SE 350 - Object-Oriented Software Development

Object-Oriented Programming Principles: SOLID

Instructor: Stefan Mitsch

Learning Objectives

- Understand some common design flaws
- Understand design principles
- Be able to discuss benefits and drawbacks of design principles

Design Principles

- Robert C. Martin: Design Principles and Design Patterns (2000)
- Goal: understandable, readable, and testable code
- Work collaboratively on code

S	O	L	I	D
Single Responsibility Principle	Open-Closed Principle	Liskov Substitution Principle	Interface Segregation Principle	Dependency Inversion Principle

Design Problems

- Rigidity: implementing a change is difficult because it translates into a cascade of changes
- Fragility: any change tends to break code in many, even conceptually unrelated, places
- Immobility: unable to reuse modules across projects because of too many dependencies
- Viscosity: developers will prefer easy changes even if they break design

Design Problems

Improper design leads to

S Singleton	**T** Tight Coupling	**U** Untestability	**P** Premature Optimization	**I** Indescriptive Names	**D** Duplication

Principles to Avoid Design Problems

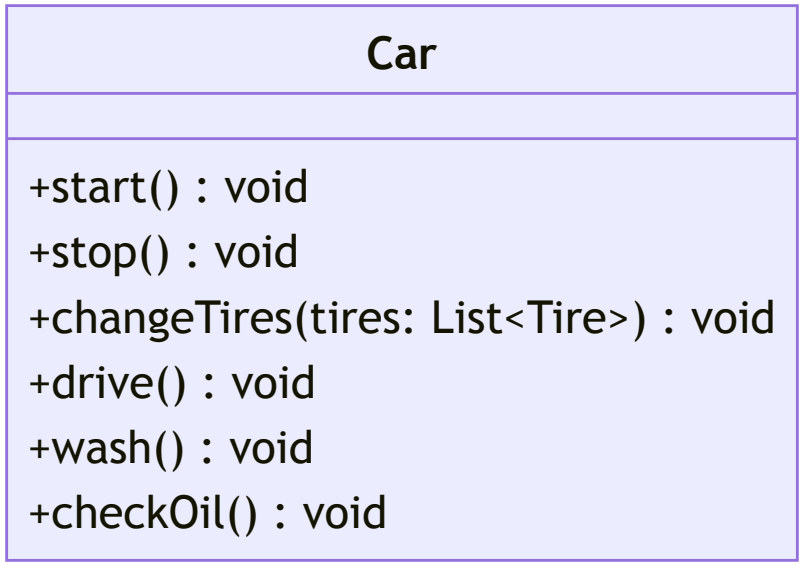
S	O	L	I	D
Single Responsibility Principle	Open-Closed Principle	Liskov Substitution Principle	Interface Segregation Principle	Dependency Inversion Principle

Single Responsibility Principle

A class should do one thing and have only a single reason to change

- Gather together things that change for the same reason
- Benefits
  - Easier testing and maintenance
  - Less internal complexity
- How to follow
  - Write small classes with specific names and purposes
- Tradeoffs
  - Cohesion vs. coupling

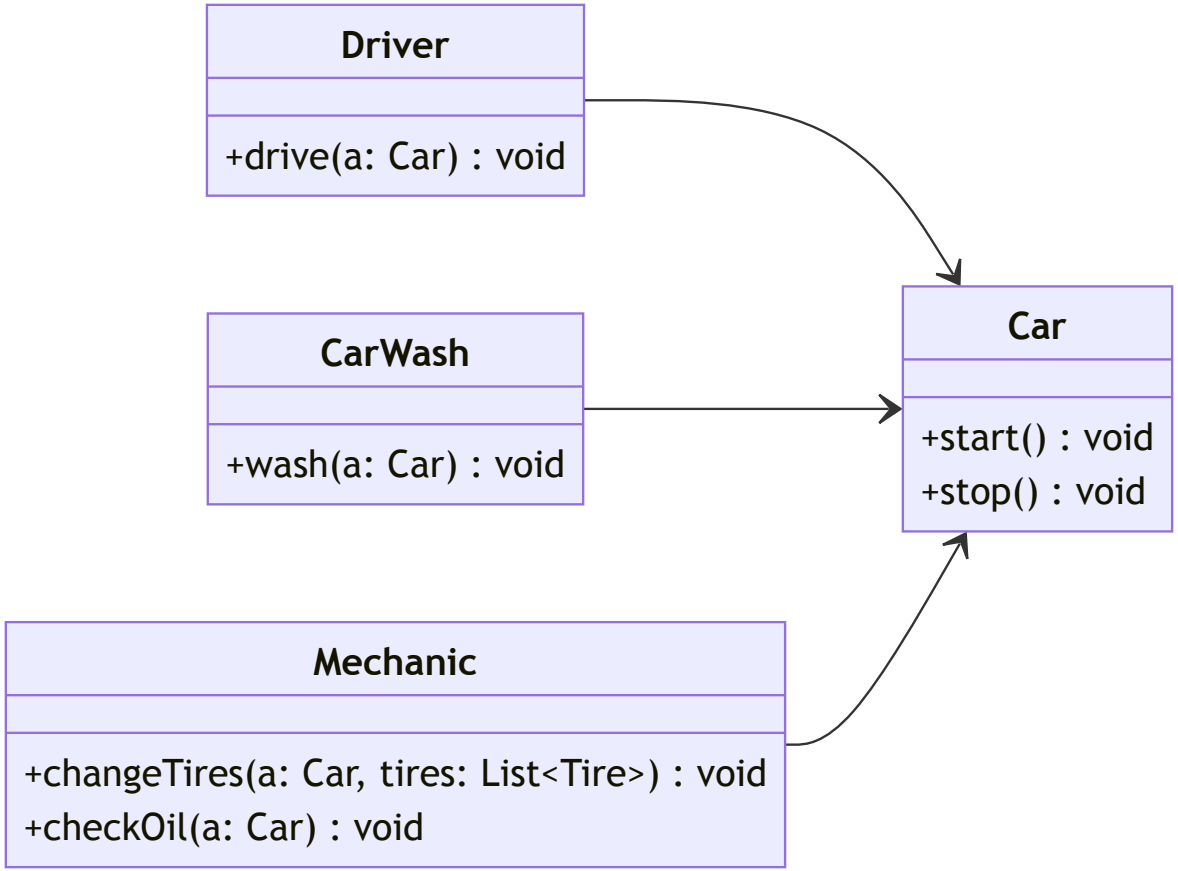
Single Responsibility Principle: Car



What is bad about this design?

