't Mean You Should

Single Responsibility Principle

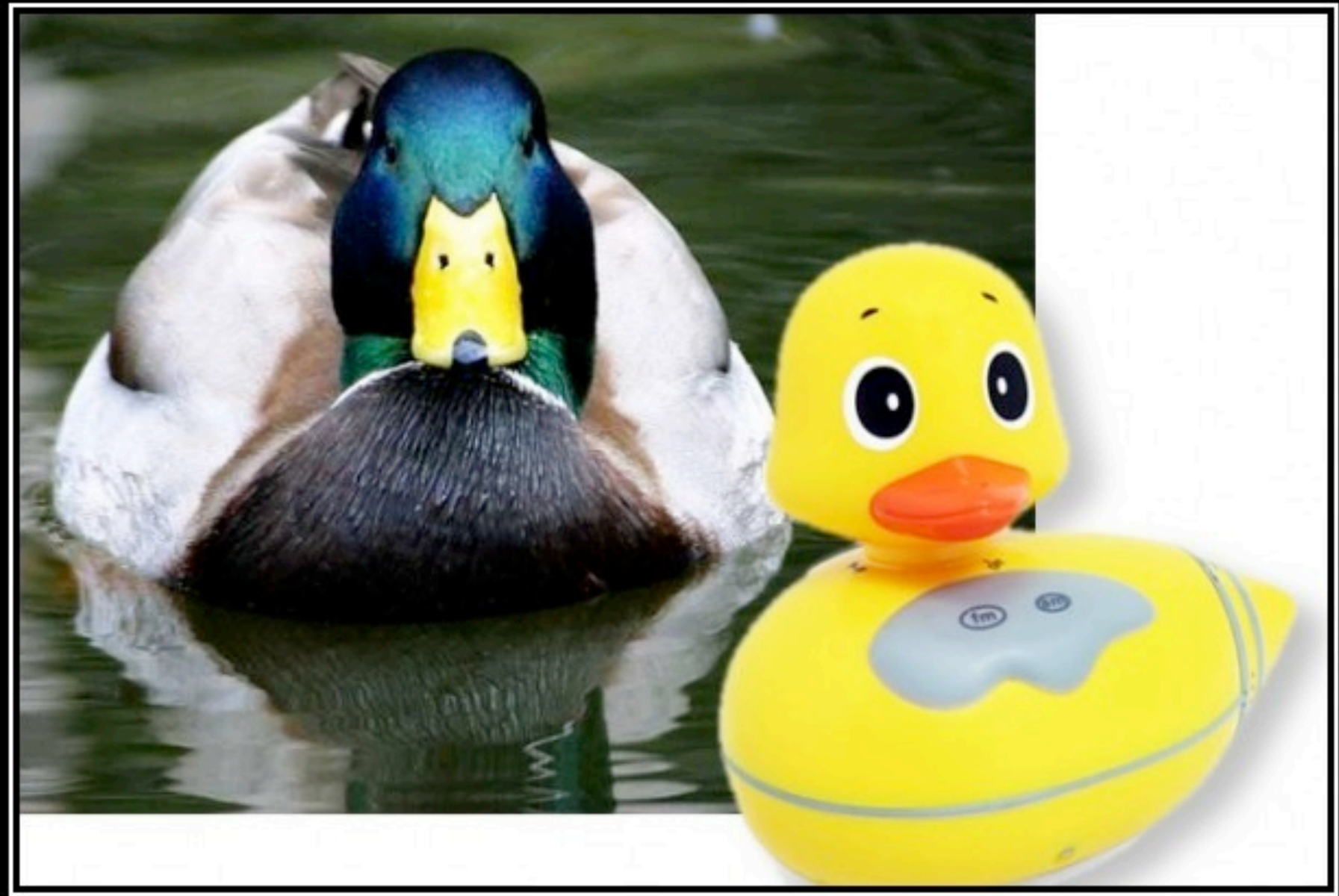
- For a given class there should be only one reason for change
- Component / class / method should have exactly one, well-defined responsibility

OPEN CLOSED PRINCIPLE

Open Chest Surgery Is Not Needed When Putting On A Coat

Open-Closed Principle

- Modules, classes, functions should be open for extensions, but closed for modifications.
- It should be easy to change a class' behaviour without modifying the class.



