# Robotics Corner

# Robotics Corner

## SOLID Principles of OOD

01 Single Responsibility Principle

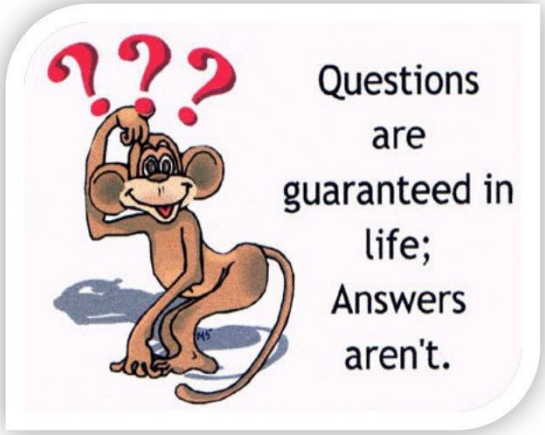
02 Open Closed Principle

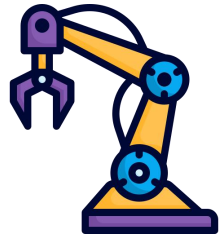
03 Liskov Substitution Principle

04 Interface Segregation Principle

05 Dependency Inversion Principle

# 00 What is and why SOLID principles?

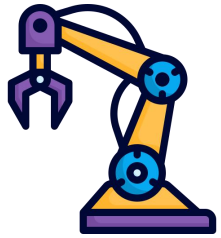




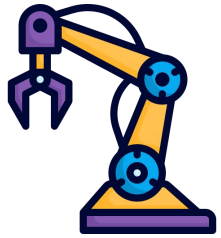
# Engineering Philosophy

- Beautiful, Maintainable Code
- We want to “Care” about our code

# Norman Rockstars follow Kaizen



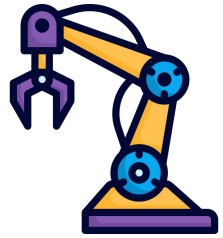
- Yesterday we have code that works
- **Today we want to improve it**



# SOLID Design Principles

What is the only constant in software development?


ROBOTICS CORNER

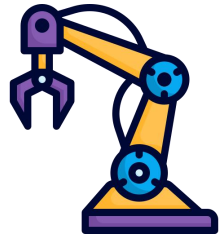


# How does our software respond to CHANGE?

## TWO KINDS OF PROBLEMS

**Code Problems a.k.a Code Smells**

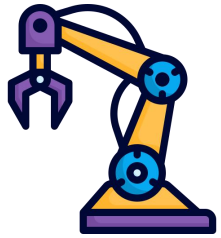
- Refactoring – Improving the design of existing code [Martin Fowler]
- Catalog of smells



# Design Smells

- Agile Principles & Practices[Bob Martin]

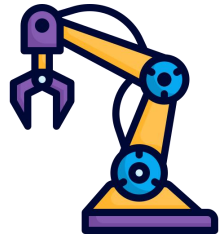
- Rigidity
- Fragility
- Viscosity
- Immobility



# Design?

- In 1992, Jack Reeves wrote a seminal article "**What Is Software Design**?"in the *C++ Journal.[1] In this* article, Reeves argued that the design of a software system is documented primarily by its source code, that diagrams representing the source code are ancillary to the design and are not the design

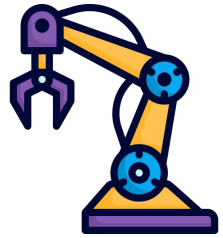
  itself. **The source code *is the design***.
- *'In school, they teach the opposite'.*



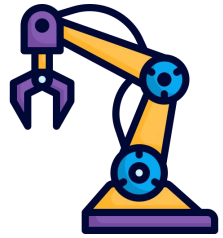
"Rigidity is the opposite of Flexibility. Our Design Sucks if it is not Flexible"


ROBOTICS CORNER

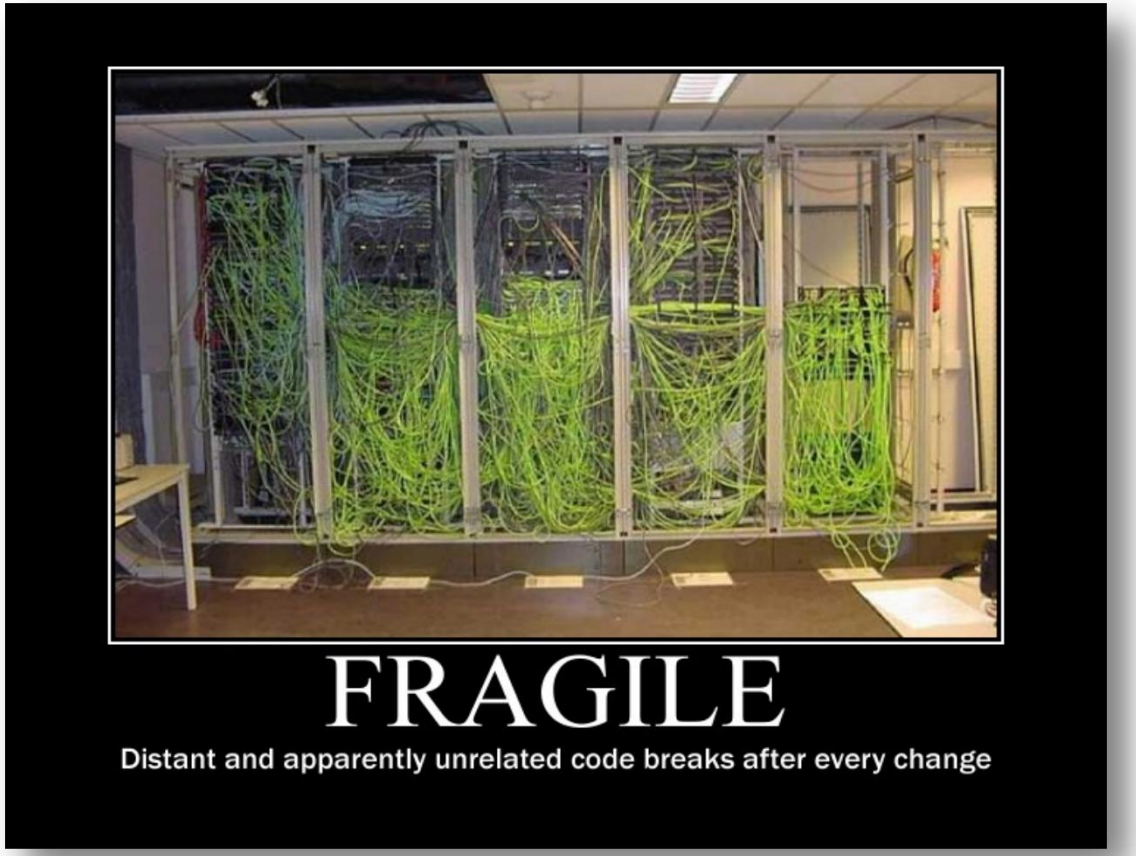
# RIGIDITY



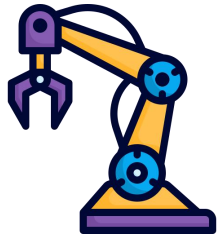
- Rigidity is the tendency for software to be difficult to change, even in simple ways. A design is rigid if a single change causes a cascade of subsequent changes in dependent modules. The more modules that must be changed, the more rigid the design.