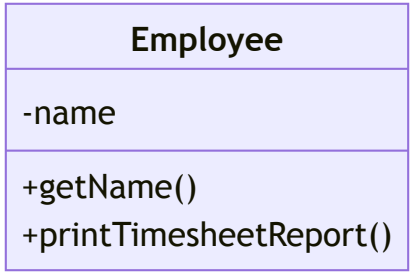
- Solution

Single Responsibility Principle: Car



- A `Driver` drives the car, not the car itself
- A `CarWash` can handle washing a car
- A `Mechanic` can change tires and check oil

Single Responsibility Principle: Employee



▶ Solution

What is bad about this design and how to fix it?

Single Responsibility Principle: Bank Customer

```
public class BankCustomer {
  private long personId;
  private String firstName;
  private String lastName;
  private List<Long> accountIds;
  private List<String> accountNumbers;
}
```

- What is bad about this design and how to fix it?

▶ Solution

Single Responsibility Principle: Bank Customer

```
public class Person {
  private long id;
  private String firstName;
  private String lastName;
}

public class Account {
  private long id;
  private String number;
}

public class BankCustomer {
  private Person p;
  private List<Account> accounts;
}
```

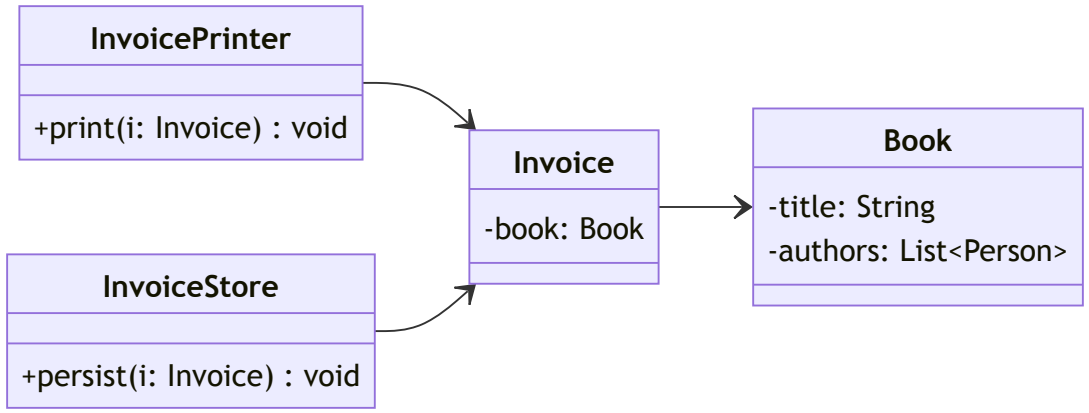
- Now `Person` is a class that is reusable by itself in other non-bank applications
- This design can be criticized for premature flexibility!

Single Responsibility Principle: Book Invoice

```
public class Book {
  private String title;
  private List<Person> authors;
}
public class Invoice {
  private Book book;
  private int quantity;
  private double itemPrice;

  public double totalPrice() {
    return itemPrice*quantity;
  }

  public void print() {
    System.out.println("Invoice " + book.title);
    System.out.println("Total price = $" + totalPrice + " (" + quantity + " * $" + itemPrice);
  }

  public void save() {
    // persist to a file
  }
}
```

- What is bad about this design?

Single Responsibility Principle: Book Invoice



What is the benefit?

▶ Solution

S	O	L	I	D
Single Responsibility Principle	Open-Closed Principle	Liskov Substitution Principle	Interface Segregation Principle	Dependency Inversion Principle

Open-Closed Principle

Software components should be open for extension, but closed for modification

- Modification: change existing code
- Extension: add new functionality
- Open for extension: inheritance; add new functionality with new (sub)classes
- What if there is a bug in existing code, can we not fix it?
- Benefits:
  - Existing code does not need to change to add new functionality
- How to follow
  - Provide interfaces and abstract classes, program to an interface
- Tradeoffs
  - Needed vs. premature flexibility

