SOLID

Intro

Donatas Kukta

Kazimieras Slivka

These slides are based on ***C#: From Zero To Hero bootcamp - SOLID chapter***

Also checkout ***The C# Workshop: Kickstart your career as a software developer with C#***

You can find these slides at *https://github.com/DonatasKukta/SOLID*

What does SOLID stand for?

- **S**ingle-responsibility: *One reason to change*
- **O**pen–closed: *Open for extension, closed for modification*
- **L**iskov substitution: *Use derived classes without knowing it*
- **I**nterface segregation: *Fine grained client-specific interfaces*
- **D**ependency inversion: *Depend on abstractions, not concretions*

Why SOLID is important?

- Best practices to follow
- Avoid complexity headache
- Better code quality:
  - Improved readability
  - Better flexibility
  - Lower maintainability overhead
- Applicable for software in general, not just source code
- It is *industry-standard* and common question in interviews

Single Responsibility principle

A function, class or a module should have a single reason to change.

Just Because You Can, Doesn’t Mean You Should!

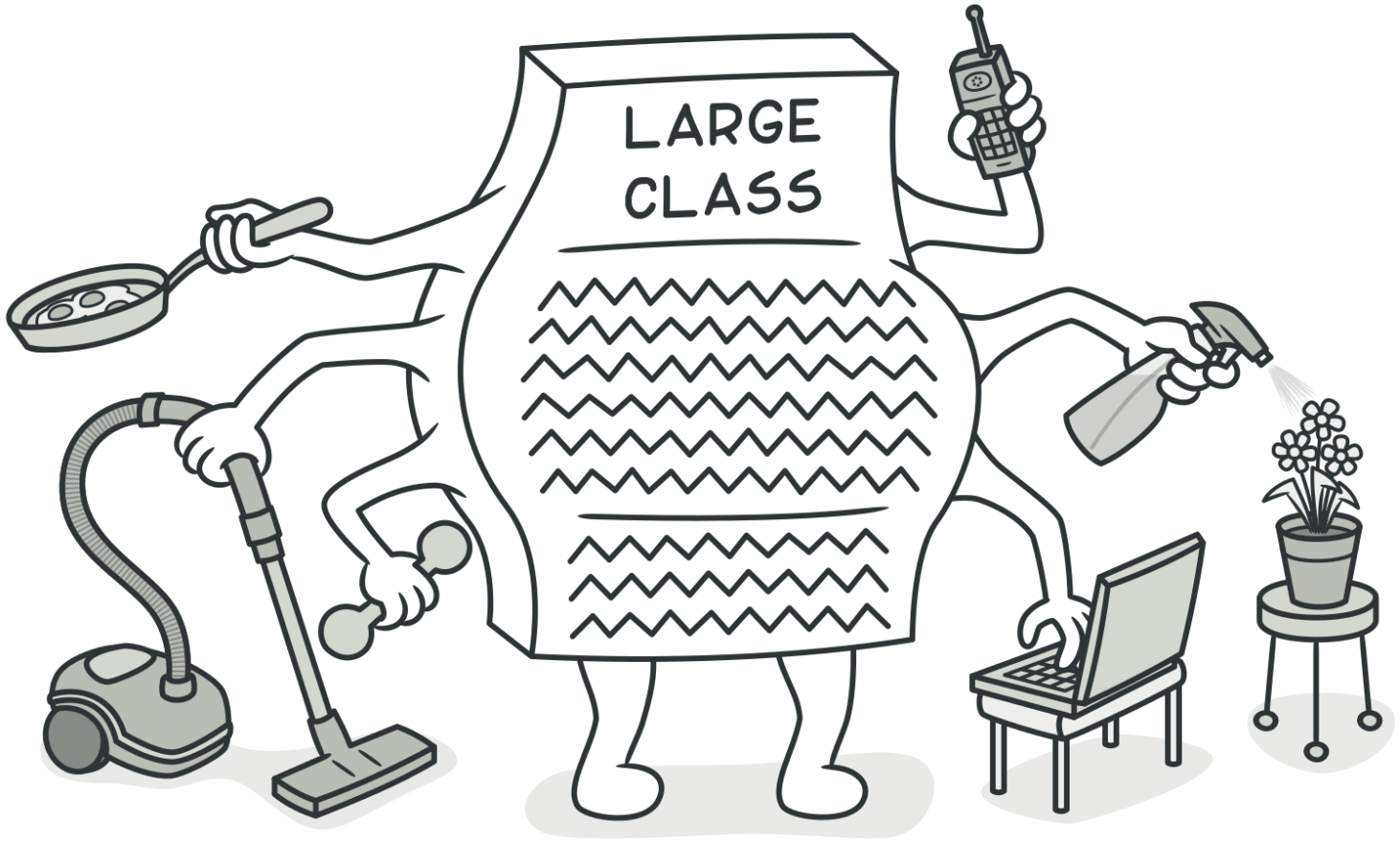
Single Responsibility principle

The God Class

- Obvious flow of behavior
- Structure is easy to understand
- It's easy to insert changes

