# System Design

## DRY

## SOLID

Good reference from MSDN: [Link](https://learn.microsoft.com/en-us/archive/msdn-magazine/2014/may/csharp-best-practices-dangers-of-violating-solid-principles-in-csharp" \l "the-single-responsibility-principle)

### Single Responsibility Principle

An object should only have one reason to change

*public IList<IList<Nerd>> ComputeNerdClusters(*

*List<Nerd> nerds,*

*IPlotter plotter = null) {*

*...*

*foreach (var nerd in nerds) {*

*...*

*if (plotter != null)*

*plotter.Draw(nerd.Location,*

*Brushes.PeachPuff, radius: 10);*

*...*

*}*

*...*

*}*

The above code 1. Computes nerd cluster and 2. Draws the cluster.

### Open-Closed Principle

Open for extension, closed for modification. Ability to add new functionality without changing existing code.

*void DrawNerd(Nerd nerd) {*

*if (nerd.IsSelected)*

*DrawEllipseAroundNerd(nerd.Position, nerd.Radius);*

*if (nerd.Image != null)*

*DrawImageOfNerd(nerd.Image, nerd.Position, nerd.Heading);*

*if (nerd is IHasBelt) // a rare occurrence*

*DrawBelt(((IHasBelt)nerd).Belt);*

*// Etc.*

*}*

**Corrected Code**

*readonly IList<IRenderer> \_renderers = new List<IRenderer>();*

*void Draw(Nerd nerd)*

*{*

*foreach (var renderer in \_renderers)*

*renderer.DrawIfPossible(\_context, nerd);*

*}*

**Other common rules**

Base classes should never directly reference their inheritors

### Liskov Substitution Principle

Defines some guidelines for maintaining inheritor substitution. Objects of a superclass should be able to be replaced with objects of a subclass without affecting the correctness of the program.

### Interface Segregation Principle

Each interface should have a specific purpose. You shouldn’t be forced to implement an interface when your object doesn’t share that purpose

### Dependency Inversion Principle

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.

**// Violating Example - Dependency Inversion Principle**

public class BusinessLogic

{

private DataAccess \_dataAccess;

public BusinessLogic()

{

\_dataAccess = new DataAccess();

}

public void PerformBusinessLogic()

{

// Use the data access layer directly

\_dataAccess.GetData();

// Perform business logic

}

}

public class DataAccess

{

public void GetData()

{

// Retrieve data from the database

}

}

**// Corrected Example - Dependency Inversion Principle**

public interface IDataAccess

{

void GetData();

}

public class DataAccess : IDataAccess

{

public void GetData()

{

// Retrieve data from the database

}

}

public class BusinessLogic

{

private IDataAccess \_dataAccess;

public BusinessLogic(IDataAccess dataAccess)

{

\_dataAccess = dataAccess;

}

public void PerformBusinessLogic()

{

// Use the data access layer via the interface

\_dataAccess.GetData();

// Perform business logic

}

}

## Microservices

## Kubernates

## Docker

## High Level

## Low Level