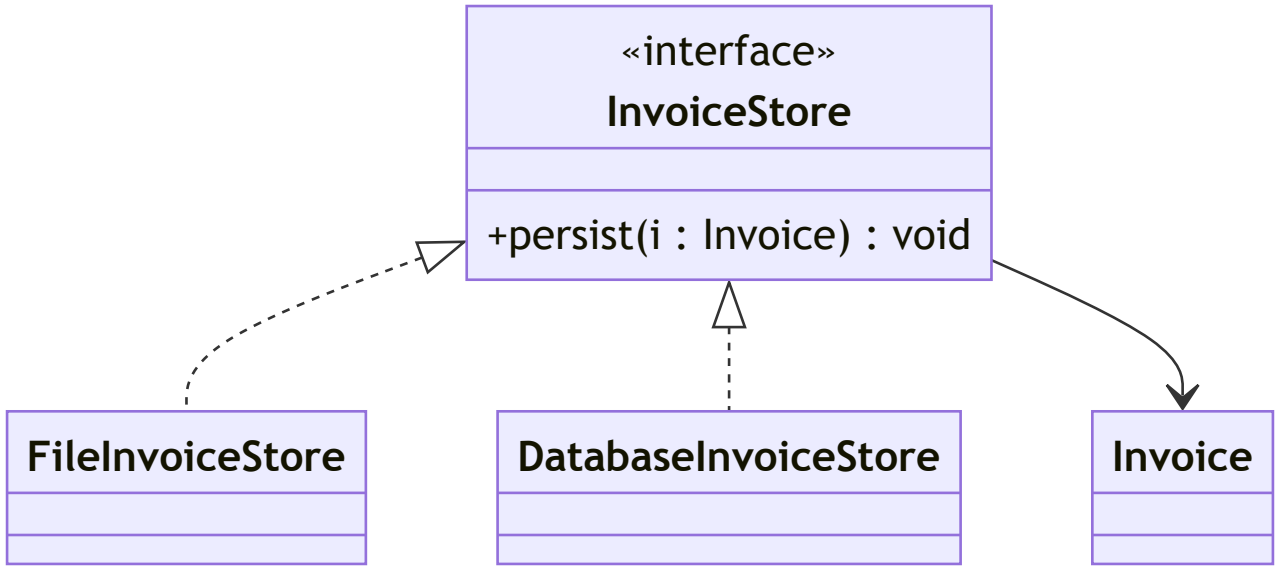
Open-Closed Principle: Bookstore

```
public class InvoiceStore {
  private Invoice invoice;
  public InvoiceStore(Invoice invoice) {
    this.invoice = invoice;
  }

  public void saveToFile(String filename) {}
  public void saveToDatabase() {}
}
```

What is bad about this design?

▶ Solution

Open-Closed Principle: Bookstore



Open-Closed Principle: Calculator

```
public abstract class BinCalcOp {
  public double l;
  public double r;
}
public class Plus implements CalcOp {}
public class Minus implements CalcOp {}
```

```
public class Calculator {
  public double calculate(BinCalcOp op) {
    if (op == null) {
      throw new IllegalArgumentException("Unable to perform op");
    } else if (op instanceof Plus) {
      return op.l + op.r;
    } else if (op instanceof Minus) {
      return op.l - op.r;
    }
  }
}
```

What is bad about this design?

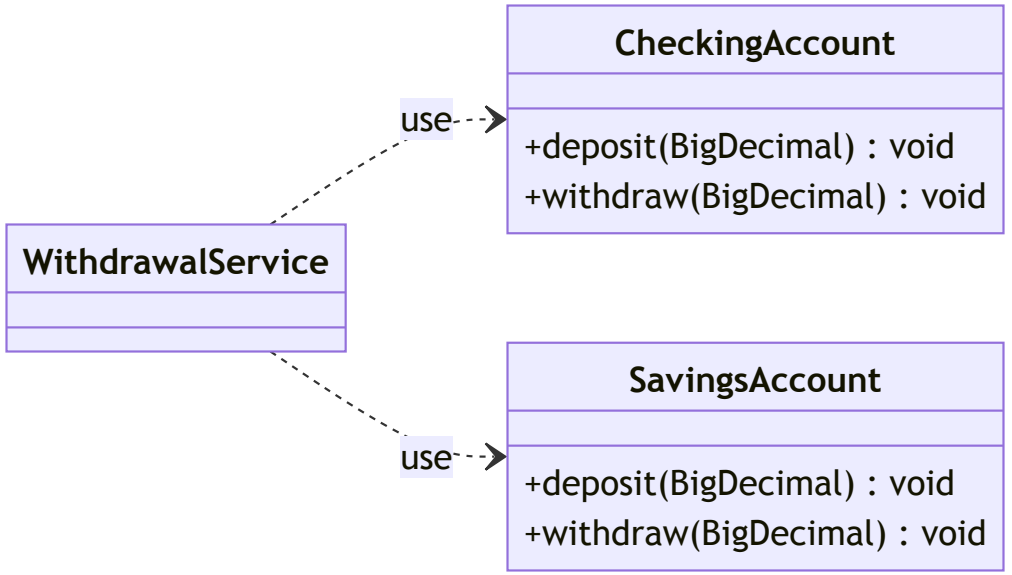
▶ Solution

Open-Closed Principle: Calculator

```
public abstract class BinCalcOp {
  private double l;
  private double r;
  // constructors, getters and setters
  public abstract double calc();
}
public class Plus implements CalcOp {
  @Override public double calc() { return getL() + getR(); }
}
public class Minus implements CalcOp {
  @Override public double calc() { return getL() - getR(); }
}
```

```
public class Calculator {
  public double calculate(BinCalcOp op) {
    if (op == null) throw new IllegalArgumentException("Unable to perform op");
    return op.calc();
  }
}
```

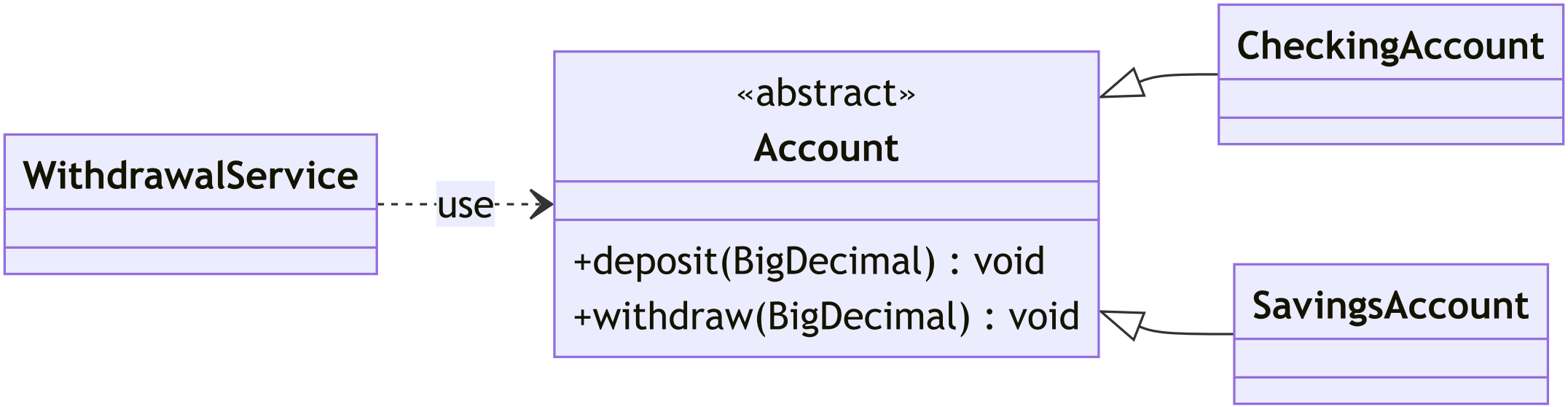
Open-Closed Principle: Banking System



What is bad about this design?

