Fundamentals:
The Single Responsibility Principle

Outline

- SRP Defined
- The Problem
- An Example
- Refactoring to Apply SRP
- Related Fundamentals

SRP: The Single Responsibility Principle

The Single Responsibility Principle states that every object should have a single responsibility, and that responsibility should be entirely encapsulated by the class.

Wikipedia

There should never be more than one reason for a class to change.

Robert C. “Uncle Bob” Martin

SINGLE RESPONSIBILITY PRINCIPLE

Just Because You Can, Doesn't Mean You Should

Cohesion and Coupling

- Cohesion: how strongly-related and focused are the various responsibilities of a module
- Coupling: the degree to which each program module relies on each one of the other modules

Strive for low coupling and high cohesion!

Responsibilities are Axes of Change

- Requirements changes typically map to responsibilities
- More responsibilities == More likelihood of change
- Having multiple responsibilities within a class couples together these responsibilities
- *The more classes a change affects, the more likely the change will introduce errors.*

Demo

The Problem With Too Many Responsibilities

The Problem

- **Cash Transactions Don't Need Credit Card Processing**
- **Point of Sale Transactions Don't Need Inventory Reservations**
  - Store inventory is updated separately in our system
- **Point of Sale Transactions Don't Need Email Notifications**
  - The customer doesn't provide an email
  - The customer knows immediately that the order was a success
- **Any change to notifications, credit card processing, or inventory management will affect Order as well as the Web and Point of Sale implementations of Order!**

Demo

Refactoring to a Better Design

What is a Responsibility?

- “a reason to change”
- A difference in usage scenarios from the client’s perspective
- Multiple small interfaces (follow ISP) can help to achieve SRP

Summary

- **Following SRP leads to lower coupling and higher cohesion**
- **Many small classes with distinct responsibilities result in a more flexible design**
- **Related Fundamentals:**
  - Open/Closed Principle
  - Interface Segregation Principle
  - Separation of Concerns
-

Credits

Fundamentals:
The Open / Closed Principle

Outline

- OCP Defined
- The Problem
- An Example
- Refactoring to Apply OCP
- Related Fundamentals

OCP: The Open/Closed Principle

The Open / Closed Principle states that software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification.

Wikipedia

OPEN CLOSED PRINCIPLE

Open Chest Surgery Is Not Needed When Putting On A Coat

The Open / Closed Principle

Open to Extension

- New behavior can be added in the future

Closed to Modification

- Changes to source or binary code are not required

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

Change behavior without changing code?

Rely on abstractions

No limit to variety of implementations of each abstraction

In .NET, abstractions include:

- Interfaces
- Abstract Base Classes

In procedural code, some level of OCP can be achieved via parameters

The Problem

- **Adding new rules require changes to the calculator every time**
- **Each change can introduce bugs and requires re-testing, etc.**
- **We want to avoid introducing changes that *cascade* through many modules in our application**
- **Writing new classes is less likely to introduce problems**
  - Nothing depends on new classes (yet)
  - New classes have no legacy coupling to make them hard to design or test

Three Approaches to Achieve OCP

- **Parameters (Procedural Programming)**
  - Allow client to control behavior specifics via a parameter
  - Combined with delegates/lambda, can be very powerful approach
- **Inheritance / Template Method Pattern**
  - Child types override behavior of a base class (or interface)
- **Composition / Strategy Pattern**
  - Client code depends on abstraction
  - Provides a “plug in” model
  - Implementations utilize Inheritance; Client utilizes Composition

When do we apply OCP?

- **Experience Tells You**

  If you know from your own experience in the problem domain that a particular class of change is likely to recur, you can apply OCP up front in your design

Otherwise – "Fool me once, shame on you; fool me twice, shame on me"

- **Don't apply OCP at first**
- **If the module changes once, accept it.**
- **If it changes a second time, refactor to achieve OCP**

Remember *TANSTAAFL*

- ***There Ain't No Such Thing As A Free Lunch***
- **OCP adds complexity to design**
- **No design can be closed against all changes**

Summary

- **Conformance to OCP yields flexibility, reusability, and maintainability**
- **Know which changes to guard against, and resist premature abstraction**
- **Related Fundamentals:**
  - Single Responsibility Principle
  - Strategy Pattern
  - Template Method Pattern
-