# “Does our software break when we try to adapt to a change?”

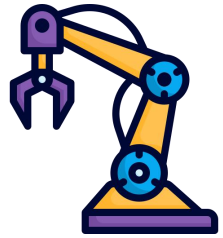



ROBOTICS CORNER



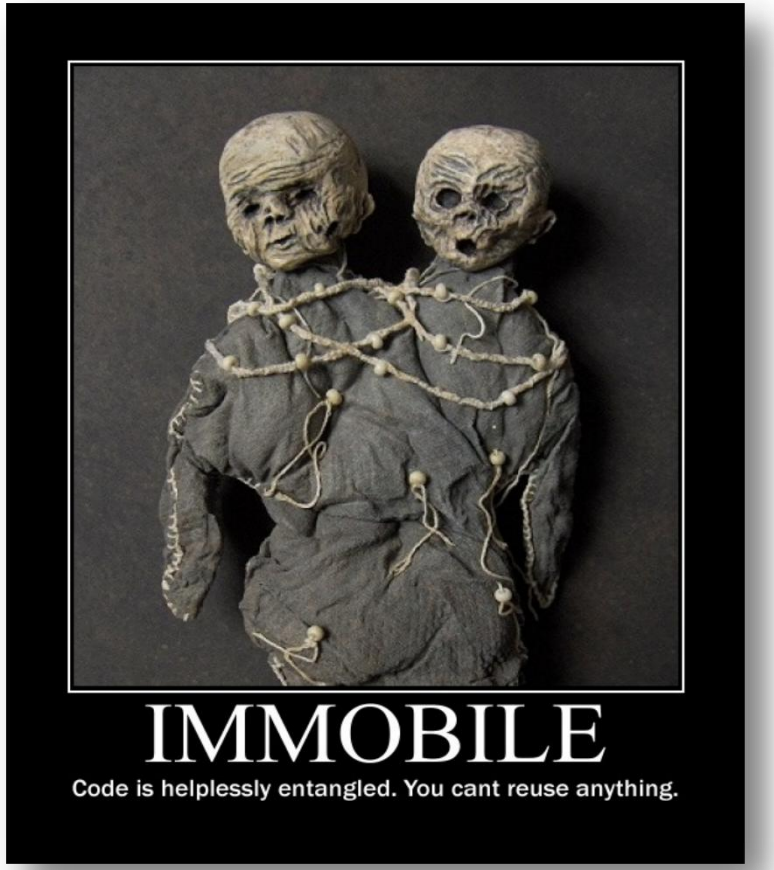
# Fragility

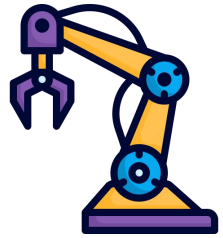
- Fragility is the tendency of a program to break in many places when a single change is made. Often, the new problems are in areas that have no conceptual relationship with the area that was changed. Fixing those problems leads to even more problems, and the development team begins to resemble a dog chasing its tail.



# “We can’t reuse our beautiful code”

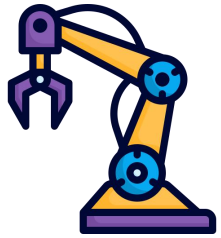
[Or what we think is so beautiful, until our peers challenge it in a code review]



