Are You a SOLID Coder?

Principles of Software Design

Steve Green

steve.p.green@outlook.com

@stephenpgreen

/stevepgreen

Who is this Guy?

"If it's work, we try to do less.

If it's art, we try to do more."

- Seth Godin

My professional experience

I consider code an art form, and have worked over the last 15 years at some of the finest technology companies in Kansas City, trying to master it as a craft.

PlattForm

OAKWOOD

jack henry & ASSOCIATES INC.

Clickfarm DAMN GOOD DIGITAL

avanade Results Realized

VML

What is Software Design?

To understand what software design "is", it is important to contrast it with software analysis.

Software Analysis

The process and effort used to determine "**what**" should be built is software analysis. This is generally a single activity conducted before the build cycles start.

Software Design

The constant decisions surrounding "**how**" to build an application are its design. These are made both before the build starts and continue through delivery.

Design Is **Not** A One-Time Activity

* Design is an evolutionary process throughout the construction of an application.

Why Is Good Design Important?

Good Design Is Easily Consumed

"Any fool can write code a computer can understand. Good programmers write code that humans can understand." – Fowler

1. **Delivery**: maintaining good design allows us to deliver solutions faster given it is easier to work with the code.

2. **Change**: a solution that preserves good design is more flexible which makes it easier for it to manage change.

3. **Complexity**: well designed solutions strive for simplicity, allowing developers to handle complex challenges.

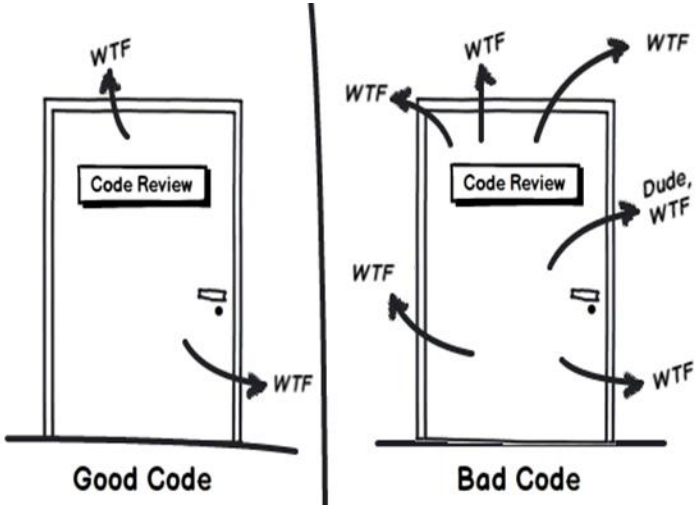
Bad design often leads to frustrated, depressed, and de-motivated teams.

How Does Bad Design Happen?

Bad design builds quietly over time until it becomes unmanageable.

* No one sets out to design software poorly; rather it happens slowly over time. Consider this messy kitchen. The designer didn't set out to create this disaster. Instead, it happened naturally over time as the result of small bad decisions that incrementally destroyed this space.

Identifying Good Design



Metrics are Key to Early Detection

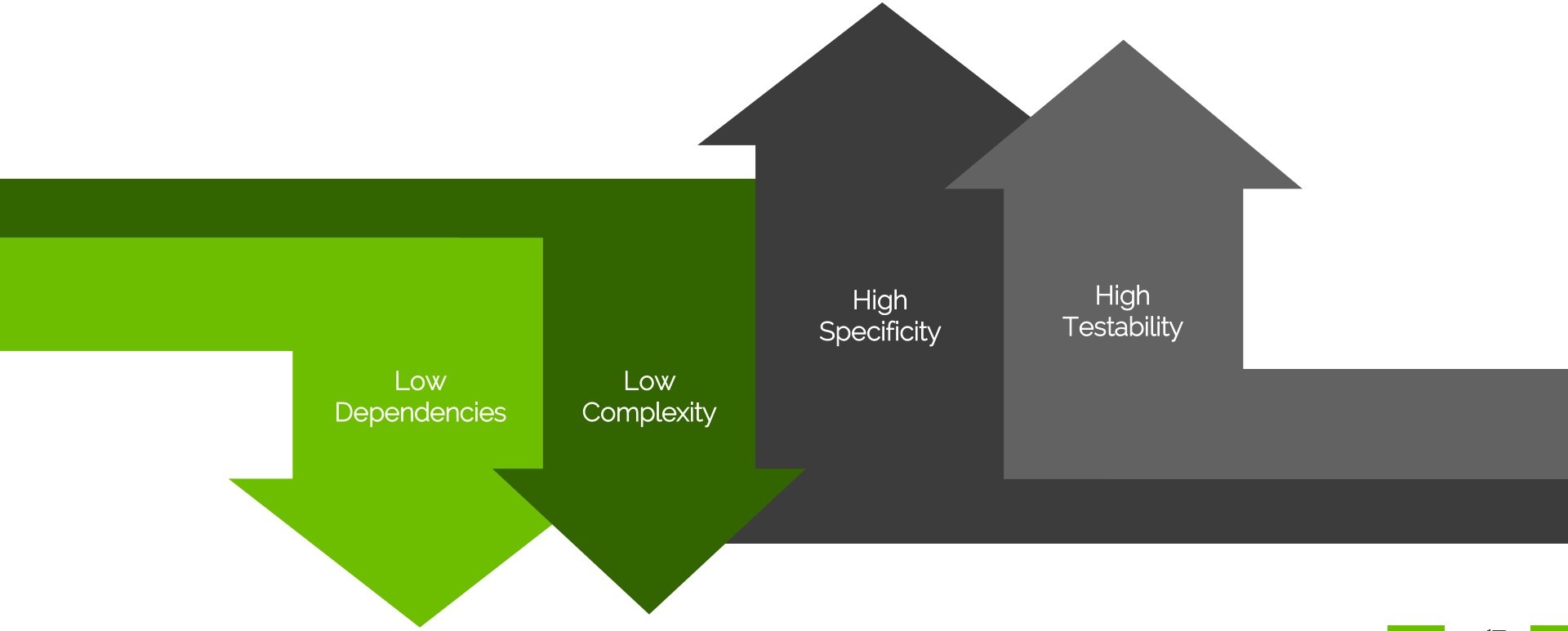
Many teams have difficulty developing an early metric for good design, instead relying mostly on the "eye" test during code reviews.