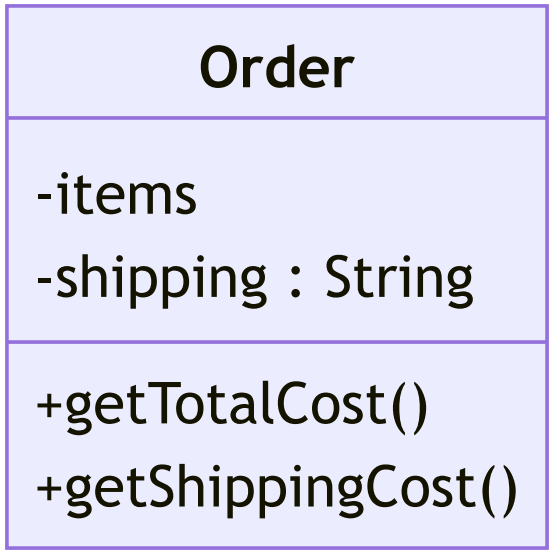
- Solution

Open-Closed Principle: Banking System



Now `WithdrawalService` can withdraw and deposit to any account; doesn't need to change for new account types

Open-Closed Principle: Store



▶ Solution

```
if (shipping == "ground") {
  if (getTotal() > 100) return 0;
  else return 10;
} else if (shipping == "air") return 50;
```

What is bad about this implementation?

S	O	L	I	D
Single Responsibility Principle	Open-Closed Principle	Liskov Substitution Principle	Interface Segregation Principle	Dependency Inversion Principle

Liskov Substitution Principle

Derived types must be completely substitutable for their base types

- Strong behavioral subtyping
- Let $\phi(x)$ be a property provable about objects x of type T. Then $\phi(y)$ should be true for objects y of type S where S is a subtype of T.
- Symbolically $S \leq T \rightarrow \forall x : T\big(\phi(x) \rightarrow \forall y : S\phi(y)\big)$
- Desirable properties ϕ are e.g. correctness, termination
- Benefits
  - Helps conform to "is-a" relationship
  - Extension does not break correctness
- How to follow
  - Make class and method contracts explicit, and satisfy them on extension
  - Avoid extension with unsupported methods

Liskov Substitution Principle: Square

A square is a special rectangle?

```
public class Rectangle {
  protected double length;
  protected double width;
  public Rectangle(double length, double width) { /* ... */ }
  public void getLength() { return length; }
  public void setLength(double l) { length = l; }
  public void getWidth() { return width; }
  public void setWidth(double w) { width = w; }
}

public class Square extends Rectangle {
  public Square(double side) { super(side, side); }
  public void setSide(double s) { length = s; width = s; }
  @Override
  public void setLength(double l) { setSide(l); }
  @Override
  public void setWidth(double w) { setSide(w); }
}
```

What is bad about this design?

▶ Solution

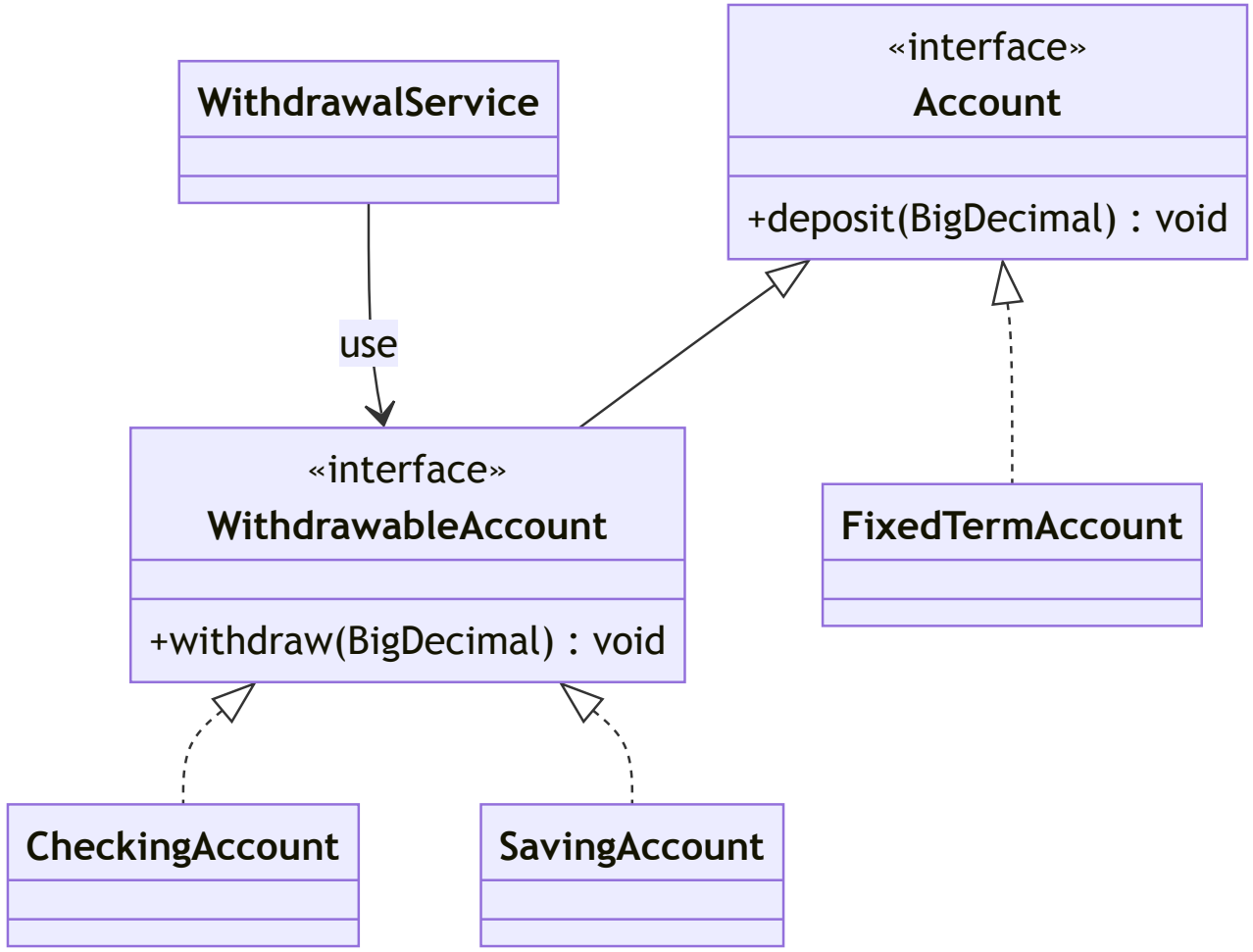
Liskov Substitution Principle: Banking System

```
public class FixedTermAccount extends Account {
  @Override
  public void deposit(BigDecimal amount) { }
  public void withdraw(BigDecimal amount) {
    throw new UnsupportedOperationException("Cannot withdraw from fixed-term account");
  }
}
```

