Interface Segregation Principle

Steve Smith
FORCE MULTIPLIER FOR DEV TEAMS
@ardalis | ardalis.com | weeklydevtips.com

SOLID Principles

SRP

Single Responsibility Principle

OCP

Open / Closed Principle

LSP

Liskov Substitution Principle

Interface Segregation Principle

ISP

Dependency Inversion Principle

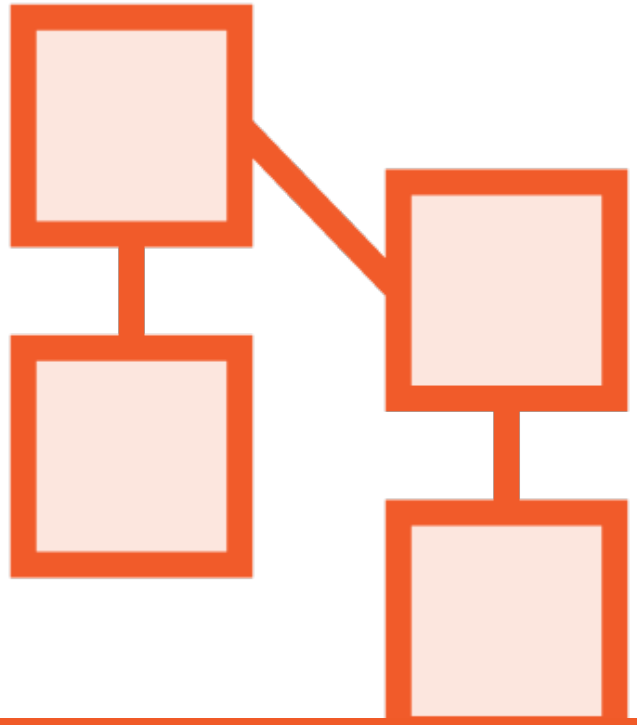
DIP

Interface Segregation Principle

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

Prefer small, cohesive interfaces to large, “fat” ones.

What Does Interface Mean in ISP?



C# `interface` type/keyword

Public (or accessible) interface of a `class`

A type's interface in this context is whatever can be accessed by client code working with an instance of that type.

What's a Client?

In this context, the client is the code that is interacting with an instance of the interface. It's the calling code.

