SOLID Principles for C# Developers

SINGLE RESPONSIBILITY PRINCIPLE

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

SOLID

SRP - Single Responsibility Principle

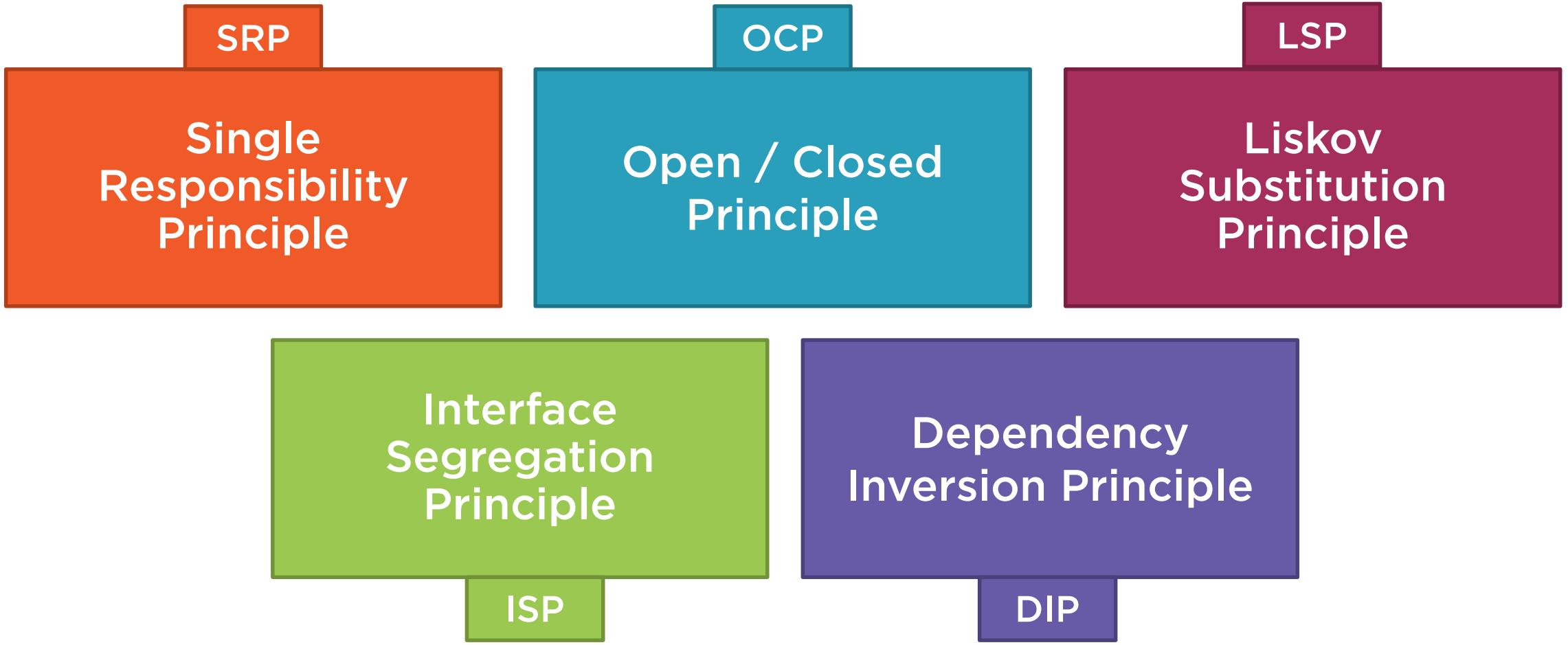
OCP - Open Closed Principle

LSP - Liskov Substitution Principle

ISP - Interface Segregation Principle

DIP - Dependency Inversion Principle

SOLID Principles



Practice PDD

Pain Driven Development (PDD)

Avoid premature optimization

If current design is painful to work with, use principles to guide redesign

Single Responsibility Principle

Each software module should have one and only one reason to change.

The individual classes and methods in our applications define what the application does, and how it does it.

Multipurpose tools don’t perform as well as dedicated tools

Dedicated tools are easier to use

A problem with one part of a multipurpose tool can impact all parts

What Is a Responsibility?

Persistence

Logging

Validation

Business Logic

Reasons to Change

Persistence

Logging

Validation

Business Logic

Responsibilities change
at different times
for different reasons.

Each one is an axis of change.

Axes of Change

Chief Information / Technology Officer

Chief Security Officer

Chief Operations Officer

Chief Marketing Officer

Tight Coupling

Binds two (or more) details together in a way that's difficult to change.

Loose Coupling

Offers a modular way to choose which details are involved in a particular operation.

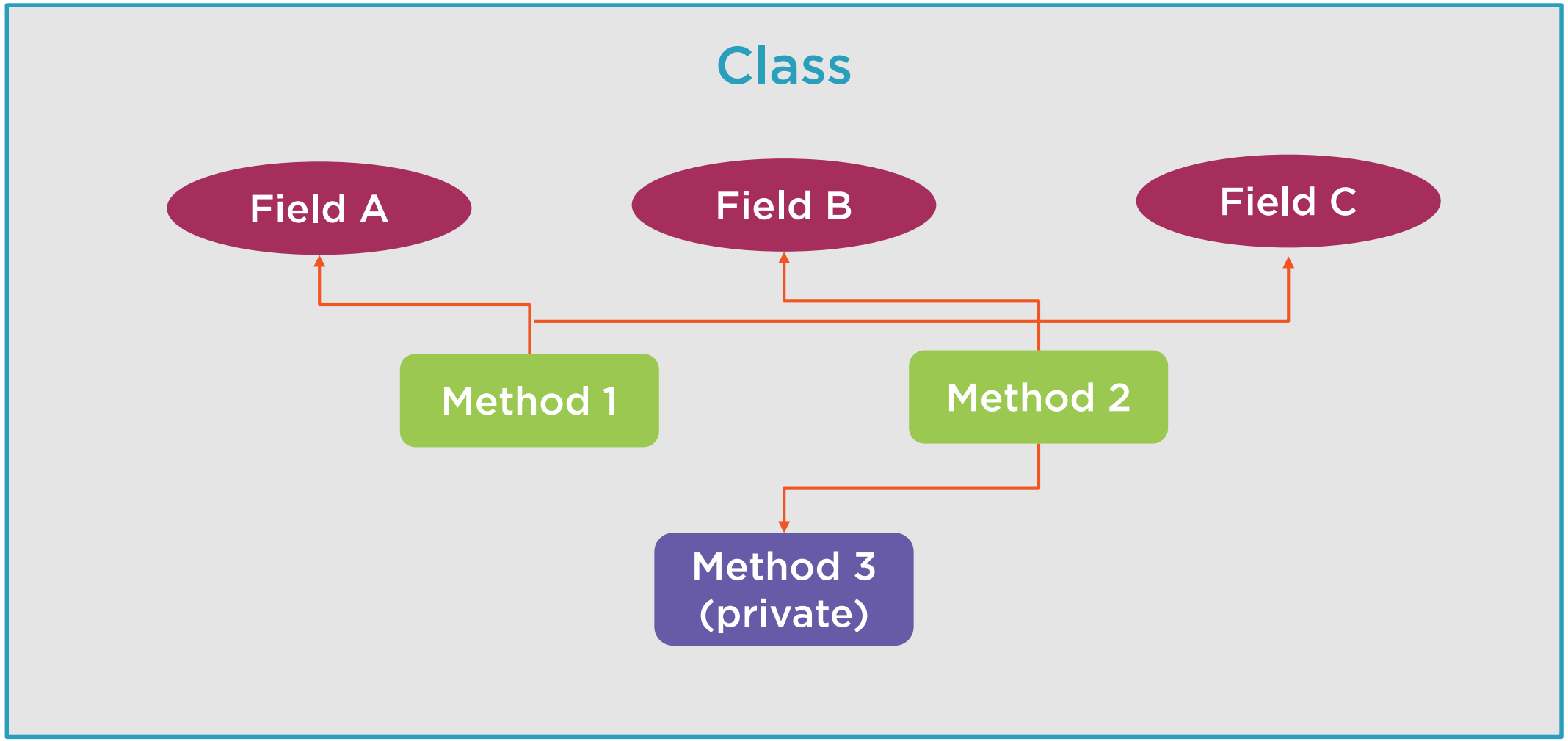
Separation of Concerns

Programs should be separated into distinct sections, each addressing a separate concern, or set of information that affects the program.

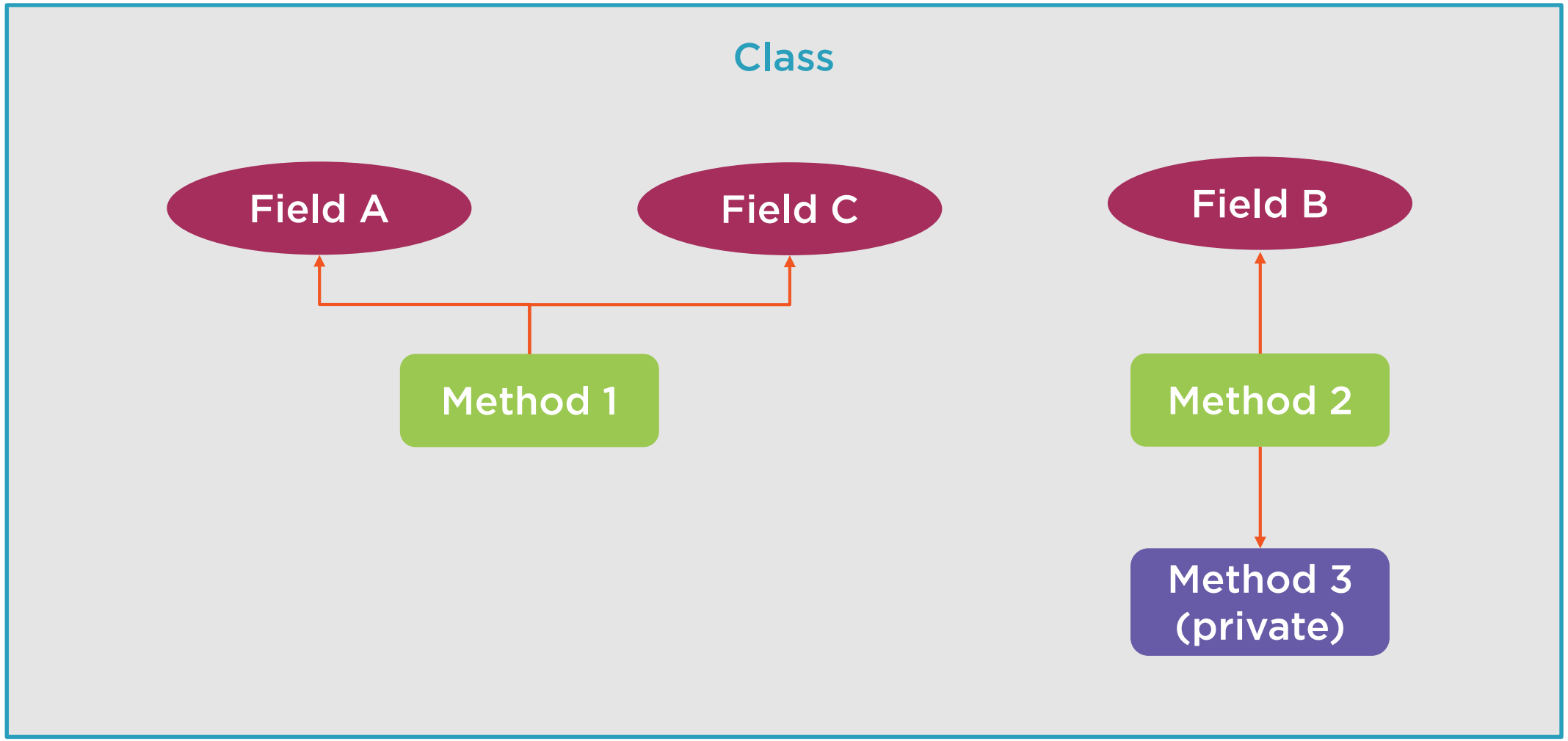
Keep plumbing code separate from high level business logic

Class elements that belong together are cohesive.