A Better Design Criteria

Identifying good design is easier if we use a rubric based on common code smells.

- **Rigidity** – How easy is the design to change?
- **Fragility** – Is the design easy to break?
- **Immobility** – Can the design be re-used?
- **Viscosity** – Is it hard to do the right thing?

Characteristics of Good Design



Good Design Is Everyone's Responsibility

The Boy Scout Rule

"Always leave the code you're editing a little better than you found it." – Bob Martin

* When teams accept the responsibility for the design, the system as a whole improves over time. As the software evolves, developers take an active approach to enhancing the system. Much like picking up trash on the side of the road, if everyone does just a little bit, the highway remains clean for everyone.

Being a SOLID coder means you have the skills necessary to carry out this promise.

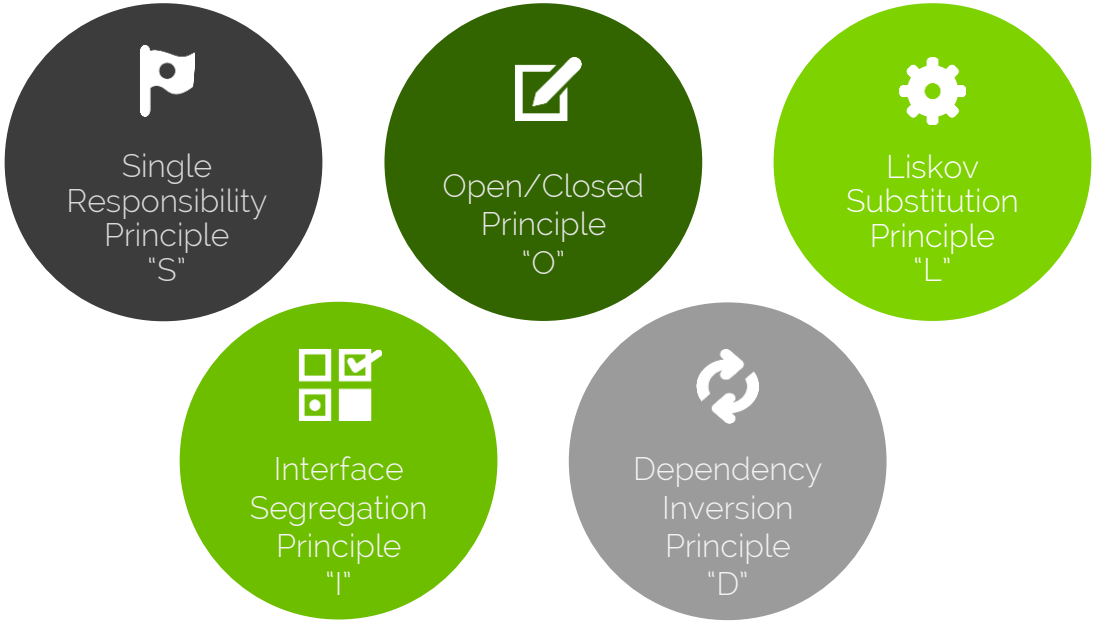
Meet Uncle Bob Martin

Uncle Bob's SOLID Principles of Design

Bob Martin provided us with well accepted guidelines to help deliver solutions based on best practices. These principles are referred to as the SOLID principles of object oriented design.

The SOLID Principles of Software Design

SOLID is an acronym that can help us remember the concepts of good design. In a single word, we can reference some of the most important patterns and practices to use when building applications.



What is the Single Responsibility Principle?

You should focus on doing one thing really well.

Avoid Trying to Do Too Much

Code that tries to do too much becomes hard to understand, maintain, and use.

Single Responsibility Principle

A unit of code (whether it is a class, method or function) should try to do **ONLY** one thing.

If the code adheres to this idea, then it is only susceptible to a single change in the software's specification, but no more.

What is a Responsibility?

A Responsibility Is a Reason for Change

A responsibility is an axis of change. When viewed from the perspective of the client, it generally reflects a difference in the possible use of the software.

1. Software requirements map to responsibilities.
2. The more responsibilities, the more likely the need for change.
3. Grouping multiple responsibilities tightly couples the code.
4. The more classes a change effects, the higher chance it will produce errors.

A Quick Example: The Modem

The Modem Interface

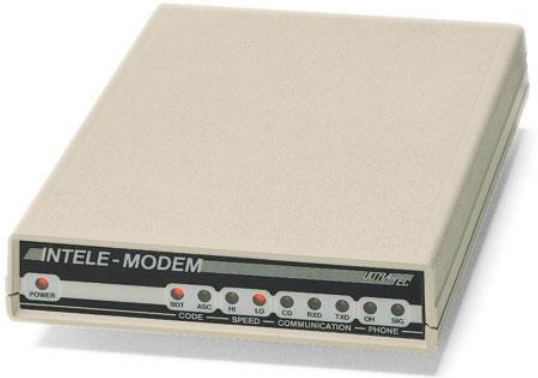
Below we design a simple interface that will represent a modem in our project.

```
public interface IModem
{
    0 references
    bool Dial(string number);

    0 references
    void Hangup();

    0 references
    void Send(char data);

    0 references
    char Recieve();
}
```



Let's Consider a Modem

A modem has two main responsibilities, establish connections and transmit data.

A Quick Example: The Modem

Let's Refactor Towards the SRP

Once we refactor our example to adhere to the single responsibility principle, we see the modem represented as multiple discrete interfaces.

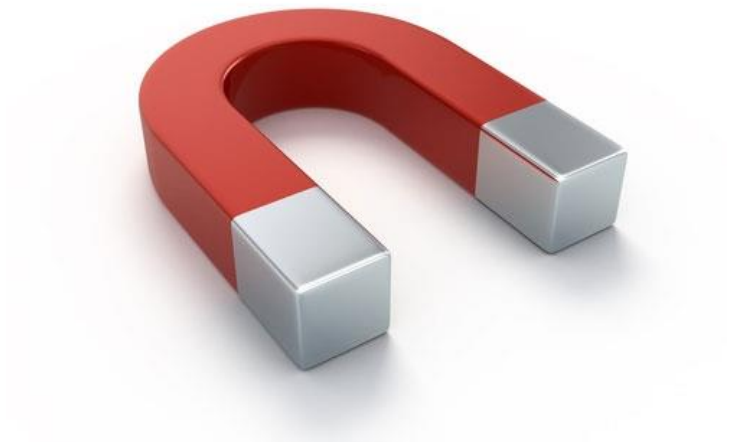
```
public interface IModemConnection
{
    0 references
    bool Dial(string number);

    0 references
    void Hangup();
}
```

```
public interface IModemChannel
{
    0 references
    void Send(char data);

    0 references
    char Recieve();
}
```


Maximizing Class Cohesion



Magnet Classes Attract The Lazy

A class that has low cohesion acts like a magnet attracting lazy developers. These classes quickly become "dumping grounds".