The Problem with Large Interfaces

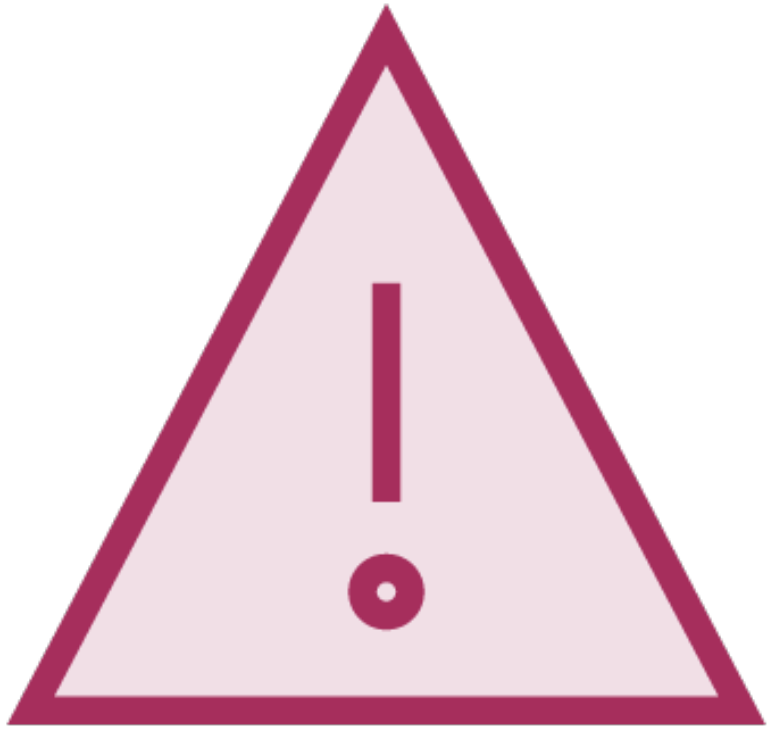
```
public class MyMembershipProvider : MembershipProvider
{
    0 references
    public override bool EnablePasswordRetrieval => throw new System.NotImplementedException();
    0 references
    public override bool EnablePasswordReset => throw new System.NotImplementedException();
    0 references
    public override bool RequiresQuestionAndAnswer => throw new System.NotImplementedException();
    0 references
    public override string ApplicationName { get => throw new System.NotImplementedException(); set => throw new System.NotImplementedException(); }
    0 references
    public override int MaxInvalidPasswordAttempts => throw new System.NotImplementedException();
    0 references
    public override int PasswordAttemptWindow => throw new System.NotImplementedException();
    0 references
    public override bool RequiresUniqueEmail => throw new System.NotImplementedException();
    0 references
    public override MembershipPasswordFormat PasswordFormat => throw new System.NotImplementedException();
    0 references
    public override int MinRequiredPasswordLength => throw new System.NotImplementedException();
    0 references
    public override int MinRequiredNonAlphanumericCharacters => throw new System.NotImplementedException();
    0 references
    public override string PasswordStrengthRegularExpression => throw new System.NotImplementedException();
    0 references
    public override bool ChangePassword(string username, string oldPassword, string newPassword)...
    0 references
    public override bool ChangePasswordQuestionAndAnswer(string username, string password, string newPasswordQuestion, string newPasswordAnswer)...
    0 references
    public override MembershipUser CreateUser(string username, string password, string email, string passwordQuestion, string passwordAnswer,
        bool isApproved, object providerUserKey, out MembershipCreateStatus status)...
    0 references
    public override bool DeleteUser(string username, bool deleteAllRelatedData)...
    0 references
    public override MembershipUserCollection FindUsersByEmail(string emailToMatch, int pageIndex, int pageSize, out int totalRecords)...
    0 references
    public override MembershipUserCollection FindUsersByName(string usernameToMatch, int pageIndex, int pageSize, out int totalRecords)...
    0 references
```

What if all your code needs is to log the user in?

Violating ISP results in classes that depend on things they don’t need.

What if you want to implement your own custom Pluralsight login provider?

More Dependencies Means



- More coupling
- More brittle code
- More difficult testing
- More difficult deployments

Detecting ISP Violations in Your Code

Large interfaces

`NotImplementedException`

Code uses just a small subset of a larger interface

A Poorly-designed Interface

```
public interface INotificationService
{
    void SendText(string SmsNumber, string message);

    void SendEmail(string to, string from,
                   string subject, string body);
}
```

Interface Lacks Cohesion

```
public class SmtpNotificationService : INotificationService
{
    public void SendEmail(string to, string from,
                          string subject, string body)
    {
        // actually send email here
    }

    public void SendText(string SmsNumber, string message)
    {
        throw new NotImplementedException();
    }
}
```

