# System Design

[System Design 1](#_Toc18419)

[DRY 2](#_Toc18111)

[SOLID 2](#_Toc9427)

[Single Responsibility Principle 2](#_Toc19998)

[Open-Closed Principle 3](#_Toc20181)

[Liskov Substitution Principle 3](#_Toc16730)

[Interface Segregation Principle 4](#_Toc3201)

[Dependency Inversion Principle 4](#_Toc217)

[Design Pattern 4](#_Toc10397)

[Creational Patterns 4](#_Toc20571)

[Factory Method 5](#_Toc13234)

[Abstract Factory 5](#_Toc4840)

[Builder 5](#_Toc20163)

[Prototype 5](#_Toc28458)

[Singleton 5](#_Toc12477)

[Structural Patterns 5](#_Toc26525)

[Adapter 5](#_Toc25704)

[Bridge 5](#_Toc3384)

[Composite 5](#_Toc27361)

[Decorator 6](#_Toc26759)

[Facade 6](#_Toc3814)

[Flyweight 6](#_Toc21569)

[Proxy 6](#_Toc22395)

[Behavioral Patterns 6](#_Toc7647)

[Chain of Responsibility 7](#_Toc15532)

[Command 7](#_Toc9166)

[Iterator 7](#_Toc6012)

[Mediator 7](#_Toc9762)

[Memento 7](#_Toc20041)

[Observer 7](#_Toc26109)

[State 7](#_Toc2331)

[Strategy 7](#_Toc20736)

[Template method 7](#_Toc21958)

[Visitor 7](#_Toc8580)

[Microservices 7](#_Toc8647)

[Kubernates 7](#_Toc24949)

[Docker 7](#_Toc24633)

[High Level 7](#_Toc27892)

[Low Level 7](#_Toc9312)

## MVC

## MVVM

## 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. New functionality should be implemented by adding new classes, attributes and methods instead of changing the current ones.

*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);*

*}*

### Liskov Substitution Principle

S is a subtype of T, then objects of type T may be replaced with objet of type S

Derived type must be completely substitutable for their base types

It’s a definition of a subtyping relation, called (string) behavioral subtyping.

