# Modularizing Abstractions with the Interface Segregation Principle (ISP)

**Dan Geabunea**
PASSIONATE SOFTWARE DEVELOPER | BLOGGER

@romaniancoder www.romaniancoder.com

# Overview

What is the Interface Segregation Principle?

Identifying “fat” interfaces

Refactoring code that depends on large interfaces

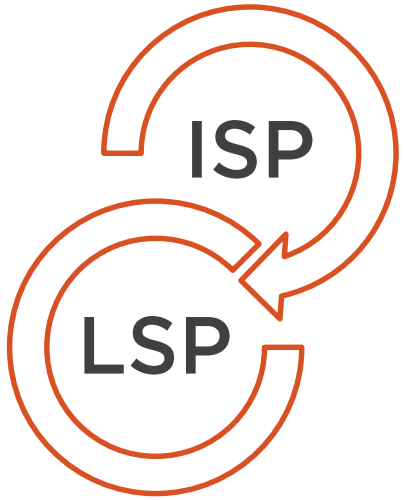
Demo: Apply the Interface Segregation Principle to fix clients that depend on a large interface

# Interface Segregation Principle

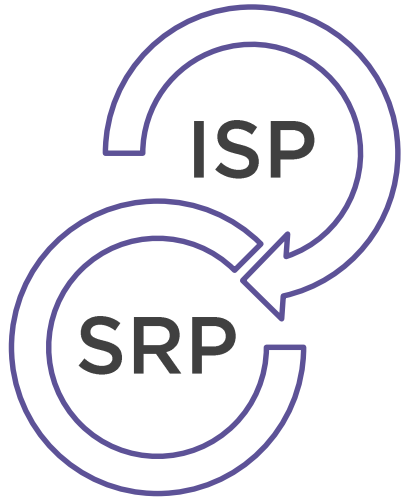
Clients should not be forced to depend on methods that they do not use.

The “interface” word in the principle name does not strictly mean a Java interface.

# The ISP Reinforces Other Principles




By keeping interfaces small, the classes that implement them have a higher chance to fully substitute the interface

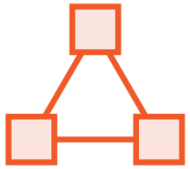



Classes that implement small interfaces are more focused and tend to have a single purpose

# Benefits of Applying the ISP

Lean interfaces minimize dependencies on unused members and reduce code coupling

Code becomes more cohesive and focused



It reinforces the use of the SRP and LSP

# Identifying “Fat” Interfaces

# Interfaces with Many Methods

```
interface LoginService{
    void signIn();
    void signOut();
    void updateRememberMeCookie();
    User getUserDetails();
    void setSessionExpiration(int seconds);
    void validateToken(Jwt token);
    …
}
```

# Throwing Exceptions

```
class GoogleLoginService implements LoginService{
    …
    public void updateRememberMeCookie(){
        throw new UnsupportedOperationException();
    }
    public void setSessionExpiration(int seconds){
        throw new UnsupportedOperationException();
    }
```

# Interfaces with Low Cohesion

```
interface ShoppingCart{
    void addItem(Item item);
    void removeItem(Item item);
    void processPayment();
    bool checkItemInStock(Item item);
}
```

# Increased Coupling

```
class ShoppingCartImpl implements ShoppingCart{
    public void processPayment(){
        PaymentService ps = new PaymentService();
        ps.pay(this.totalAmount);
        User user  = UserService.getUserForTransaction();
        EmailService emailService = new EmailService();
        emailService.notify(user);
    }
}
```

# Not Just About Interfaces

```
public abstract class Account{
    abstract double getBalance();
    abstract void processLocalPayment(double amnt);
    abstract void processInternationalPayment(double amnt);
}
```

# Empty Methods

```
public class SchoolAccount extends Account{
    …
    void processInternationalPayment(double amount){
        // Do nothing. Better than throwing an error, right?
    }
}
```

# Symptoms of Interface Pollution

Interfaces with lots of methods

Interfaces with low cohesion

Client throws exception instead of implementing method

Client provides empty implementation

Client forces implementation and becomes highly coupled

# Refactoring Code That Depends on Large Interfaces

# Fixing Interface Pollution

## Your own code

Breaking interfaces is pretty easy and safe due to the possibility to implement as many interfaces as we want

## External legacy code

You can't control the interfaces in the external code, so you use design patterns like "Adapter"

# From a Fat Interface

```
interface Account{
    double getBalance();
    void processLocalPayment(double amount);
    void processInternationalPayment(double amount);
}
```

# To Three Lean Interfaces

```
interface BaseAccount{
    double getBalance();
}
interface LocalMoneyTransferCapability{
    void processLocalPayment(double amount);
}
interface IntlMoneyTransferCapability{
    void processInternationalPayment(double amount);
}
```

# Cohesive Clients

```
public class SchoolAccount
implements Account, LocalMoneyTransferCapability{
    // We can now correctly implement all the methods
}
```

# Interface Reuse

```
public class InternationalLoanService
implements IntlMoneyTransferCapability {
    …
    public void processInternationalPayment(double amount){
        // Money transfer logic here
    }
}
```

# Demo

**Refactor code that uses fat interfaces**

- Observe the symptoms of inappropriate interfaces
- Modularize interfaces and refactor using the ISP

# Summary

**The ISP is linked with the LSP and SRP**

**Don't confuse the word "interface" in the name with a Java interface**

**Pay attention to the symptoms of large interfaces and take action**

**Break large interfaces into many focused ones for code that you own**

- Use the "Adapter Pattern" for external code

“Fat interfaces lead to inadvertent couplings between clients that ought otherwise to be isolated.”

**Robert C. Martin**

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