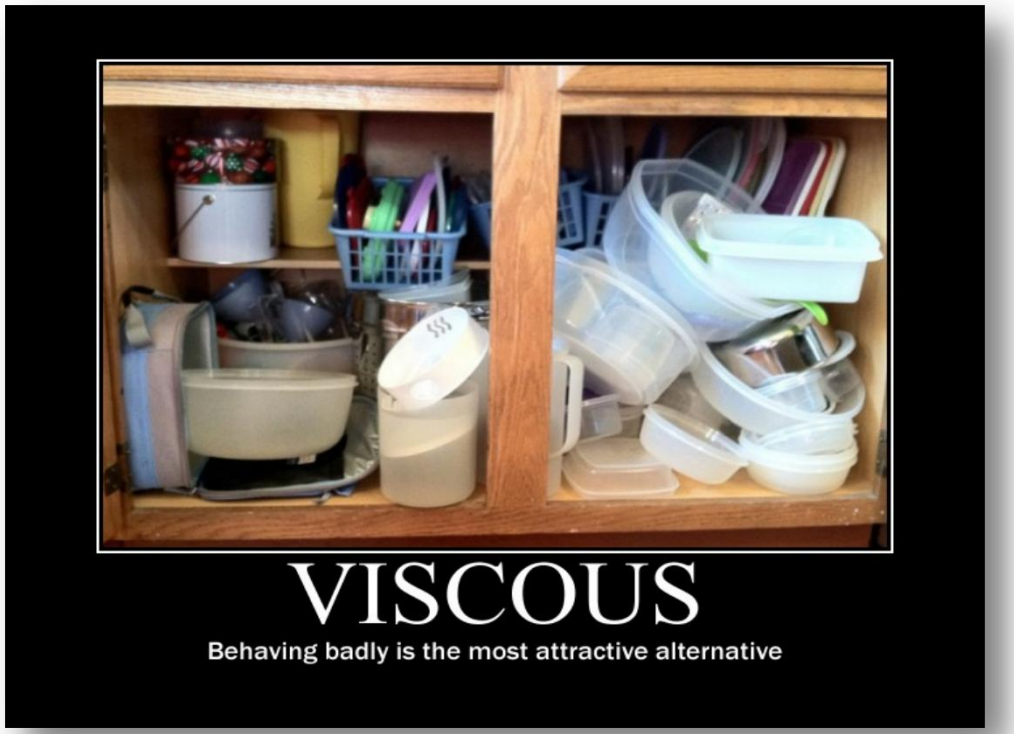


# IMMOBILITY

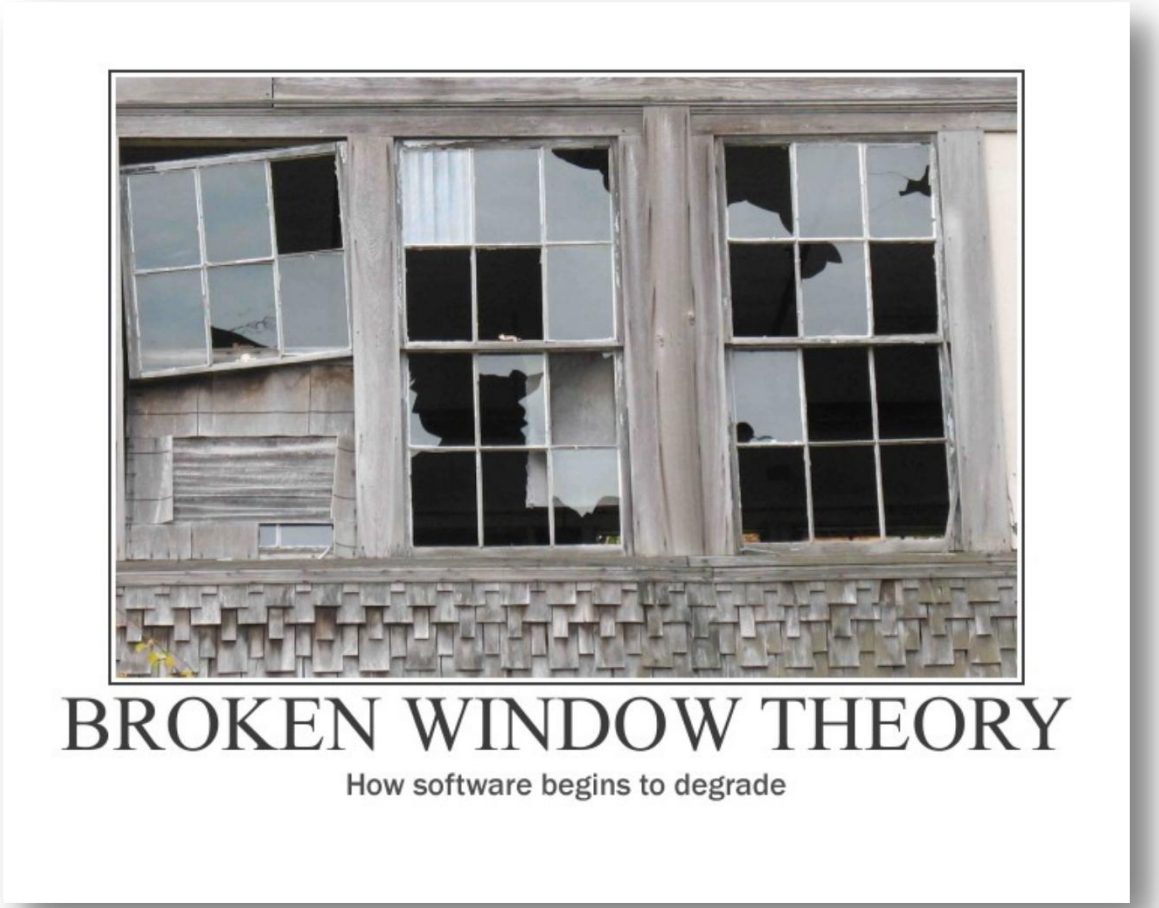
- A design is immobile when it contains parts that could be useful in other systems, but the effort and risk involved with separating those parts from the original system are too great. This is an unfortunate but very common occurrence.

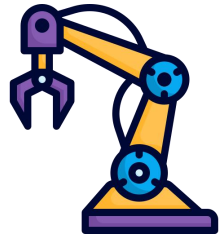


"We can't maintain/continue/persist a good design"



# What is and why SOLID principles?

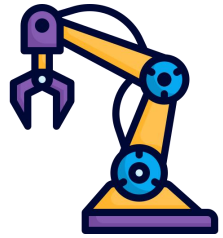




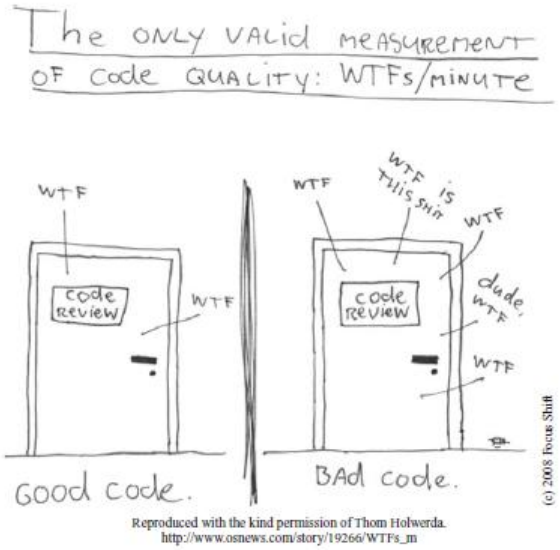
# VISCOSITY

- A viscous project is one in which the design of the software is difficult to preserve. Friction develops and slows us down. We want to create systems and project environments that make it easy to preserve and improve the

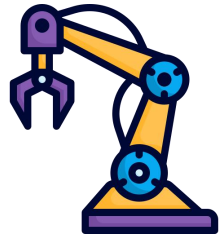
  design.



# Clean Code




Reproduced with the kind permission of Thom Holwerda.
http://www.osnews.com/story/19266/WTFs_m

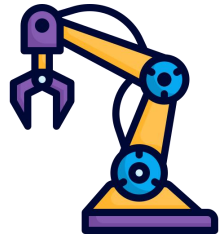


# SOLID Principles of Object Oriented Design

- **S**ingle **R**esponsibility **P**rinciple
- **O**pen/**C**losed **P**rinciple
- **L**iskov **S**ubstitution **P**rinciple
- **I**nterface **S**egregation **P**rinciple
- **D**ependency **I**nversion **P**rinciple

http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod

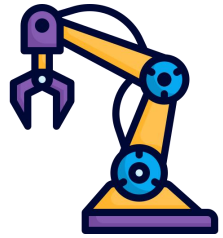
ROBOTICS CORNER



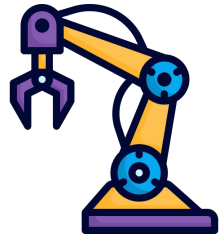
# The Acronym **SOLID** was given by

- Micheal Feathers
- Author of Working Effectively with Legacy Code

http://butunclebob.com/ArticleS.MichaelFeathers



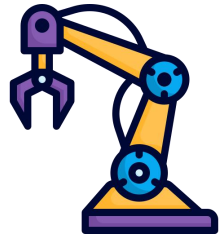
SO, WHAT ARE THESE SOLID PRINCIPLES ABOUT?



# Better question would be

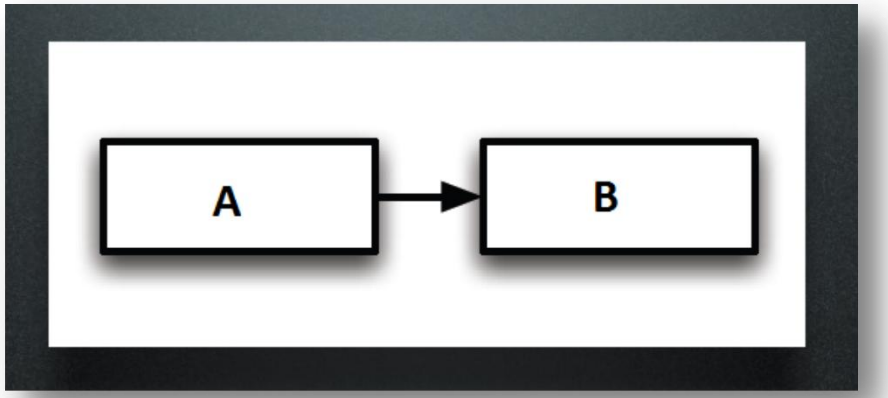
- What problems are they trying to solve?

