Aims at creating a system that is easy to maintain and extend over time

SOLID PRINCIPLES OF OBJECT ORIENTED DESIGN

42skillz

SOLID IS A BAD ACRONYM

SOLID IS A REACTION TO DESIGN SMELLS

THE AIM IS TO ACHIEVE A FLEXIBLE DESIGN TO ADDRESS FREQUENTS CHANGES

Single Responsibility

Open/Closed

SOLID Principles …

Liskov Substitution

Interface Segregation

Dependency Inversion

SINGLE RESPONSIBILITY PRINCIPLE

Just Because You Can, Doesn't Mean You Should

SINGLE RESPONSIBILITY PRINCIPLE

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

Each time a class is modified the risk of
introducing bugs grows
By concentrating on a single responsibility this risk is limited

SINGLE RESPONSIBILITY PRINCIPLE?

```
public class TripService
{
  public List<Trip> GetTripsByUser(User user)
  {
    List<Trip> tripList = new List<Trip>();
    User loggedUser = UserSession.GetInstance().GetLoggedUser();
    bool isFriend = false;
    if (loggedUser != null)
    {
      foreach(User friend in user.GetFriends())
      {
        if (friend.Equals(loggedUser))
        { isFriend = true; break; }
      }

      if (isFriend)
        tripList = TripDAO.FindTripsByUser(user);
    }
    return tripList;
  }
}
```

❌ Feature Envy

42skillz

THE SRP VIOLATION SOLVED

```
public class TripService
{
  public List<Trip> GetTripsByUser(User user)
    List<Trip> tripList = new List<Trip>();
    User loggedUser = UserSession.GetInstance().GetLoggedUser();
    if (loggedUser != null)
    {
      if (user.IsFriendWith(loggedUser))
        tripList = TripDAO.FindTripsByUser(user);
    }
    return tripList;
  }
}
```

1. **Extract Method**
2. **Move to User type**

SINGLE RESPONSIBILITY PRINCIPLE

If you have …

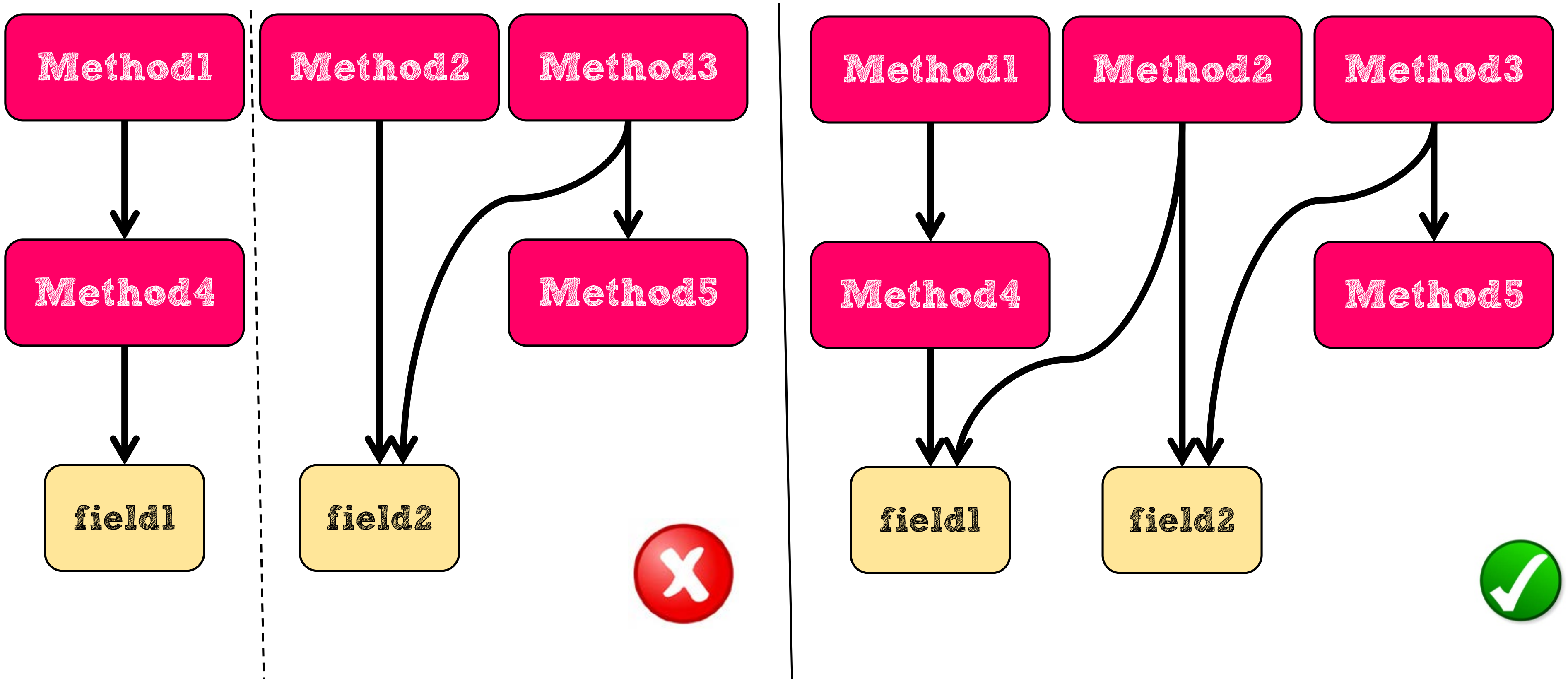
- a very a big class (LOC, Total of methods …)
- a lack of cohesion of methods (LCOM)