Example in github

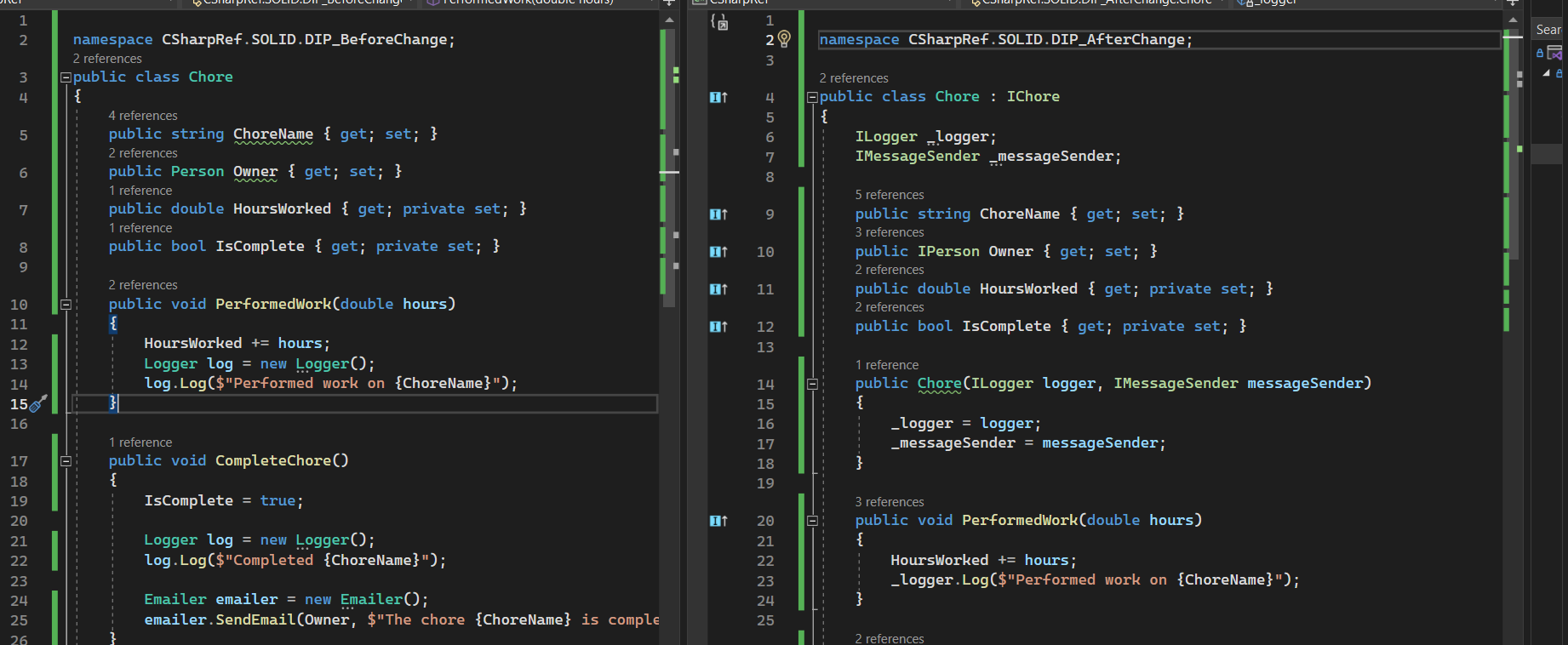
### Interface Segregation Principle

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.

Example in github



## Design Pattern

### Creational Patterns

Creational design patterns abstract the instantiation process. These patterns provide various object creation mechanisms, which increase flexibility and reuse of existing code.

#### Factory Method

Provides an interface for creating objects in a superclass, but allows subclasses to alter the type of objects that will be created.

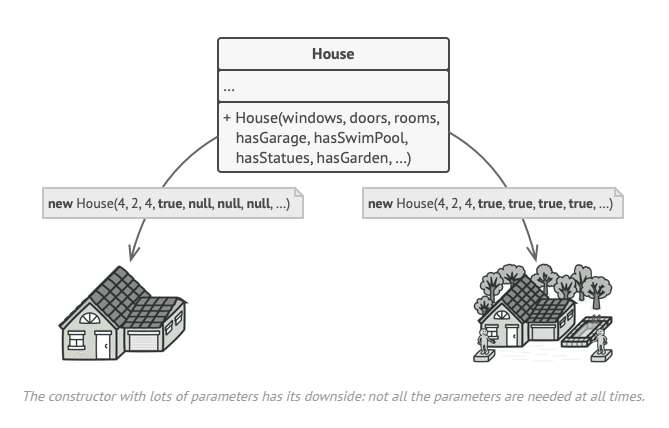
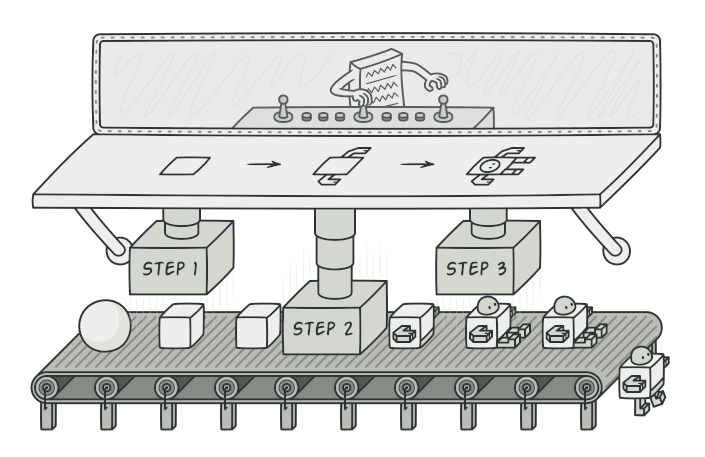
#### Abstract Factory

Produce families of related objects without specifying their concrete classes.