- They are trying to solve a specific set of problems, namely



# DEPENDENCY MANAGEMENT ISSUES

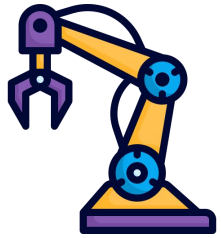
## On Dependencies




**A IS DEPENDENT ON B. If something happens to B, what effect does it have on A?**

**DEPENDENCIES GIVE US SOME DEGREE OF REASON TO SEE HOW CHANGES PROPOGATE**

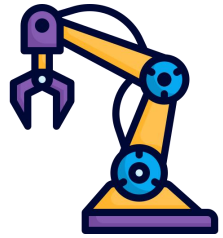
ROBOTICS CORNER



# What is Dependency Management?

- “Dependency Management is an issue that most of us have faced. Whenever we bring up on our screens a nasty batch of tangled legacy code, we are experiencing the results of poor dependency management. “

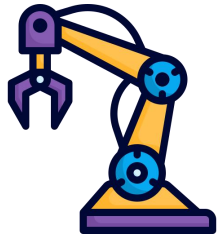
ROBOTICS CORNER



Why is Dependency Management so important?

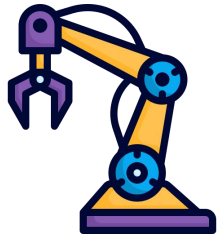
- Poor dependency management leads to code that is hard to change, fragile, and non-reusable
- On the other hand, when dependencies are well managed, the code remains flexible, robust, and reusable

# What is and why SOLID principles?



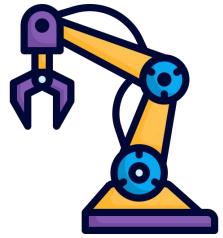
WHAT IS THE ONE TECHNICAL DEPENDENCY THAT TELOGICAL WANTS TO GET AWAY FROM ASAP?


ROBOTICS CORNER



# Other OO Developers

- Have committed the same kind of design mistakes
- Have learnt from those mistakes
- Some have specified some GUIDELINES
- We can choose to follow or not!
- If we choose to follow, we have to understand the problems and where they apply.
- If we encounter the same issues, we can meditate on these guidelines and apply them.



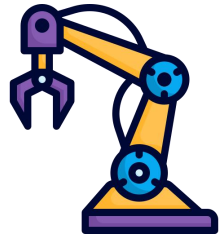
*“Talk is cheap, show me the code”*

- Linus Torvalds


STATIC (C#), DYNAMIC (RUBY)

ROBOTICS CORNER

# 01 SINGLE RESPONSIBILITY PRINCIPLE

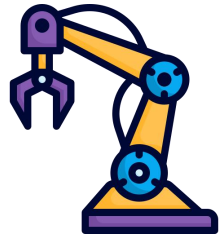


- *A CLASS SHOULD HAVE ONE AND ONLY ONE RESPONSIBILITY.*

- *THERE SHOULD NEVER BE MORE THAN ONE REASON FOR A CLASS TO CHANGE.*

ROBOTICS CORNER

# 01 SINGLE RESPONSIBILITY PRINCIPLE

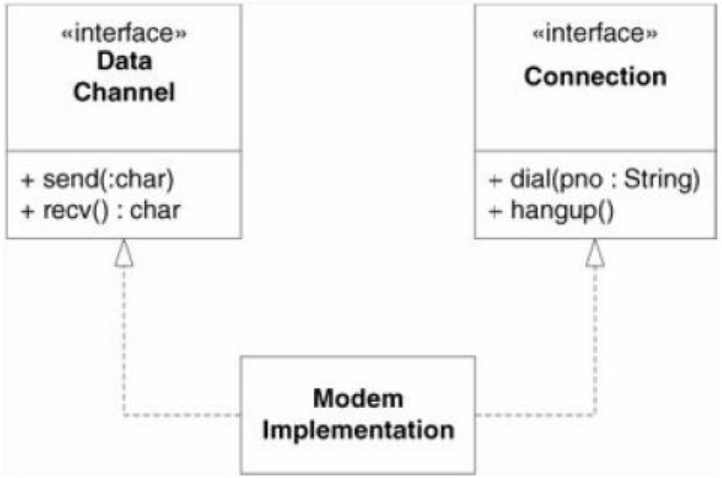


# Example from Uncle Bob's paper

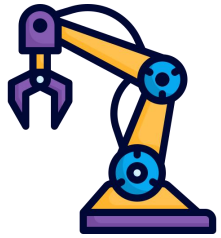
**Listing 8-1.** Modem.cs -- SRP violation

```
public interface Modem
{
  public void Dial(string pno);
  public void Hangup();
  public void Send(char c);
  public char Recv();
}
```

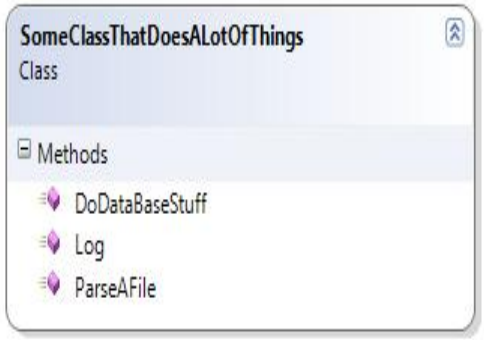
# 01 SINGLE RESPONSIBILITY PRINCIPLE



# SINGLE RESPONSIBILITY PRINCIPLE

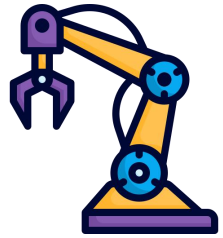


- This principle was described in the work of Tom DeMarco and Meilir Page-Jones.
- They called it **cohesion**, which they defined as the functional relatedness of the elements of a module.




ROBOTICS CORNER

# 01 SINGLE RESPONSIBILITY PRINCIPLE



Why is it important to separate two responsibilities into separate classes?

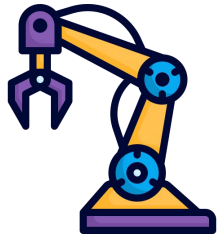
Because each responsibility is an axis of change. When the requirements change, that change will be manifest through a change in responsibility amongst the classes. If a class assumes more than one responsibility, then there will be more than one reason for it to change.

If a class has more then one responsibility, then the responsibilities become coupled. Changes to one responsibility may impair or inhibit the class' ability to meet the others. This kind of coupling leads to fragile designs that break in unexpected ways when

changed.

ROBOTICS CORNER

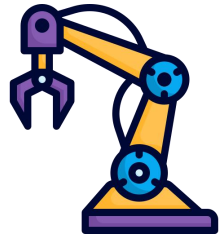
# 01 SINGLE RESPONSIBILITY PRINCIPLE



- Responsibility means a reason to change. Responsibility is an axis for change.
- Change in requirements are often the case.
- Following the Single Responsibility Principle can lead us to our goal of Strong Cohesion & Loose Coupling.
- Many small classes with distinct responsibilities results in a more flexible design.
- The SRP is one of the simplest of the principle, and one of the hardest to get right.
- Conjoining responsibilities is something that we do naturally.
- Finding and separating those responsibilities from one another is much of what software design is really about.
- Multiple small interfaces that follow Interface Segregation Principle can help in achieving SRP.