You might be breaking The SR Principle!

LACK COHESION OF METHODS

Degree to which methods and fields within a class are related to one another

Only methods and attributes that rely on each other should be kept in the same class



OPEN CLOSED PRINCIPLE

Open Chest Surgery Is Not Needed When Putting On A Coat

OPEN / CLOSE PRINCIPLE

Software entities should be open for extension but closed for modification

As the SRP, this principle reduces the risk of

new errors

being introduced by limiting changes to existing code

OPEN / CLOSE PRINCIPLE?

```
public double Price(int daysRented)
{
    double thisAmount = 0;
    switch (KindOfMovie)
    {
        case KindOfMovie.Regular:
            thisAmount += 2;
            if (daysRented > 2)
                thisAmount += (daysRented - 2)*1.5;
            break;
        case KindOfMovie.NewRelease:
            thisAmount += daysRented*3;
            break;
        case KindOfMovie.Children:
            thisAmount += 1.5;
            if (daysRented > 3)
                thisAmount += (daysRented - 3)*1.5;
            break;
    }
    return thisAmount;
}
```

❌ Switch Statement

42skillz

THE OPC VIOLATION SOLVED

Apply The Pattern Strategy

```
public double Price(int daysRented)
{
    return _strategyPricing
                .GetPricingMovie(KindOfMovie)
                  .Price(daysRented);
}
```

1. Extract Method for each KindOfMovie
2. Extract Class StrategyPricing with each new method
3. Extract Class for each method (wrapped)
4. For each rename the method name: Price
5. For each new class apply interface IPricingMovie with Price()
6. On StrategyPricing create a factory method called GetPricingMovie

THE OPC VIOLATION SOLVED

```
public class StrategyPricing
{
    private readonly Dictionary<KindOfMovie, IPricingMovie>
    _strategyPricing = new Dictionary<KindOfMovie, IPricingMovie>
    {
        [KindOfMovie.NewRelease] = new PricingNewRelease(),
        [KindOfMovie.Children] = new PricingChildren(),
        [KindOfMovie.Regular] = new PricingRegular()
    };

    public IPricingMovie GetPricingMovie(KindOfMovie kindOfMovie)
    {
        return _mappingPricingMovies[kindOfMovie];
    }
}
```

OPEN / CLOSE PRINCIPLE

If you have …

a high cyclomatique complexity

too much conditionals instructions type based

You might be breaking The OC Principle!

