Object-Oriented Programming

SOLID

Michael L. Collard, Ph.D.

Department of Computer Science, The University of Akron

SOLID Principles for Object-Oriented Design

- Five basic principles (guidelines) for Object-Oriented Design

- Leads to systems that are:

  Easy to maintain

  Easy to extend

- SOLID is a guide for:

  Creating designs from scratch

  Improving existing designs

SOLID Principles

- ***S***ingle Responsibility Principle (SRP)
- ***O***pen/Closed Principle (OCP)
- ***L***iskov Substitution Principle (LSP)
- ***I***nterface Segregation Principle (ISP)
- ***D***ependency Inversion Principle (DIP)

Single Responsibility Principle

Every class should have a single responsibility

- Responsibility - a reason to change

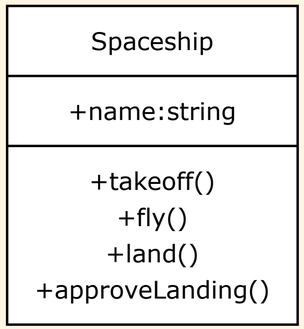
- The class should entirely encapsulate responsibility

- All of the class should focus on that single responsibility

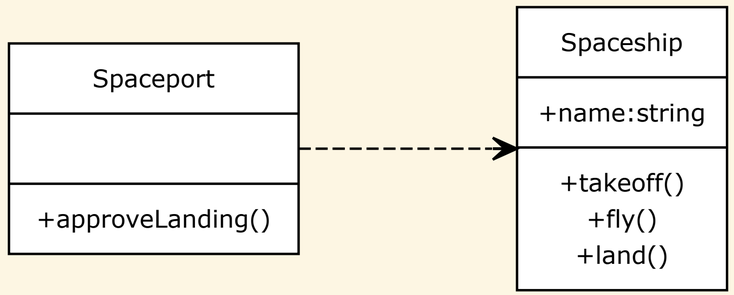
- Why? More cohesive. Easier to understand. Easier to maintain.

SRP Examples

Multiple Responsibilities



Single Responsibility



Open/Closed Principle

Software entities (classes, functions, etc.) should be open for extension but closed for modification

- Closed - as can be compiled, stored in a library, and used by client classes

- Open - as any new class can inherit and add new features

- Why? Client code dependent on base (closed) class unaffected. Less testing. Less code to review.

Meyer's Open/Closed Principle

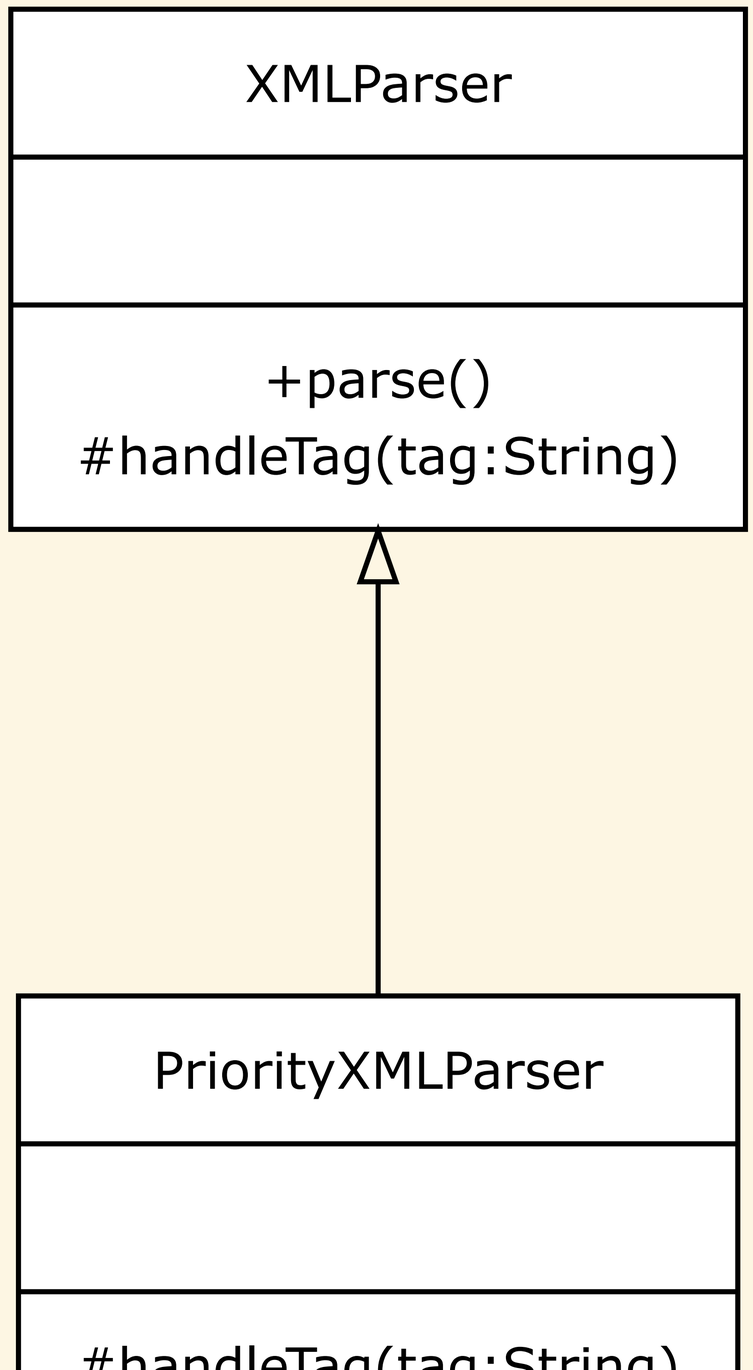
Implementation is extended through inheritance