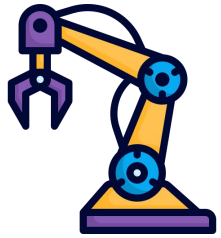
ROBOTICS CORNER

# OPEN/CLOSED PRINCIPLE



*SOFTWARE ENTITIES (CLASSES, MODULES, FUNCTIONS, ETC.) SHOULD BE OPEN FOR EXTENSION, BUT CLOSED FOR MODIFICATION.*

# 02 OPEN/CLOSED PRINCIPLE



- Open to Extension: New Behavior can be added in the future

- Closed to Modification: Changes to existing code is not required.


ROBOTICS CORNER

# Example from Uncle Bob's paper

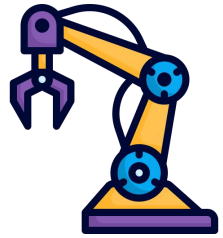


**Figure 9-1.** `Client` **is not open and closed.**

# “Abstraction” is the key

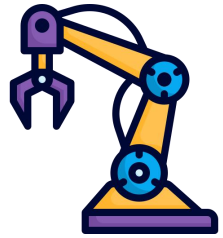
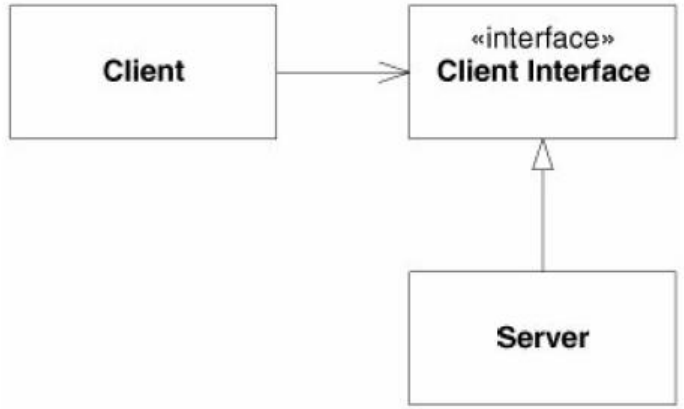
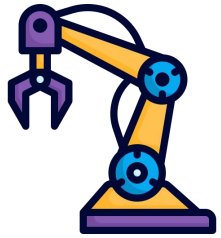


**Figure 9-2. STRATEGY pattern: `Client` is both open and closed.**



# OCP

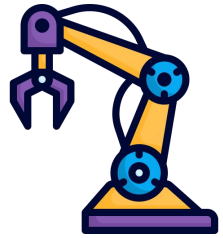


- The OCP Principle was given by Bertrand Meyer
- OCP is at the heart of Object Oriented design
- Conformance to OCP yields the greatest benefits of OOD, reusability & maintainability
- Key to OCP is programming to abstractions.
- Do not depend on implementations, depend on abstractions.
- 100 % Conformance to OCP is not possible. There might be a requirement that may render our code not closed.
- Since closure cannot be complete, it should be strategic.
- Developer must choose what changes the code can be closed to.
- Experience would help a developer to make choices and design code for the most probable causes of change.

ROBOTICS CORNER

*FUNCTIONS THAT USE POINTERS OR REFERENCES TO BASE CLASSES MUST BE ABLE TO USE OBJECTS OF DERIVED CLASSES*

*WITHOUT KNOWING IT.*



# LSP

The above is a paraphrase of the Liskov Substitution Principle

Barbara Liskov first wrote it as follows :

> If for each object o1 of type S there is an object o2 of type T such that for all programs P defined in terms of T, the behavior of P is unchanged when o1 is substituted for o2 then S is a subtype of T.

# Example from Uncle Bob's paper

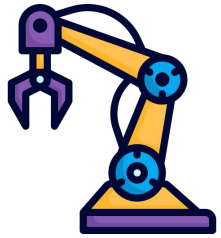
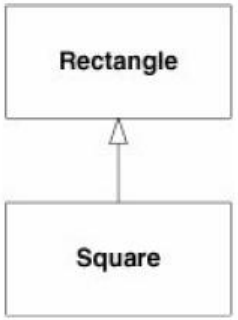
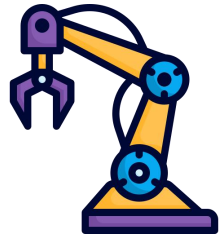


**Figure 10-1.** `Square` **inherits from** `Rectangle`




ROBOTICS CORNER

# 03 LISKOV SUBSTITUTION PRINCIPLE



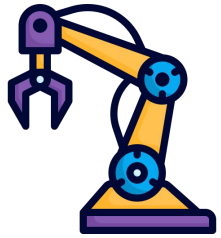
```
public class Rectangle
{
    public virtual double Length { get; set; }
    public virtual double Width { get; set; }

    public double CalculateArea()
    {
        return Length*Width;
    }
}
```

```
public class Square : Rectangle
{
    private double _length;
    private double _width;

    public override double Length
    {
        get { return _length; }

        set { _length = value;
            _width = value;
        }

    }
    public override double Width
    {
        get { return _width; }

        set
        {
            _length = value;
            _width = value;
        }

    }

}
```

```
static void Main(string[] args)
{
    Console.WriteLine("Kids tool to learn areas.. ");

    //Kids are Programming against our API
    var kidslist = new List<Rectangle>()
                        {
                            new Square(){Length = 12,Width = 14},
                            new Square(){Length = 4, Width = 8}
                        };

    foreach (var rectangle in kidslist)
    {
        ShowAreas(rectangle);
    }
    Console.ReadKey();

}

private static void ShowAreas(Rectangle r)
{
    Console.WriteLine(r.CalculateArea());
}
```

What we thought to be code reuse with inheritance causes a problem

ROBOTICS CORNER



# LSP

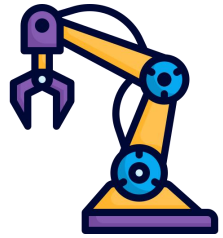
- Named after Barbara Liskov, who gave the principle in 1988.
- Substitutability
- Calling code must not know the difference between a derived type and a base type.
- Sub-types must be substitutable for their base types.
- Child classes should not remove base class behavior.
- Native OOP: "IS-A" relationship
- LSP : "IS-SUBSTITUTABLE=FOR“

- **LSP surfaces the undesired behavior problems in subtypes caused due to inheritance**