LISKOV SUBSTITUTION PRINCIPLE

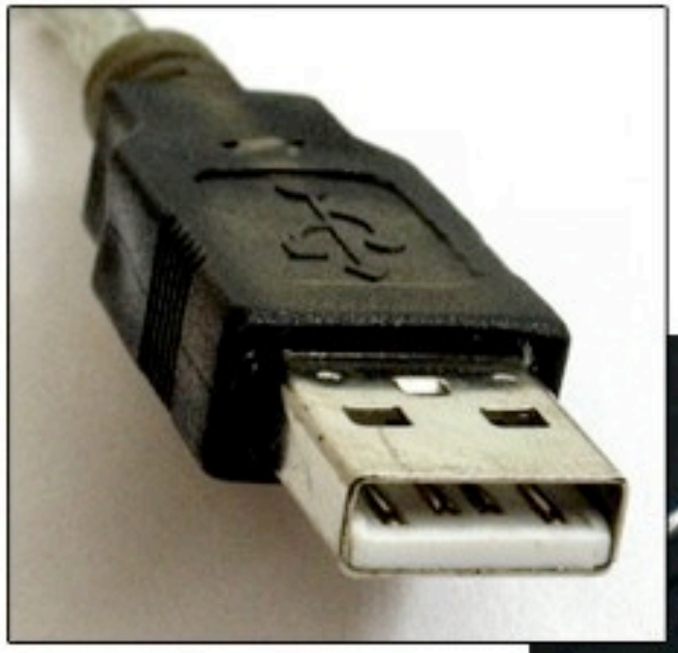
If It Looks Like A Duck, Quacks Like A Duck, But Needs Batteries - You Probably Have The Wrong Abstraction

Liskov Substitution Principle

- Subclasses must always be possible to use just like they were the superclass.
- You can use a subclass everywhere, where client uses the base class.

Liskov Substitution Principle

- You can use a subclass in place of a superclass if:
  - entry conditions in the subclass are not stronger than the the implementation in the superclass
  - exit conditions in the subclass are not weaker than the implementation in the superclass
- The overwriten methods should not expect more nor provide less