CYCLOMATIQUE COMPLEXITY

The cyclomatique complexity of a section of source code is the count of the number of linearly independent paths through the source code

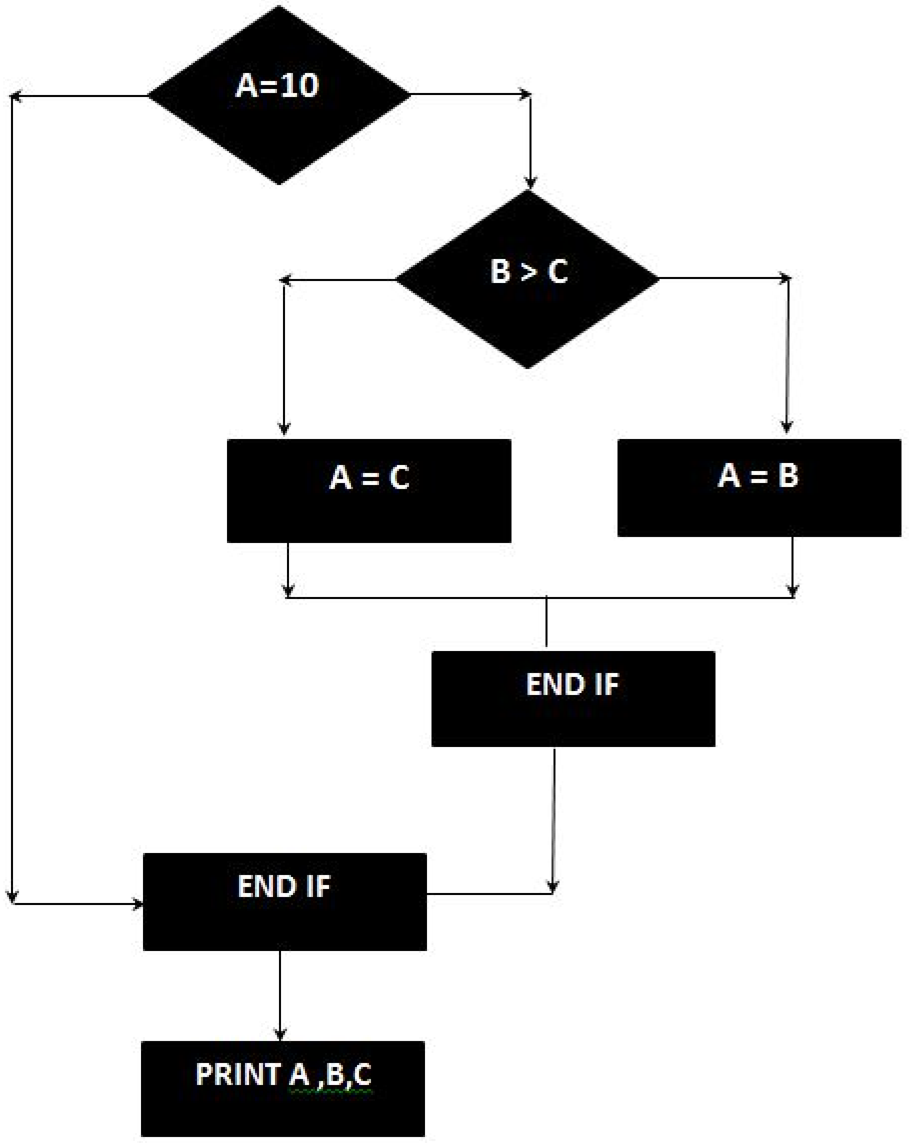
If the source code contained no decision points such as IF statements or FOR loops, the complexity would be 1, since there is only a single path through the code.