Fundamentals:
The Liskov Substitution Principle

Outline

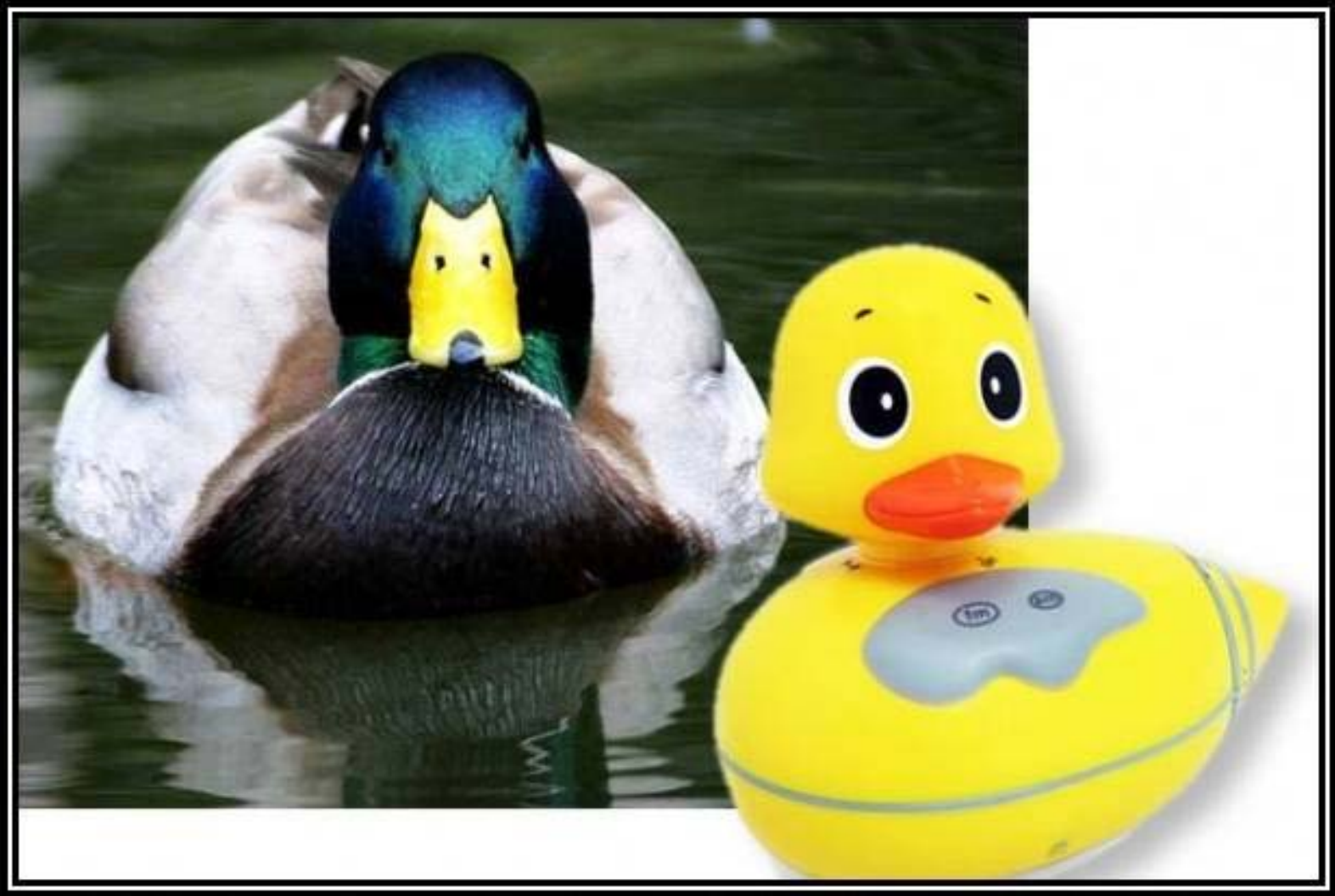
- LSP Defined
- The Problem
- An Example
- Refactoring to Apply LSP
- Related Fundamentals

LSP: The Liskov Substitution Principle

The Liskov Substitution Principle states that Subtypes must be substitutable for their base types.