Methods Must Be Strongly-Related

Maximizing cohesion ensures that all the code within the class works together to accomplish a single task.

- High cohesion enhances the readability of your code.
- Classes that expose singular functionality are easier to reuse.
- Code that is well organized avoids attracting pollution.

Why is Cohesion Important?

Detroit's Michigan Theater

This theater used to be a crown jewel for the city of Detroit. Now, it has been converted into a downtown parking lot.

This is extremely odd. These things clearly don't go together. In fact, using a theater this way just leads to confusion and more questions.

Identifying Low Cohesion

Features of Low Cohesion

Below are some traits of low cohesion within your classes:

- Watch out for methods that don't interact with the rest of the class.
- Check your fields and make sure they are used by more than one method.
- Monitor your source control to find classes that have a higher frequency of change.

The Name Matters

Remember that cohesion is about logical groupings. If you're having trouble naming your class, chances are you have low cohesion.

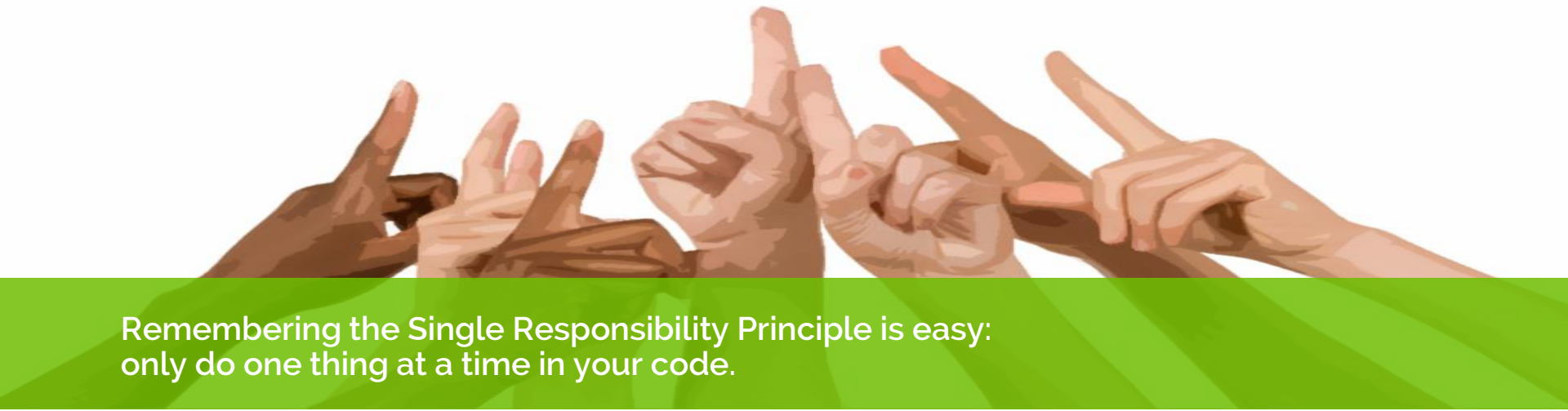
Minimizing Code Coupling

Loosely Coupled Code

Classes that maintain the single responsibility principle naturally become loosely coupled.

When code is tightly coupled, any modification becomes a risky proposition, as the change has a high chance of interfering with the other responsibilities of the class.

Summarizing Single Responsibility Principle



Remembering the Single Responsibility Principle is easy: only do one thing at a time in your code.

* If you follow this principle, you will notice that you have very few "if", "switch" or other branching statements because you have isolated and segregated those other functions into different classes. Those classes have their own context and understand their unique behaviors.

What is the Open/Closed Principle?

New behaviors should be added by extending a class.

Avoid Getting Under the Hood

Classes that require a change to internally contained code for new behavior is inflexible.

Open/Closed Principle

Software entities should be open for extension, but closed to modification.

We shouldn't need to dig into the internals of the software in order to add or change its behavior. Instead, we should add behavior by extension.

Why Is the Open/Closed Principle Important?

Change Management

The open/closed principle helps to ensure that the risks that code edits endure are properly mitigated. This helps us build highly flexible systems.

1. Calling code is isolated from edits.
2. Sometimes the dependent libraries cannot be modified.
3. New classes are less likely to produce errors with the existing software.
4. Reduces the number of necessary tests for any change.

The Open/Closed Principle Using Parameters

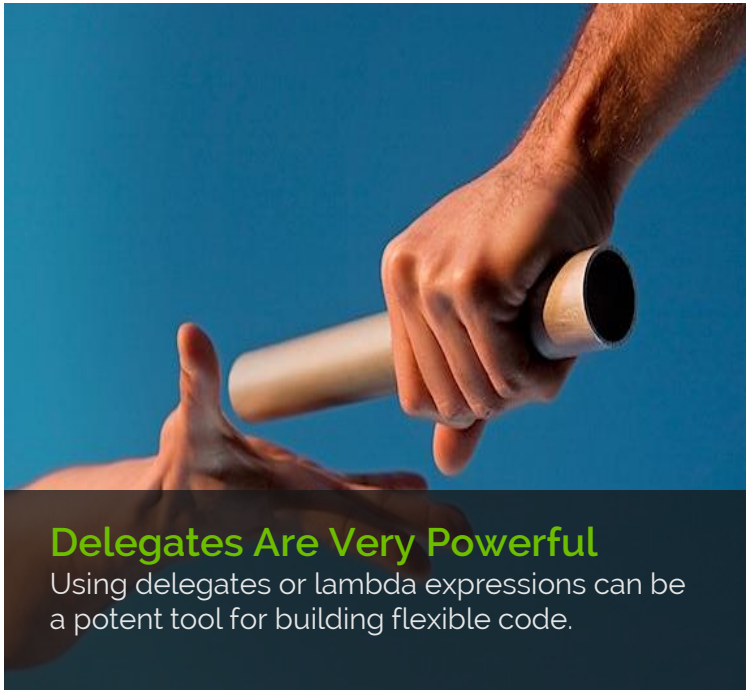
Allow clients to control behavior by passing contextual parameters to a method.