The complexity M is then defined as $M = E - N + 2$

Where

E = the number of edges of the graph
N = the number of nodes of the graph

CYCLOMATIQUE COMPLEXITY

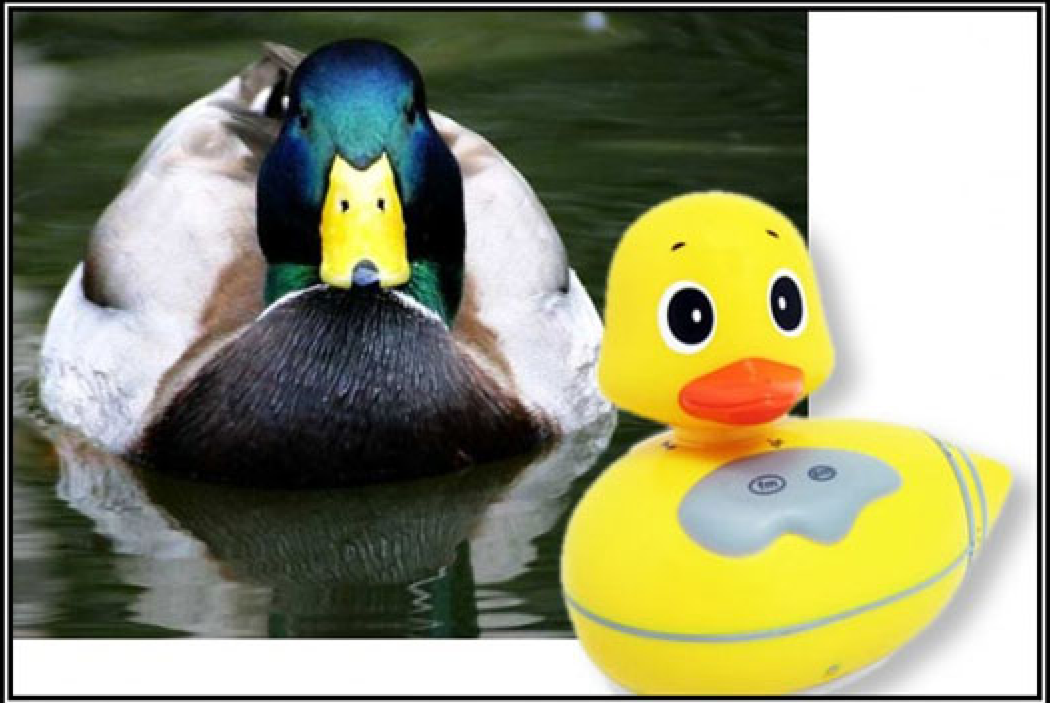


The control flow diagram shows :

- seven nodes
- eight edges (lines),

Hence the cyclomatique complexity is 8 - 7 + 2 = 3

42skillz



LISKOV SUBSTITUTION PRINCIPLE

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

LISKOV SUBSTITUTION PRINCIPLE

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

This principle reduces the risk of

new errors

being introduced by limiting changes behaviors when a new derived class is introduced

LISKOV SUBSTITUTION PRINCIPLE?

```
public abstract class Bird
{
    public abstract void Fly();
    ...
}

public class Pelican: Bird
{
    public override void Fly()
    {
       Console.Writeline("Let’s fly …");
    }
    ...
}

public class Ostrich : Bird
{
    public override void Fly()
    {
       throw new System.InvalidOper  ionException("An Ostrich can not fly !");
    }
    ...
}
```

Refused Bequest

THE LSP VIOLATION SOLVED

```
public abstract class Bird
{
    public abstract void Eat();
}

public interface class IFlyBird
{
    public void Fly();
}
```

1. **Extract interface IFlyBird**
2. **Move Method Fly To IFlyBird**

```
public class Pelican: Bird, IFlyBird
{
    public void Fly()
    {
       Console.Writeline("Let’s fly …");
    }

    public void Eat()
    {
       Console.Writeline("Let’s eat …");
    }
}
```

```
public class Ostrich : Bird
{
    public void Eat()
    {
       Console.Writeline("Let’s eat …");
    }
}
```

42skillz

LSP IN DEPTH

If the class contains a state, LSP comes with more rules

Contract rules

- Pre conditions cannot be strengthened in subtype
- Post conditions cannot be strengthened in a subtype
- Invariants – conditions that must remain true for the hierarchy

Variance rules - Based on the variance of arguments and return type

- There must be contra variance of method argument in the subtype
- There must be covariance of the return type in the subtype
- No new exception can be thrown be the subtype

LISKOV SUBSTITUTION PRINCIPLE

If you …

Call a method on derived type gives some unexpected results

Get a bad type but still have to check what type is the actual type!