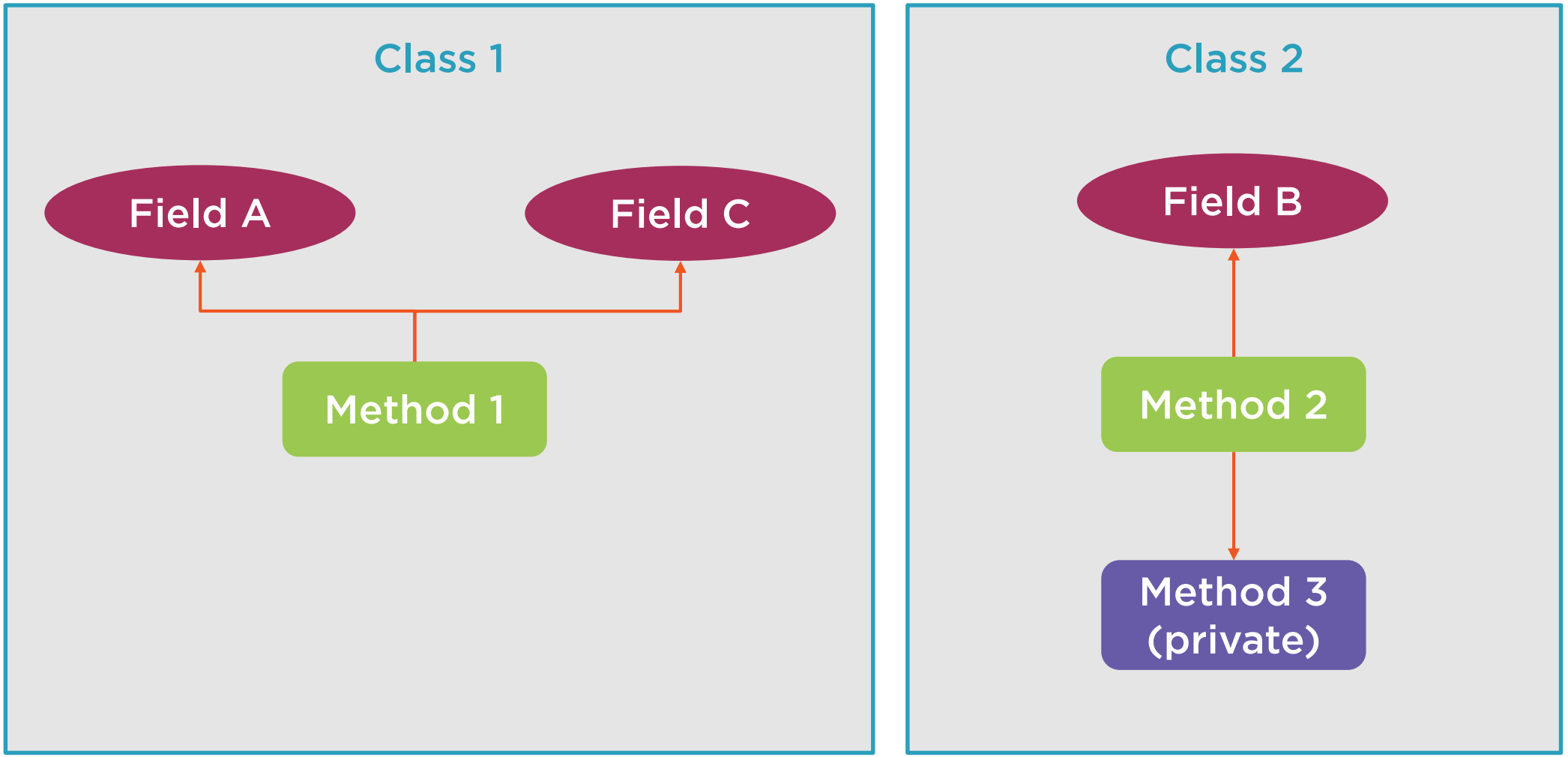
Class Cohesion



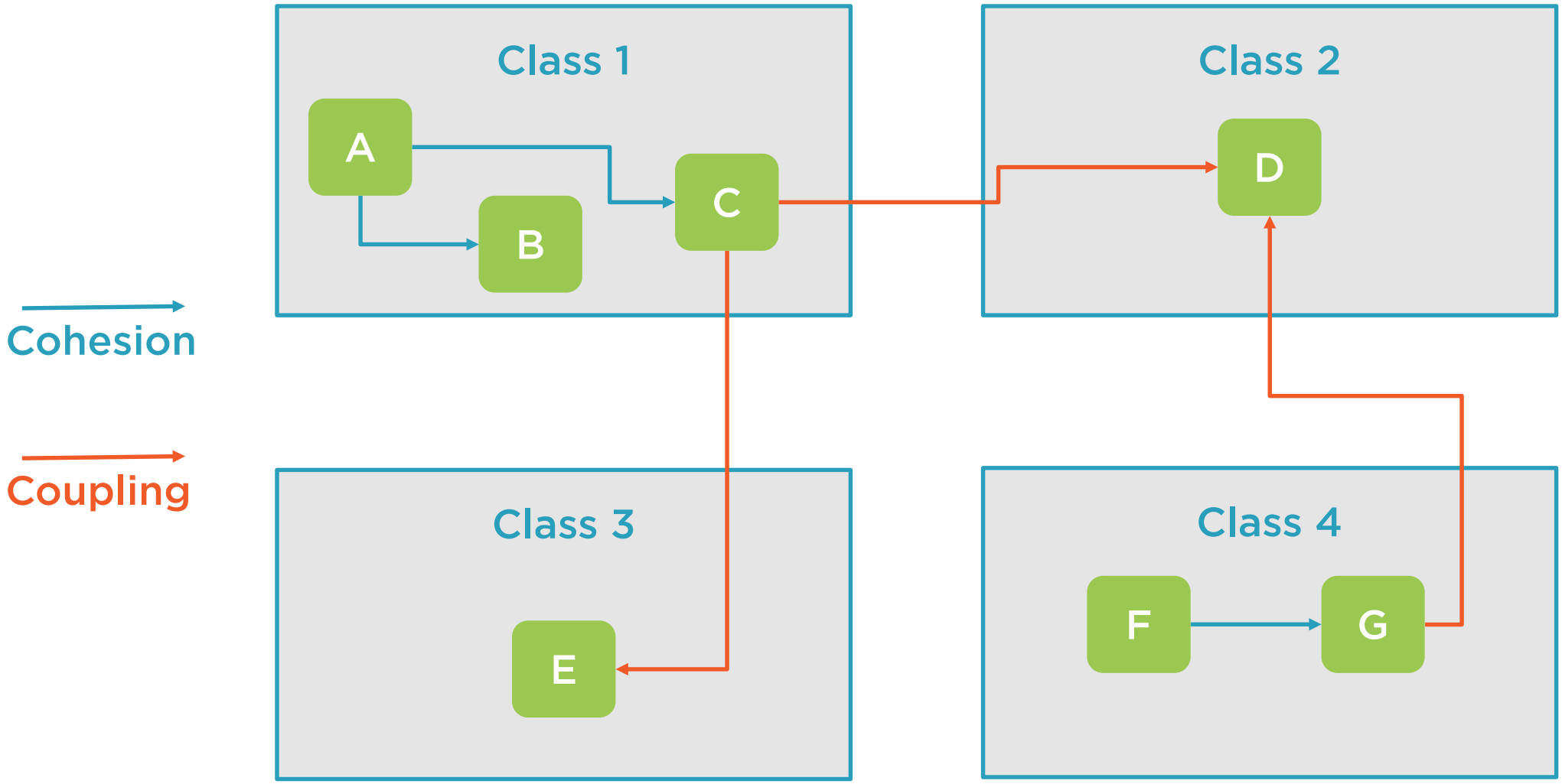
Class Cohesion (Low)



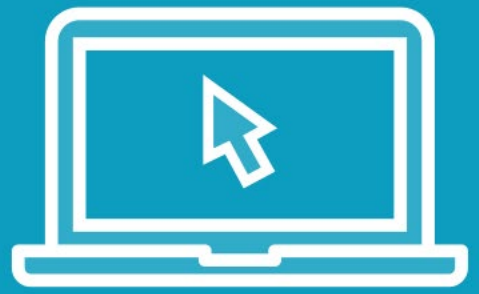
Class Cohesion (High)



Class Coupling and Cohesion



Demo



An Insurance Rating Service

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

How many responsibilities did you find in `RatingEngine`?

```
Console.WriteLine("Starting
rate.");
```

```
string policyJson =
File.ReadAllText("policy.json
");
```

```
var policy =
JsonConvert.DeserializeObject
<Policy>(policyJson,
new StringEnumConverter());
```

- Logging
- Persistence
- Encoding Format

```
case PolicyType.Auto:
```

```
if
(String.IsNullOrEmpty(policy.
Make))
```

```
int age = DateTime.Today.Year -
policy.DateOfBirth.Year;
if (policy.DateOfBirth.Month ==
DateTime.Today.Month &&
DateTime.Today.Day <
policy.DateOfBirth.Day ||
DateTime.Today.Month <
policy.DateOfBirth.Month)
{
    age--;
}
```

- Business Rule – Type of Policy (several of these)
- Validation (many examples)
- Age Calculation

Responsibilities and Testability

Difficult to test one responsibility in isolation

Tests become

- Longer
- More complex
- Brittle
- Coupled to implementation

Testing

```
[Fact]
public void ReturnsRatingOf10000For200000LandPolicy()
{
    var policy = new Policy { Type = PolicyType.Land,
        BondAmount = 200000,Valuation = 200000 };
    string json = JsonConvert.SerializeObject(policy);
    File.WriteAllText("policy.json", json);

    var engine = new RatingEngine();
    engine.Rate();
    var result = engine.Rating;

    Assert.Equal(10000, result);
}
```

Applying SRP to `RatingEngine`

Logging

```
public class ConsoleLogger
{
    public void Log(string message)
    {
        Console.WriteLine(message);
    }
}
```

Logging

```
public class RatingEngine
{
    public ConsoleLogger Logger { get; set; } = new
ConsoleLogger();
    …
}
```

Logging

```
Console.WriteLine("Rating LAND policy...");
Console.WriteLine("Validating policy.");
if (policy.BondAmount == 0 || policy.Valuation == 0)
{
    Console.WriteLine("Land policy must specify Bond Amount
and Valuation.");
    return;
}
```

Logging

```
Logger.Log("Rating LAND policy...");
Logger.Log("Validating policy.");
if (policy.BondAmount == 0 || policy.Valuation == 0)
{
    Logger.Log("Land policy must specify Bond Amount and
Valuation.");
    return;
}
```

Persistence

```
public class FilePolicySource
{
    public string GetPolicyFromSource()
    {
        return File.ReadAllText("policy.json");
    }
}
```

Persistence

```
public class RatingEngine
{
    public FilePolicySource PolicySource { get; set; } =
new FilePolicySource();
    …
}
```

Persistence

```
string policyJson = PolicySource.GetPolicyFromSource();
```

```
public void Rate()

{

    Logger.Log("Starting rate.");
    Logger.Log("Loading policy.");


    string policyJson =
PolicySource.GetPolicyFromSource();


    var policy =
PolicySerializer.GetPolicyFromJsonS
tring(policyJson);

    …
}
```

- Logging (how is delegated)
- Persistence (how is delegated)
- Encoding Format (how is delegated)

Learning More

Pluralsight courses

- “Refactoring Fundamentals”
- “Microsoft Azure Developer: Refactoring Code”

Improved Testability

The 3 new classes are easily tested

The RatingEngine can now swap in test implementations of these 3 dependencies

Testing Serializer in Isolation

```
[Fact]
public void ReturnsDefaultPolicyFromEmptyJsonString()
{
    var inputJson = "{}";
    var serializer = new JsonPolicySerializer();

    var result = serializer.GetPolicyFromJsonString(inputJson);

    var policy = new Policy();
    AssertPoliciesEqual(result, policy);
}
```

SOLID Principles

Single Responsibility Principle

Open / Closed Principle

Liskov Substitution Principle

Interface Segregation Principle

Dependency Inversion Principle

Key Takeaways

Practice Pain Driven Development!