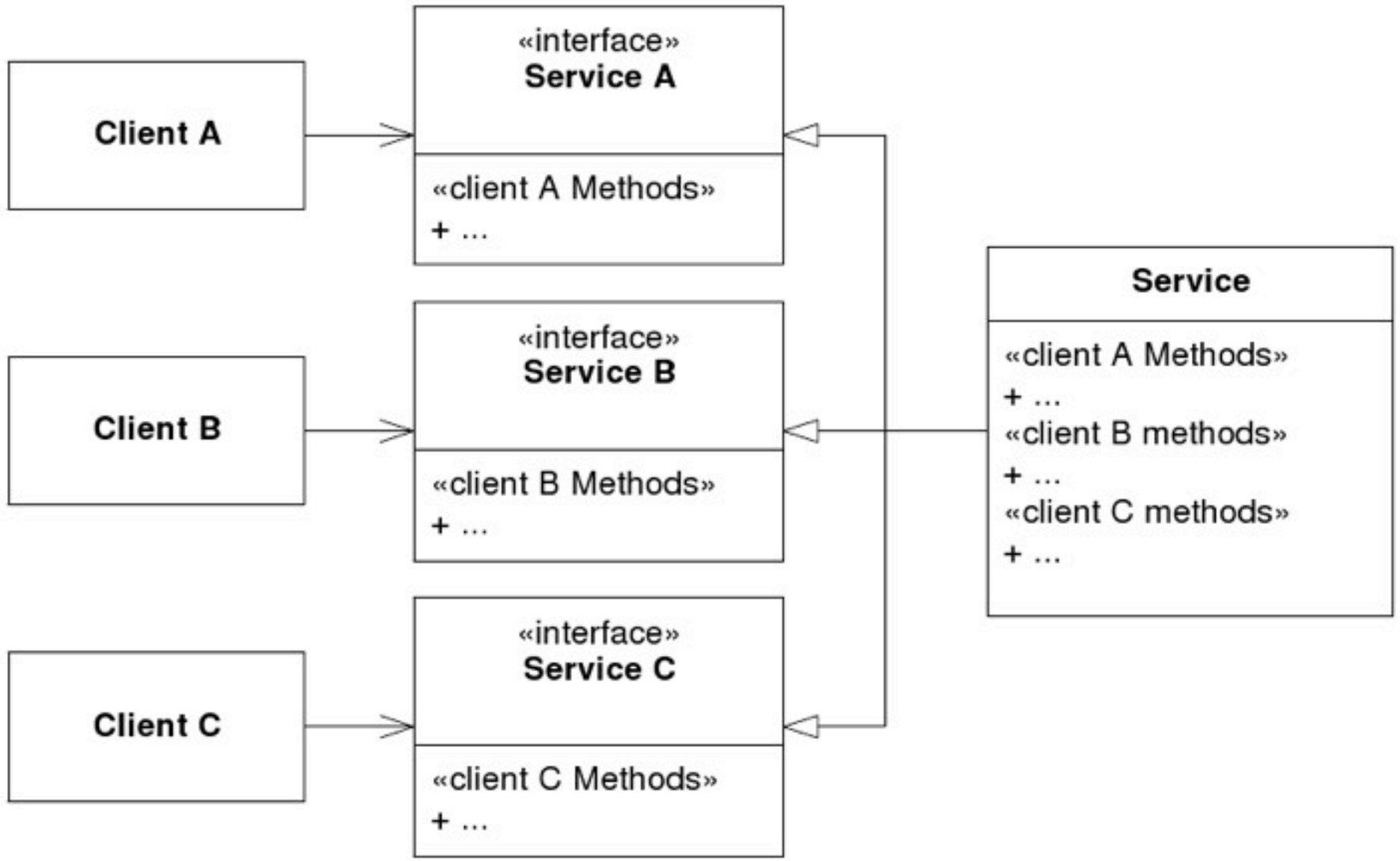
INTERFACE SEGREGATION PRINCIPLE

You Want Me To Plug This In, Where?

