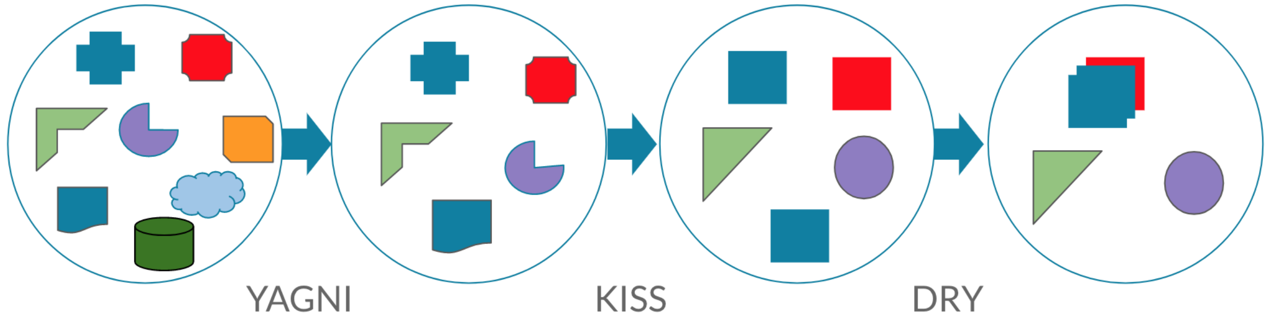
Make you code better

Simple rules and simple words

There are lots of acronyms such as

- **DRY / DIE** – Don’t repeat yourself / Duplication is Evil
- **SSOT** – single source of truth (store it once)
- **KISS** – keep it simple, stupid
- **YAGNI** – you aren’t gonna need it



Questions:

What are DRY and KISS?

What are YAGNI and SSOT?

S.O.L.I.D.

Letter	Meaning
S	Single responsibility principle
O	Open/closed principle
L	Liskov substitution principle
I	Interface segregation principle
D	Dependency inversion principle

Single responsibility

Class, method, module must take only one responsibility. There shouldn't be more than one reason to change a class.

So we can be sure that there is only one reason to change these parts of our code.

An extreme example of non-compliance with the principle ("Antipattern")

- “God object”, divine object - knows and knows too much
- unreasonable complexity, smearing logic

Single responsibility

Bad, so bad...

```
public class Employee
{
    public int Employee_Id { get; set; }
    public string Employee_Name { get; set; }

    public bool InsertIntoEmployeeTable(Employee em)
    {
        // Insert into employee table.
        return true;
    }
    public void GenerateReport(Employee em)
    {
        // Report generation with employee data using crystal report.
    }
}
```

Single responsibility

Not so bad…

```
public class Employee
{
    public int Employee_Id { get; set; }
    public string Employee_Name { get; set; }

    public bool InsertIntoEmployeeTable(Employee em)
    {
        // Insert into employee table.
        return true;
    }
}
public class ReportService
{
    …
    public void GenerateReport(Employee em)
    {
        // Report generation with employee data using crystal report.
    }
}
```

Single responsibility

Not so bad…

```
public class Employee
{
    public int Employee_Id { get; set; }
    public string Employee_Name { get; set; }

    public bool InsertIntoEmployeeTable(Employee em)
    {
        // Insert into employee table.
        return true;
    }
}
public class ReportService
{
    ...
    public void GenerateReport(Employee em)
    {
        // Report generation with employee data using crystal report.
    }
}
```

Right way

```
public class Employee
{
    public int Employee_Id { get; set; }
    public string Employee_Name { get; set; }
}
public class EmployeeEngine
{
    public bool InsertIntoEmployeeTable(Employee em)
    {
        // Insert into employee table.
        return true;
    }
}
public class ReportService
{
    public void GenerateReport(Employee em)
    {
        // Report generation with employee data using crystal report.
    }
}
```

Open/closed principle

Class must be closed for modification and can be opened for extension

- Closeness - stability of interfaces
- Openness - you can change behavior without class source code changes (rebuilds)

Extreme case of non-compliance

- complete lack of abstraction
- excessive number of levels of abstraction
  - "factory of factories"

Open/closed principle

BAD

```
public class ReportGeneration

{

    public string ReportType { get; set; }


    public void GenerateReport(Employee em)

    {

        if (ReportType == "CRS")

        {

            // Report generation with employee data in Crystal Report.

        }

        if (ReportType == "PDF")

        {

            // Report generation with employee data in PDF.

        }

    }

}
```

GOOD

```
public interface IReportGeneration
{
    public virtual void GenerateReport(Employee em);
}
/// <summary>Class to generate Crystal report</summary>
public class CrystalReportGeneraion : IReportGeneration
{
    public override void GenerateReport(Employee em)
    {
        // Generate crystal report.
    }
}
/// <summary>Class to generate PDF report</summary>
public class PDFReportGeneraion : IReportGeneration
{
    public override void GenerateReport(Employee em)
    {
        // Generate PDF report.
    }
}
```

Liskov substitution principle

Child class should no break parent class’s type definition and behavior. Child class’s instance can used as instance of parent class.