Illusion vs reality

- Nobody understands how it works
- Everyone is afraid to change existing code
- Everyone just stich new code to the class
- Tech debt keeps growing



Single Responsibility principle. Examples

Bad

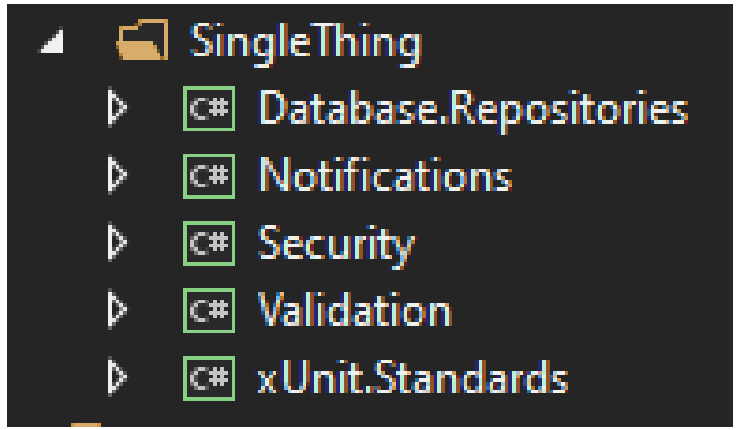
Good

Single Responsibility principle. Examples

Bad

- Unknown, abstract contents
- Semantical use (common things, third party things)
- Question: what about common infrastructure?

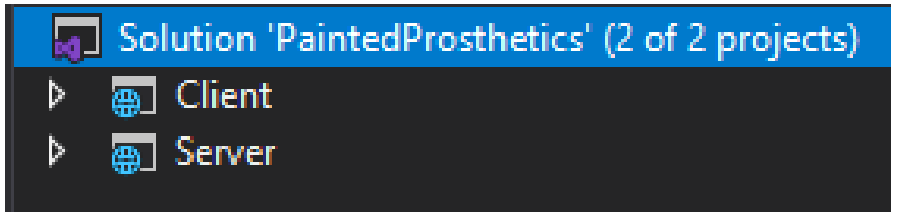
Good



- Clear contents
- Logical use (by project purpose)
- No questions

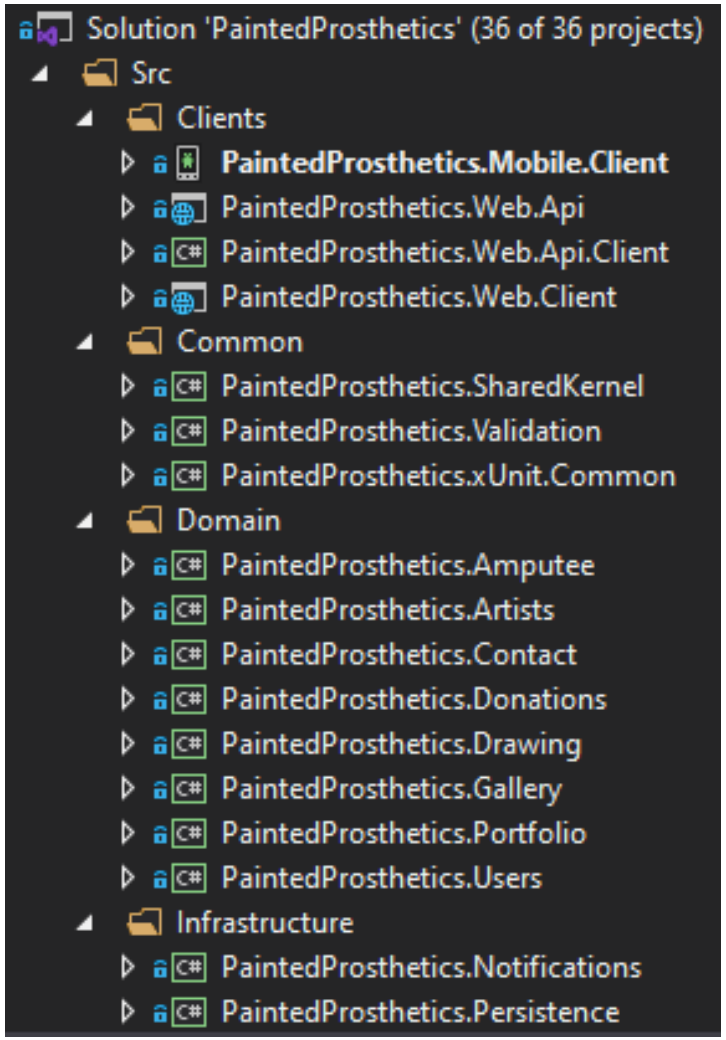
Single Responsibility principle. Examples