Such parameters generally contain some type of state information that allows a method to behave differently.

Parameters Are Explicit Tools

Extending code using parameters can be a powerful tool in procedural languages.

Extending the Modem Using Parameters



The Modem Channel

Here we have refactored the modem channel interface to demonstrate the open/closed principle using parameters.

Note that new behaviors for any channel instance would not require modifications to the internal implementation.

```
public interface IModemChannel
{
    0 references
    void Send(char data, Action<char> transmitter);

    0 references
    char Receive(Func<char> reciever);
}
```


The Template Method Pattern

Deferred Code Execution

The template method behavioral pattern allows some pieces of an algorithm to be deferred to a subclass.

- A skeleton of the algorithm is defined in a base or root class.
- Some implementations can contain default behavior in template methods.
- Subclasses then override the template with different behavior without modifying the root.

Overriding Inherited Code

The most common form of extension is class inheritance using object oriented mechanisms.

The Strategy Pattern

Extension With Pluggable Modules

The strategy pattern forces clients to depend upon abstractions that can be swapped in or out without effecting the caller.

* When implementing a strategy pattern, it is common to see behaviors exposed to clients through an abstraction such as an interface or base class. The calling code can then use various mechanisms to plug the concrete module in at run time, including: dependency injection, composition, etc.

The Strategy Pattern: Extending the Modem


```
public class Modem
{
    private readonly IModemChannel _channel;

    private readonly IModemConnection _connection;

    0 references
    public Modem(IModemConnection connection, IModemChannel channel)
    {
        if(null == connection)
            throw new ArgumentNullException("connection");

        if(null == channel)
            throw new ArgumentNullException("channel");

        _channel = channel;
        _connection = connection;
    }

    0 references
    public IModemChannel Channel { get { return _channel; } }

    0 references
    public IModemConnection Connection { get { return _connection; } }
}
```


When Should You Apply the Open/Closed Principle?

New problem domains

When dealing with a new problem domain start with simplicity and work towards complexity.

- Begin your initial implementation without using the open/closed principle.
- If a change happens, just accept it and modify the code.
- If a change happens again, then refactor utilizing the open/closed principle.

Don't Fool Me Twice

Apply the "Fool me once shame on you, fool me twice, shame on me" rule.

Avoid Complexity When Possible

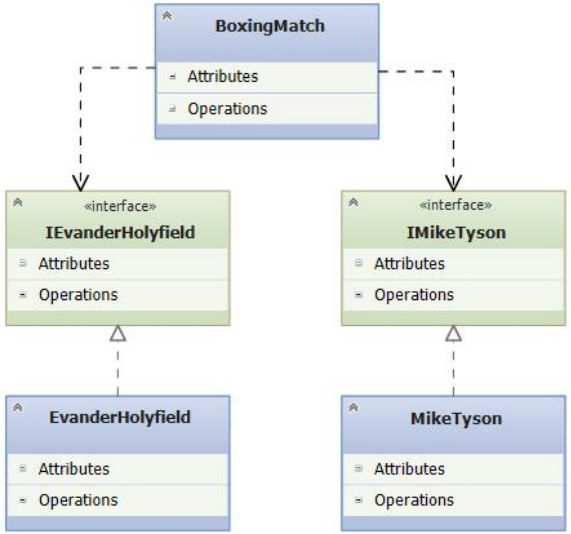
Remember that there is no such thing as a free lunch.

- The open/closed principle adds the cost of complexity to the design. So make sure that the complexity is justified and is providing a benefit to the system.

Be Careful Not To Overuse Abstractions

The boxing match model

Obviously, the below model is silly. However, solutions like this are quite common as developers feel the need to provide an interface for every object.



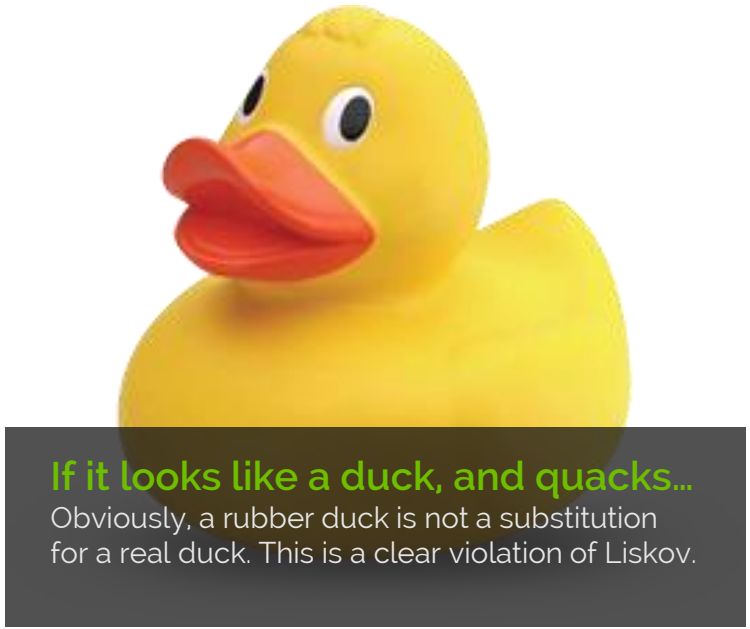
Summarizing the Open/Closed Principle

Remembering the Open/Closed Principle is easy:
guard against change by forcing modifications as extensions.

* Conformance to the open/closed principle adds flexibility, reusability, and maintainability to your software. The key is to decide what changes to guard against while resisting premature abstraction.

What is the Liskov Substitution Principle?

Proper abstractions should be substitutable in all cases.



If it looks like a duck, and quacks...

Obviously, a rubber duck is not a substitution for a real duck. This is a clear violation of Liskov.

Liskov Substitution Principle

Subtypes must be substitutable for their base types in all cases.

Calling code should not need to be aware that it is consuming a subtype rather than the base type. Such transactions must remain completely transparent.

What Does Substitutable Mean?

Integrity as the Metric

The Liskov Substitution Principle helps ensure the integrity of the system by ensuring abstractions behave as expected.