You might be breaking The LS Principle!

Interface Segregation

Tailor interfaces to individual clients' needs.

INTERFACE SEGREGATION PRINCIPLE

Clients should not be forced to depend upon interfaces that they do not use

The dependents are linked to these for looser coupling, increasing robustness, flexibility and the possibility of reuse

INTERFACE SEGREGATION PRINCIPLE

```
public class FileRepository
{
    public virtual void Write(id, string message)
    {
      // Write to file here
    }

    public virtual string Read(int id)
    {
       // Read from file here
    }

    public virtual FileInfo GetFileInfo(int id)
    {
      // Retrieve information from file
    }
}
```

EXTRACT INTERFACE

```
public interface IRepository
{
    void Write(id, string message);

    string Read(int id);

    FileInfo GetFileInfo(int id);
}
```

INTERFACE SEGREGATION PRINCIPLE?

```
public class SqlRepository : IRepository
{
    public void Write(id, string message)
    {
        // Write to database here
    }

    public string Read(int id)
    {
        // Read from database here
    }

    public FileInfo GetFileInfo(int id)
    {
        throw new NotImplementedException();
    }
}
```

Refused Bequest

42skillz

COMMAND QUERY SEGREGATION PRINCIPLE

This principle separate

- Command
  - modify data
- Query
  - retrieve data

COMMAND QUERY SEGREGATION PRINCIPLE

```
// Command
public interface IRepositoryWriter
{
    void Write(int id, string message);
}

// Query
public interface IRepositoryReader
{
    string Read(int id);
}
```

THE ISP VIOLATION SOLVED

```
public class FileStore : IStoreReader, IStoreWriter
{
     public void Write(id, string message)
     {
       // Write to file here
     }

     public string Read(int id)
     {
        // Read from file here
     }

     public FileInfo GetFileInfo(int id)
     {
       // Retrieve information from file
     }
}
```

42skillz

THE ISP VIOLATION SOLVED

```
public class SqlStore : IStoreReader, IStoreWriter
{
     public void Write(id, string message)
     {
       // Write to database here
     }

     public string Read(int id)
     {
        // Read from database here
     }
}
```

INTERFACE SEGREGATION PRINCIPLE

If ...

- interface has multiple responsibilities
- difficulties to expose a subset of responsibilities

You might be breaking The IS Principle!

