Rigorous Software Engineering

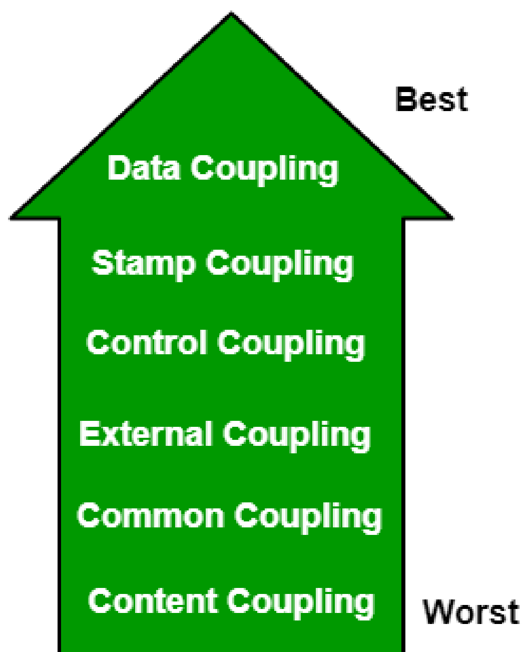
Gnkgo, Computer Science, Bsc 6. Semester

1 Coupling

Coupling is the measure of the degree of interdependence between modules. A good software design will have low coupling.

1.1 Types of Coupling

- Data coupling - Modules exchange elements, and the receiving end uses all of them.
- Procedural coupling
- Class coupling



1.2 Data Coupling

Dependency between modules occurs by passing only data, making them data coupled. Components are independent and communicate through data.
Problems:

- Changes in data representation
- Unexpected side effects
- Concurrency

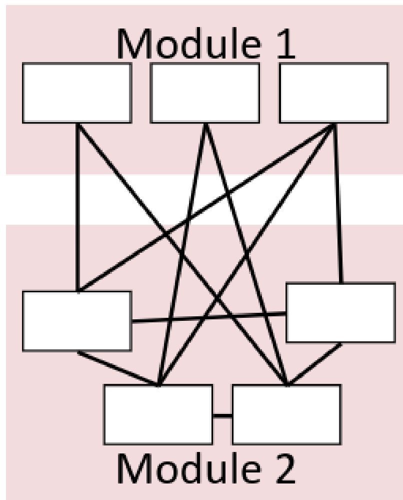
Example: customer billing system

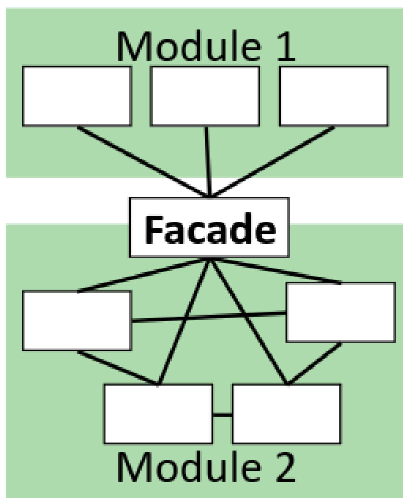
- Access on public variables

- Hide implementation details behind the interface (make private)
- Don't give links, copy references

1.2.1 Facade Pattern

- Restricts and simplifies access
- Provides a single, simplified interface





1.2.2 Flyweight Pattern

- Maximizes sharing of immutable objects
- Invariant: if two objects are structurally equal, they are the same object

1.3 Procedural Coupling

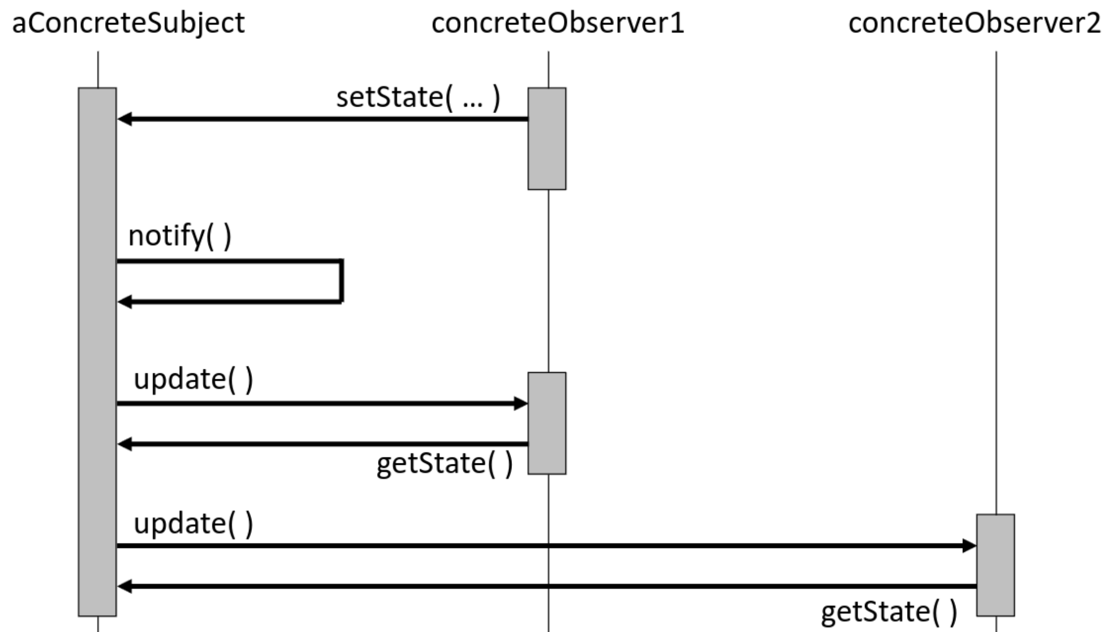
Modules are coupled to other modules whose methods they call.
Problems:

- Changing a signature in the callee requires changing the caller
- Callers cannot be reused without callee modules

Approach:

- Moving code may reduce procedural coupling
- Duplicating functionality

1.3.1 Observer Pattern



The *Observer Pattern* defines one-to-many dependency between objects, so when one object changes state, all of its dependents are notified and updated automatically.

Event-Based Communication: Discussion

Strengths
- Loose (dynamic) coupling via events
- Strong support for reuse: plug in new components by registering them for events
- Change: add, remove, and replace components with minimum effect on other components

Weaknesses
- Loss of control
  - What components will respond to an event?
  - In which order will components be invoked?
  - Are invoked components finished?

  Loss of control complicates ensuring correctness

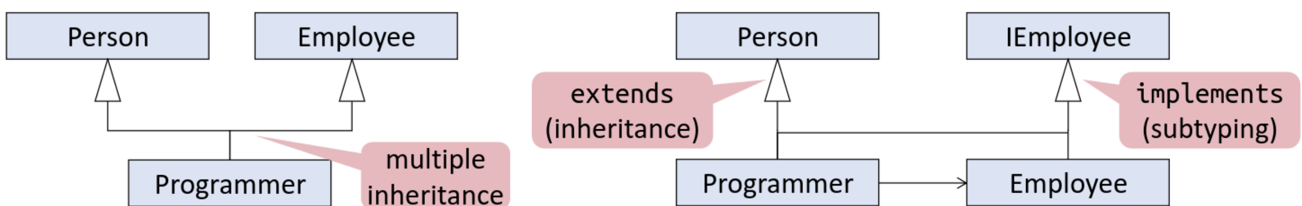
Variants
- Register only for specific event types → fine-grained control over messages to receive
- Introduce an intermediary *message broker* or *event bus* → filter, log, prioritize, … (search for *Publish-Subscribe* pattern)
- …

1.4 Class Coupling

Inheritance couples the subclass to the superclass.
Solution:

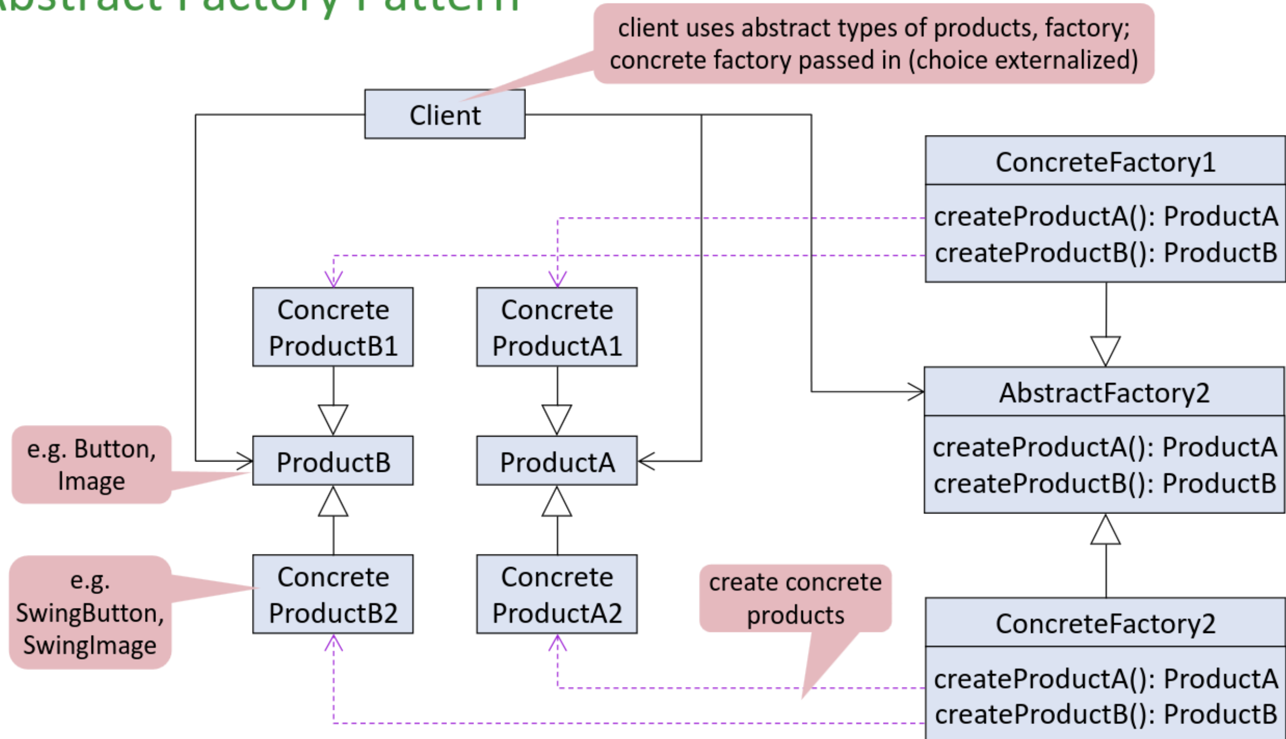
- Delegation can be used to avoid coupling through inheritance
- Use type declarations as generic as possible
- Use interfaces (Instead of TreeMap, just use Map, the most general supertype)