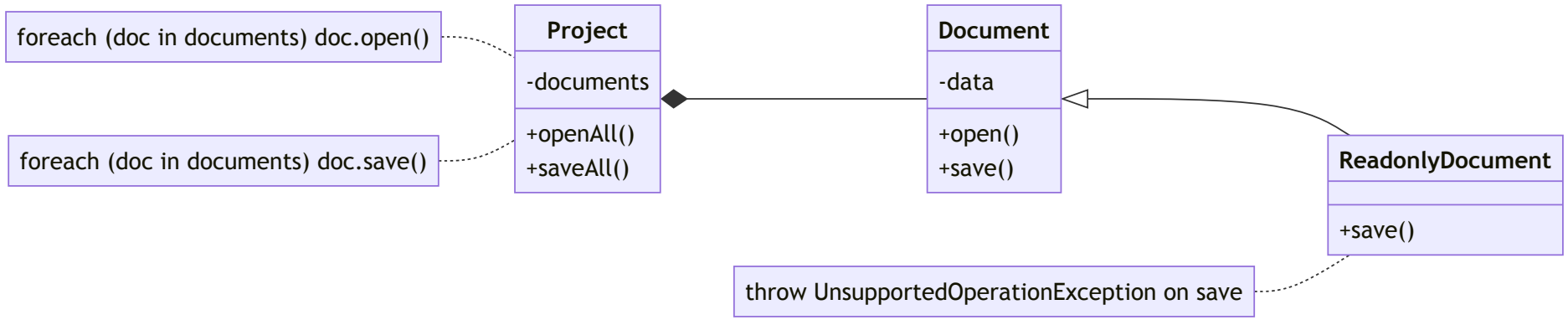
Why does this code violate the Liskov Substitution Principle?

▶ Solution

Liskov Substitution Principle: Banking System



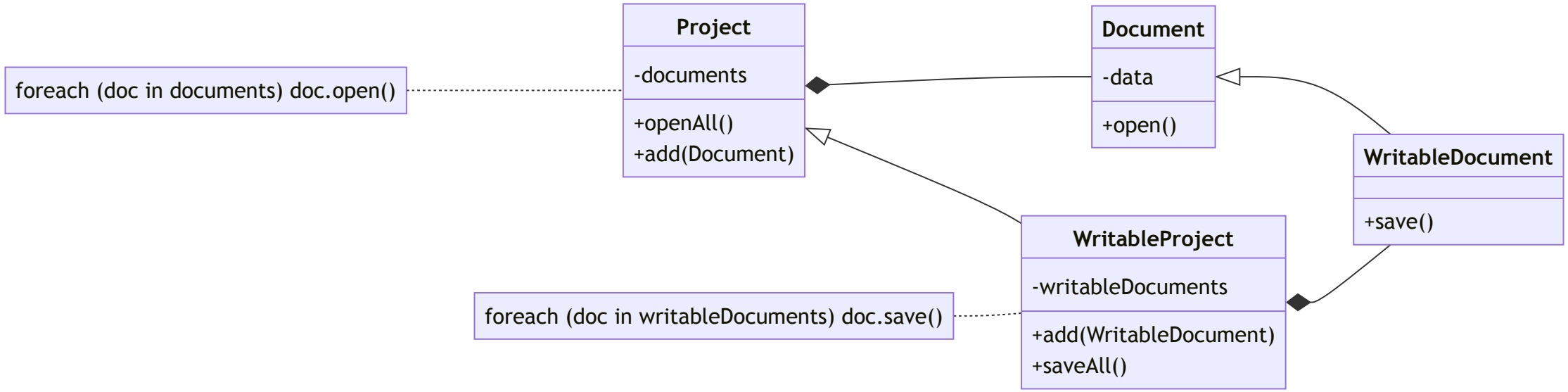
Liskov Substitution Principle: Documents



- Why is this design violating the Liskov Substitution Principle?