- "Open" means available for extension (generalization/inheritance)

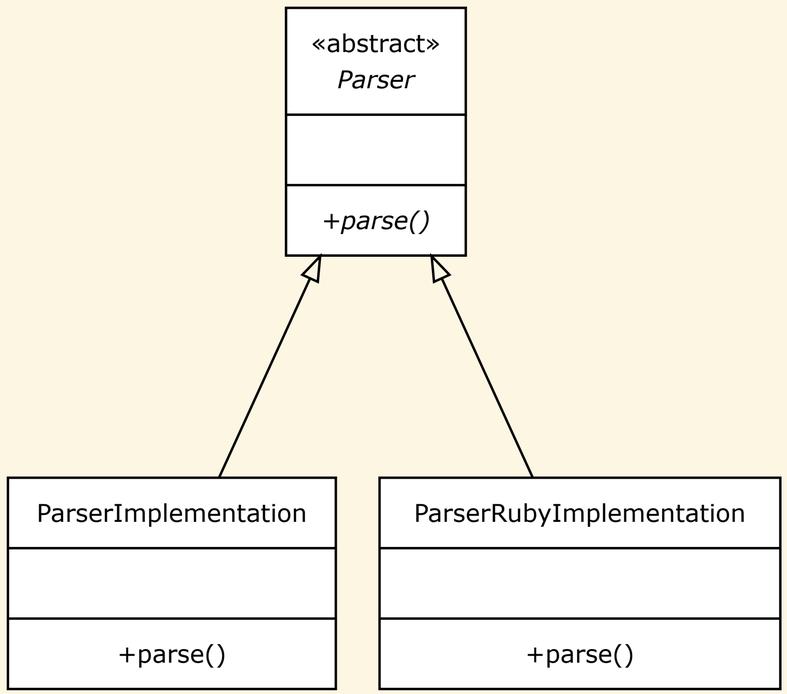
- "Closed" because we do not change the original class

- New functionality by adding new classes, not changing the current ones

Meyer's OCP Example UML



Polymorphic Open/Closed Principle

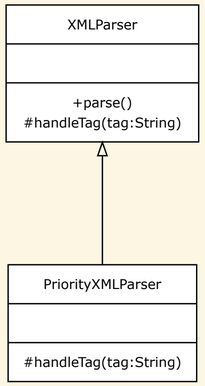


Abstract base class and multiple implementations that we can substitute for each other

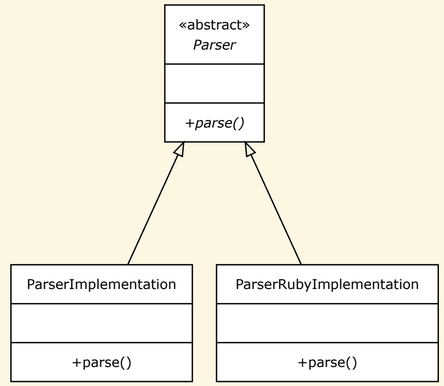
- Base design on abstract base classes
- Focus on sharing the interface, not the implementation
- Reuse implementation via delegation

OCP Comparison

Meyer



Polymorphic



Liskov Substitution Principle

Object in a program should be replaceable with an instance of subtypes without affecting program correctness

- Preconditions cannot be strengthened in a subtype
- Postconditions cannot be weakened in a subtype
- Invariants of supertype must be preserved in subtype
- History constraint - new methods in subtype cannot introduce state changes in a way that is not permissible in the supertype
- Why? Knowledge/assumptions about the base class apply to the subclass. Easier to understand. Easier to maintain.