Each class should have a single responsibility, or reason to change

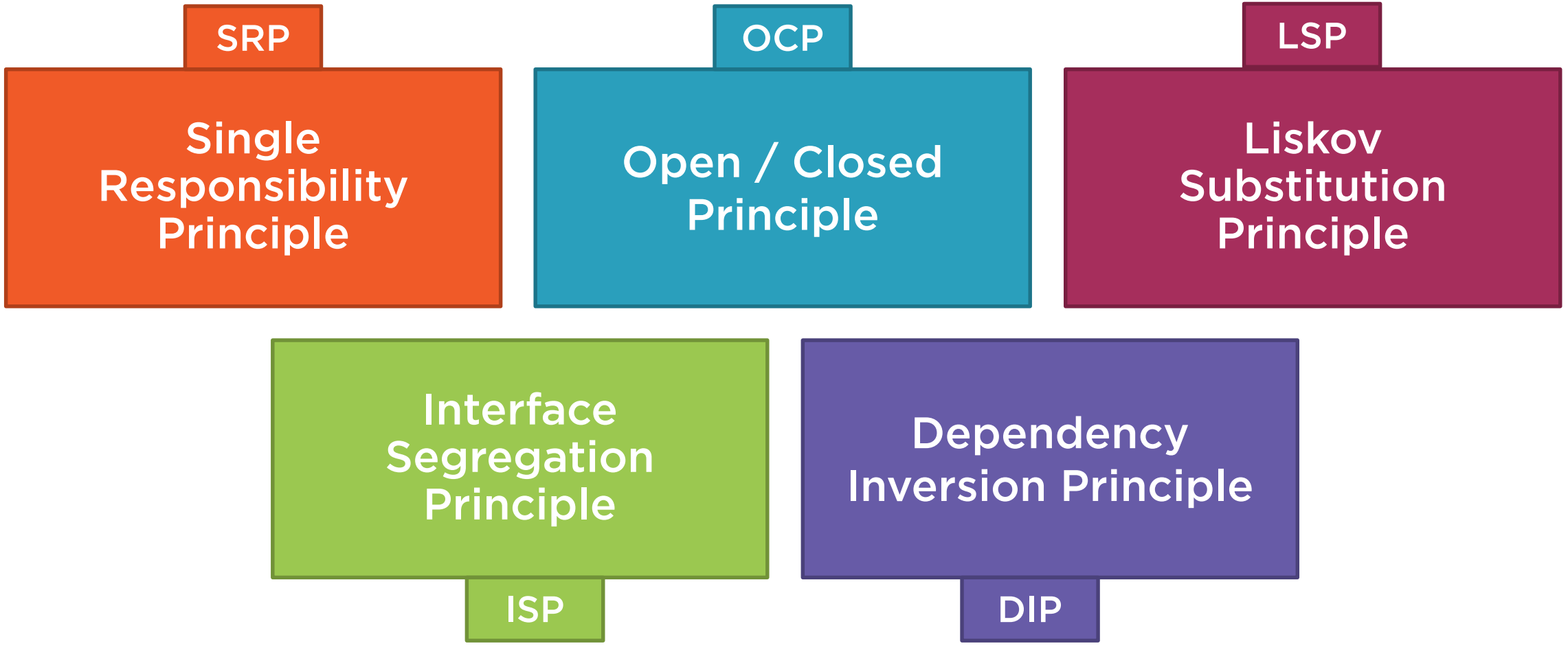
Strive for high cohesion and loose coupling

Keep classes small, focused, and testable

Open / Closed Principle

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

SOLID Principles



Open / Closed Principle

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

Dr. Bertrand Meyer originated the term in his 1988 book, *Object-Oriented Software Construction*

It should be possible to change the behavior of a method without editing its source code.

Open / Closed

Open to extension

New behavior can be added in the future

Code that is closed to extension has fixed behavior

Closed to modification

Changes to source or binary code are not required

The only way to change the behavior of code that is closed to extension is to change the code itself

Why Should Code Be Closed to Modification?

Less likely to introduce bugs in code we don't touch or redeploy

Less likely to break dependent code when we don't have to deploy updates

Fewer conditionals in code that is open to extension results in simpler code

Bug fixes are ok

How Do We Add Another Policy Type?

```
switch (policy.Type)
{
    case PolicyType.Auto:

    case PolicyType.Land:

    case PolicyType.Life:
}
```

Balance abstraction and concreteness

Abstraction adds complexity

Predict where variation is needed and apply abstraction as needed

Extremely Concrete

```
public class DoOneThing
{
    public void Execute()
    {
        Console.WriteLine(“Hello world.”);
    }
}
```

Extremely Concrete

```
public class DoSomethingElse
{
    public void SomethingElse()
    {
        var doThing = new DoOneThing();
        doThing.Execute();
        // other stuff
    }
}
```

Extreme Extensibility

```
public class DoAnything<TArg, TResult>
{
    private Func<TArg, TResult> _function;
    public DoAnything(Func<TArg, TResult> function)
    {
        _function = function;
    }
    public TResult Execute(TArg a)
    {
        return _function(a);
    }
}
```

Extreme Extensibility

```
public abstract class DoAnything<TResult, TArg>
{
    public abstract TResult Execute(TArg a);
}
```

How Can You Predict Future Changes?

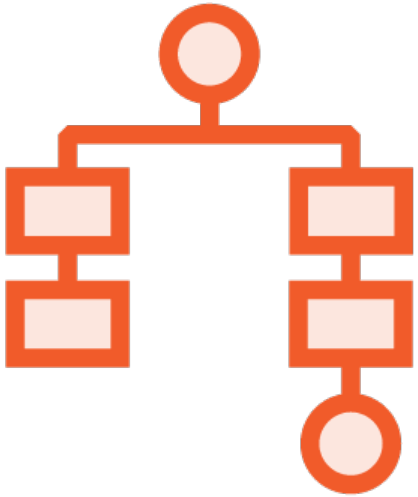
Start concrete

Modify the code the first time or two

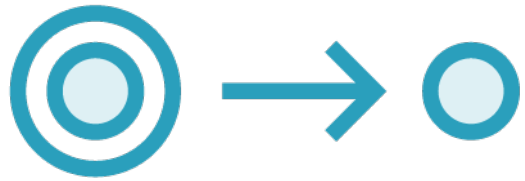
By the third modification, consider making the code open to extension for that axis of change

Typical Approaches to OCP

Parameters



Inheritance



Composition / Injection

Extremely Concrete

```
public class DoOneThing
{
    public void Execute()
    {
        Console.WriteLine(“Hello world.”);
    }
}
```

Parameter-Based Extension

```
public class DoOneThing
{
    public void Execute(string message)
    {
        Console.WriteLine(message);
    }
}
```

Inheritance-based Extension

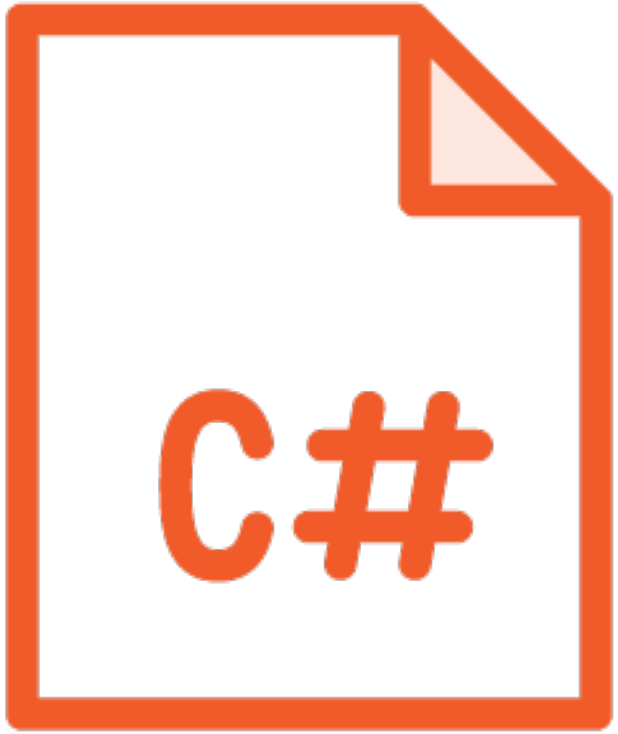
```
public class DoOneThing
{
    public virtual void Execute()
    {
        Console.WriteLine(“Hello world.”);
    }
}
public class DoAnotherThing
{
    public override void Execute()
    {
        Console.WriteLine(“Goodbye world!”);
    }
}
```

Composition/Injection Extension

```
public class DoOneThing
{
    private readonly MessageService _messageService;
    public DoOneThing(MessageService messageService)
        => _messageService = messageService;

    public void Execute()
    {
        Console.WriteLine(_messageService.GetMessage());
    }
}
```

Prefer implementing
new features in new classes.

Why Use a New Class?



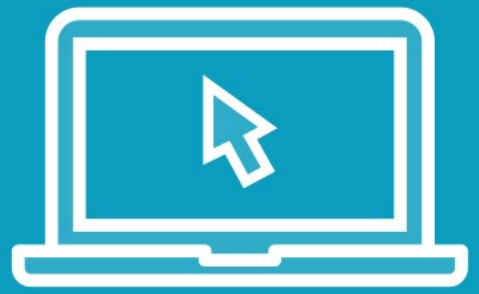
- Design class to suit problem at hand
- Nothing in current system depends on it
- Can add behavior without touching existing code
- Can follow Single Responsibility Principle
- Can be unit-tested

Learn More

Maintain Legacy Code with New Code

- weeklydevtips.com/015

Demo



Applying OCP to RatingService

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

Packages and Libraries

Closed for modification

- Consumers cannot change package contents

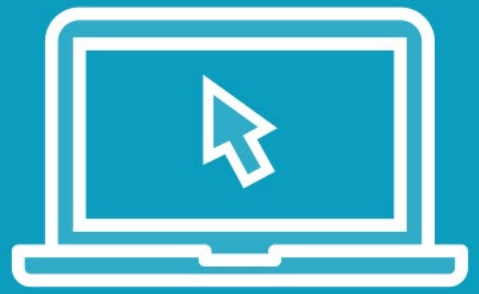
Closed for modification

- Should not break consumers when new behavior is added

Open to extension

- Consumers should be able to extend the package to suit their own needs

Demo



NuGet package example: Guard Clauses

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

More Resources on OCP

Why you need to know OCP but don't

- https://bit.ly/2LSXOuo

Open Closed Principle by Robert Martin

- https://bit.ly/2Gmxg1Z

Open Closed Principle by Jon Skeet

- https://bit.ly/2AMmprC

SOLID Principles

Single Responsibility Principle

Open / Closed Principle

Liskov Substitution Principle

Interface Segregation Principle

Dependency Inversion Principle

Key Takeaways

Solve the problem first using simple, concrete code

Identify the kinds of changes the application is likely to continue needing

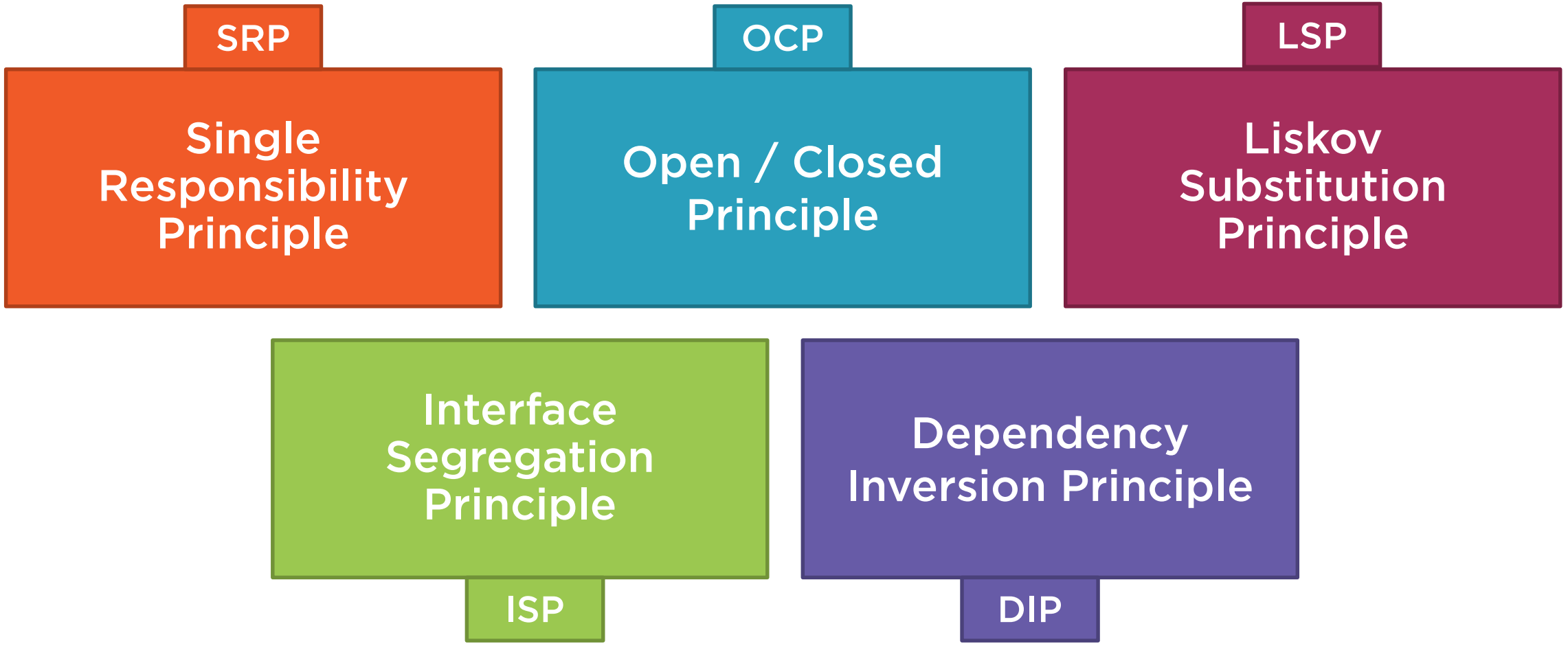
Modify code to be extensible along the axis of change you've identified