DEPENDENCY INVERSION PRINCIPLE

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

DEPENDENCY INVERSION PRINCIPLE

The DIP makes two statements

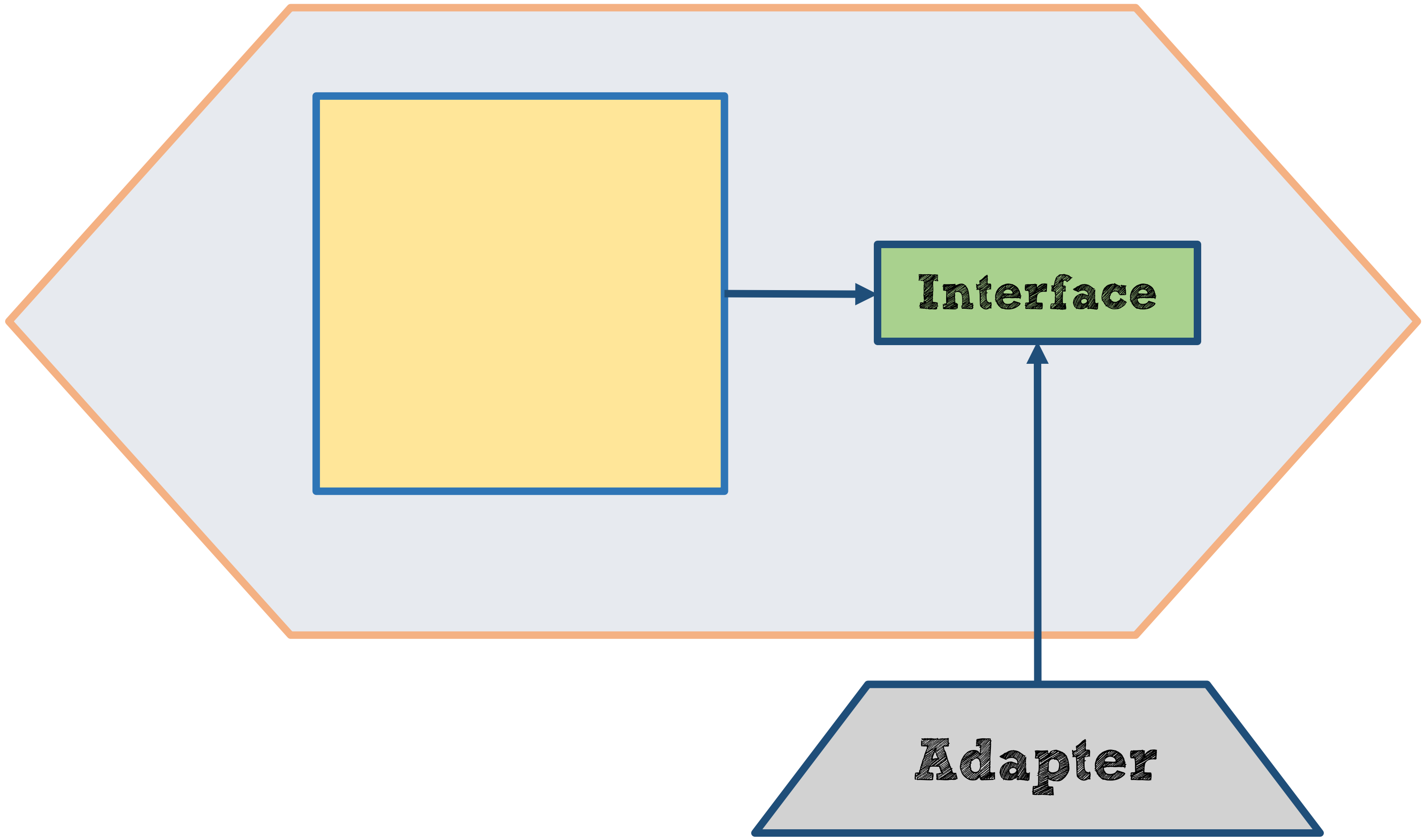
I) high level modules should not depend upon low level modules. Both should depend upon abstractions

II) the rule is that abstractions should not depend upon details. Details should depend upon abstractions

This reduces fragility caused by changes in low level modules introducing bugs in the higher layers

The DIP is often met with the use of dependency injection

DEPENDENCY INVERSION PRINCIPLE



DEPENDENCY INVERSION PRINCIPLE?

```
public class Alarm
{
  private const double LowPressureTreshold = 17;
  private const double HighPressureTreshold = 21;
  private readonly Sensor _sensor;

  public Alarm()
  {
    _sensor = new Sensor();
  }
  public void Check()
  {
    double pressure = _sensor.PopNextPressurePsiValue();

    if (pressure < LowPressureTreshold ||
        pressure > HighPressureTreshold)
    {
       AlarmOn = true;
    }
  }
}
```

Inappropriate Intimacy

42skillz

THE DIP SOLVED

```
public class Alarm
{
  private const double LowPressureTreshold = 17;
  private const double HighPressureTreshold = 21;
  private readonly ISensor _sensor;

  public Alarm(ISensor sensor)
  {
    _sensor = sensor;
  }
  public void Check()
  {
    double pressure = _sensor.PopNextPressurePsiValue();

    if (pressure < LowPressureTreshold ||
        pressure > HighPressureTreshold)
    {
        AlarmOn = true;
    }
  }
}
```