Bad



- For long term projects, it’s worth tackling complexity by splitting different system concerns into subsystems
- Developers should strive to minimize the things we need to care about at once.

Good



Bad

```
61      void stayButton_Click(object sender, EventArgs e)
62      {
63          while (true)
64          {
65              dealer.AddCard(deck.DrawCard());
66              dealer.GetCards();
67              showDealerScoreLabel.Text = dealer.Score.ToString();
68              showDealerCardsLabel.Text = dealer.GetCards();
69
70              if (dealer.Score > 21)
71              {
72                  showDealerScoreLabel.Text = dealer.Score.ToString();
73                  showDealerCardsLabel.Text = dealer.GetCards();
74                  var form = new EndScreen("YOU WIN!");
75                  form.ShowDialog();
76                  ResetState();
77                  return;
78              }
79              else if (dealer.Score > me.Score)
80              {
81                  showDealerScoreLabel.Text = dealer.Score.ToString();
82                  showDealerCardsLabel.Text = dealer.GetCards();
83                  var form = new EndScreen("YOU LOSE!");
84                  form.ShowDialog();
85                  ResetState();
86                  return;
87              }
88              else if (dealer.Score == me.Score && dealer.Score == 21)
89              {
90                  showDealerScoreLabel.Text = dealer.Score.ToString();
91                  showDealerCardsLabel.Text = dealer.GetCards();
92                  var form = new EndScreen("TIE!");
93                  form.ShowDialog();
94                  ResetState();
95                  return;
96              }
97          }
98      }
```

Better

```
61      readonly Dictionary<Outcome, string> outcomeMessages = new(){
62          {Outcome.PlayerWon, "YOU WIN!"},
63          {Outcome.DealerWon, "YOU LOOSE!"},
64          {Outcome.Tie, "TIE!"},
65      };
66
67      void stayButton_Click(object sender, EventArgs e)
68      {
69          while (true)
70          {
71              dealer.AddCard(deck.DrawCard());
72              // UI logic in it's own method
73              UpdateDealerLabels();
74              // Game logic in it's own method
75              var outcome = DetermineOutcome(dealer.Score, me.Score);
76
77              if(outcome == Outcome.None)
78                  continue;
79
80              // Outcomes and messages strongly coupled together
81              var message = outcomeMessages[outcome];
82              // UI logic in it's own methods
83              ShowEndScreen(message);
84              ResetState();
85              break;
86          }
87      }
```