Split It Up

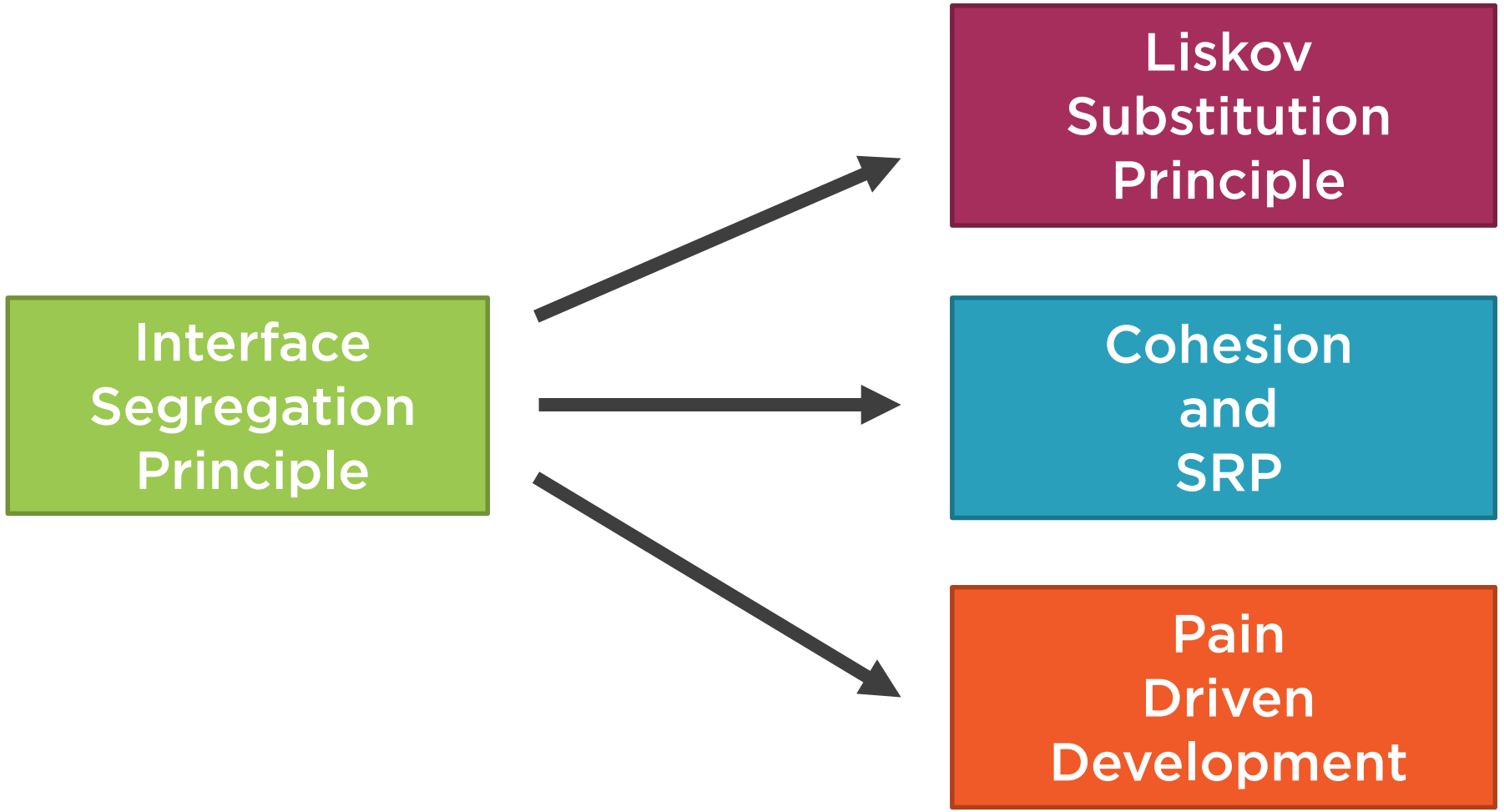
```
public interface IEmailNotificationService
{
    void SendEmail(string to, string from,
                   string subject, string body);
}

public interface ITextNotificationService
{
    void SendText(string SmsNumber, string message);
}
```

What about legacy code that's coupled to the original interface?

Multiple Interface Inheritance

```
public interface INotificationService :
                                IEmailNotificationService,
                                ITextNotificationService
{
}
```

Related Concepts



Fixing ISP Violations

Break up large interfaces into smaller ones

- Compose fat interfaces from smaller ones for backward compatibility

To address large interfaces you don't control

- Create a small, cohesive interface
- Use the Adapter design pattern so your code can work with the Adapter

Clients should own and define their interfaces

Where Do Interfaces Live in Our Apps?