- Implementing "correct" inheritance
- Demand less, guarantee more

Extreme case: incomprehensible inheritance (either too much or not at all) and therefore non-obvious behavior

Liskov substitution principle

Child class should no break parent class's type definition and behavior. Child class's instance can used as instance of parent class.

```
public abstract class Employee
{
    public virtual string GetProjectDetails(int employeeId)
    {
        return "Base Project";
    }
    public virtual string GetEmployeeDetails(int employeeId)
    {
        return "Base Employee";
    }
}
public class CasualEmployee : Employee
{
    public override string GetProjectDetails(int employeeId)
    {
        return "Child Project";
    }
    public override string GetEmployeeDetails(int employeeId)
    {
        return "Child Employee";
    }
}
```

```
public class ContractualEmployee : Employee
{
    public override string GetProjectDetails(int employeeId)
    {
        return "Child Project";
    }

    public override string GetEmployeeDetails(int employeeId)
    {
        throw new NotImplementedException();
    }
 }
```

```
List<Employee> employeeList = new List<Employee>();
employeeList.Add(new ContractualEmployee());
employeeList.Add(new CasualEmployee());
foreach (Employee e in employeeList)
{
    e.GetEmployeeDetails(1245);
}
```

Liskov substitution principle

Child class should no break parent class's type definition and behavior. Child class's instance can used as instance of parent class.

```
public abstract class Employee
{
    public virtual string GetProjectDetails(int employeeId)
    {
        return "Base Project";
    }
    public virtual string GetEmployeeDetails(int employeeId)
    {
        return "Base Employee";
    }
}
public class CasualEmployee : Employee
{
    public override string GetProjectDetails(int employeeId)
    {
        return "Child Project";
    }
    public override string GetEmployeeDetails(int employeeId)
    {
        return "Child Employee";
    }
}
```

```
public class ContractualEmployee : Employee
{
    public override string GetProjectDetails(int employeeId)
    {
        return "Child Project";
    }
}
```

```
List<Employee> employeeList = new List<Employee>();
employeeList.Add(new ContractualEmployee());
employeeList.Add(new CasualEmployee());
foreach (Employee e in employeeList)
{
    e.GetEmployeeDetails(1245);
}
```

Interface segregation principle

KISS is its brother!

Better to have lots of simple interfaces than one complex and large.

In this case you don’t need to implement or stub useless methods.

Extreme cases:

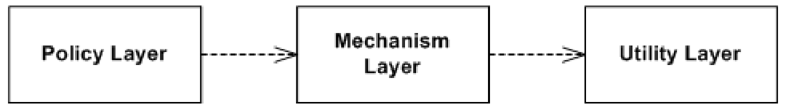
- “Fat" interface: all functional capabilities in one interface (customers separated, but the interface is not)
- a thousand interfaces (interface per method)

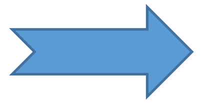
Dependency inversion principle

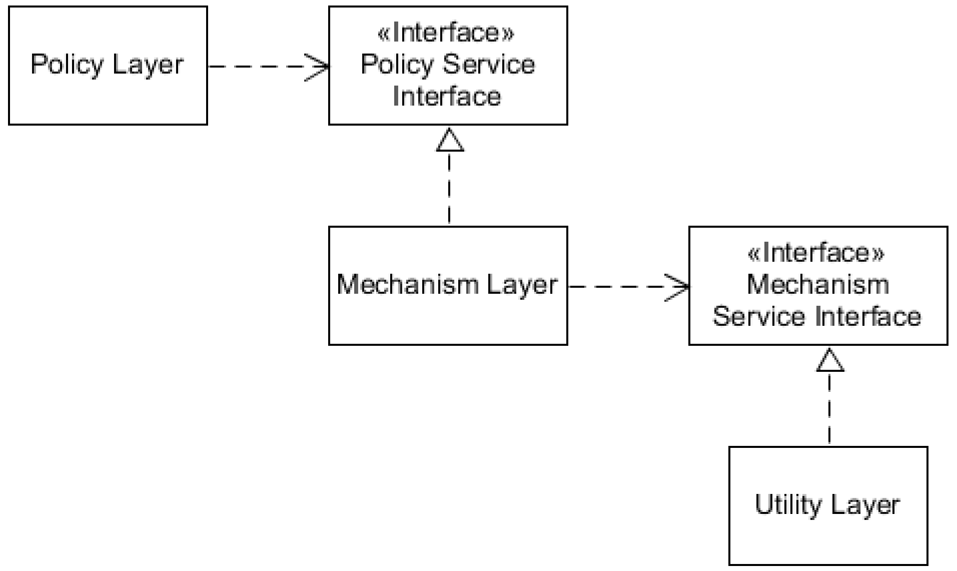
Abstractions should not depend on details.

