5. SOLID and Agile Design

COMP1531 • KC Notes

5.1 Coupling and Cohesion

Coupling

✔ **loose coupling**

Interdependence *between* classes – you can use and modify classes independently of each other

Cohesion

✔ **high cohesion**

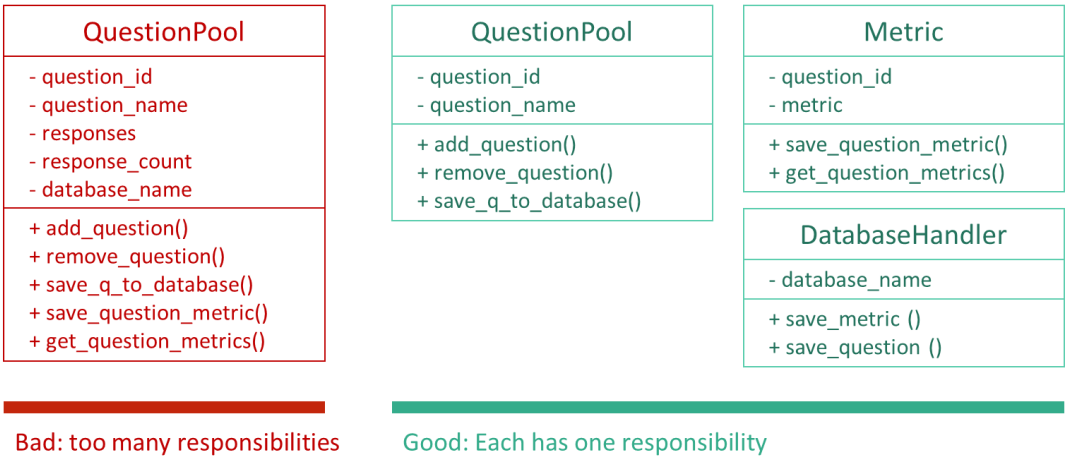
Focus *within* a class – how focused and how well elements of a class work together as a functional unit

- Good software aims for **loose coupling and high cohesion** among its components
  - Software that has tight coupling/low cohesion are often **fragile**
- **SOLID Goals**: principles that support this so that components are easy to understand, maintain, extend, reuse and test
  - **Single Responsibility Principle**
  - **Open Closed Principle**
  - Liskov Substitution Principle
  - Interface Segregation Principle
  - **Dependency Inversion Principle**

5.2 Single Responsibility Principle

A class should have **one and only one** reason to change

- Every class should have **only one responsibility** (reason for change)
- A class with **more than one responsibility** means responsibilities become coupled
  - Changes to one responsibility impairs the class's ability to meet the others
  - Design is fragile and may break
- SRP does not imply "do only one thing" – class can **call other functions** from other classes, but should **not be responsible**
- Examples:
  - Mixing **business logic** with **work orchestration** (how a particular job is done, e.g. generating a report)
  - Grouping **business logic** with **persistence control** (storing data, vs. calculations)

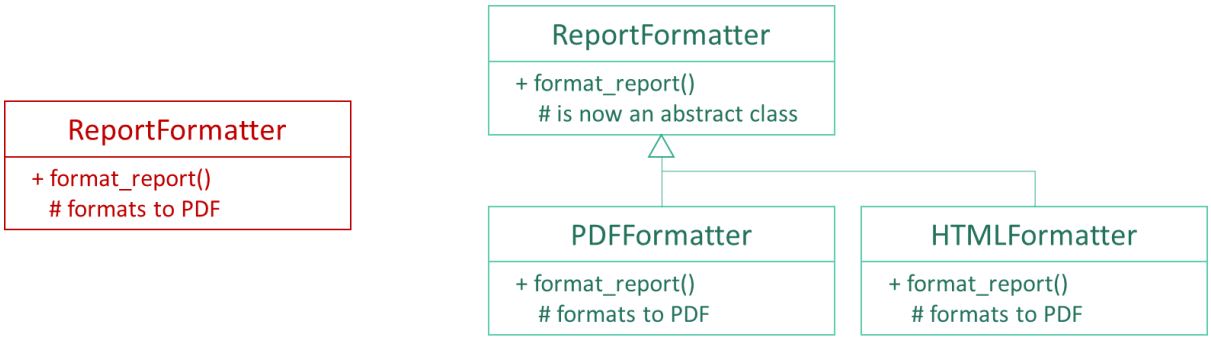


- Benefits:
  - **Readability**: easy to identify the single responsibility
  - **Reusability**: code can be reused in different contexts
  - **Testability**: responsibilities can be tested separately

5.3 Open Closed Principle

Software entities should be **open for extension** but **closed for modification**

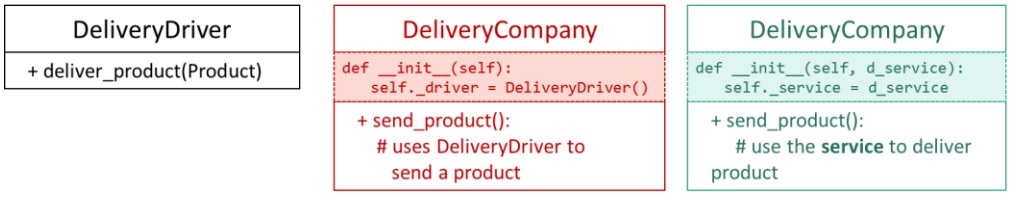
- Modules have two attributes:
  - **Open for extension**: when requirements change, class is extended with new behaviours to adapt to these changes
  - **Closed for modification**: extending the behaviour does not require changing the original source
- ReportFormatter is **not open for extension** – the option on the right is more appropriate and allows extension without modification of existing classes



5.4 Dependency Inversion Principle

High level modules should not depend on low level modules,
Both should **depend upon abstractions**

- E.g. soldering a lamp directly to electrical wiring in the wall, vs. plugs and sockets
  - Classes that directly hard-wired to others are **hard to replace**
  - Low level classes should be **easily replaceable** – high level modules should therefore not rely on low-level classes
- Object A (client) depends on object B (service) – object A must not create or import object B directly
  - A should provide a way for **injecting** B
  - This injection should be done independently in the **dependency injector**



Driver is a low-level module (company > driver)
Company should not rely on Driver

We now **inject** the Driver
e.g. DeliveryCompany(DeliveryDriver)
So, we can also inject other low-level modules like
DeliveryCompany(DeliveryTrucker)

5.5 OO and Agile Design

- **Domain-based organisation**: when requirements change in a story in Agile, they change on the **domain level**
- OO also allows more focus on the **problem domain –** software should model the customer's requirements
- Allows **faster modification** of code associated with changes in requirements