ROBOTICS CORNER

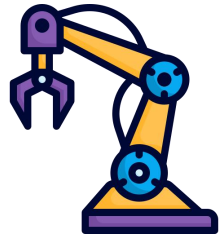
*CLIENTS SHOULD NOT BE FORCED TO DEPEND UPON INTERFACES THAT THEY DO NOT USE.*

# 04 INTERFACE SEGREGATION PRINCIPLE



```
public interface IPackagesService
{
    AdvertisedPrice GetAdvertisedPrice(IBasePackage basePackage, int numberOfMonthsToShow, bool allowOnlineOnlyPromotions);
    List<IBasePackage> GetAllPackages(PackageCriteria packageCriteria);
    TermCommitment GetTermCommitment(IBasePackage basePackage, int numberOfMonthsToShow, bool allowOnlineOnlyPromotions);
    Summary GetSummary(IBasePackage basePackage);
    void SetDataSource(string dataSource);
    string GetPackageName(int packageId);
    IBasePackage GetPackage(int packageId, string zipcode, int accountId);
    PricesPromotions GetPricesPromotions(IBasePackage basePackage, int numberOfMonthsToShow, bool allowOnlineOnlyPromotions);
    bool DoesClientHavePromotionalPriceAdvantage(PricesPromotions clientPricesPromotions, PricesPromotions competitorPricesPromotion
    string GetStandardPrice(PricesPromotions pricesPromotions);
    bool DoesClientHaveStandardPriceAdvantage(PricesPromotions clientPricesPromotions, PricesPromotions competitorPricesPromotions);
    PricesPromotionsForOneTimeFee GetOneTimeFee(IBasePackage basePackage, int numberOfMonthsToShow, bool allowOnlineOnlyPromotions);
    bool DoesClientHaveOneTimeFeeAdvantage(PricesPromotionsForOneTimeFee clientPricesPromotions, PricesPromotionsForOneTimeFee compet
    Contract GetContract(IBasePackage basePackage, int numberOfMonthsToShow, bool allowOnlyPromotions);
    bool DoesClientHaveContractAdvantage(Contract clientContract, Contract competitorContract);
    CashBack GetCashBack(IBasePackage basePackage, int numberOfMonthsToShow, bool allowOnlyPromotions);
    bool DoesClientHaveCashBackAdvantage(CashBack clientCashBack, CashBack competitorCashBack);
    EarlyTerminationFee GetEarlyTerminationFee(IBasePackage basePackage, int numberOfMonthsToShow, bool allowOnlyPromotions);
    bool DoesPackageHaveTVReceiverIncluded(IBasePackage basePackage);
    HdService GetHDService(IBasePackage basePackage);
    List<LineupArea> GetLineupAreas(int packageId, string zipcode, int accountId);
    bool DoesClientHaveEarlyTerminationFeeAdvantage(EarlyTerminationFee clientEarlyTerminationFee, EarlyTerminationFee competitorEar
    Voicemail GetVoicemail(IBasePackage basePackage);
    List<OptionalChannelPackage> GetOptionalChannelPackages(int packageId, string zipcode, int accountId);
    bool DoClientFeaturesHaveAdvantageOverCompetitorFeatures(List<Domain.BasePackageAggregate.Feature> clientFeatures, List<Domain.Ba
    Modem GetModem(IBasePackage basePackage);
    string GetInternetSpeed(IBasePackage basePackage);
    List<DirectMail> GetDirectMails(int packageId, string zipcode, int accountId);
    List<Domain.BasePackageAggregate.Feature> GetFeaturesForInternationalCallingRate(IBasePackage basePackage);
    string GetAnytimeMinutes(IBasePackage basePackage);
    List<string> GetFreeMinutes(IBasePackage basePackage);
    string GetWirelessDataUsage(IBasePackage basePackage);
    List<FeaturePackage> GetFeaturePackages(IBasePackage basePackage, string productType);
    bool DoesPackageHaveTVReceiverFromFeaturePackageIncluded(IBasePackage basePackage);
    bool DoesPackageHaveIncludedChannelLineup(IBasePackage basePackage);
    bool DoesPackageHavePromotionsThatAffectTVReceivers(IBasePackage basePackage);
}
```

ROBOTICS CORNER

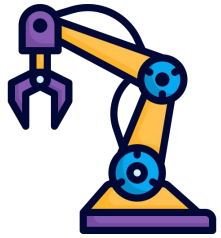


# ISP

- Fat/Polluted interfaces
- Classes that have “fat” interfaces are classes whose interfaces are not cohesive.
- Let the client drive the interface.
- In other words, the interfaces of the class can be broken up into groups of member functions. Each group serves a different set of clients. Thus some clients use one group of member functions, and other clients use the other groups.
- The ISP acknowledges that there are objects that require non-cohesive interfaces; however it suggests that clients should not know about them as a single class. Instead, clients should know about abstract base classes that have cohesive interfaces.
- Some languages refer to these abstract base classes as “interfaces”, “protocols” or “signatures”.

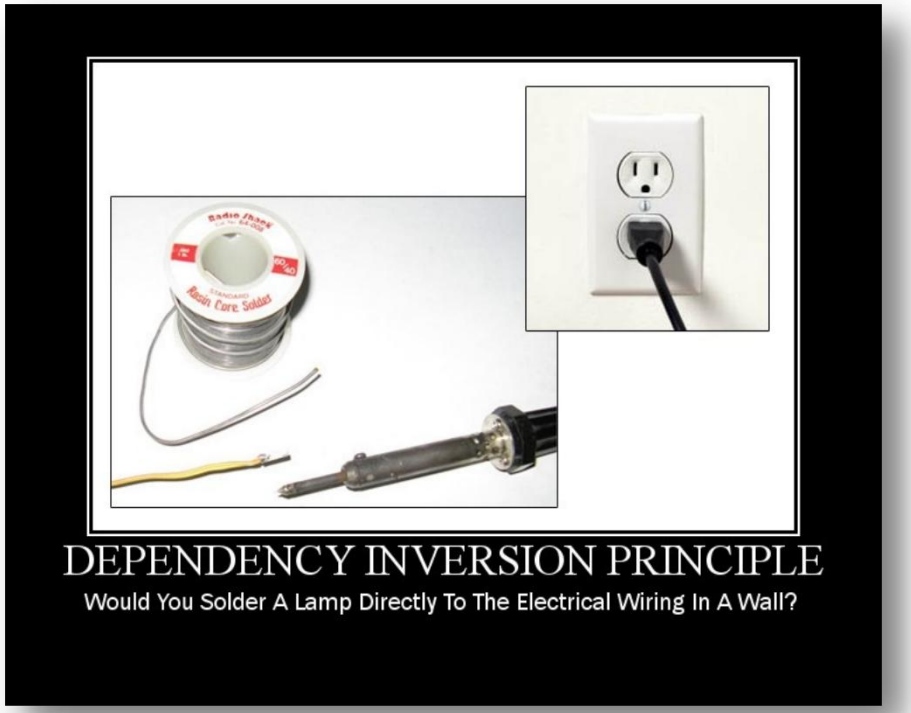
ROBOTICS CORNER

# DEPENDENCY INVERSION PRINCIPLE



- *A. HIGH-LEVEL MODULES SHOULD NOT DEPEND UPON LOW-LEVEL MODULES. BOTH SHOULD DEPEND UPON ABSTRACTIONS.*

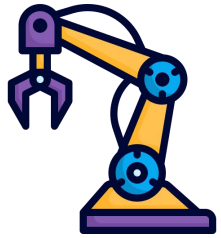
- *B. ABSTRACTIONS SHOULD NOT DEPEND UPON DETAILS. DETAILS SHOULD DEPEND UPON ABSTRACTIONS.*