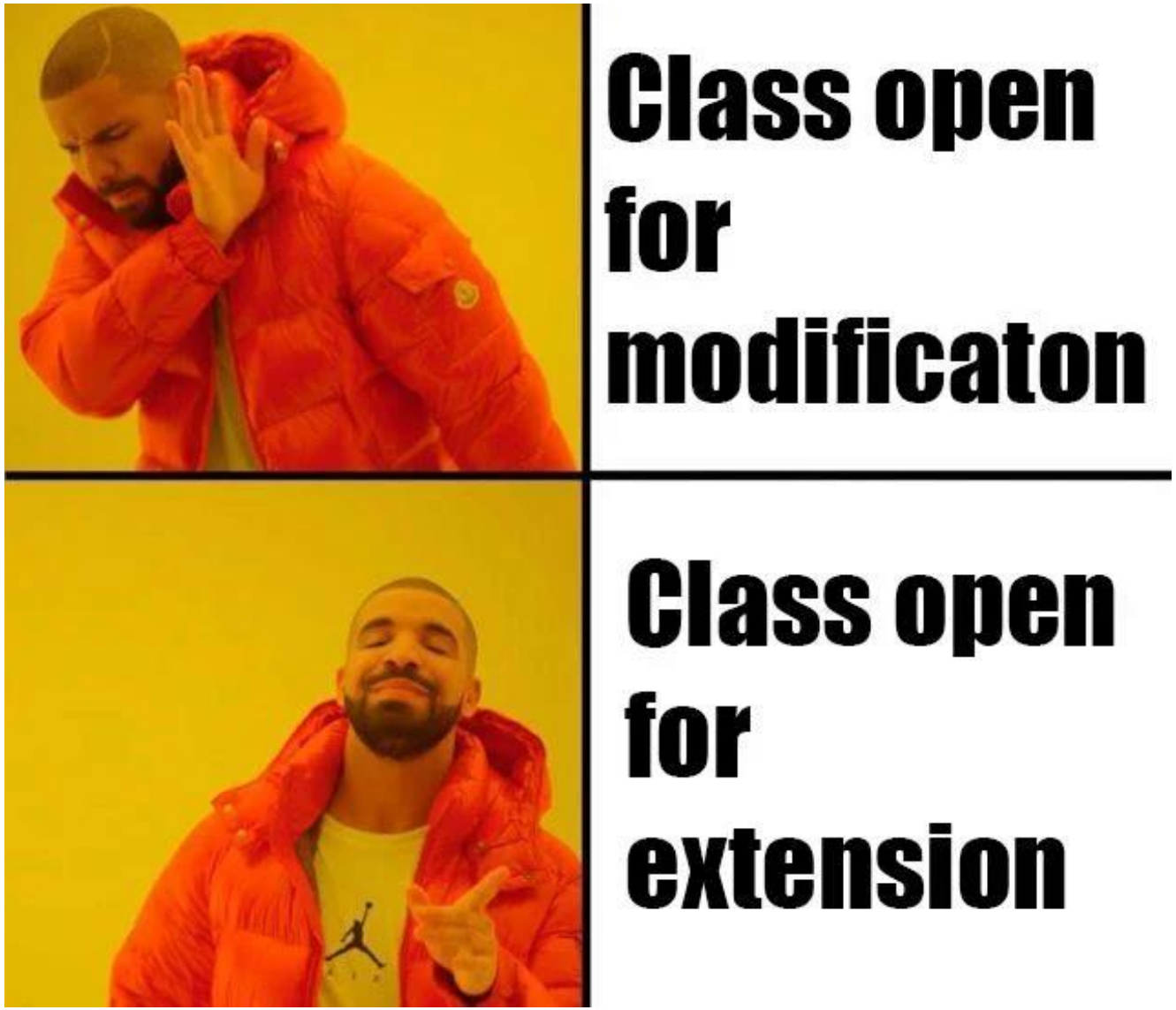
Open-Closed principle

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

Lights can be attached without disassembling the engine

Open-Closed principle

- Less changes in existing code
- Less chances of breaking anything with new functionality
- Easier to find different functionality
- New changes has isolated complexity (no accumulated complexity)



Open-Closed principle

How to add functionality without changing existing code?

- Accepting functions as parameters
- Using Extension Methods
- Using Inheritance
- Using Generics
- Using Composition

1

```
public static decimal CalculateTax(this Order order, decimal taxRate)
  => order.totalSum * taxRate;

void BillClient(Order order, decimal taxRate)
{
  var tax = order.CalculateTax(taxRate);
  // ...
}
```

2

```
interface ITaxCalculator { decimal CalculateTax(Order order); }
class LithuaniaTaxCalculator : ITaxCalculator…
class PolandTaxCalcualtor : ITaxCalculator…
void BillClient(Order order, ITaxCalculator taxCalculator)
{
  var tax = taxCalculator.CalculateTax(order);
  // ...
}
```

3

```
abstract class Order
{
  protected decimal TotalSum;
  public abstract decimal CalculateTax();
}
class OrderInLithuania : Order
{
  public override decimal CalculateTax() => TotalSum * 0.21;
}
class OrderInPoland : Order
{
  public override decimal CalculateTax() => TotalSum * 0.23;
}
void BillClient(Order order)
{
  var tax = order.CalculateTax();
  // ...
}
```

4

```
void BillClient(Order order, Func<decimal, decimal> calculateTax)
{
  var tax = calculateTax(order.totalSum);
  // ...
}
```

Open-Closed principle: Composition vs Inheritance

Inheritance

- Strongly couples Parent and Child
- New behavior is implemented in Child
- Child is stuck forever with one Parent
- Should not be used as a mean to reuse code
- Should be used as a mean to create logical hierarchies
- Deep inheritance (2+ levels) increases complexity
- *Usually try to* prefer composition over Inheritance ☺

```
abstract class Order
{
  protected decimal TotalSum;
  public abstract decimal CalculateTax();
}
class OrderInLithuania : Order
{
  public override decimal CalculateTax() => TotalSum * 0.21;
}
class OrderInPoland : Order
{
  public override decimal CalculateTax() => TotalSum * 0.23;
}
void BillClient(Order order)
{
  var tax = order.CalculateTax();
  // ...
}
```