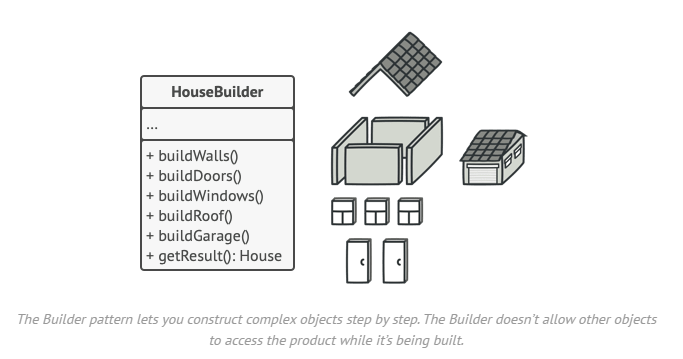
#### Builder

Builder is a creational design pattern that lets you construct complex objects step by step. The pattern allows you to produce different types and representations of an object using the same construction code.

Problem



Solution



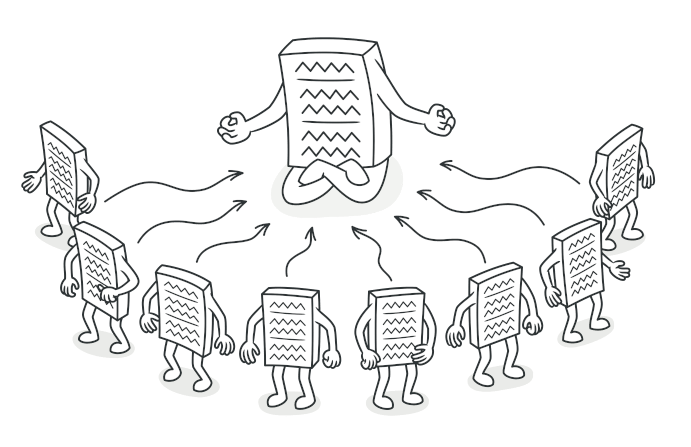
#### Prototype

A partially or fully initialized object that you copy (clone) and make use of. Some of the ways to copy are,

1. ICloneable - inbuild interface
2. Copy constructor
3. Explicit deep copy interface
4. Prototype inheritance
5. Serialization (best way)
   1. Only concern is relevant models should be marked with [Serializable] attribute if binary serialization is used
      1. Adding new variables or removing old variables
         1. Refer the C2V approach

#### Singleton

Component which is instantiated only once, while providing a global access point to this instance.



1. Lazy<>
2. Use ThreadLocal<T> to create a singleton instance for each thread
3. Use DI to inject the singleton instance to all the models
4. Define the singleton lifetime using DI container

### Structural Patterns

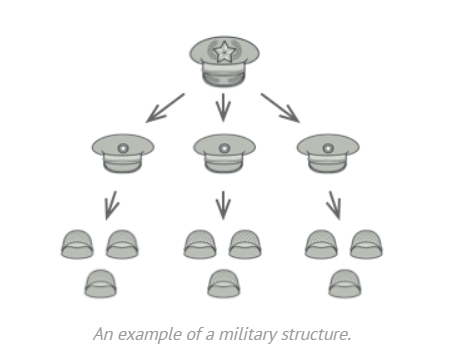
These patterns explain **how to assemble** objects and classes into larger structures while keeping these structures flexible and efficient.

#### Adapter

#### Bridge

#### Composite

Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly. Can be done only if core model of app can be represented in tree format.



#### Decorator

#### Facade - Mugappu

Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higherlevel interface that makes the subsystem easier to use.

#### Flyweight

#### Proxy

### Behavioral Patterns

These patterns are concerned with **algorithms and the assignment of responsibilities between objects**.

#### Chain of Responsibility

#### Command

#### Interpreter

#### Iterator

#### Mediator

#### Memento

#### Observer

#### State

#### Strategy

#### Template method

#### Visitor

## Microservices

## Kubernates

## Docker

## High Level

## Low Level