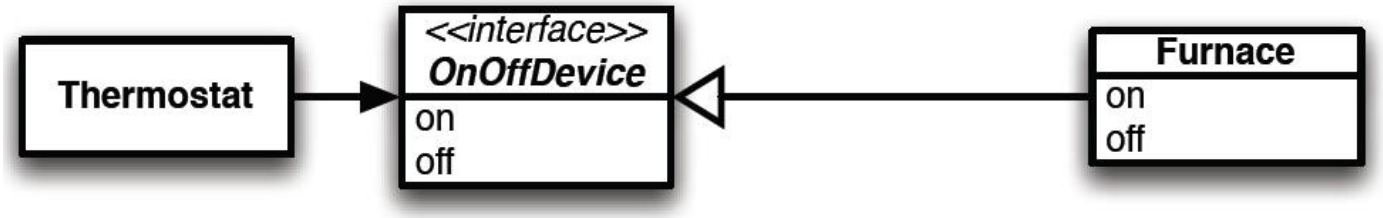



ROBOTICS CORNER

# The Thermostat Problem

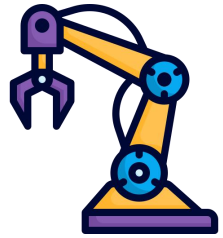
# 05 DEPENDENCY INVERSION PRINCIPLE

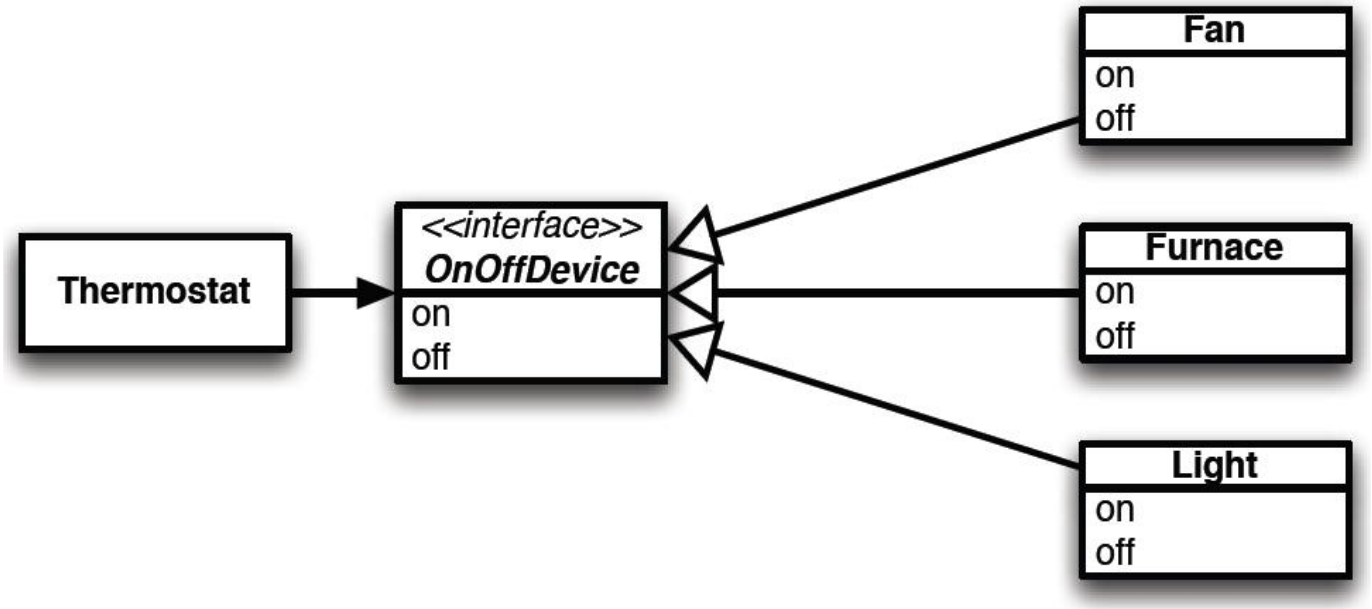




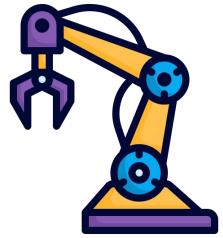

ROBOTICS CORNER

# 05 DEPENDENCY INVERSION PRINCIPLE



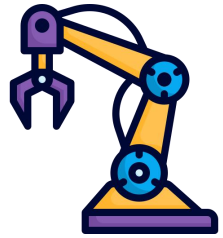


# DIP

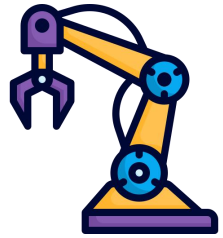


- Code to an abstraction, not an implementation
- Dependencies can also include libraries, files, file system, mail service, ftp, sftp, databases etc
- Tools to recognize dependencies: Ndepend, VS Ultimate Architecture tab

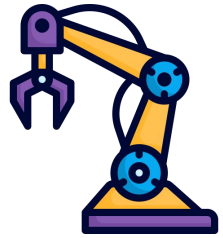
ROBOTICS CORNER



In the API, How can we swap the existing ORM NHibernate with another ORM such as Entity FW or more still MongoDB as the DB?

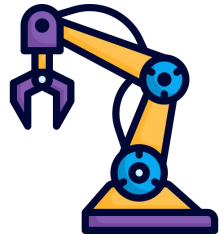


# STATIC vs. DYNAMIC



# Why do we need an Interface?

# 05 DEPENDENCY INVERSION PRINCIPLE

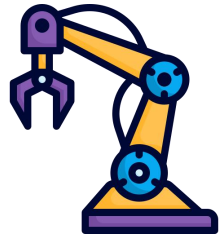


In static languages,
We must know about Types being passed around!

```
public string SomeBehavior(string input)
{
    //Code to perform whatever is necessary to behave in a certain way
    return "Behavior has been achieved!";
}
```

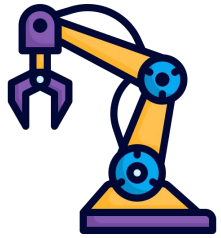
ROBOTICS CORNER

# 05 DEPENDENCY INVERSION PRINCIPLE



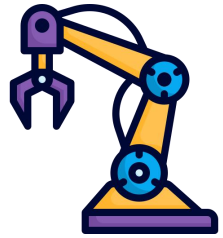
A dynamic language like Ruby is designed to never worry about types in the first place

```ruby
def some_behavior(input)

# I am ruby. I am Dynamic. And I dont have to worry about input's type details guys.
# Code to perform what ever is necessary to attain the behavior

return "behavior achieved.", "i shall take a break bro"
end
```



For Discussion

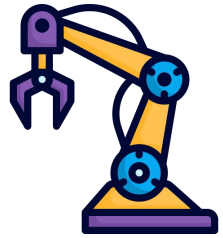
Some SOLID principles are an awkward fit for Ruby.
Are there other SOLID-like principles that are specific to dynamic languages?