Composition

- Classes refer to nested objects instead of inheriting them
- Class can depend on multiple other objects
- Should be used as a mean to reuse code
- Loosely coupled objects gives better flexibility
- Easier to write unit tests

```
interface ITaxCalculator { decimal CalculateTax(Order order); }
class LithuaniaTaxCalculator : ITaxCalculator…
class PolandTaxCalcualtor : ITaxCalculator…
void BillClient(Order order, ITaxCalculator taxCalculator)
{
  var tax = taxCalculator.CalculateTax(order);
  // ...
}
```

Open-Close Principle

Design patterns used with minimal changes in existing code:

- Decorator
- Strategy
- Factory Method
- Template Method

What other things could help?

Liskov Substitution principle

Objects of a superclass shall be replaceable with objects of its subclasses without breaking the application.

TLDR; *Use derived classes without knowing it.*

It looks like a duck, quacks like a duck, but it needs batteries!

You probably have the wrong abstraction.

Liskov Substitution principle

Any class that is the child of a parent class should be usable in place of its parent **without any unexpected behavior**

Subtyping: implementing a class which follows requirements specified by an interface.

Inheritance: implementing a Child class that specializes Parent class to a particular use, by reusing it's behavior and possibly overriding parts of it.

- Behavior reuse (or override) should be easy to justify
- Good for enforcing business logic, creating real world relations
- Allows creating restrictions that must be followed

What do you think about the example on the right in terms for enforcing business logic?

```
interface ILogger { void Log(string message); }
class SqlLogger : ILogger{
  public void Log(string message)
  { // What's wrong here?
    Database.Execute($"INSERT INTO Logs VALUES({message})");
  }
}
class FileLogger : ILogger{
  public void Log(string message)
  {
    File.Open();
    File.Append(message);
    File.Close();
  }
}
class ConsoleLogger : ILogger…
```

Use derived classes without knowing it.

```
abstract class Order
{
  protected decimal TotalSum;
  public abstract decimal CalculateTax();
}
class OrderInLithuania : Order
{
  public override decimal CalculateTax() => TotalSum * 0.21;
}
class OrderInPoland : Order
{
  public override decimal CalculateTax() => TotalSum * 0.23;
}
void BillClient(Order order)
{
  var tax = order.CalculateTax();
  // ...
}
```

Liskov Substitution principle

```
abstract class Bird
{
  public void LayEgg() { }
  public void Fly() { }
}
class Duck : Bird { }
class Eagle : Bird { }
class Penguin : Bird { }
class Ostrich : Bird { }
```

- Resembles real-life relation (they are all Birds)
- Gives restrictions that are followed (Lay an Egg and Fly)
- But what happens when Penguin or Ostrich tries to Fly?
- Parent and Child classes are not always interchangeable.

```
abstract class Bird
{
  public void LayEgg() { }
}
abstract class FlyingBird : Bird
{
  public void Fly() { }
}

class Duck : FlyingBird { }
class Eagle : FlyingBird { }
class Penguin : Bird { }
class Ostrich : Bird { }
```

- Resembles real-life relation (they are all Birds)
- Gives restriction that are followed (all can Lay an Egg and some can Fly)
- Parent and Child classes are always interchangeable.
- Be cautious about growing inheritance depth.

Interface Segregation principle

Fine-grained client-specific interfaces.

or

Clients should not be forced to depend on interfaces they don't use.

There is no combined power & water connector

Interface Segregation principle

- Avoid interface pollution
- A single interface for everything is too vague and different implementations or clients might not use all that it provides

```
interface IVehicle
{
    void Drive();
    void Fly();
    void FillTank();
    void Charge();
}
```

- How many unrelated responsibilities exist in this interface?
- What can we expect from the IVehicle?
- What happens to the consumer when new methods are added?