Agile Principles, Patterns, and Practices in C#

Named for Barbara Liskov, who first described the principle in 1988.



LISKOV SUBSTITUTION PRINCIPLE

If It Looks Like A Duck, Quacks Like A Duck, But Needs Batteries - You Probably Have The Wrong Abstraction

Substitutability

Child classes must not:

1) Remove base class behavior
2) Violate base class invariants

And in general must not require calling code to know they are different from their base type.

Inheritance and the IS-A Relationship

Naïve OOP teaches use of IS-A to describe child classes' relationship to base classes

LSP suggests that IS-A should be replaced with

IS-SUBSTITUTABLE-FOR

Invariants

- Consist of reasonable assumptions of behavior by clients
- Can be expressed as preconditions and postconditions for methods
- Frequently, unit tests are used to specify expected behavior of a method or class
- *Design By Contract* is a technique that makes defining these pre- and post-conditions explicit within code itself.
- To follow LSP, derived classes must not violate any constraints defined (or assumed by clients) on the base classes

The Problem

- Non-substitutable code breaks polymorphism
- Client code expects child classes to work in place of their base classes
- “Fixing” substitutability problems by adding if-then or switch statements quickly becomes a maintenance nightmare (and violates OCP)

LSP Violation “Smells”

```
foreach (var emp in Employees)
{
  if(emp is Manager)
  {
      _printer.PrintManager(emp as Manager);
  }
  else
  {
    _printer.PrintEmployee(emp);
  }
}
```

LSP Violation "Smells"

```
public abstract class Base
{
     public abstract void Method1();
     public abstract void Method2();
}

public class Child : Base
{
    public override void Method1()
    {
      throw new NotImplementedException();
    }
    public override void Method2()
    {
      // do stuff
    }
}
```

Follow ISP!

Use small interfaces so you don't require classes to implement more than they need!

When do we fix LSP?

- If you notice obvious smells like those shown
- If you find yourself being bitten by the OCP violations LSP invariably causes

LSP Tips

- **"Tell, Don't Ask"**
  - Don't interrogate objects for their internals – move behavior to the object
  - Tell the object what you want it to do

- **Consider Refactoring to a new Base Class**
  - Given two classes that share a lot of behavior but are not substitutable…
  - Create a third class that both can derive from
  - Ensure substitutability is retained between each class and the new base

Summary

- **Conformance to LSP allows for proper use of polymorphism and produces more maintainable code**
- **Remember IS-SUBSTITUTABLE-FOR instead of IS-A**
- **Related Fundamentals:**
  - Polymorphism
  - Inheritance
  - Interface Segregation Principle
  - Open / Closed Principle
- **Recommended Reading:**
  - Agile Principles, Patterns, and Practices by Robert C. Martin and Micah Martin [http://amzn.to/agilepppcsharp]

Fundamentals:
The Interface Segregation Principle

Outline

- ISP Defined
- The Problem
- An Example
- Refactoring to Apply ISP
- Related Fundamentals

ISP: The Interface Segregation Principle

The Interface Segregation Principle states that Clients should not be forced to depend on methods they do not use.

Agile Principles, Patterns, and Practices in C#

Corollary:

Prefer small, cohesive interfaces to “fat” interfaces

INTERFACE SEGREGATION PRINCIPLE

You Want Me To Plug This In, Where?

What's an Interface?

Interface keyword/type

```
public interface IDoSomething { … }
```

Public interface of a class

```
public class SomeClass { … }
```

What does the *client* see and use?