Client code should define and own the interfaces it uses

Interfaces should be declared where both client code and implementations can access it

Learn More

Microsoft Reference Application + eBook

- github.com/dotnet-architecture/eShopOnWeb

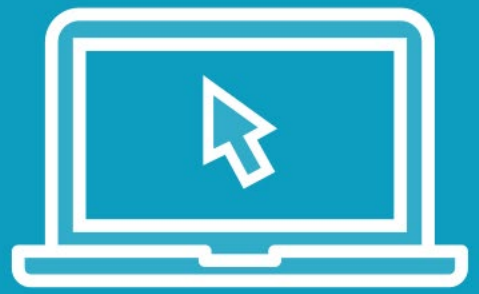
Clean Architecture Solution Template

- github.com/ardalis/CleanArchitecture

On Pluralsight

- “Creating N-Tier Applications in C#”
- “Domain-Driven Design Fundamentals”
- “Design Patterns Library”

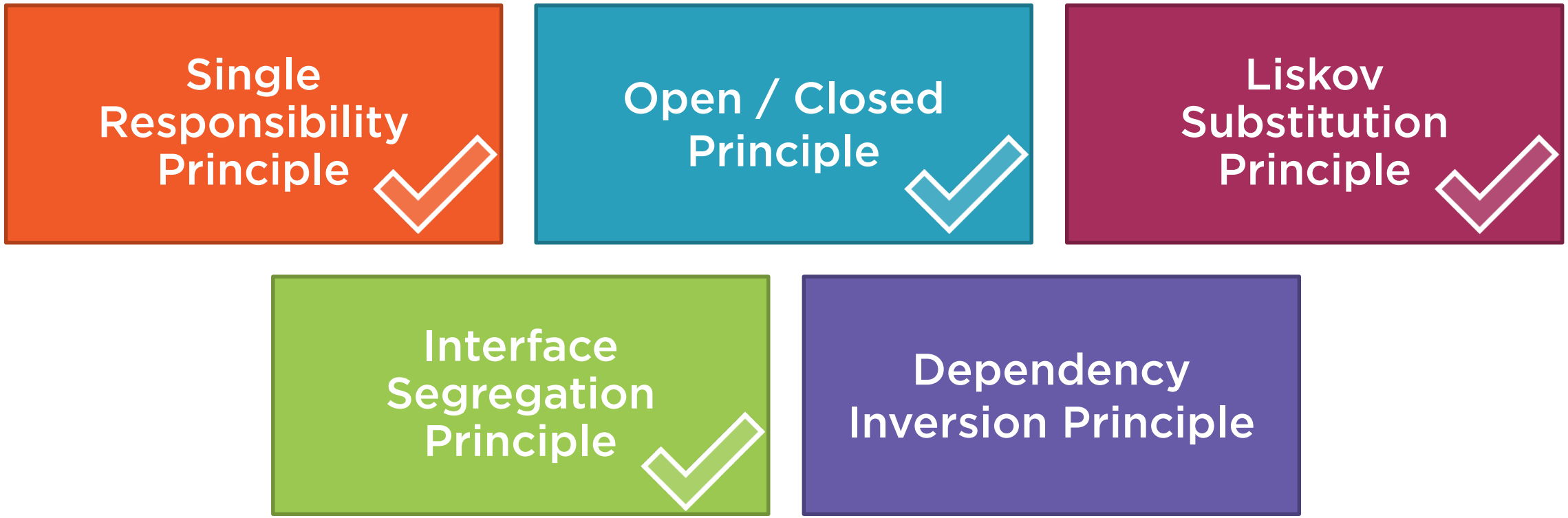
Demo



Applying ISP to ArdalisRating

Available at
https://github.com/ardalis/solidsample

SOLID Principles



Key Takeaways

Prefer small, cohesive interfaces to large, expansive ones

Following ISP helps with SRP and LSP

Break up large interfaces by using

- Interface inheritance
- The Adapter design pattern