Interface Segregation principle: the problem

```
class Airplane : IVehicle
{
  public void Drive()
  {
    throw new InvalidOperationException("Airplane cannot drive");
  }
  public void FillTank()
  {
    // ...
  }
  public void Fly()
  {
    // ...
  }
  public void Charge()
  {
    throw new InvalidOperationException("This airplane has fuel engine");
  }
}
```

```
class Car : IVehicle
{
  public void Drive()
  {
    // ...
  }
  public void FillTank()
  {
    // ...
  }
  public void Fly()
  {
    throw new InvalidOperationException("Car cannot fly");
  }
  public void Charge()
  {
    throw new InvalidOperationException("This car has fuel engine");
  }
}
```

Interface Segregation principle: the solution

- Interfaces are split into logically separated parts
- Clients are allowed to use only what's needed for them
- Better distinction of responsibilities
- Easier to compose multiple interfaces together
- Flexible object usability

```
interface ICar
{
  void Honk();
  void Drive();
}
interface IAirplane
{
  void Fly();
}
interface IFuelEngine
{
  void FillTank();
}
interface IElectricEngine
{
  void Charge();
}

class Airplane : IAirplane, IFuelEngine { }
class Car : ICar, IFuelEngine { }
class Tesla : ICar, IElectricEngine { }
```

Dependency Inversion Principle

Depend on abstractions, not concrete implementations.

Excavator design doesn’t depend on attachments

The idea

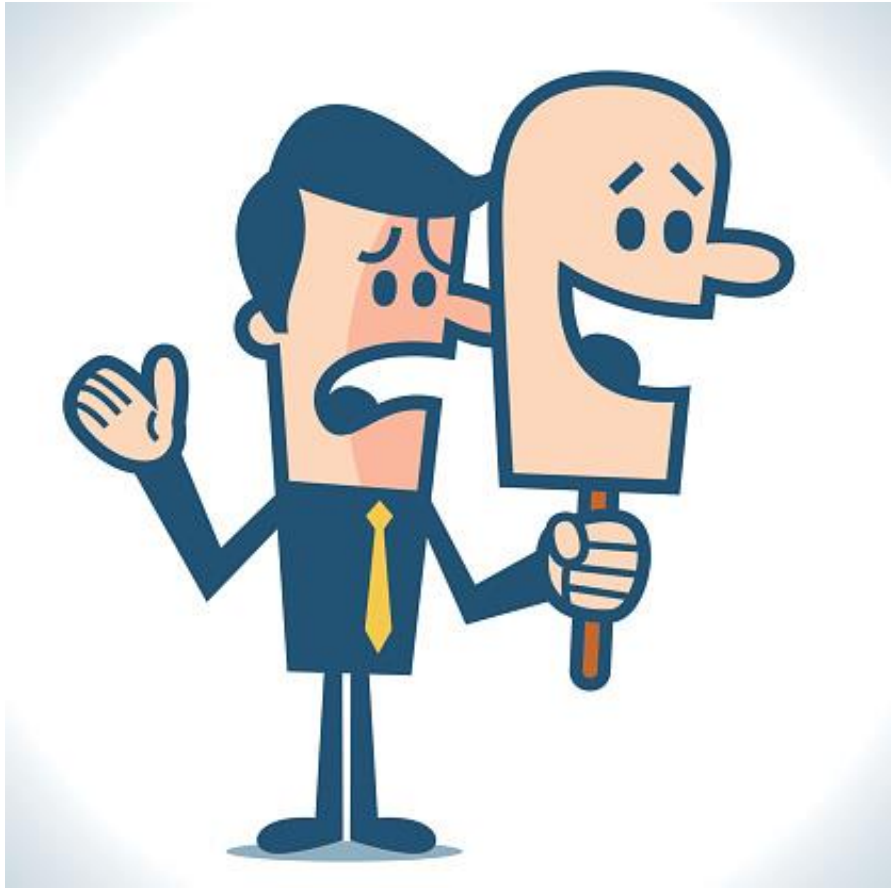
- High level modules should not depend on low level modules. Both should depend on abstractions.
- Abstractions should not depend on details. Details should depend on abstractions.
- Avoid hardcoding dependencies
- Use abstractions that your domain needs

DIP - Maintainability

- With DIP, we achieve simplicity (isolate the details)
- We get a living documentation (a contract that tells what a class can do)
- We don’t marry any framework-specific implementation (or abstraction), so if we need a change, we can do it without rewriting