The Problem

Client Class (Login Control) Needs This:

```
private void AuthenticateUsingMembershipProvider(AuthenticateEventArgs e)
{
    e.Authenticated = LoginUtil.GetProvider(this.MembershipProvider).ValidateUser(this.UserNameInternal, this.PasswordInternal);
}
```

Forced to use this

```
public class CustomMembershipProvider : MembershipProvider
{
    public override MembershipUser CreateUser(string username, string password, string
bershipCreateStatus status)...
    public override bool ChangePasswordQuestionAndAnswer(string username, string passw
    public override string GetPassword(string username, string answer)...
    public override bool ChangePassword(string username, string oldPassword, string ne
    public override string ResetPassword(string username, string answer)...
    public override void UpdateUser(MembershipUser user)...
    public override bool ValidateUser(string username, string password)...
    public override bool UnlockUser(string userName)...
    public override MembershipUser GetUser(object providerUserKey, bool userIsOnline)
    public override MembershipUser GetUser(string username, bool userIsOnline)...
    public override string GetUserNameByEmail(string email)...
    public override bool DeleteUser(string username, bool deleteAllRelatedData)...
    public override MembershipUserCollection GetAllUsers(int pageIndex, int pageSize,
    public override int GetNumberOfUsersOnline()...
    public override MembershipUserCollection FindUsersByName(string usernameToMatch,
    public override MembershipUserCollection FindUsersByEmail(string emailToMatch, int
    public override bool EnablePasswordRetrieval...
    public override bool EnablePasswordReset...
    public override bool RequiresQuestionAndAnswer...
    public override string ApplicationName { get; set; }
    public override int MaxInvalidPasswordAttempts...
    public override int PasswordAttemptWindow...
    public override bool RequiresUniqueEmail...
    public override MembershipPasswordFormat PasswordFormat...
    public override int MinRequiredPasswordLength...
    public override int MinRequiredNonAlphanumericCharacters...
    public override string PasswordStrengthRegularExpression...
}
```

The Problem

- AboutPage simply needs ApplicationName and AuthorName

- Forced to deal with huge ConfigurationSettings class
- Forced to deal with actual configuration files

Interface Segregation violations result in classes that depend on things they do not need, increasing coupling and reducing flexibility and maintainability

ISP Smells

- Unimplemented interface methods:

```
public override string ResetPassword(
  string username, string answer)
 {
      throw new NotImplementedException();
 }
```

Remember these violate Liskov Substitution Principle!

ISP Smells

- Client references a class but only uses small portion of it

When do we fix ISP?

- **Once there is pain**
  - If there is no pain, there's no problem to address.
- **If you find yourself depending on a "fat" interface you own**
  - Create a smaller interface with just what you need
  - Have the fat interface implement your new interface
  - Reference the new interface with your code
- **If you find "fat" interfaces are problematic but you do not own them**
  - Create a smaller interface with just what you need
  - Implement this interface using an Adapter that implements the full interface

ISP Tips

- **Keep interfaces small, cohesive, and focused**
- **Whenever possible, let the client define the interface**
- **Whenever possible, package the interface with the client**
  - Alternately, package in a third assembly client and implementation both depend upon
    - Last resort: Package interfaces with their implementation

Summary

- **Don’t force client code to depend on things it doesn’t need**
- **Keep interfaces lean and focused**
- **Refactor large interfaces so they inherit smaller interfaces**
- **Related Fundamentals:**
  - Polymorphism
  - Inheritance
  - Liskov Substitution Principle
  - Façade Pattern
- **Recommended Reading:**

Fundamentals:
The Dependency Inversion Principle
Part 1

Outline

- DIP Defined
- The Problem
- An Example
- Refactoring to Apply DIP
- Related Fundamentals

DIP: The 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.

Agile Principles, Patterns, and Practices in C#

DEPENDENCY INVERSION PRINCIPLE

Would You Solder A Lamp Directly To The Electrical Wiring In A Wall?

What are dependencies?

- Framework
- Third Party Libraries
- Database
- File System
- Email
- Web Services
- System Resources (Clock)
- Configuration
- The `new` Keyword
- Static methods
- Thread.Sleep
- Random

Traditional Programming and Dependencies

- **High Level Modules Call Low Level Modules**

- **User Interface depends on**
  - Business Logic depends on
    - Infrastructure
    - Utility
    - Data Access

- **Static methods are used for convenience or as Façade layers**
- **Class instantiation / Call stack logic is scattered through all modules**
  - Violation of Single Responsibility Principle

Class Dependencies: Be Honest!