- Without the need to modify its source each time

Liskov Substitution Principle

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

SOLID Principles



Liskov Substitution Principle

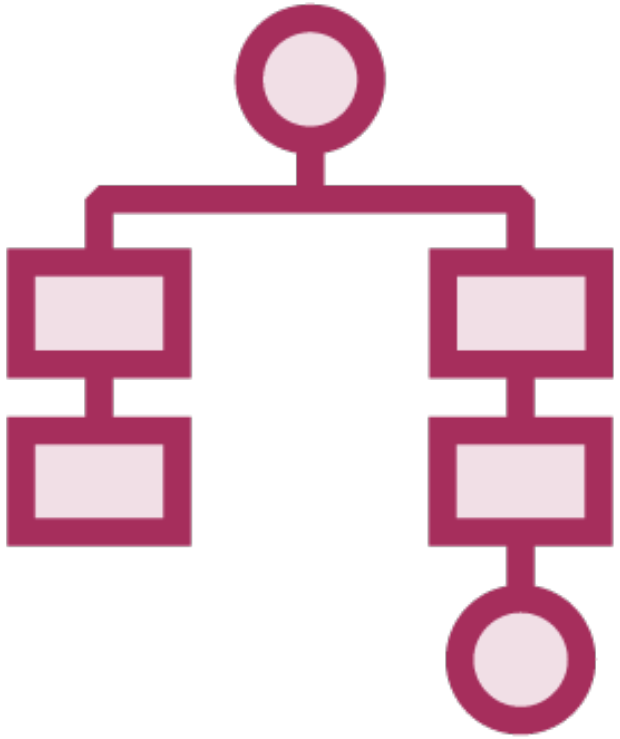
Let Φ(x) be a property provable about objects x of type T. Then Φ(y) should be true for objects y of type S where S is a subtype of T.

Liskov Substitution Principle

Subtypes must be substitutable for their base types.

Barbara Liskov introduced the principle in a conference keynote in 1987.

Basic Object-Oriented Design



Something IS-A something else

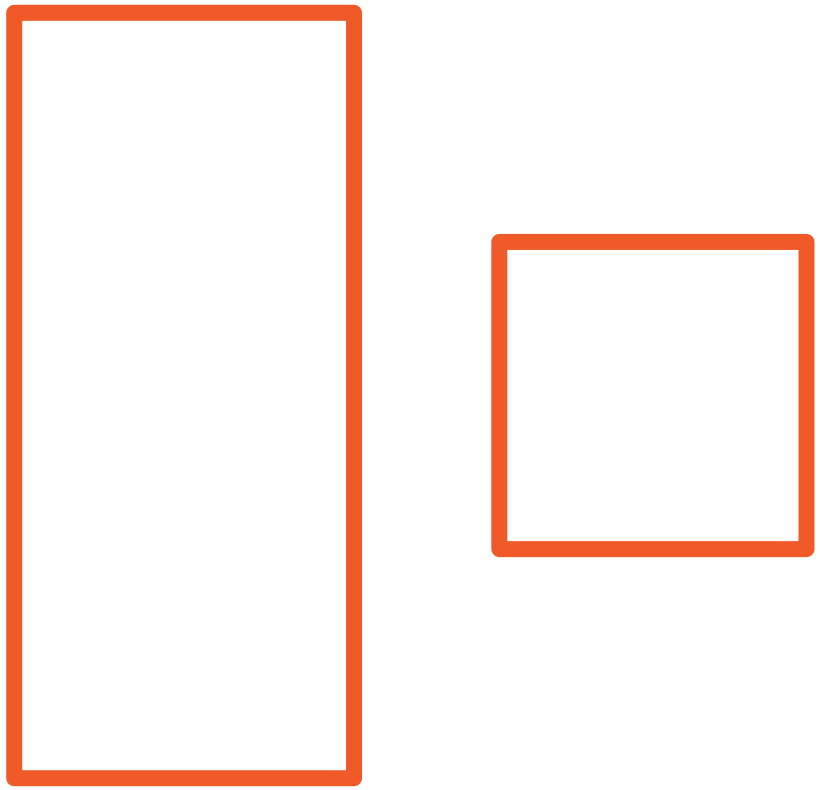
- An eagle IS-A bird

Something HAS-A property

- An address HAS-A city

LSP states that the IS-A relationship is insufficient and should be replaced with IS-SUBSTITUTABLE-FOR.

Classic Rectangle-square Problem



A rectangle has four sides and four right angles

A square has four equal sides and four right angles

Per geometry, a square is a rectangle

Rectangle

```
public class Rectangle
{
    public virtual int Height { get; set; }
    public virtual int Width { get; set; }
}
```

Area Calculation Utility

```
public class AreaCalculator
{
    public static int CalculateArea(Rectangle r)
    {
        return r.Height * r.Width;
    }
}
```

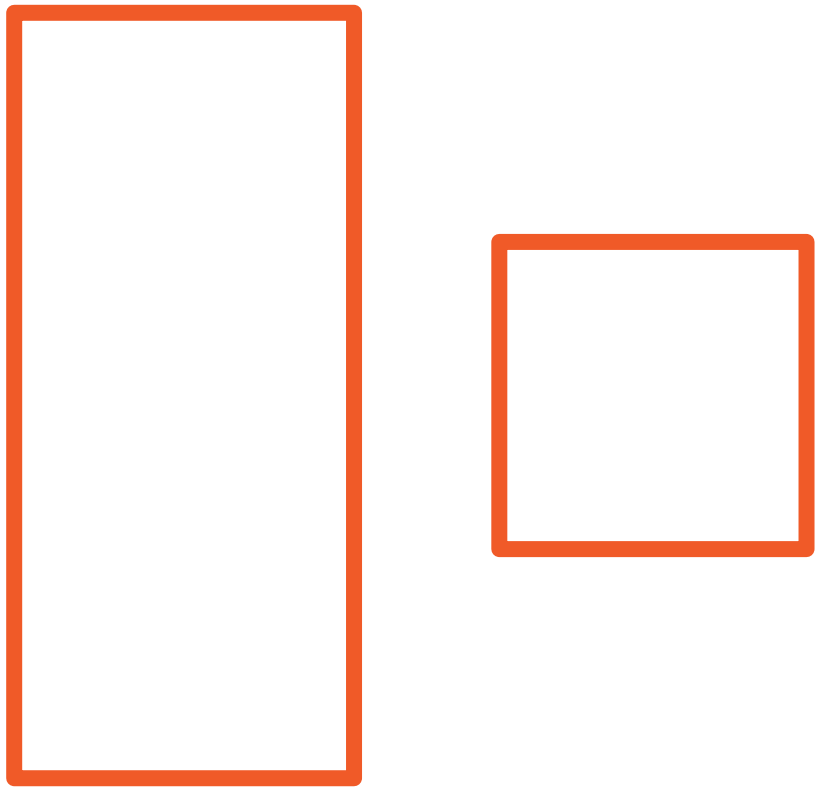
Square (a Subtype of Rectangle)

```
public class Square : Rectangle
{
    private int _height;
    public int Height
    {

        get { return _height; }
        set
        {
            _width = value;
            _height = value;
        }
    }
    // Width implemented similarly
}
```

The Problem

```
Rectangle myRect = new Square();
myRect.Width = 4;
myRect.Height = 5;

Assert.Equal(20, AreaCalculator.CalculateArea(myRect));

// Actual Result: 25
```

What Happened?



Square has an invariant

- Its sides must be equal

Rectangle has an invariant

- Its sides are independent

This design breaks rectangle's invariant and thus violates LSP

One Solution

```
public class Rectangle
{
    public int Height { get; set; }
    public int Width { get; set; }

    public bool IsSquare => Height == Width;
}
```

Another Solution

```
public class Rectangle
{
    public int Height { get; set; }
    public int Width { get; set; }
}

public class Square
{
    public int Side { get; set; }
}
```

Detecting LSP Violations in Your Code

Type checking with `is` or `as` in polymorphic code

Null checks

`NotImplementedException`

Type Checking

```
foreach(var employee in employees)
{
    if(employee is Manager)
    {
        Helpers.PrintManager(employee as Manager);
        break;
    }
    Helpers.PrintEmployee(employee);
}
```

Type Checking (Corrected)

```
foreach(var employee in employees)
{
    employee.Print();
}

// OR

foreach(var employee in employees)
{
    Helpers.PrintEmployee(employee);
}
```

Null Checking

```
foreach(var employee in employees)
{
    if(employee == null)
    {
        Console.WriteLine("Employee not found.");
        break;
    }
    Helpers.PrintEmployee(employee);
}
```

Type Checking

```
foreach(var employee in employees)
{
    if(employee is Manager)
    {
        Helpers.PrintManager(employee as Manager);
        break;
    }
    Helpers.PrintEmployee(employee);
}
```

Learn More

Nulls Break Polymorphism

- ardalis.com/nulls-break-polymorphism

Pluralsight courses

- “Design Patterns Library”

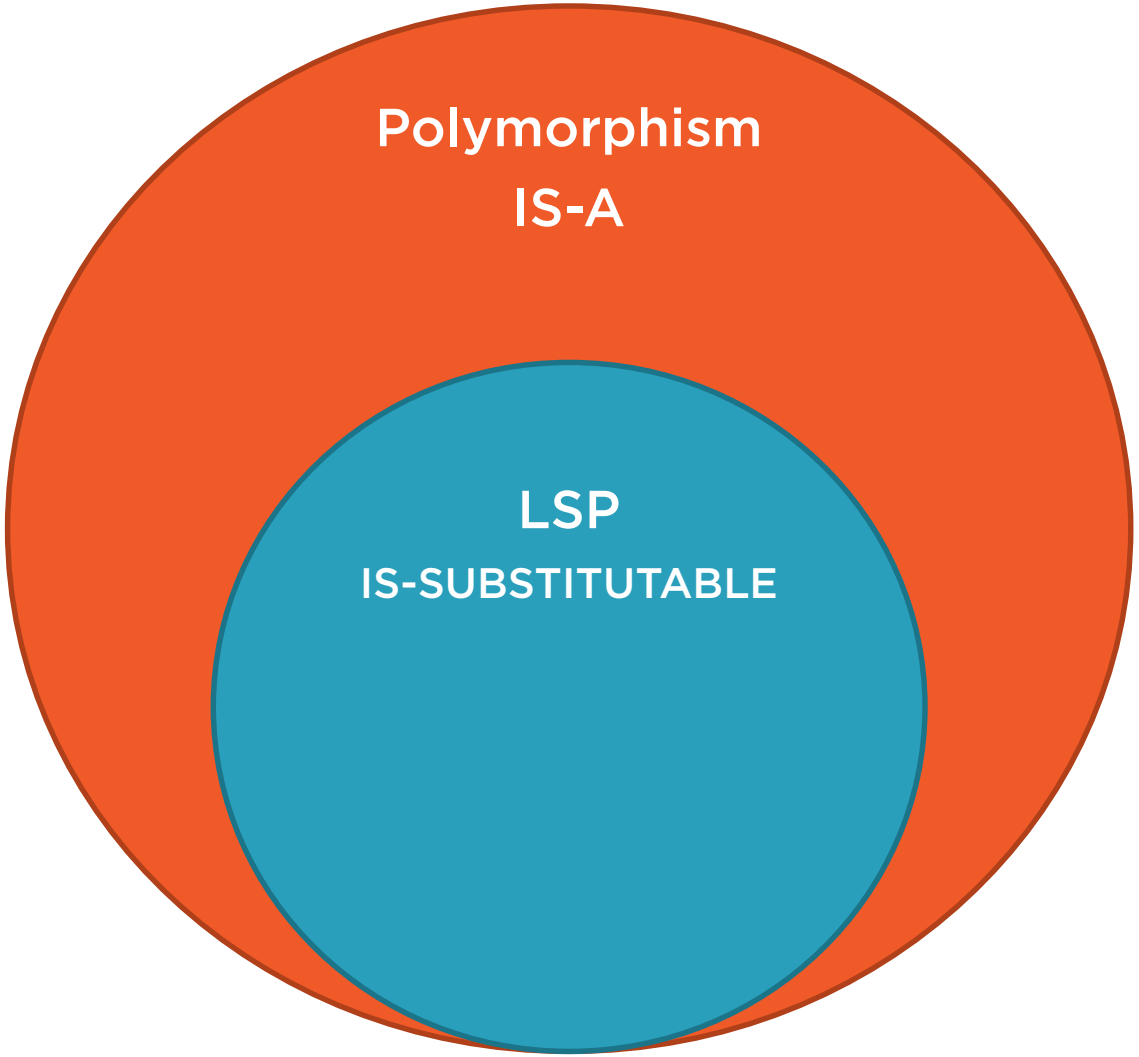
Not Implemented Exceptions

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

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

Not Implemented Exceptions

```
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();
    }
}
```