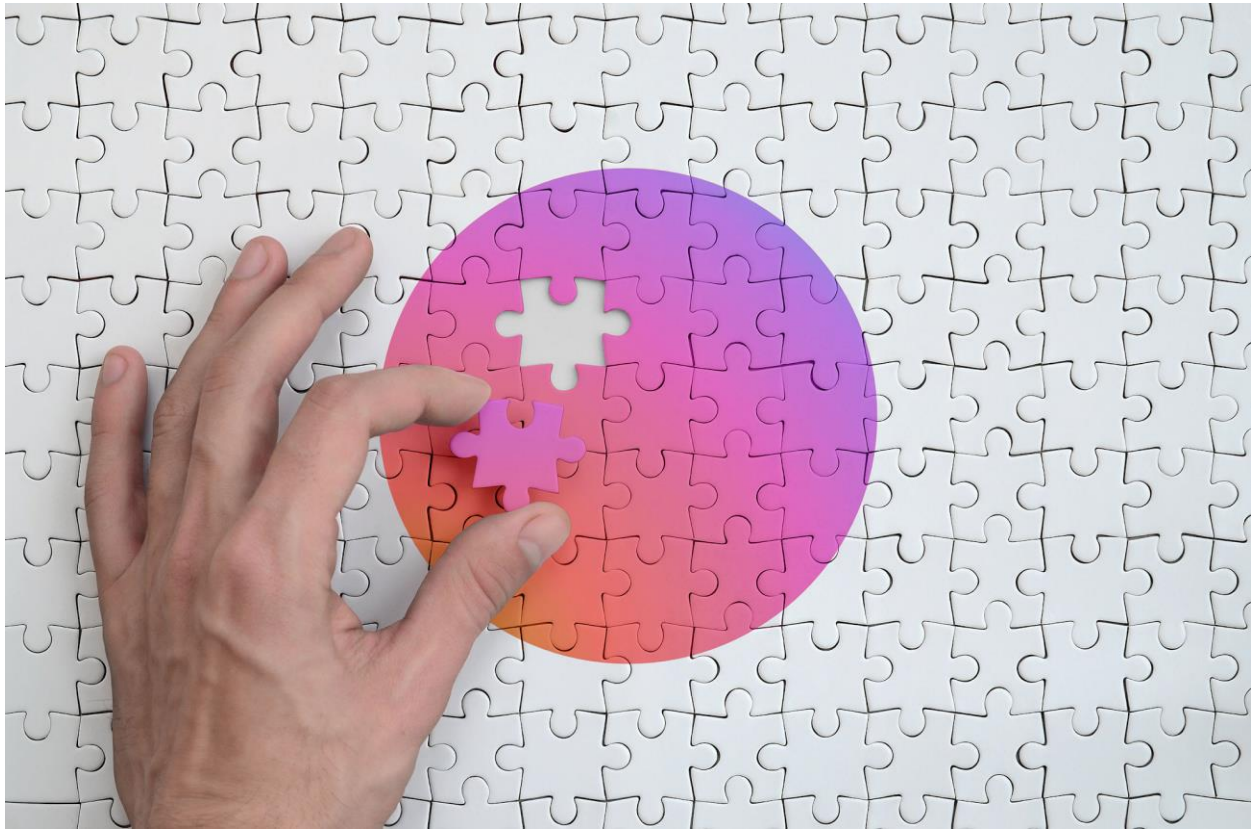
DIP - Testability

- It’s easy to mock abstractions (so we can write unit tests for whatever consumes them)
- If we weren’t depending on abstractions, we would need to create fake doubles in order to be able to test



DIP - Reusability

- Components (especially lower level) can be swapped
- We can swap system as a whole or a part of it



What are the ways of injecting external dependencies?

Constructor injection

- The most common way of achieving dependency inversion
- Expose abstractions in ctor

```
public class SchoolTerminal : ISchoolTerminal
{
    private readonly IStudentsRepository _studentsRepository;
    private readonly ITeachersRepository _teachersRepository;
    private readonly IGradesRepository _gradesRepository;
    private readonly ILessonClassesRepository _lessonClassesRepository;

    public SchoolTerminal(
        IStudentsRepository studentsRepository,
        ITeachersRepository teachersRepository,
        IGradesRepository gradesRepository,
        ILessonClassesRepository lessonClassesRepository)...
```

Method injection

- Expose abstractions in a method
- Way less popular than ctor injection

```
public class OrderService
{
    public void Create(Order order, ILogger logger)...
}
```

```
var service = new OrderService();
service.Create(new Order(), new Logger());
```

Property injection

- Expose abstraction in a property
- The least popular

```
public class OrderService
{
    public ILogger Logger { get; set; }

    public void Create(Order order)
    {
    }
}
```

```
var service = new OrderService();
service.Logger = new Logger();
```

Inversion Of Control

- In a single place
- At the startup of your application
- Create once (recommended)

```
public static ISchoolTerminal InitializeSchoolTerminal()
{
    var services = new ServiceCollection();

    services.AddTransient<ISchoolMemoryContext, SchoolMemoryContext>();
    services.AddTransient<ITeachersRepository, TeachersRepository>();
    services.AddTransient<IGradesRepository, GradesRepository>();
    services.AddTransient<ILessonClassesRepository, LessonClassesRepository>();
    services.AddTransient<IStudentsRepository, StudentsRepository>();

    services.AddTransient<ISchoolTerminal, SchoolTerminal>();

    var provider = services.BuildServiceProvider();

    return provider.GetService<ISchoolTerminal>();
}
```