▶ Solution

Liskov Substitution Principle: Documents



S	O	L	I	D
Single Responsibility Principle	Open-Closed Principle	Liskov Substitution Principle	Interface Segregation Principle	Dependency Inversion Principle

Interface Segregation Principle

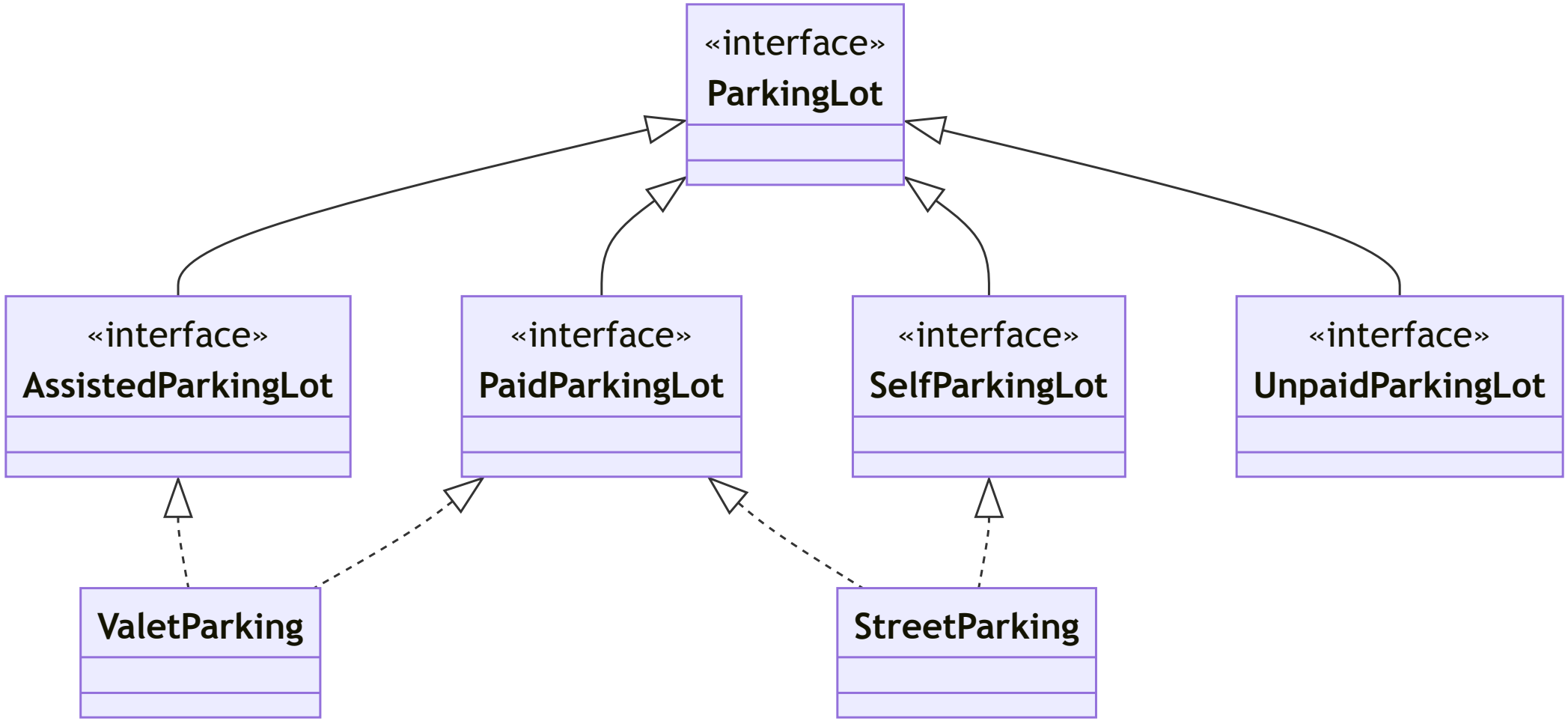
Clients should not be forced to implement unnecessary methods which they will not use

- Lots of client-specific interfaces are better than 1 general-purpose interface
- Related to Single Responsibility Principle
- Violating it often violates also Liskov Substitution Principle
- Benefits
  - Avoids empty/unsupported implementations
  - Allows objects to take on many different roles
- How to follow
  - Avoid general-purpose interfaces, provide specific interfaces
- Tradeoffs
  - Cohesion vs. coupling