LSP Is a Subset of Polymorphism



Fixing LSP Violations

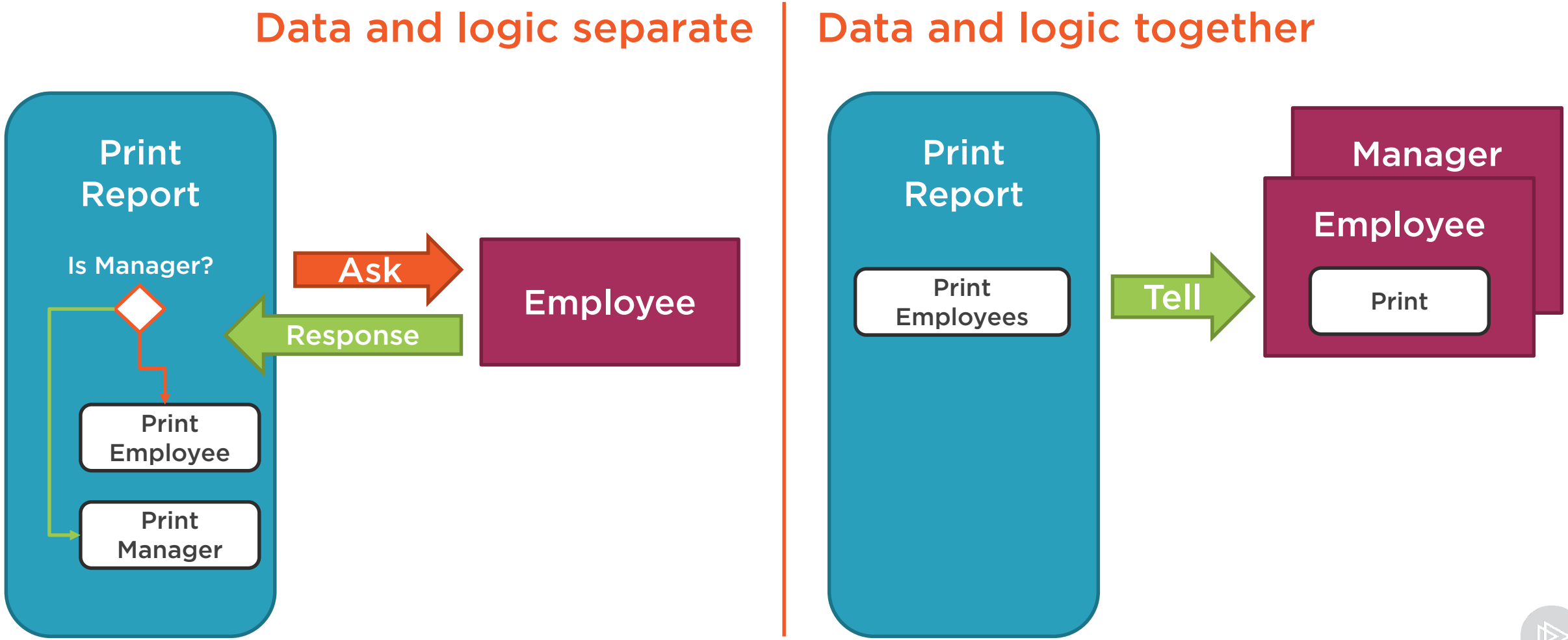
Follow the "Tell, Don't Ask" principle

Minimize null checks with

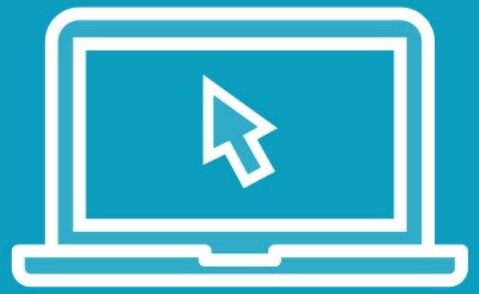
- C# features
- Guard clauses
- Null Object design pattern