- Multiple inheritance can be replaced by subtyping and delegation



1.4.1 Abstract Factory Pattern

Situation: Construction of families of objects.



Java Implementation:

```
interface MealFactory {
    Pizza createPizza(); // no inheritance needed
    Burger createBurger(); // no inheritance needed
}
```

Vegan Meal Factory Implementation:

```
public class VeganMealFactory implements MealFactory {
    @Override
    public Pizza createPizza() {
        return new VeganPizza();
    }

    @Override
    public Burger createBurger() {
        return new VeganBurger();
    }
}
```

Non-Vegan Meal Factory Implementation:

```
public class NonVeganMealFactory implements MealFactory {
    @Override
    public Pizza createPizza() {
        return new NonVeganPizza();
    }

    @Override
    public Burger createBurger() {
        return new NonVeganBurger();
    }
}
```

1.5 Adaptation

1.5.1 Strategy Pattern

The Strategy Pattern is a design pattern that allows defining a family of algorithms, encapsulating each one, and making them interchangeable.

1.5.2 Visitor Pattern

The Visitor pattern is a behavioral design pattern that lets you separate algorithms from the objects on which they operate.

2 Documentation

2.1 Why?

Essential vs Incidental Properties

- Stable Properties: Should not change during software evolution, ensuring consistent behavior.
- Unstable Properties: Can change without affecting core functionality, used to improve performance, readability, or adaptability.

2.2 What?

- How to use the code - document the interface.
- How the code works - document the implementation.

3 Testing

3.1 Test Stages

- Parameterized Unit Tests

- Functional Testing
- Structural Testing

Testing Strategies: Summary

Functional testing
- Goal: Cover all the requirements
- Black-box

Structural testing
- Goal: Cover all the code
- White-box

Random testing
- Goal: Cover corner cases
- Black-box

Exhaustive testing
- Hardly ever possible

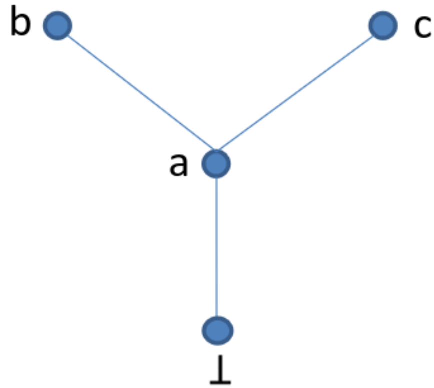
4 Analysis

Join is the least upper bound.

4.1 Analysis Math

- Partial order: binary relation on a set with properties.
  - Reflexive
  - Transitive
  - Anti-symmetric

Bounds: example



upper bounds {b, c}: none

lower bounds {b, c}: a and ⊥

$\sqcap$ {b, c}: a

no T element

Fixed point iff $f(x) = x$
Post-fixed point iff $f(x) \sqsubseteq x$

$$\forall z \in A : \alpha(F(\gamma(z))) \sqsubseteq_A F'(z) \tag{4.1}$$

Approximating a Function: Definition 2

So we have the 2 functions:

$$F: C \longrightarrow C$$

$$F^{\#}: A \longrightarrow A$$

But what if α and γ do not form a Galois Connection ? For instance, α is not monotone. Then, we can use the following definition of approximation:

$$\forall z \in A : F(\gamma(z)) \sqsubseteq_c \gamma(F^{\#}(z))$$

If not monotone:

$$\forall z \in A : F(\gamma(z)) \sqsubseteq_C \gamma(F'(z)) \tag{4.2}$$

5 Analysis Intervals

$$[a, b]\nabla_i[c, d] = [e, f] \tag{5.1}$$

where:

$$\text{if } c < a, \text{ then } e = -\infty, \text{ else } e = a$$
$$\text{if } d > b, \text{ then } f = \infty, \text{ else } f = b$$

6 Pointer Analysis

Flow-sensitive: respects the program control flow.
Flow-insensitive: assumes all execution orders are possible.

Statement/Expression	Description
`p = q`	compare two pointers
`p := alloc`$^{\ell}$	create new object
`p := q`$^{\ell}$	assign pointers
`p.f := q`$^{\ell}$	pointer heap store
`p := q.f`$^{\ell}$	pointer heap load