If there is a drastic change in behavior for a subclass, then the system is said to be volatile.

1. Child subtypes should never remove base class behavior.
2. Derived classes must not violate any base class invariants.
3. Child classes should not throw new types of exceptions.
4. Subtypes can freely add new methods that extend the root class.

Semantics Matter

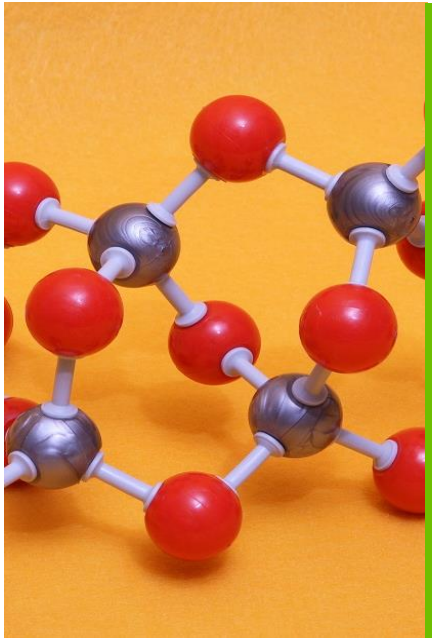
Be Careful with Naïve Language

Classic object oriented techniques might describe a subclass as an "IS-A" relationship. Instead, thing of it as an "IS-SUBSTITUTABLE-FOR".

* Event though we all learned object oriented programming by thinking in concrete terms such as "a car is a vehicle" or "a square is a rectangle", this is dangerous. Instead, it is safer to consider if a "noun" can be substituted for another "noun". Can a square be substituted for a rectangle in all cases?

What is an Invariant?

1. Assumptions about a class are expressed as pre/post-conditions.
2. Conditions are rarely expressed in code, but instead unit tests.
3. Derived classes must not violate any base constraints.



It's All About the Model

Preserving the invariant behavior of the base class is really about ensuring the integrity of the models our classes represent.

This consists of the reasonable assumptions that clients make when calling the base class.

A Quick Example: The Square

Is a Square a Rectangle?

Obviously it is; however, is a square a suitable replacement for a rectangle? Maybe not in all cases.

Let's start with a rectangle

A rectangle has two simple properties: height and width. By its definition, a rectangle can have a height that differs from its width. This is depicted in the code below:

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

    4 references
    public virtual int Width { get; set; }
}
```


A Quick Example: The Square

Is a Square a Rectangle?

Obviously it is; however, is a square a suitable replacement for a rectangle? Maybe not in all cases.

Now consider a square

A square changes the basic constraints expected by a rectangle: height and width must always be equal:

```
public class Square : Rectangle
{
    4 references
    public override int Height
    {
        get { return base.Height; }
        set
        {
            base.Height = value;
            Width = value;
        }
    }

    4 references
    public override int Width
    {
        get { return base.Width; }
        set
        {
            base.Width = value;
            Height = value;
        }
    }
}
```


Identifying and Resolving Liskov Violations Can Be Tricky

Discovering Liskov issues can be difficult. However, once an issue is detected, here is some practical advice to resolve the issue.

Tell, Don't Ask

Do not interrogate objects about their internal state. Instead, tell them the behavior to run.

Use New Base Types

When two objects seem related but are not substitutable, consider adding a new root class for both.

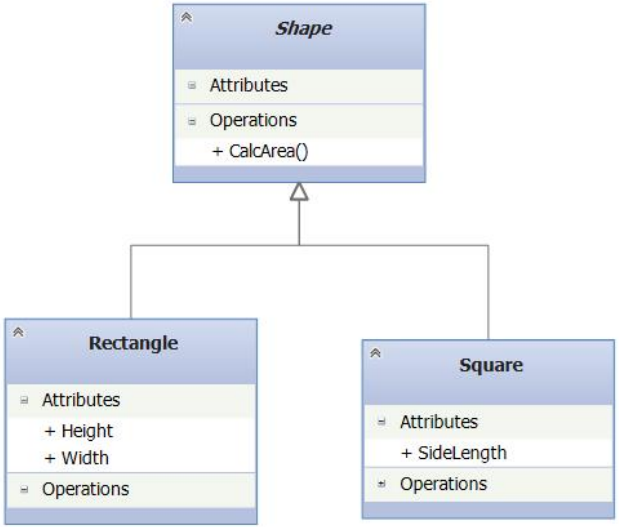
Tests Are Your Friend

Remember, testing your objects for post-conditions can be the easiest way to find issues.

A Quick Example: The Square

A Better Model

A better model is to introduce a new base class that both rectangle and square extend. In this way, both are always substitutable for a "shape".



Thinking About Behavior

Sometimes it helps to think about behavior instead of state, and abstract from there. An area calculation is a primary use of a shape.

Summarizing the Liskov Substitution Principle

Remembering the Liskov Substitution Principle is easy:
ask if an "object" can be substituted for another "object" before inheriting.

* Conformance to the Liskov Substitution Principle ensures proper use of polymorphism that maintains the integrity to your software model. This helps produce more maintainable code and reliable tests.

What is the Interface Segregation Principle?

Avoid fat interfaces that have unused members.

Grouping a Bunch of Interfaces

Using a bunch of discrete interfaces makes our code more explicit and easier to understand.

Interface Segregation Principle

Clients should not be forced to depend on methods they do not use. Instead, they should consume clean cohesive interfaces.

This prevents client objects from depending on members they don't use.

Why Does It Matter?

Giant Interfaces Are Extra Baggage

The Interface Segregation Principle ensures that our code is lean and nimble. When a dependency has too many unused members, it just creates extra baggage we have to deal with.

1. Interfaces with too many members are less reusable.
2. Useless members leads to inadvertent coupling.
3. Additional members leads to increased complexity.
4. Multiple small interfaces ensure single responsibilities.

How to Identify?

Look for Unused Code

To find interface segregation violations just look for unused code. Once found, isolate the unused blocks in separate logical interfaces.

Some Simple Techniques

Identifying good design is easier if we use a rubric based on common code smells.

- **Reference Utilization**
  If a client references a class, but only uses a very small subset of its functionality.

- **Not Implemented Exceptions**
  If a client only implements a few an interface and throws errors when a client calls non-implemented members.

A Quick Example: A Bird

Consider a Simple Bird Model

We create an interface that expresses what we believe are the behaviors of a bird.