LSP Violation: Preconditions cannot be strengthened in a subtype

```cpp
class Base {
public:
    virtual void process(int x) {
        // precondition: x must be positive
        assert(x > 0);

        // ...
    }
};

class Derived : public Base {
public:
    void process(int x) override {
        // precondition: x must be positive and even
        assert(x > 0 && x % 2 == 0);

        // ...
    }
};
```

LSP Violation: Postconditions cannot be weakened in a subtype

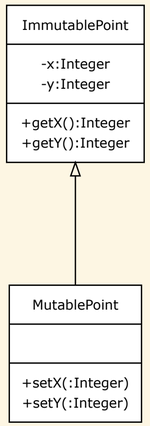
```cpp
class Base {
public:
    virtual int getValue() {

        // postcondition: always >= 0
        return std::abs(data);
    }
};

class Derived : public Base {
public:
    int getValue() override {

        // postcondition: any int value
        return data;
    }
};
```

LSP Violation: Invariants of supertype must be preserved in subtype

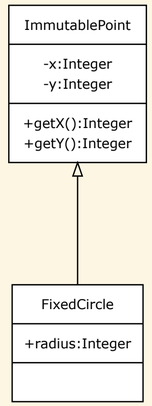
```cpp
class Stack {
protected:
    int size = 0;
public:
    // invariant: size >= 0, always

    virtual void push() {
        ++size;
    }
    virtual void pop() {
        if (size > 0)
            --size;
    }
};

class BadStack : public Stack {
public:
    void pop() override {
        --size;
    }
};
```

LSP Violation: History Constraint

Not substitutable



Substitutable



Interface Segregation Principle

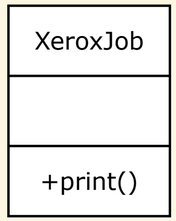
Many client-specific interfaces are better than one general-purpose interface

A client should not be forced to depend on methods it does not use

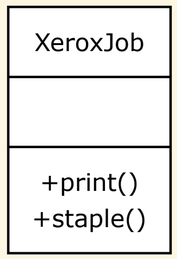
- More cohesive
- Lower coupling
- Easier to understand
- Easier to maintain

ISP Examples

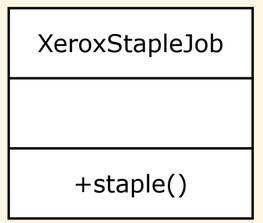
Original Interface

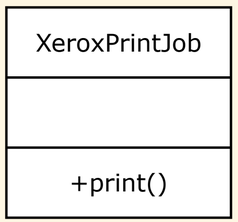


General Interface



Multiple Specific Interfaces





Dependency Inversion Principle

Depend upon abstractions, not concretions

- *Concretions* are concrete classes (i.e., non-abstract)

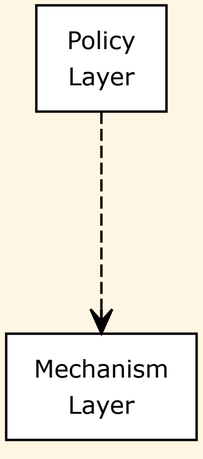
- Abstractions should not depend on details, but details on abstractions

- High-level modules are independent and should not depend on low-level modules

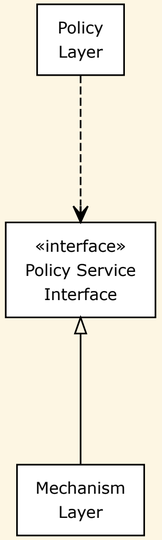
- Why? Lower coupling. Reuse. Easier to test. Easer to understand. Easier to maintain.

DIP Examples

Traditional Layered Architecture (violates principle)



Ownership Inversion (upholds principle)



Conclusion

- Meant to be applied together
- Make it more likely that the system is easy to maintain and extend over time
- SOLID principles are guidelines

  Do not guarantee success

  Can be misused
- Use in conjunction with other principles

  Occam's Razor (KISS)

  GRASP

  DRY