Interface Segregation Principle: Parking Lot

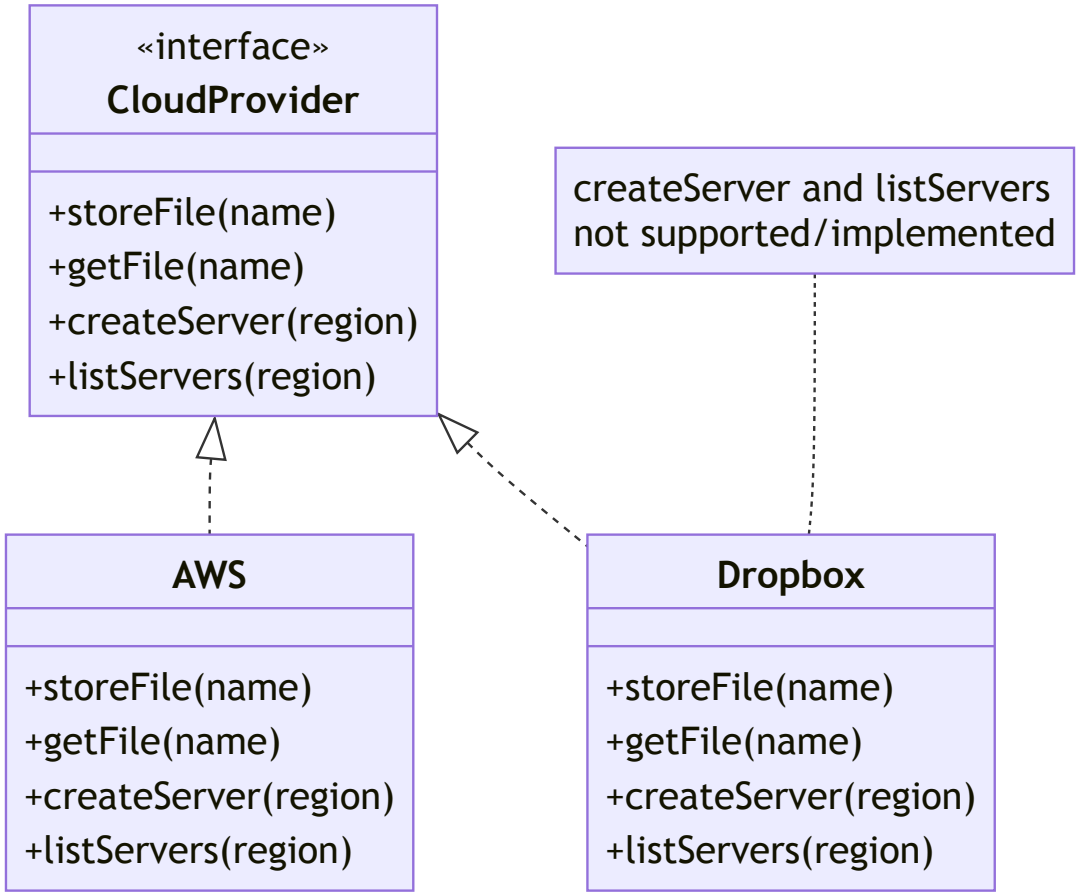
```
public interface ParkingLot {
  int park(Car c);
  Car unpark(int ticket);
  void unpark(Car c);
  int getCapacity();
  double getFee(int ticket);
  void pay(int ticket);
}

public class FreeParking implements ParkingLot {
  // can park and unpark, but fee and pay do not apply
}
public class ValetParking implements ParkingLot {
  // cannot self-park
}
public class StreetParking implements ParkingLot {
  // can only self-park
  // has unknown capacity
}
```

Interface Segregation Principle: Parking Lot



Interface Segregation Principle: Cloud Provider

- Solution



Interface Segregation Principle: Java AWT

- Lots of listener interfaces, e.g.
  - `FocusListener`
  - `KeyListener`
  - `MouseMotionListener`
  - `MouseWheelListener`
  - `TextListener`
  - `WindowFocusListener`
- Clients implement only what they need

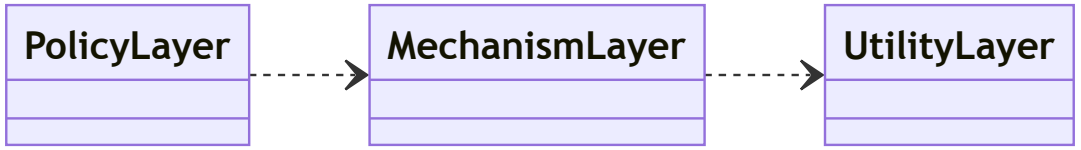
S Single Responsibility Principle	**O** Open-Closed Principle	**L** Liskov Substitution Principle	**I** Interface Segregation Principle	**D** Dependency Inversion Principle

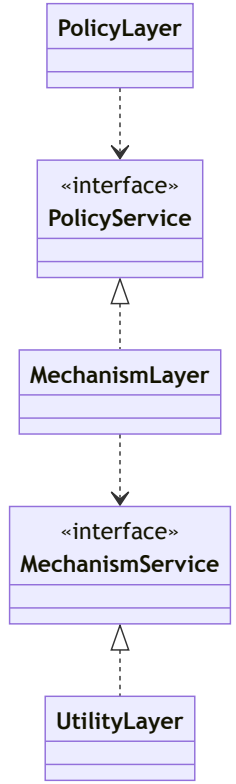
Dependency Inversion Principle

Modules should depend on interfaces/abstract classes, but not on concrete classes

- Depend on abstractions (not specializations)
  - High-level modules should not import anything from low-level modules; **both** should depend on abstractions
  - Abstractions should not depend on concrete implementations; implementations should depend on abstractions
- Benefits
  - Decouples client from implementation
  - Sets up design for Open-Closed Principle
- How to follow
  - Provide interfaces for any anticipated extension point
- Tradeoffs
  - Needed vs. premature flexibility

Dependency Inversion Principle





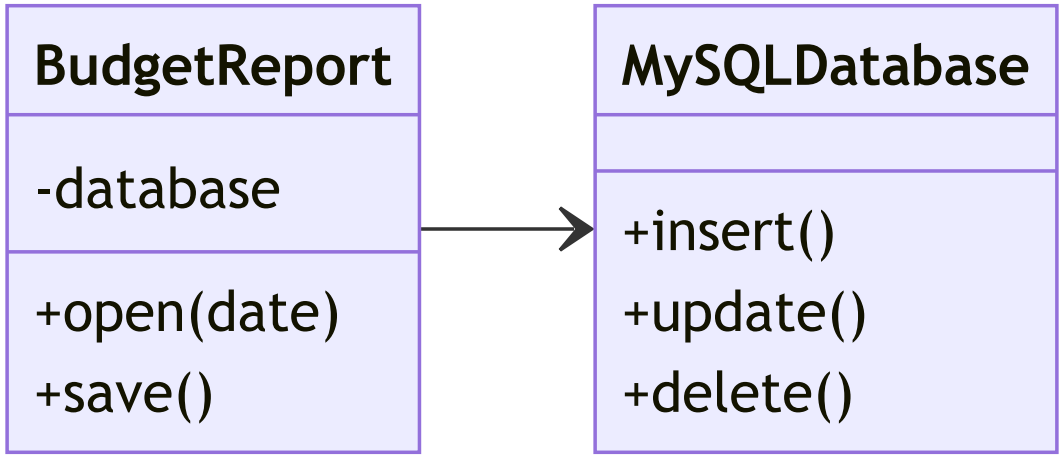
Dependency Inversion Principle: Payment

```
public class SqlProductRepo {
  public Product getProduct(String id) {}
}
public class PaymentProcessor {
  public void pay(String productId) {
    SqlProductRepo r = new SqlProductRepo();
    Product product = r.getProduct(String id);
    processPayment(product);
  }
  private void processPayment(Product product) {}
}
```