```
public interface IBird
{
    0 references
    void Eat();

    0 references
    void Fly();

    0 references
    void Walk();
}
```

A Quick Example: A Bird

A Better Solution

Below, we've refactored our solution to accommodate the different types of birds using the Interface Segregation Principle.

```
public interface IBird
{
    0 references
    void Eat();

    0 references
    void Walk();
}

public interface IFlyingBird : IBird
{
    0 references
    void Fly();
}
```

How would you model a bird?

Chances are, you would add capabilities for flight. However, what happens when you implement an Ostrich?

When Should You Fix Violations?

Is the violation painful?

As in most refactoring strategies, the only time you should really fix existing interface segregation violations is when they cause pain. Otherwise, there are likely more valuable things to spend your time building.

Summarizing the Interface Segregation Principle

Remembering the Interface Segregation Principle is easy:
look for wasteful members that are not being used and separate them.

* Conformance to the Interface Segregation Principle ensures flexible solutions that are highly mobile. This also promotes reusability as it lowers the barriers for other developers to consume your code.

What is the Dependency Inversion Principle?

Depend upon abstractions, not concretions.

Consider an Electrical Outlet

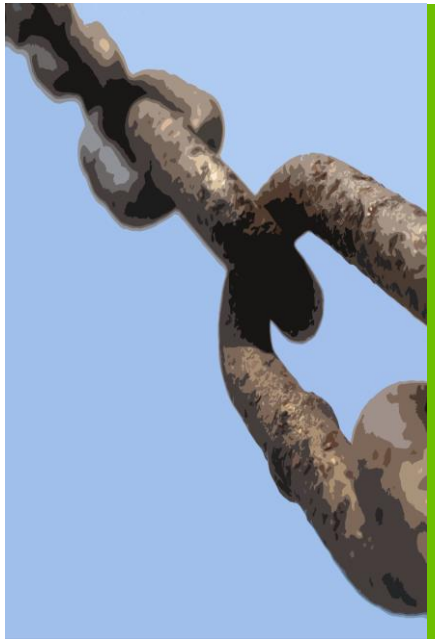
Outlets provide an easy abstraction for electronic devices. Imagine the alternative, soldering a wire directly to the power line?

Dependency Inversion Principle

High level modules should not depend upon low level modules. Instead, both should depend upon abstractions.

Likewise, abstractions should not depend upon details. Rather, details should also depend upon abstractions.

Why Does It Matter?

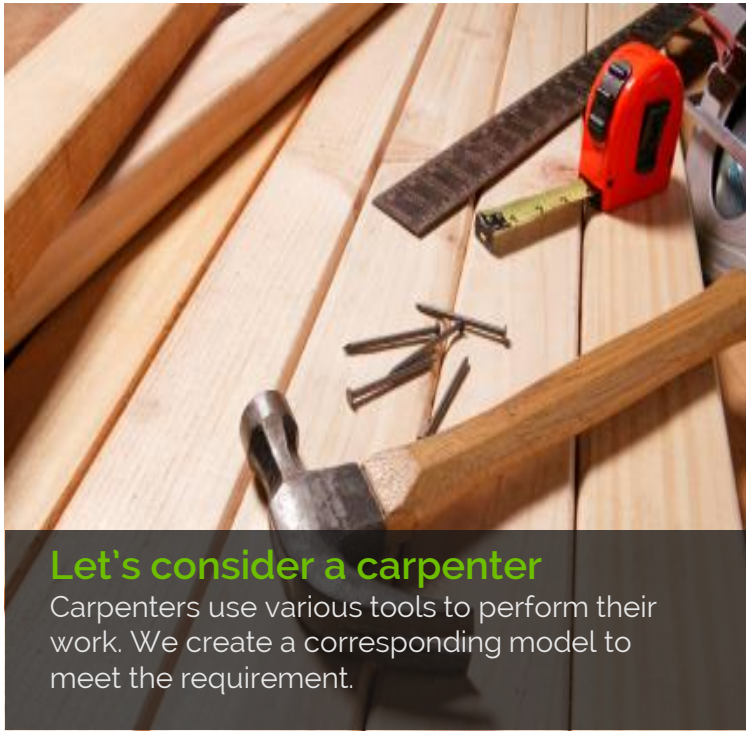


Avoid Tight Coupling

The Dependency Inversion Principle ensures that our code is flexible. When our solution is tightly coupled, any change forces a recompilation of the entire project.

1. Two classes are tightly coupled if they directly depend on each other.
2. Tightly coupled classes cannot work independently of each other.
3. Any change in one class can easily cascade to all linked classes.

A Quick Example: The Carpenter



Let's consider a carpenter

Carpenters use various tools to perform their work. We create a corresponding model to meet the requirement.

The carpenter model

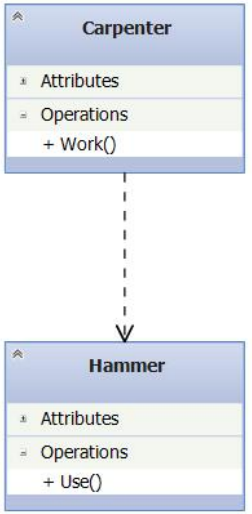
Initially, all carpenters only use hammers, as reflected in our diagram below.

High-Level Class

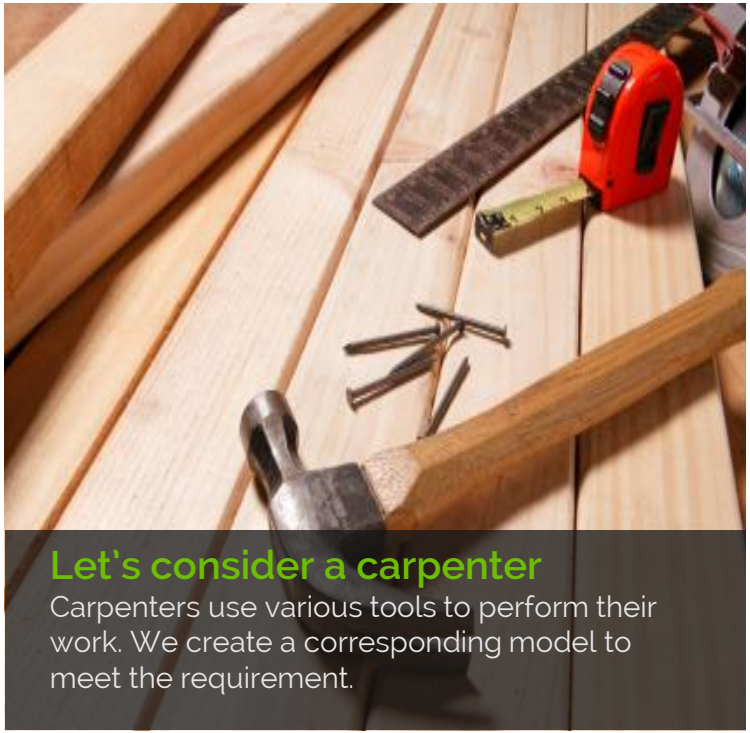
This class must conform to the public interface of the Hammer.

Low-Level Class

This class decides on its interface independent of any higher level callers.



A Quick Example: The Carpenter