Details should depend on abstractions.

SUM: Write loosely coupled code.






Policy Layer
«Interface» Policy Service Interface
Mechanism Layer
«Interface» Mechanism Service Interface
Utility Layer

Dependency inversion principle

```
public class Email
{
    public void SendEmail()
    {
        // code to send mail
    }
}
public class Notification
{
    private Email _email;
    public Notification()
    {
        _email = new Email();
    }
    public void PromotionalNotification()
    {
        _email.SendEmail();
    }
}
```

Dependency inversion principle

```
public interface IMessenger
{
    void SendMessage();
}
public class Email : IMessenger
{
    public void SendMessage()
    {
        // code to send mail
    }
}
public class Sms : IMessenger
{
    public void SendMessage()
    {
        // code to send mail
    }
}
```

Dependency inversion principle

```
public interface IMessenger
{
    void SendMessage();
}
public class Email : IMessenger
{
    public void SendMessage()
    {
        // code to send mail
    }
}
public class Sms : IMessenger
{
    public void SendMessage()
    {
        // code to send mail
    }
}
```

```
public class Notification
{
    private IMessenger _messenger;

    public Notification(IMessenger messenger)
    {
        this._messenger = messenger;
    }

    public void PromotionalNotification()
    {
        this._messenger.SendMessage();
    }
}
```

Questions

1. What means S?
2. What means O?
3. What means L?
4. What means I?
5. What means D?

GRASP

This is abbreviation means General Responsibility Assignment Software Patterns.

List of them:

- **Information expert**
- **Controller**
- **Creator**
- **Low coupling**
- **High cohesion**

- Indirection
- Polymorphism
- Protected variations
- Pure fabrication

GRASP - Information expert

Most important principle here.

It helps us to find the way we need to use for perform this operation.

Information expert (also expert or the expert principle) is a principle used to determine where to delegate responsibilities.

```
public class Customer : Entity, IAggregateRoot
{
    private readonly List<Order> _orders;
    public GetOrdersTotal(Guid orderId)
    {
        return this._orders.Sum(x => x.Value);
    }
}
```

GRASP - Creator

Class have to creating instances of classes which it can initialize, write (anywhere), use and store inside. It similar to abstract factory pattern.

```
public class Customer : Entity, IAggregateRoot
{
    private readonly List<Order> _orders;

    public void AddOrder(List<OrderProduct> orderProducts)
    {
        var order = new Order(orderProducts); // Creator

        if (this._orders.Count(x => x.IsOrderedToday()) >= 2)
            throw new BusinessRuleValidationException("You cannot order more than 2 orders on the same day");

        this._orders.Add(order);
        this.AddDomainEvent(new OrderAddedEvent(order));
    }
}
```

GRASP - Controller

A controller object is a non-user interface object responsible for receiving or handling a system event.

May be responsible for more then one use-case

```
public class UserController : Controller
{
    public IActionResult GetUser(int userId)
    {
        // return user
    }

    public IActionResult CreateUser(UserData userData)
    {
        // create user
    }
}
```

GRASP - Low coupling

Coupling is a measure of how strongly one element is connected to, has knowledge of, or relies on other elements.

GRASP - High cohesion

High cohesion is an evaluative pattern that attempts to keep objects appropriately focused, manageable and understandable. High cohesion is generally used in support of low coupling.

High cohesion means that the responsibilities of a given element are strongly related and highly focused.

GRASP - Indirection

The indirection pattern supports low coupling (and reuse potential) between two elements by assigning the responsibility of mediation between them to an intermediate object.

An example of this is the introduction of a controller component for mediation between data (model) and its representation (view) in the model-view-controller pattern (MVC)

GRASP - Polymorphism

According to polymorphism principle, responsibility of defining the variation of behaviors based on type is assigned to the type for which this variation happens.

GRASP - Protected variations

The protected variations pattern protects elements from the variations on other elements (objects, systems, subsystems) by wrapping the focus of instability with an interface and using polymorphism to create various implementations of this interface.

GRASP - Pure fabrication

A pure fabrication is a class that does not represent a concept in the problem domain, specially made up to achieve low coupling, high cohesion, and the reuse potential thereof derived

Questions

List GRASP principles.

List which of them you understand ☺