Follow ISP and be sure to fully implement interfaces

Tell, Don't Ask



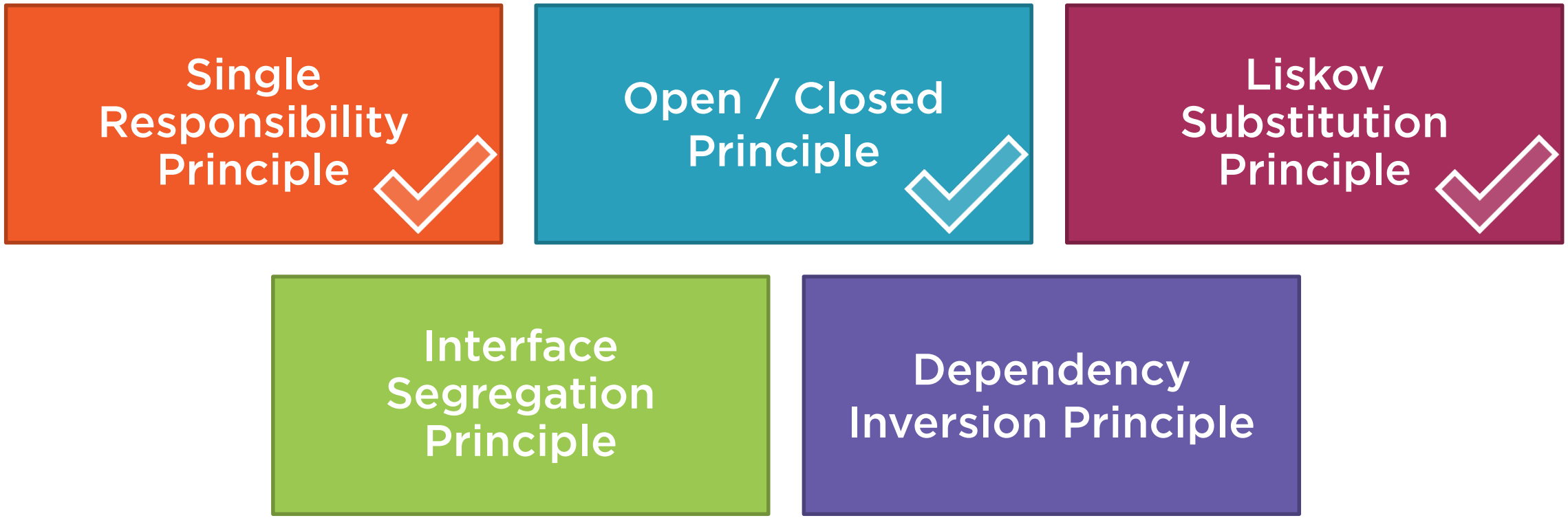
Demo



Applying LSP to ArdalisRating

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

SOLID Principles



Key Takeaways

Subtypes must be substitutable for their base types

Ensure base type invariants are enforced

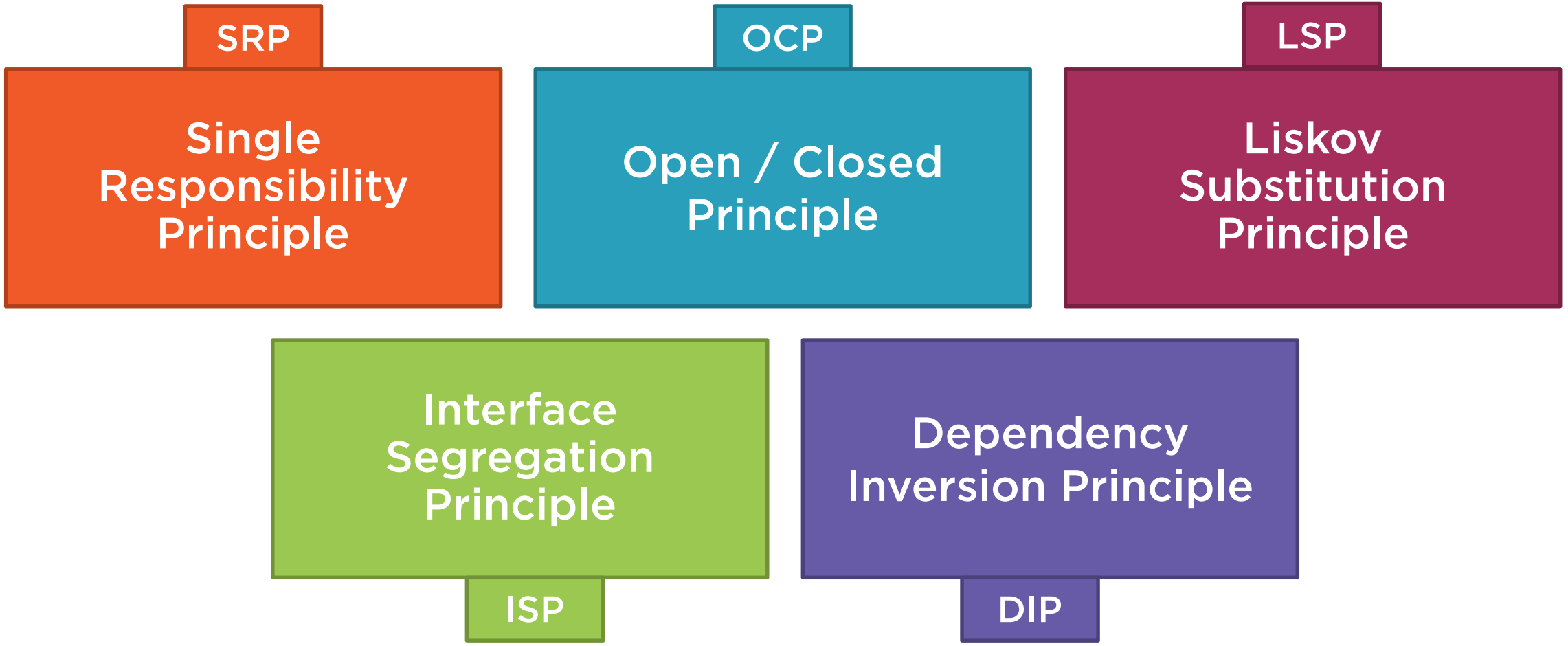
Look for

- Type checking
- Null checking
- `NotImplementedException`

Interface Segregation Principle

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

SOLID Principles

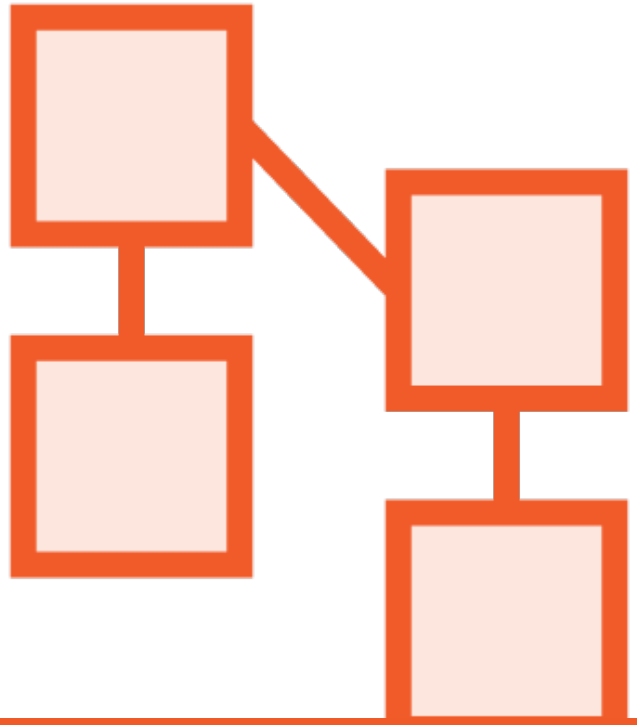


Interface Segregation Principle

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

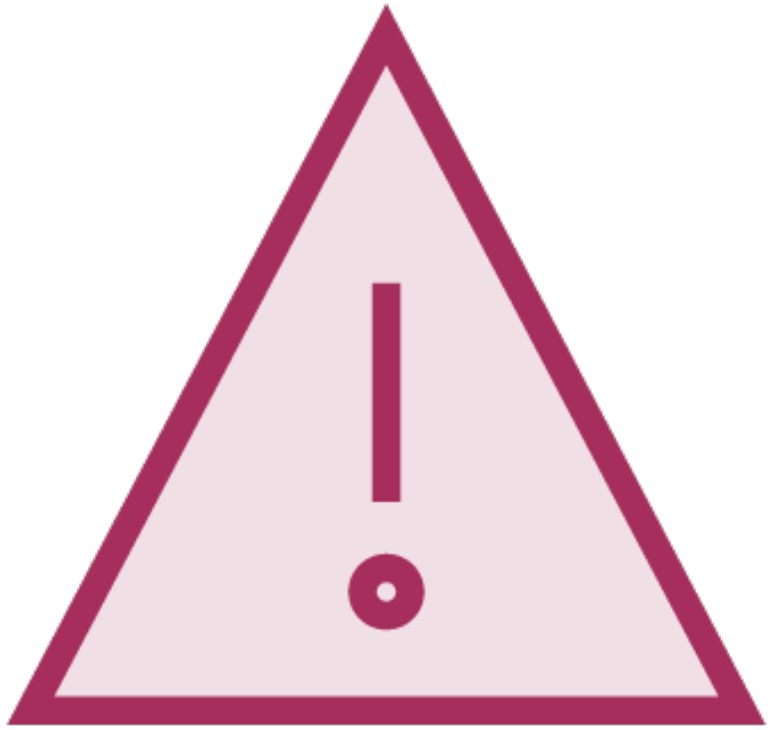
```
public class MyMembershipProvider : MembershipProvider
{
    0 references
    public override bool EnablePasswordRetrieval => throw new System.NotImplementedException();
    0 references
    public override bool EnablePasswordReset => throw new System.NotImplementedException();
    0 references
    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

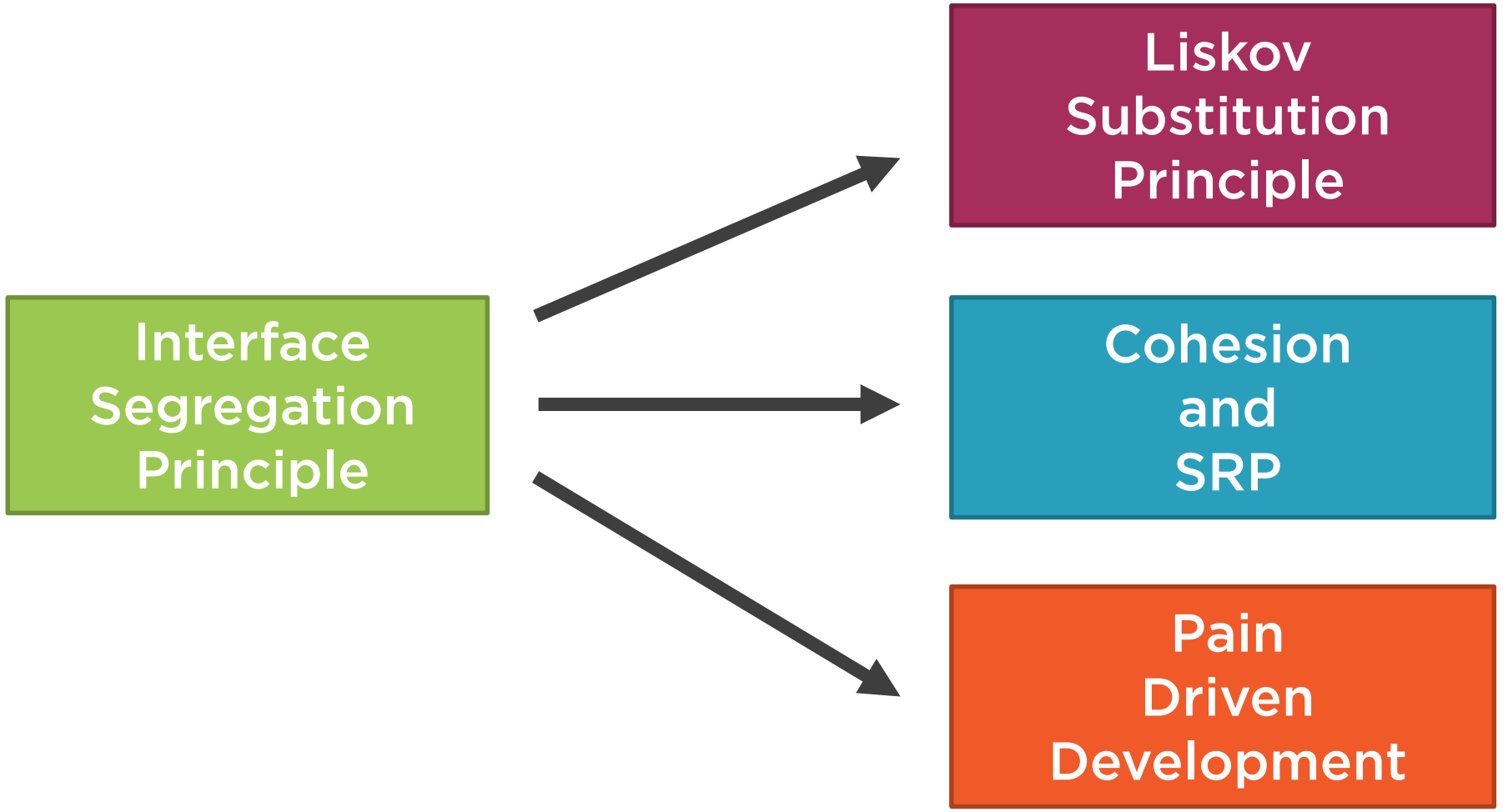
```
public interface IEmailNotificationService
{
    void SendEmail(string to, string from,
                   string subject, string body);
}