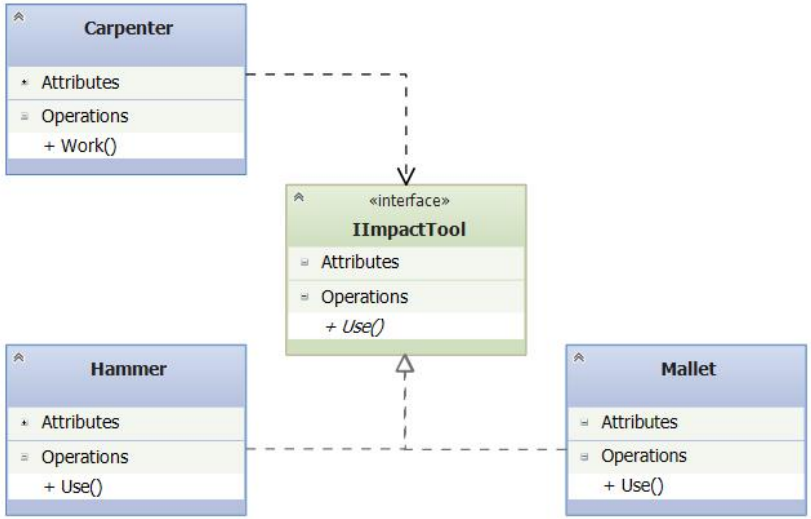


Let's consider a carpenter

Carpenters use various tools to perform their work. We create a corresponding model to meet the requirement.

The carpenter model

Carpenters have been upgraded; now, some can use hammers and others can use mallets.



Traditional Application Layering

In a traditional n-tier application, a dependency chain is created by stacking higher level components on lower level modules. This chain becomes fragile as the overall system experiences change.

Presentation Layer

Responsible for external connectivity with either a user or another computer system.

Business Layer

Responsible for logic unique to the business domain in which the system functions.

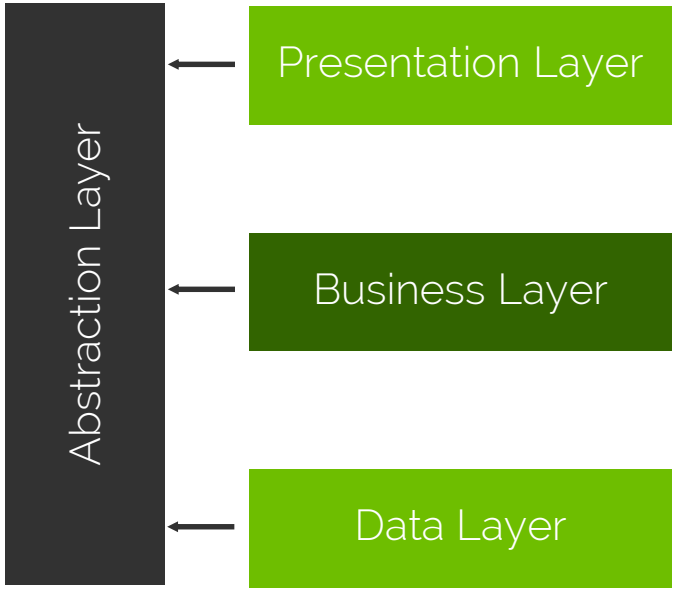
Data Layer

Responsible for managing persisted storage, or any other external data integrations.

Dependency Inversion Layering

In an application designed for inversion, the dependency chain is severed so that high-level modules no longer depend directly on low-level components. Change is easily accommodated since no explicit references exist.

Responsible for all shared interface definitions throughout the application.



Responsible for external connectivity with either a user or another computer system.

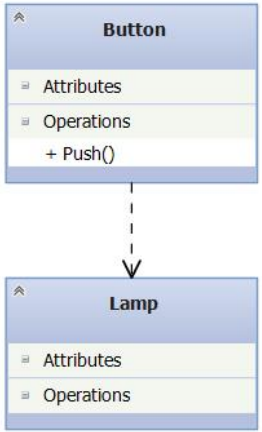
Responsible for logic unique to the business domain in which the system functions.

Responsible for managing persisted storage, or any other external data integrations.

The Button Example

The button model

Initially, our button is used to turn a lamp on and off. The Button class has an explicit dependency on the Lamp object.



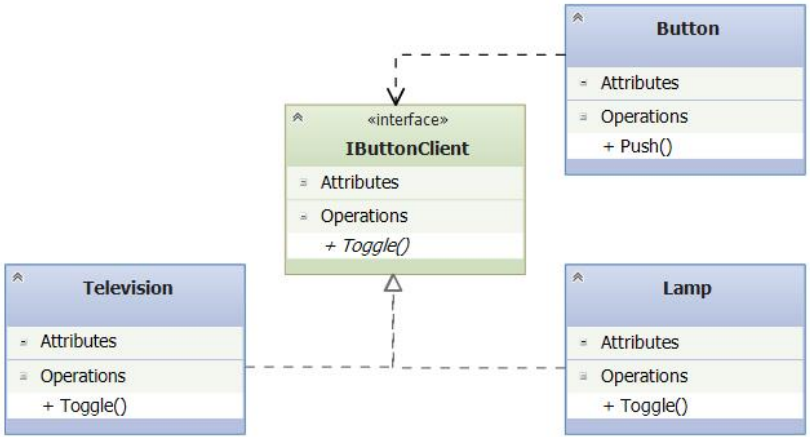
Let's consider a simple button

A button is a very general concept that should be easily reused for many applications.

The Button Example

The button model

Now, our button can be used to control any device, not just a lamp since it no longer has a direct dependency.



Let's consider a simple button

A button is a very general concept that should be easily reused for many applications.

Summarizing the Dependency Inversion Principle

Remembering the Dependency Inversion Principle is easy: use abstractions to when wiring your objects together.

* Conformance to the Dependency Inversion Principle ensures both flexibility and reusability. This allows us to create large systems that are inherently protected from change.

Summary of the SOLID Principles

Single Responsibility Principle
A class should not contain multiple responsibilities.

Open/Closed Principle
A class should be open to extension, but closed for modification.

Liskov Substitution Principle
An object should be replaceable with its sub-types without altering any system behavior.

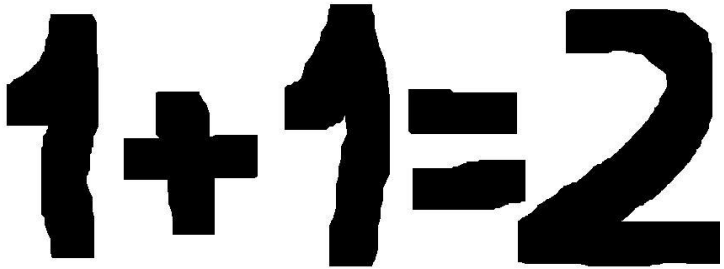
Interface Segregation Principle
Many specific interfaces are preferable to a single generic interface.