1. Extract Interface
2. Dependency Injection

42skillz

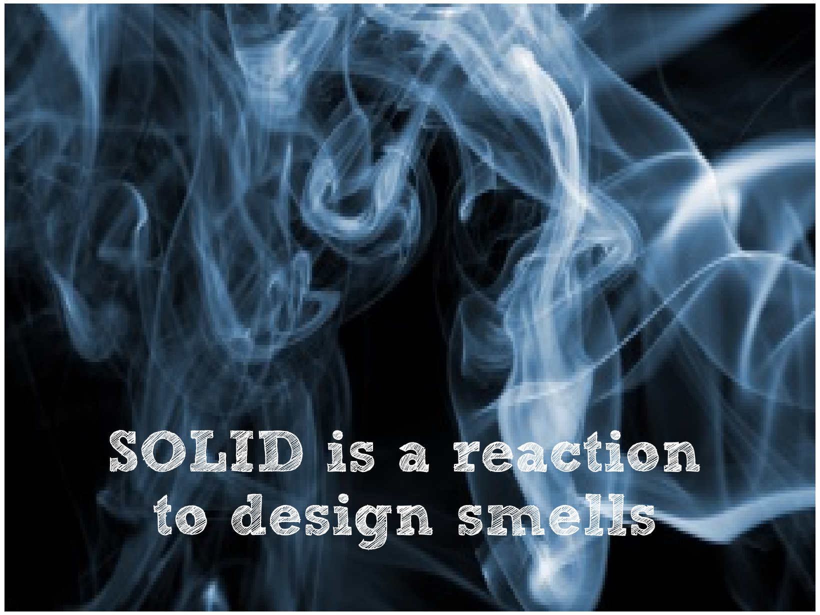
DEPENDENCY INVERSION PRINCIPLE

If you have …

- dependencies exist between classes
- a monolithic architecture

You might be breaking The DI Principle!