- **Class constructors should require any dependencies the class needs**
- **Classes whose constructors make this clear have *explicit* dependencies**
- **Classes that do not have *implicit*, hidden dependencies**

```
public class HelloWorldHidden
{
    public string Hello(string name)
    {
        if (DateTime.Now.Hour < 12) return "Good morning, " + name;
        if (DateTime.Now.Hour < 18) return "Good afternoon, " + name;
        return "Good evening, " + name;
    }
}
```

Classes Should Declare What They Need

```
public class HelloWorldExplicit
{
   private readonly DateTime _timeOfGreeting;

   public HelloWorldExplicit(DateTime timeOfGreeting)
   {
      _timeOfGreeting = timeOfGreeting;
   }

   public string Hello(string name)
   {
      if (_timeOfGreeting.Hour < 12) return "Good morning, " + name;
      if (_timeOfGreeting.Hour < 18) return "Good afternoon, " + name;
      return "Good evening, " + name;
   }
}
```

The Problem

- **Order has hidden dependencies:**
  - MailMessage
  - SmtpClient
  - InventorySystem
  - PaymentGateway
  - Logger
  - DateTime.Now

- **Result**
  - Tight coupling
  - No way to change implementation details (OCP violation)
  - Difficult to test

Dependency Injection

- ***Dependency Injection* is a technique that is used to allow calling code to *inject* the dependencies a class needs when it is instantiated.**

- **The Hollywood Principle**
  - "Don't call us; we'll call you"

- **Three Primary Techniques**
  - Constructor Injection
  - Property Injection
  - Parameter Injection

- **Other methods exist as well**

Constructor Injection

- **Dependencies are passed in via constructor**

- **Pros**
  - Classes self-document what they need to perform their work
  - Works well with or without a container
  - Classes are always in a valid state once constructed

- **Cons**
  - Constructors can have many parameters/dependencies (design smell)
  - Some features (e.g. Serialization) may require a *default constructor*
  - Some methods in the class may not require things other methods require (design smell)

Property Injection

- **Dependencies are passed in via a property**
  - Also known as “setter injection”

- **Pros**
  - Dependency can be changed at any time during object lifetime
  - Very flexible

- **Cons**
  - Objects may be in an invalid state between construction and setting of dependencies via setters
  - Less intuitive

Parameter Injection

- **Dependencies are passed in via a method parameter**

- **Pros**
  - Most granular
  - Very flexible
  - Requires no change to rest of class

- **Cons**
  - Breaks method signature
  - Can result in many parameters (design smell)

- ***Consider if only one method has the dependency, otherwise prefer constructor injection***

Refactoring

- Extract Dependencies into Interfaces
- Inject implementations of interfaces into Order
- Reduce Order's responsibilities (apply SRP)

DIP Smells

- Use of new keyword

```
foreach(var item in cart.Items)
{
  try
  {
    var inventorySystem = new InventorySystem();
    inventorySystem.Reserve(item.Sku, item.Quantity);
  }
}
```

DIP Smells

- Use of static methods/properties

```
message.Subject = "Your order placed on " +
   DateTime.Now.ToString();
```

Or

```
DataAccess.SaveCustomer(myCustomer);
```

Where do we instantiate objects?

- **Applying Dependency Injection typically results in many interfaces that eventually need to be instantiated *somewhere… but where?***

- **Default Constructor**
  - You can provide a default constructor that news up the instances you expect to typically need in your application
  - Referred to as “poor man’s dependency injection” or “poor man’s IoC”

- **Main**
  - You can manually instantiate whatever is needed in your application’s startup routine or main() method

- **IoC Container**
  - Use an “Inversion of Control” Container

IoC Containers

- **Responsible for object graph instantiation**
- **Initiated at application startup via code or configuration**
- **Managed interfaces and the implementation to be used are *Registered* with the container**
- **Dependencies on interfaces are *Resolved* at application startup or runtime**

- **Examples of IoC Containers for .NET**
  - Microsoft Unity
  - StructureMap
  - Ninject
  - Windsor
  - Funq / Munq

Summary

- **Depend on abstractions.**
- **Don't force high-level modules to depend on low-level modules through direct instantiation or static method calls**
- **Declare class dependencies explicitly in their constructors**
- **Inject dependencies via constructor, property, or parameter injection**
- **Related Fundamentals:**
  - Single Responsibility Principle
  - Interface Segregation Principle
  - Façade Pattern
  - Inversion of Control Containers
-