Let’s get our terminology straight

- Dependency inversion
  - Last principle of SRP
  - Depend on abstractions not implementations!
- Dependency injection
  - A pattern that implements the principle
  - A way of exposing dependencies through ctor, method or a property
- Inversion of control
  - A way of managing dependencies in one place
  - It’s how dependencies are configured in Startup

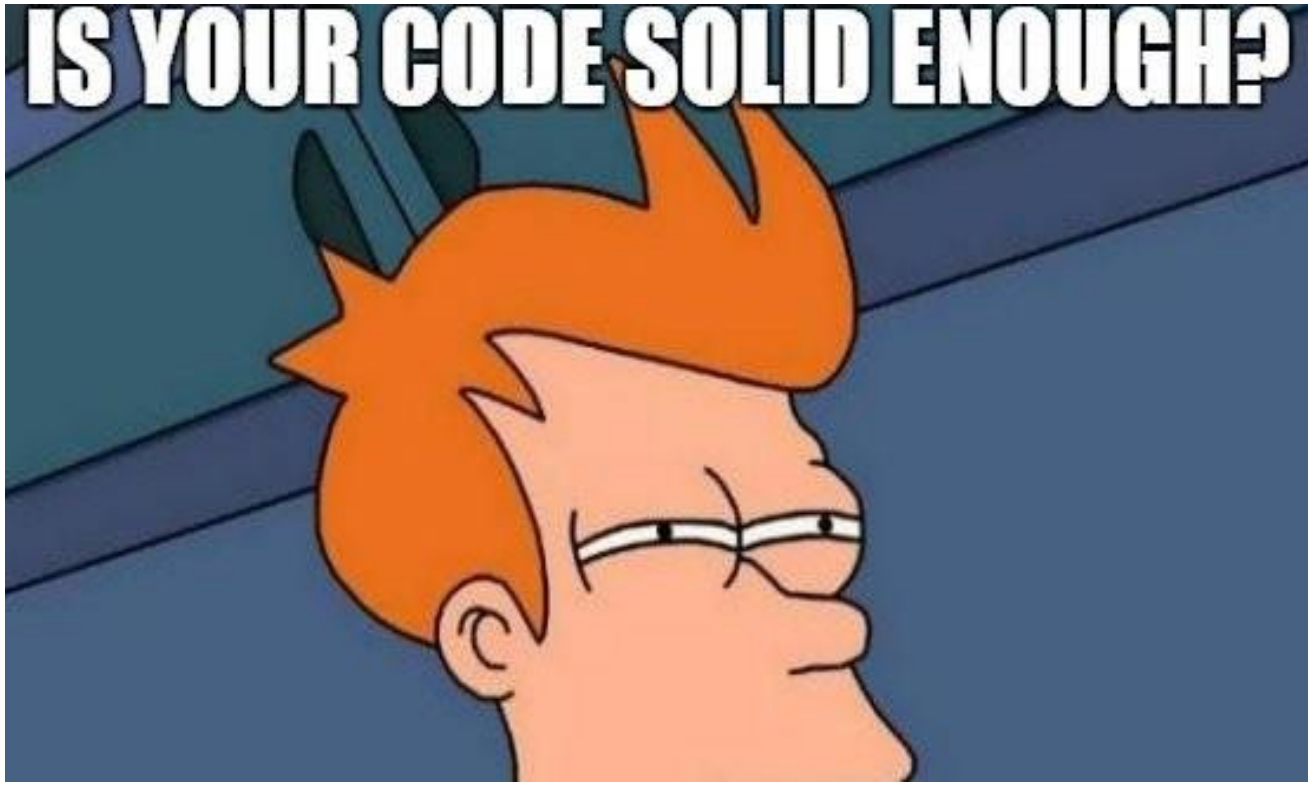
Tips to make your code SOLID!

- Try to find a balance between grouping and separating code/logic
- Prefer composition over inheritance
- Avoid long methods, classes and interfaces
- Use interfaces! Ensure they have clear reusable purpose
- Think about what needs to be done to extend the code in the future
- Assume that someone with less experience is reading your code
- Avoid hardcoding

Workshop time!

Please clone and open solution from

https://github.com/DonatasKukta/SnakeMultiplayer



Links for further reading

C#: From Zero To Hero

The C# Workshop: Kickstart your career as a software developer with C#

Gang of Four Design Patterns