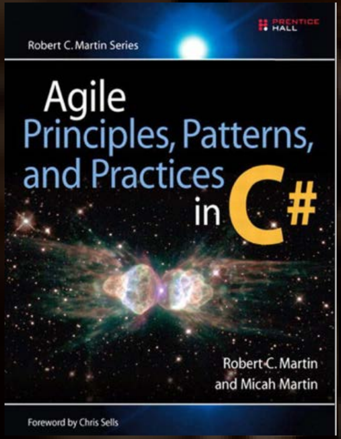
TO SUM-UP



SOLID violations are very close together

By design, Agility is friends with
SOLID Principles

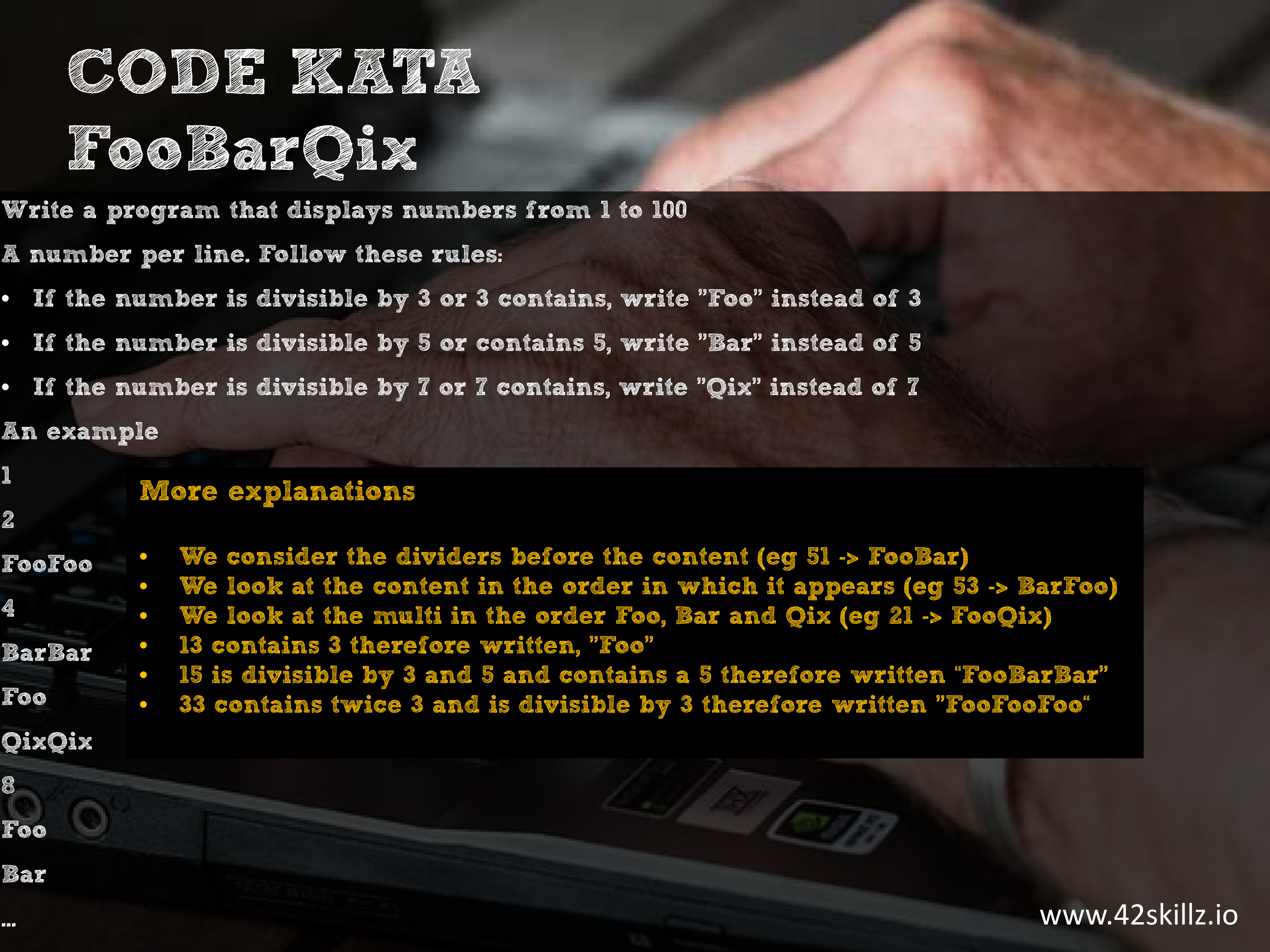
Design patterns are still useful


Robert C. Martin Series
PRENTICE HALL
Agile Principles, Patterns, and Practices in C#
Robert C. Martin and Micah Martin
Foreword by Chris Sells

Agile Principles, Patterns and Practices In C#

Robert Martin and Micah Martin

www.42skillz.io



CODE KATA FooBarQix

Write a program that displays numbers from 1 to 100

A number per line. Follow these rules:

- If the number is divisible by 3 or 3 contains, write "Foo" instead of 3
- If the number is divisible by 5 or contains 5, write "Bar" instead of 5
- If the number is divisible by 7 or 7 contains, write "Qix" instead of 7

An example

```
1
2
FooFoo
4
BarBar
Foo
QixQix
8
Foo
Bar
....
```

More explanations

- We consider the dividers before the content (eg 51 -> FooBar)
- We look at the content in the order in which it appears (eg 53 -> BarFoo)
- We look at the multi in the order Foo, Bar and Qix (eg 21 -> FooQix)
- 13 contains 3 therefore written, "Foo"
- 15 is divisible by 3 and 5 and contains a 5 therefore written "FooBarBar"
- 33 contains twice 3 and is divisible by 3 therefore written "FooFooFoo"