Interface Segregation Principle

- Clients should not depend on interfaces which they don't use

Interface Segregation Principle



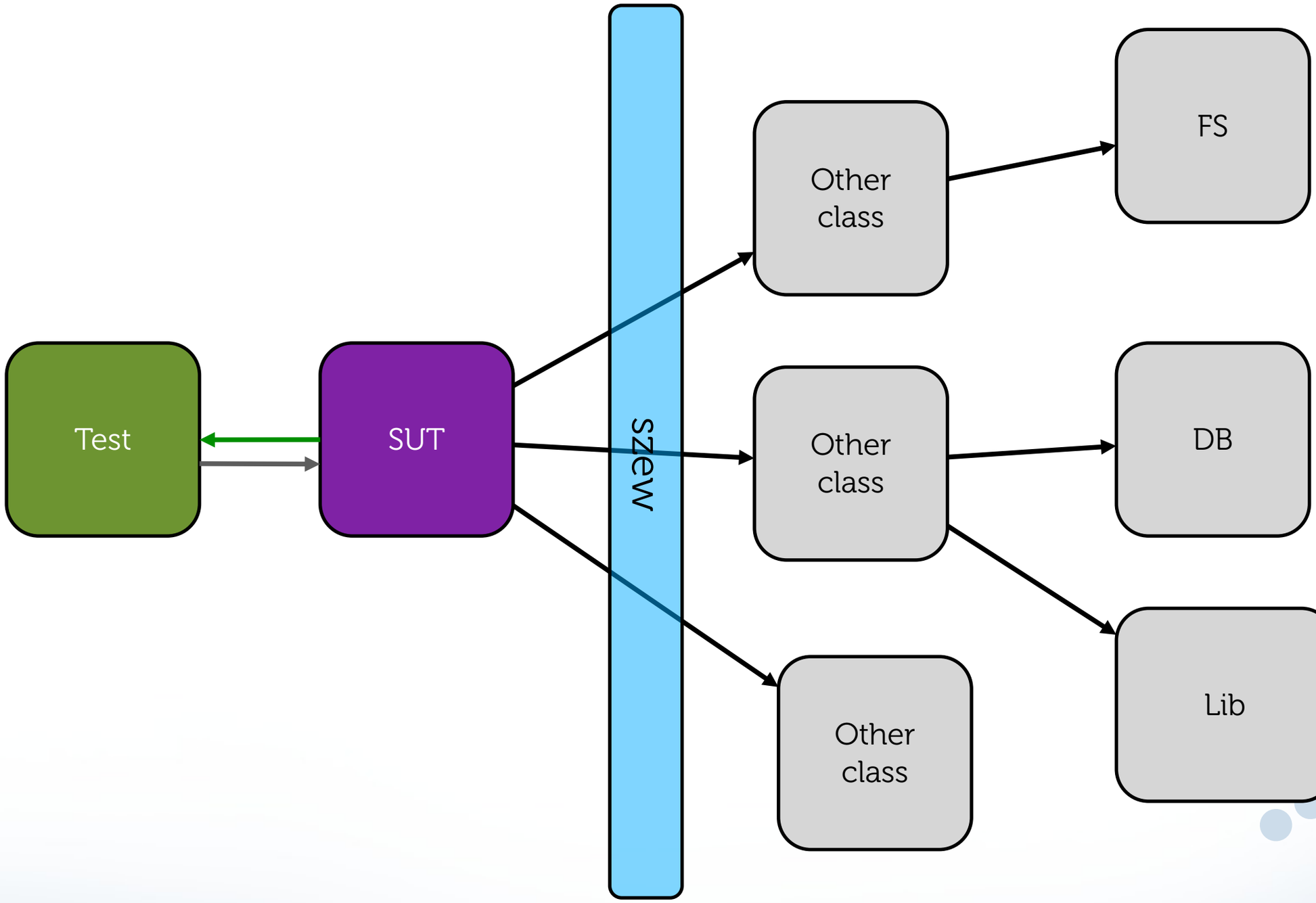
DEPENDENCY INVERSION PRINCIPLE

Would You Solder A Lamp Directly To The Electrical Wiring In A Wall?

Dependency Inversion Principle

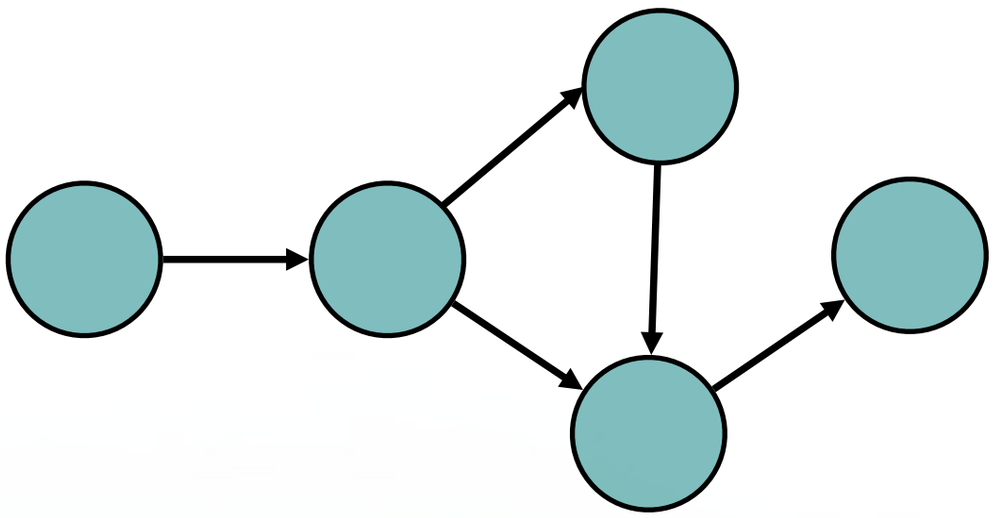
- More general modules should not depend on the more specific ones. Both should be dependent on abstractions.
- Abstractions should not depend on details. It's details which should depend on abstractions.

Testability



Detach creation from operation

Business/domain logic



Object creation

new
Factory
new
new

new
Builder
new

new
Dependency
Injection
new
new

How to write code that's hard to test?

PRAGMATISTS