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

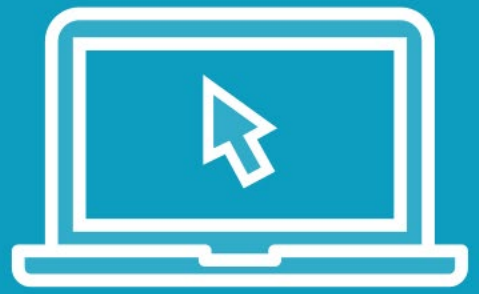
Clean Architecture Solution Template

- github.com/ardalis/CleanArchitecture

On Pluralsight

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

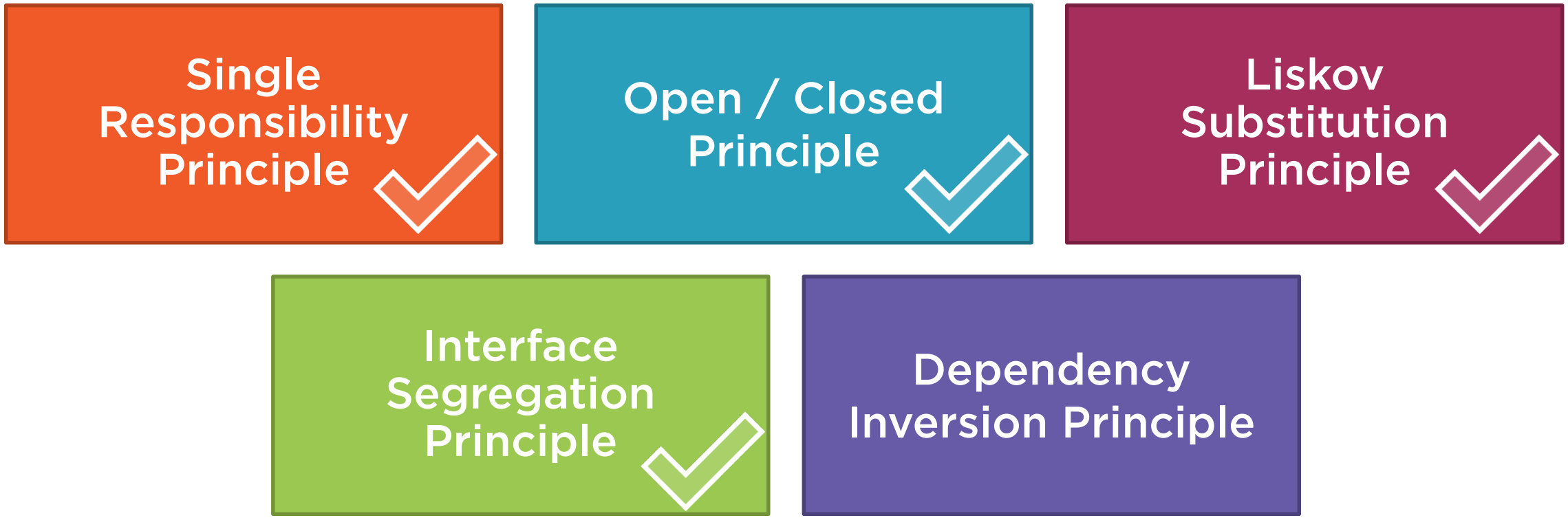
Demo



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

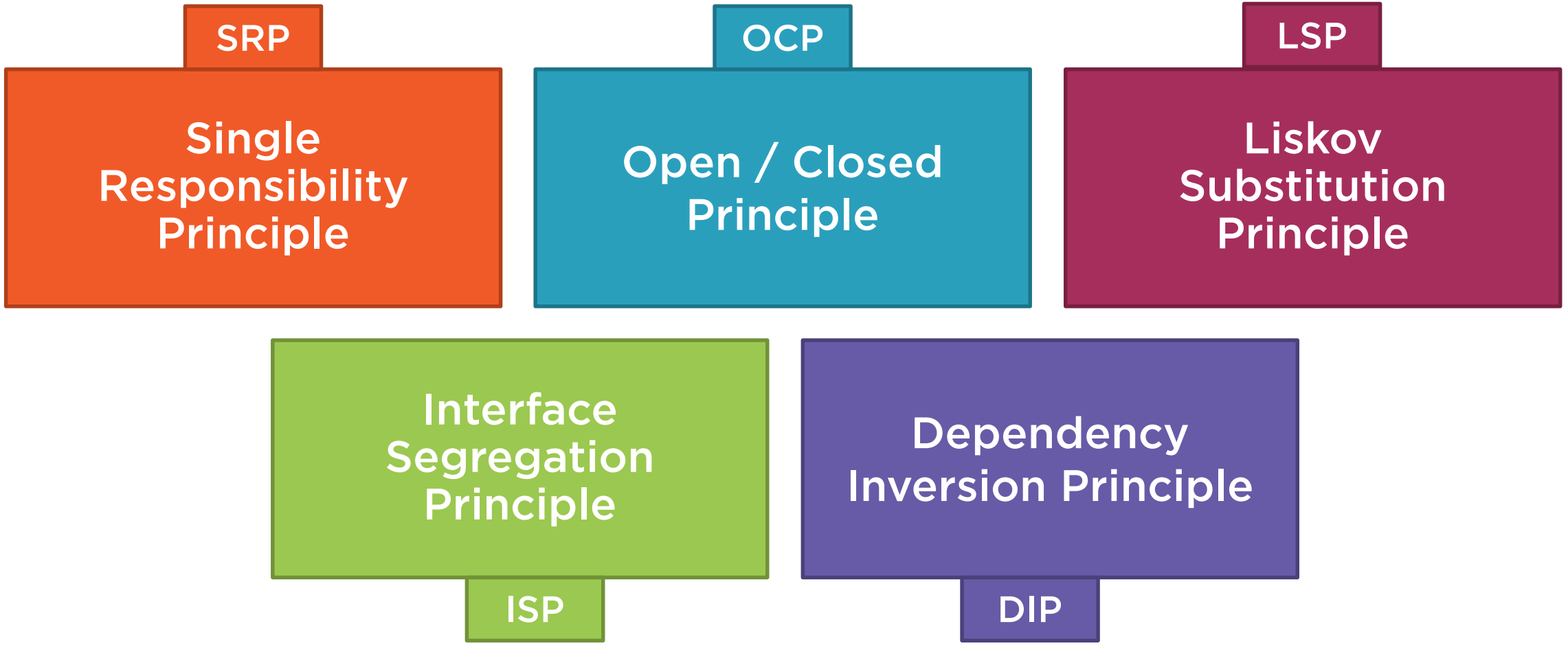
Break up large interfaces by using

- Interface inheritance
- The Adapter design pattern

Dependency Inversion Principle

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

SOLID Principles



Dependency Inversion Principle

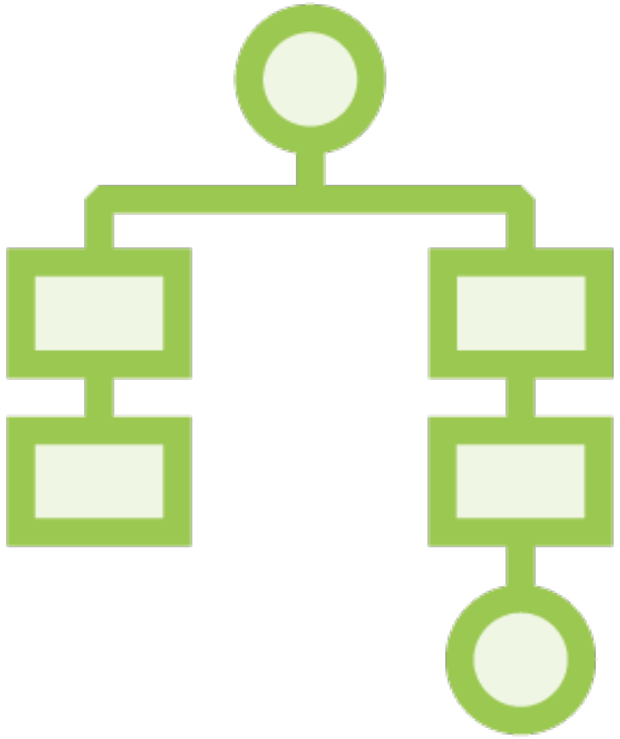
High-level modules should not depend on low-level modules. Both should depend on abstractions.

Abstractions should not depend on details.

Details should depend on abstractions.

How do I know if something depends on something else?

Dependencies in C#



References required to compile

References required to run

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”

What's the difference between "high-level" and "low-level"?

High Level

More abstract

Business rules

Process-oriented

Further from input/output (I/O)

Low Level

Closer to I/O

"Plumbing" code

Interacts with specific external systems and hardware

Separation of Concerns

Keep plumbing code separate from high level business logic

What's an abstraction?

(in C#, in this context)

Abstractions in C#

Interfaces

Abstract base classes

“Types you can’t instantiate”

What about *details*?

Abstractions should not depend on *details*.
Details should depend on abstractions.

Details and Abstractions

Abstractions shouldn't be coupled to details

Abstractions describe what

- Send a message
- Store a Customer record

Details specify how

- Send an SMTP email over port 25
- Serialize Customer to JSON and store in a text file

Depending on Details

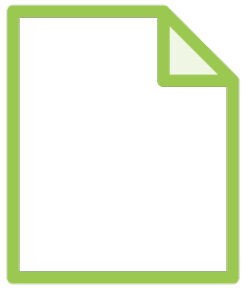
```
public interface IOrderDataAccess
{
    SqlDataReader ListOrders(SqlParameterCollection params);
}
```

Abstractions Should Not Depend on Details

```
public interface IOrderDataAccess
{
    List<Order> ListOrders(Dictionary<string,string> params);
}
```

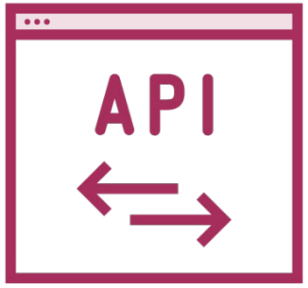
Low Level Dependencies

Database

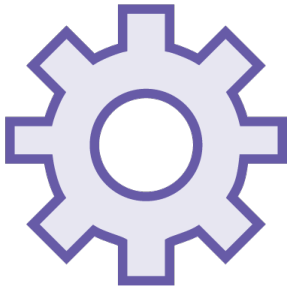


File system

Email



Web APIs



Configuration

Clock

Hidden Direct Dependencies

Direct use of low level dependencies

Static calls and `new`

Causes pain

- **Tight coupling**
- **Difficult to isolate and unit test**
- **Duplication**

New Is Glue

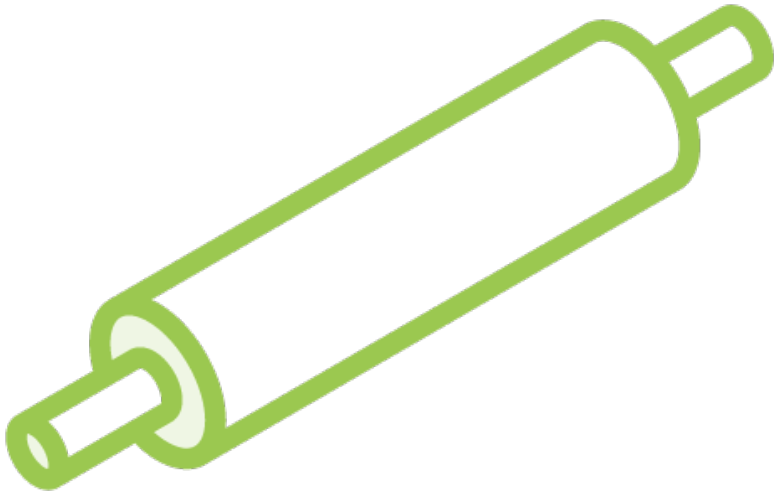
Using new to create dependencies glues your code to that dependency

- ardalis.com/new-is-glue

New isn't bad – just bear in mind the coupling it creates

- Do you need to specify the implementation?
- Could you use an abstraction instead?

Explicit Dependencies Principle

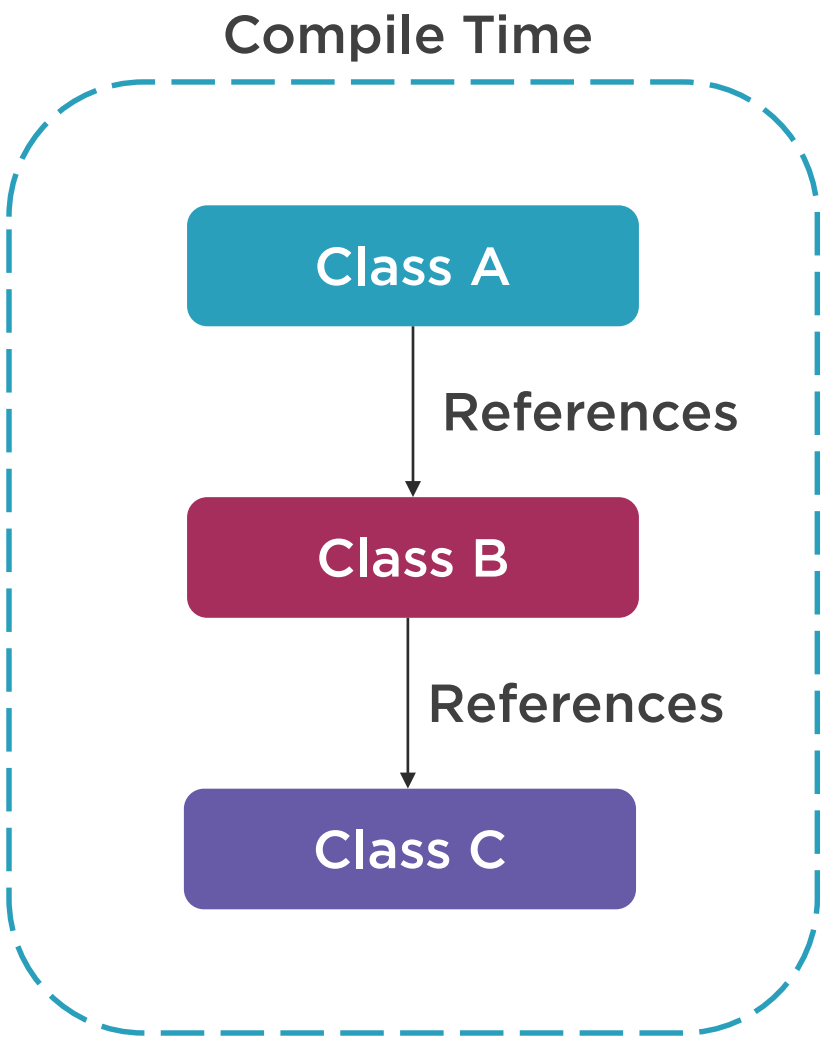


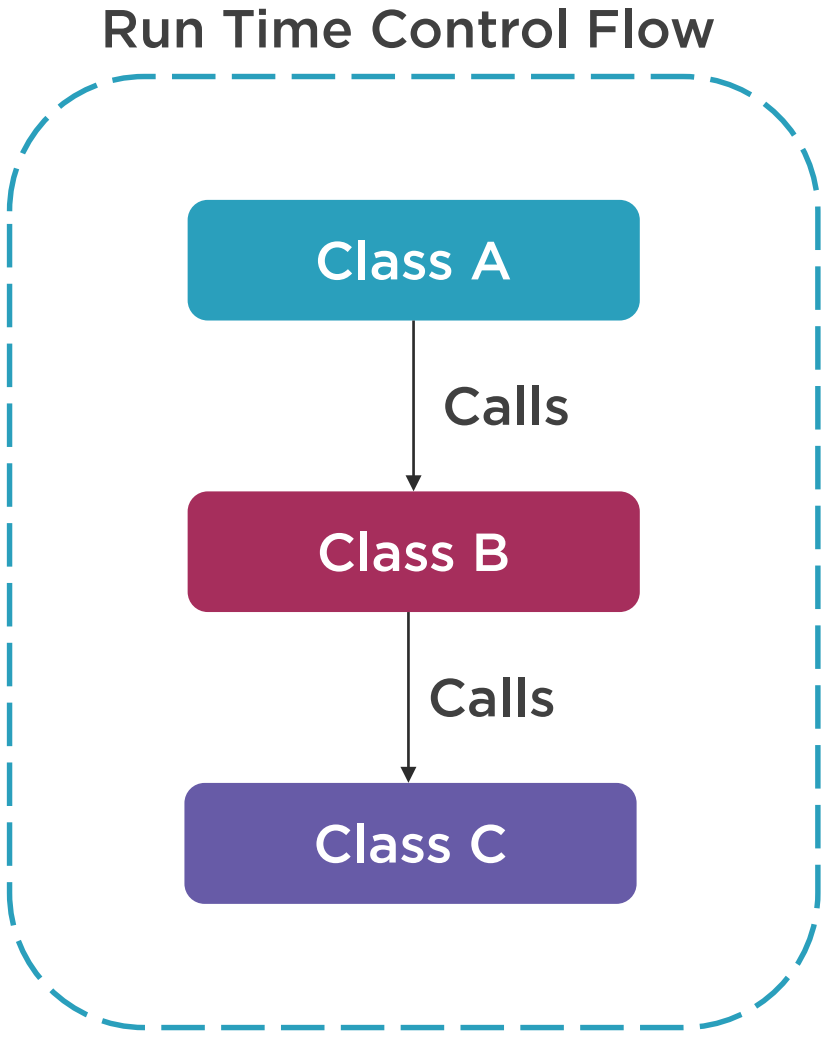
Your classes shouldn't surprise clients with dependencies