Dependency Inversion Principle
A class should only depend upon abstractions, not implementations.

In each coding task, always try to remember to be **SOLID**.

Remember that design decisions are made everyday of the coding cycle, not just at the beginning. So its important to remain SOLID.

Keep It Simple



Strive for elegant, clean designs

Complexity often creates noise in the design, making it inefficient and unmanageable.

* Avoid creating unnecessary complexity. The best coders command the simplest approaches.
* More files doesn't mean it's more complex. Often, it means its simpler.
* Always start with the simplest, easiest solution and work towards complexity.

Simplicity is Beautiful

Often, developers think for software to be valuable it must be complex. I find it help to remind myself that simple is beautiful.

Don't Repeat Yourself

Stay **DRY**: Try to ensure your design maximizes reusability throughout your solution.

* Following the DRY principle of trying not to repeat yourself leads to code that naturally conforms to the SOLID principles.

You Aren't Gonna Need It

When designing, try to ground yourself in the **Y.A.G.N.I.** Principle

If we try to build something now for a future requirement, chances are we don't know enough to do it adequately.

Avoid Coding for the Future

This can be really hard for developers, but the reality is, no value exists in building the future now.

The Principle of Least Surprise

Try to follow well established patterns and practices. The best code is like the best systems: stable and predictable.

* Realize this doesn't mean be boring. Great developers are normally very creative. However, it means if there is already an accepted way of doing something then you should have an incredibly good reason to deviate.

Remember Principles Aren't Rules

Never Be Scared To Ask Why

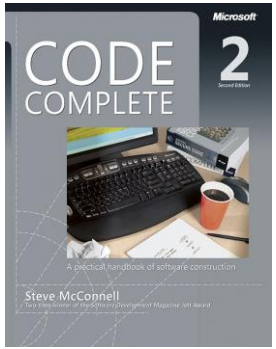
Remember these are principles are guides, not rules. There are always exceptions, so it's ok to ground yourself by asking "why".

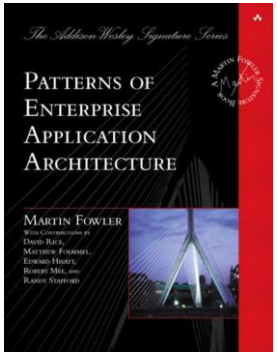
* A software principle violation is not always bad, but it is a code smell. If I see one in a code review, I will generally ask a developer "why" they choose to deviate. If the coder has a good justification, then I won't suggest a refactor.

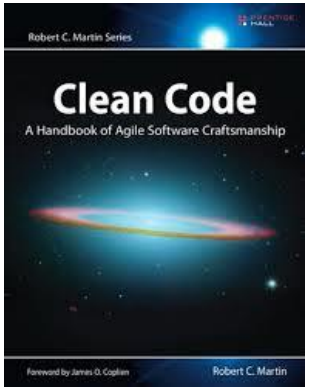
Being a SOLID coder comes down to making common sense decisions..

Next Steps

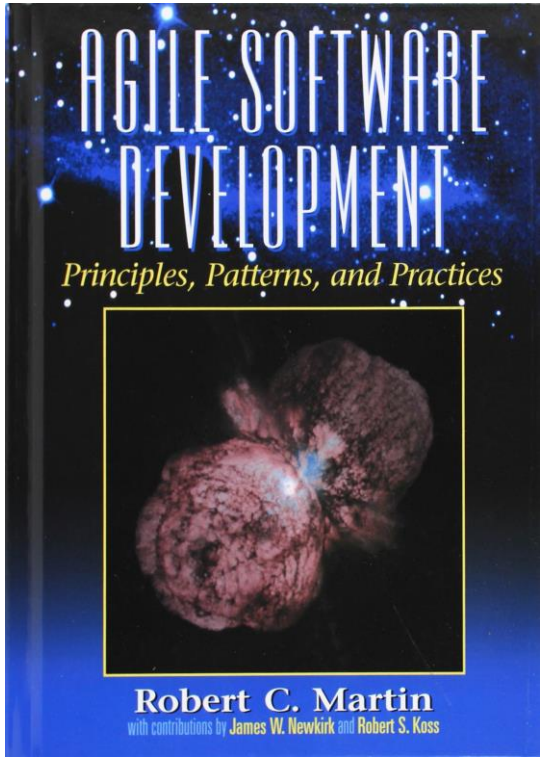
"If it's work, we try to do less. If it's art we try to do more."
– **Seth Godin**







Remember that a little knowledge can be **dangerous**. To be a great developer, I highly recommend these resources to guide you towards deeper understanding.



Are You a SOLID Coder?

Principles of Software Design

Steve Green
steve.p.green@outlook.com
@stephenpgreen
/stevepgreen