# Other design smells & principles

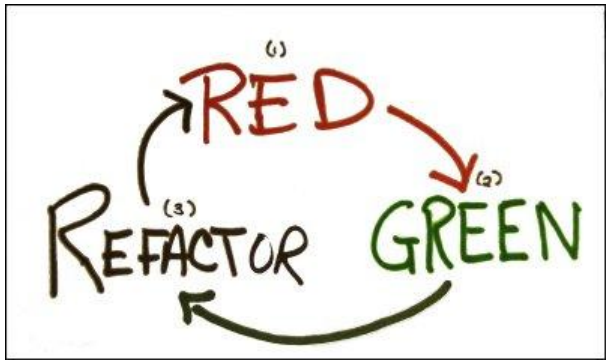
- Needless Repetition : DRY
- Complex/Over design : YAGNI, KISS

ROBOTICS CORNER



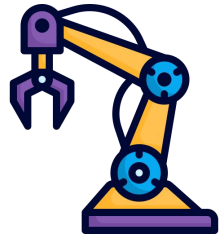
When to apply SOLID design principles

## TDD presents us an opportunity

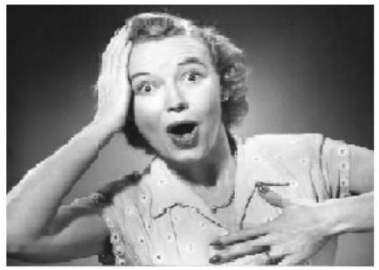


BOY SCOUTS

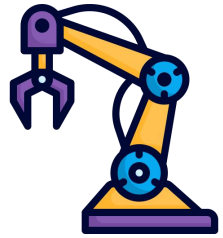
Leave the campsite better than you found it



## Clean Code

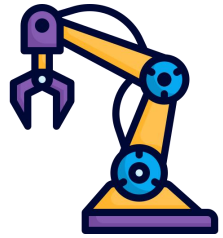


Always code as if the guy maintaining your code would be a violent psychopath and he knows where you live.



# What Next?

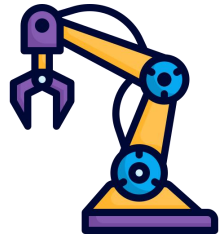
# Mediate on these principles



# For us, its constant learning

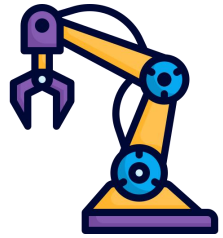
- Design Patterns
- Refactoring to Patterns
- Architectural Patterns
- Patterns in JavaScript

ROBOTICS CORNER



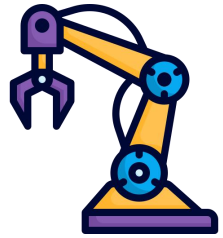
# Code, presentation can be found at

- https://github.com/chris-gibson/solid



# References

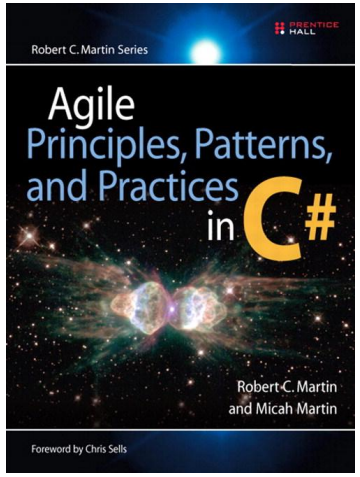

- http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod
- http://lostechies.com/derickbailey/2009/02/11/solid-development-principles-in-motivational-pictures/
- http://bighugelabs.com/motivator.php
- *http://www.developerdotstar.com/mag/bios/jack_reeves.html*

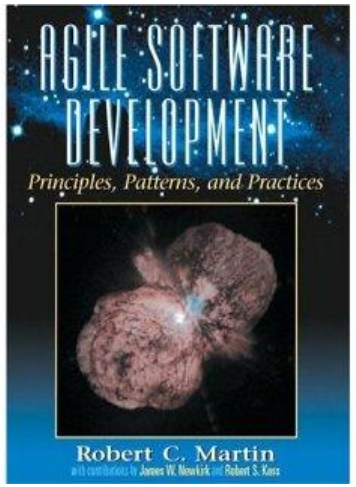


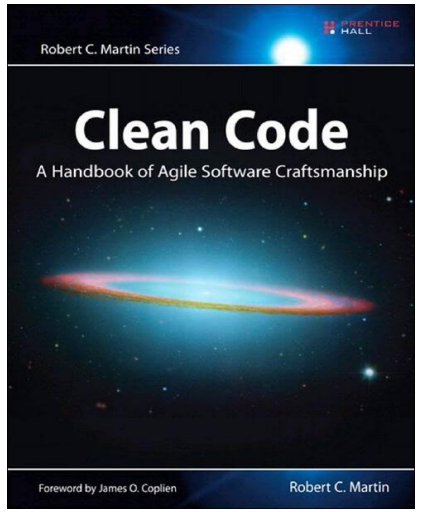
# Ruby SOLID Principles

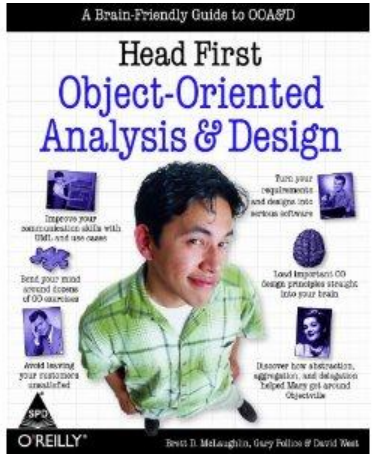
- http://blip.tv/rubynation/jim-weirich-3672101
- http://www.confreaks.com/videos/240-goruco2009-solid-object-oriented-design

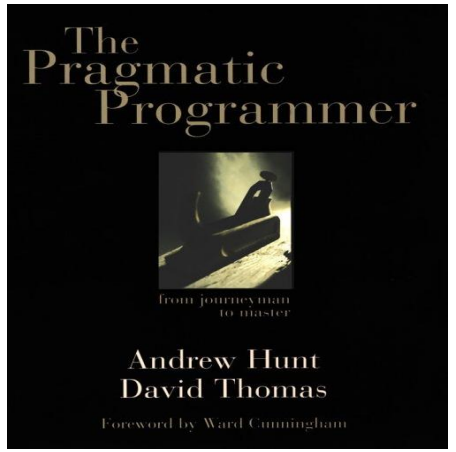
# 06 SOLID Principles

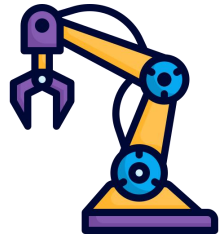




# Thank You

Do you have any questions?

ROBOTICS CORNER

ROBOTICS CORNER

01211626904

www.roboticscorner.tech

Robotics Corner

Robotics Corner

Robotics Corner

Robotics Corner