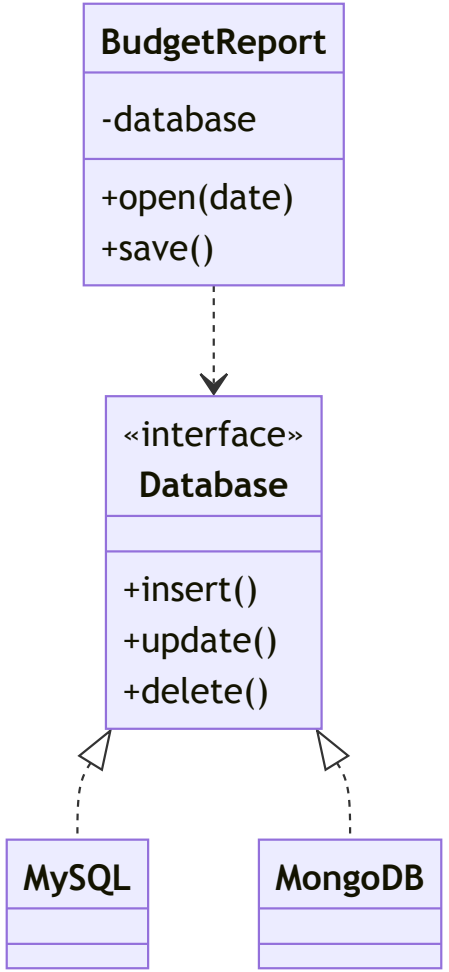
```
public interface ProductRepo {
  Product getProduct(String id);
}
public class SqlProductRepo implements ProductRepo {}
public class PaymentProcessor {
  private ProductRepo r;
  public PaymentProcessor(ProductRepo r) {}
  public void pay(String productId) {
    Product product = r.getProduct(String id);
    processPayment(product);
  }
  private void processPayment(Product product) {}
}
```

Dependency Inversion Principle: Budget Report

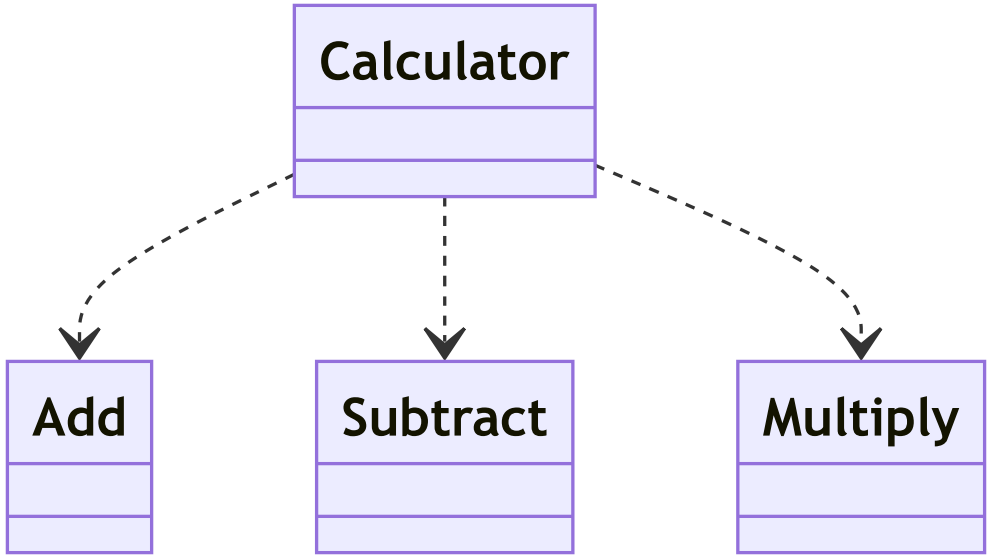
High-level references low-level

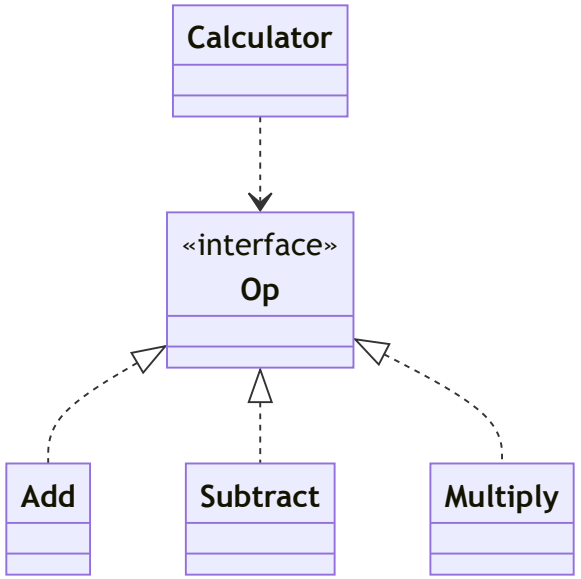


Abstraction breaks direct dependency



Dependency Inversion Principle: Calculator





Criticisms of SOLID

- Somewhat vague principles
- Focuses too much on dependencies
- Long inheritance and delegation chains
- Too much separation and abstraction can make code unreadable

Tony Marston, 2011

Marco Cecconi, 2014

Summary

S	O	L	I	D
Single Responsibility Principle	Open-Closed Principle	Liskov Substitution Principle	Interface Segregation Principle	Dependency Inversion Principle
A class should have a single responsibility and a single reason to change	Open for extension but closed for modification	Objects should be replacable with instances of their subtypes without affecting correctness	Clients should not be forced to depend on unused interfaces	Program to an interface, not to an implementation