List them up front, in the constructor

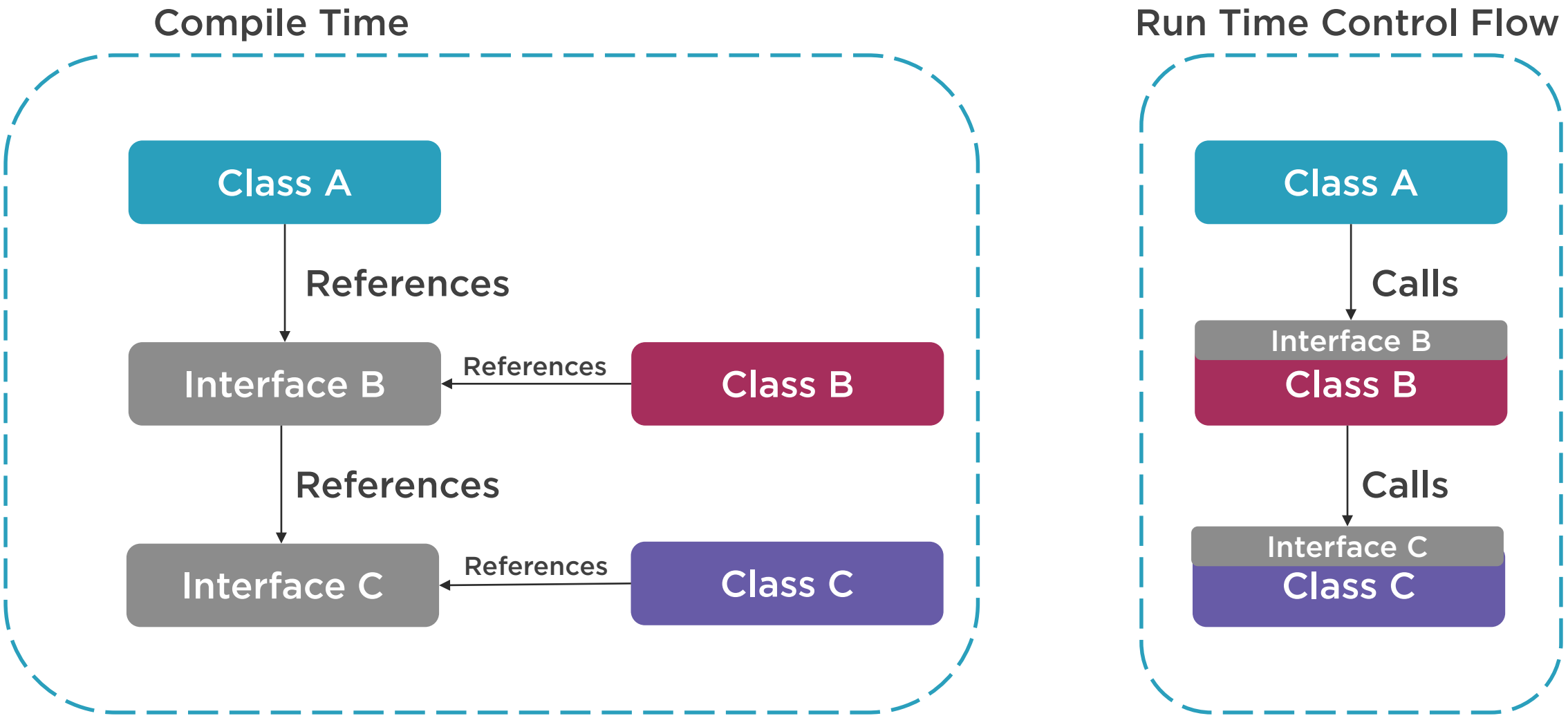
Think of them as ingredients in a cooking recipe

Dependencies without Abstractions





Dependencies with Abstractions



Learn More

Microsoft Docs: Architectural Principles

- bit.ly/2tmcebj

Explicit Dependencies Principle

- bit.ly/2Iea2Nu

Dependency Injection

Don’t create your own dependencies

- Depend on abstractions
- Request dependencies from client

Client injects dependencies as

- Constructor arguments
- Properties
- Method arguments

See also: Strategy Design Pattern

Tip: Prefer Constructor Injection

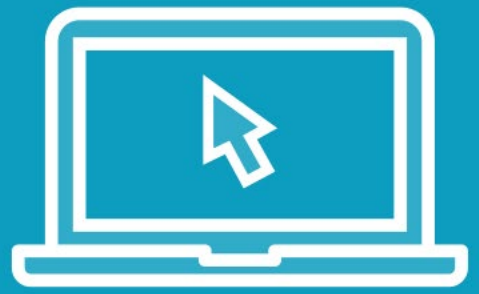
Follows Explicit Dependencies Principle

Classes are never in uninitialized state

Can leverage an IOC container to construct types and their dependencies

IOC, or “Inversion of Control” containers are sometimes called “dependency injection” (DI) containers or simply services containers.

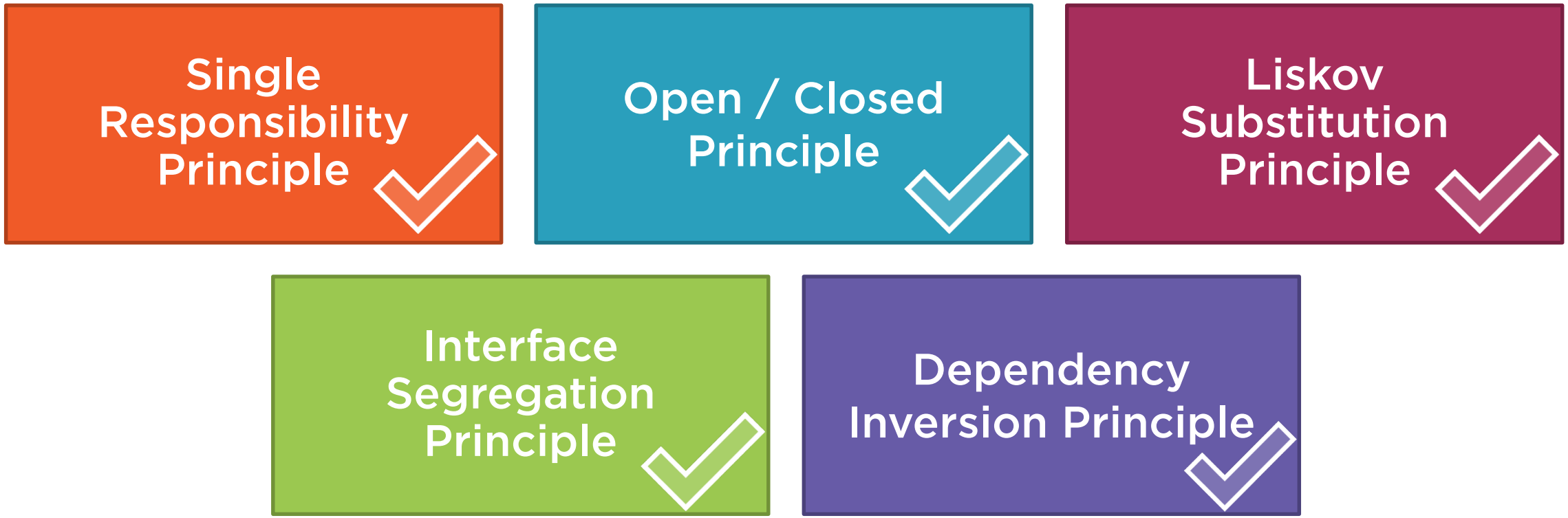
Demo



Applying DIP to ArdalisRating

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

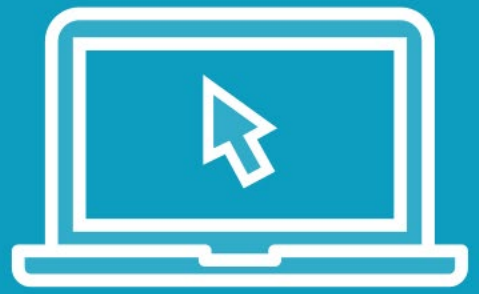
SOLID Principles



Too many files?

Use folders!

Demo



Organizing ArdalisRating and Supporting a Web Front End

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

Key Takeaways

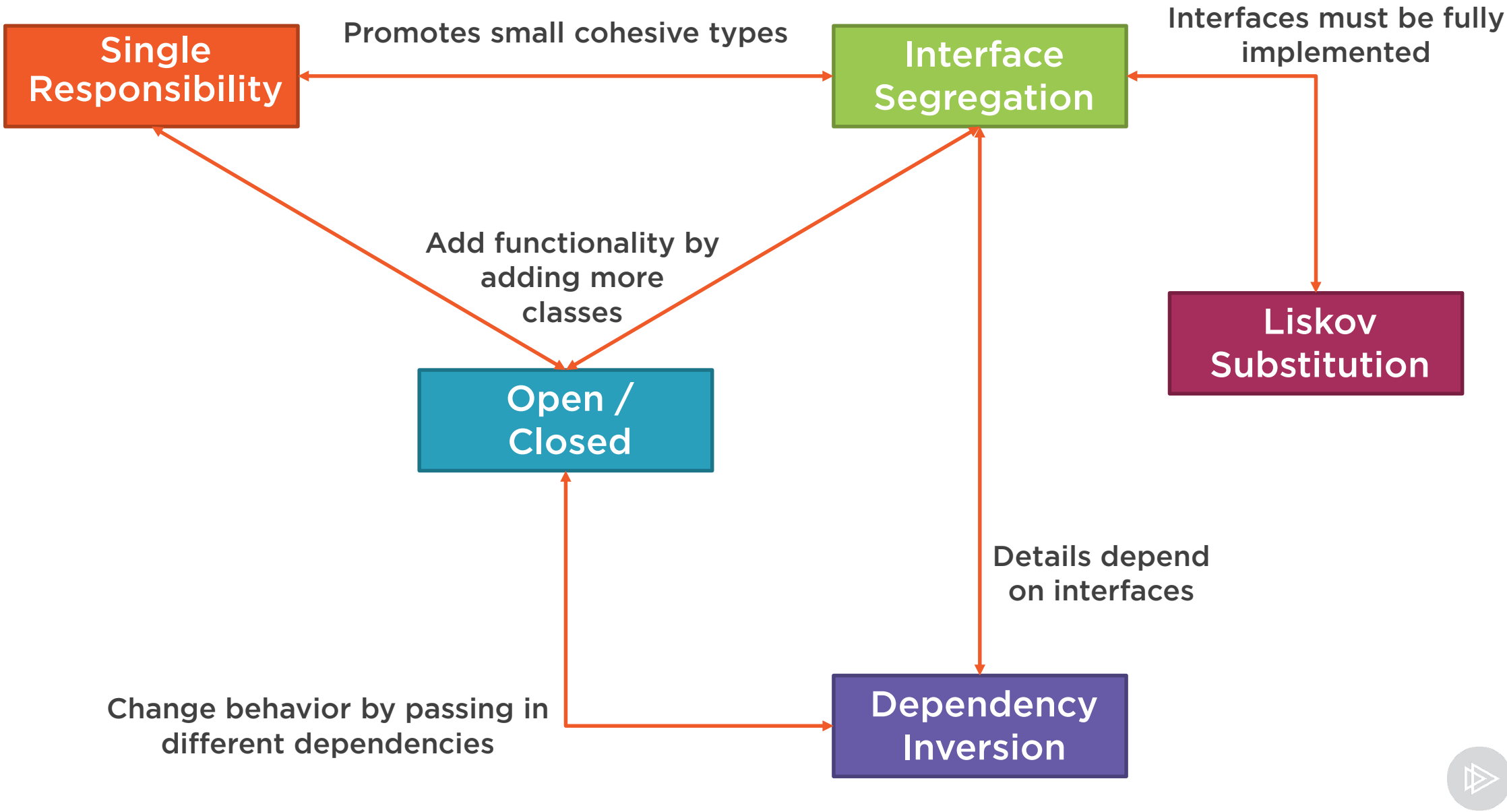
- Most classes should depend on abstractions, not implementation details
- Abstractions shouldn't leak details
- Classes should be explicit about their dependencies
- Clients should inject dependencies when they create other classes
- Structure your solutions to leverage dependency inversion

Course Summary

Review principles – see next slide

Code Samples

- https://github.com/ardalis/solidsample


Single Responsibility
Promotes small cohesive types
Interface Segregation
Interfaces must be fully implemented
Add functionality by adding more classes
Liskov Substitution
Open / Closed
Details depend on interfaces
Change behavior by passing in different dependencies
Dependency Inversion

SOLID Principles for C# Developers

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