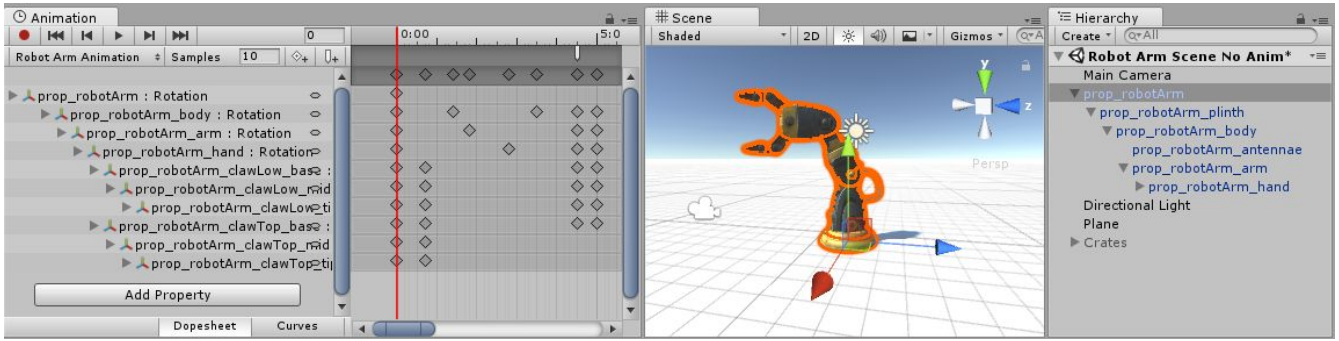
VIA University College

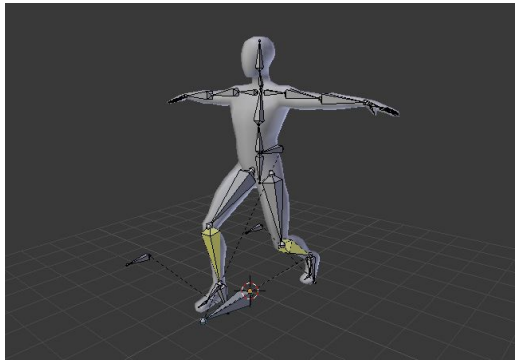
Game Architecture

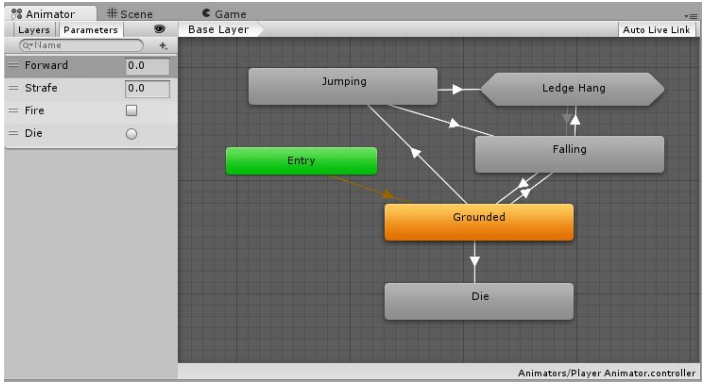
Writing maintainable and extendable code

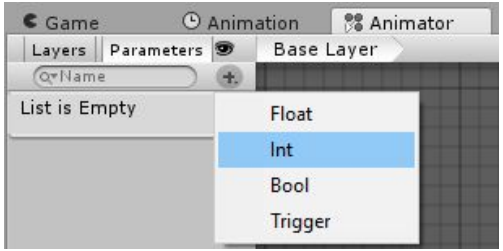
- SOLID Design Principles
- Game Design Patterns
- ScriptableObjects & Persistence
- Exercises

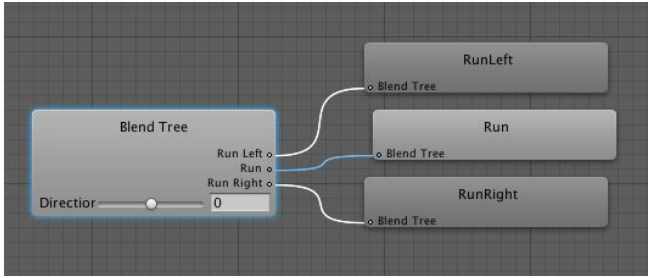
Last Week - What did we learn?











What is Architecture?

What is Architecture?

SOLID?

Unity comes with some architecture out of the box

- An Update loop
- GameObject hierarchy
- Components
- Event Functions
- Serializable objects (Prefabs, ScriptableObjects)

Why Architecture?

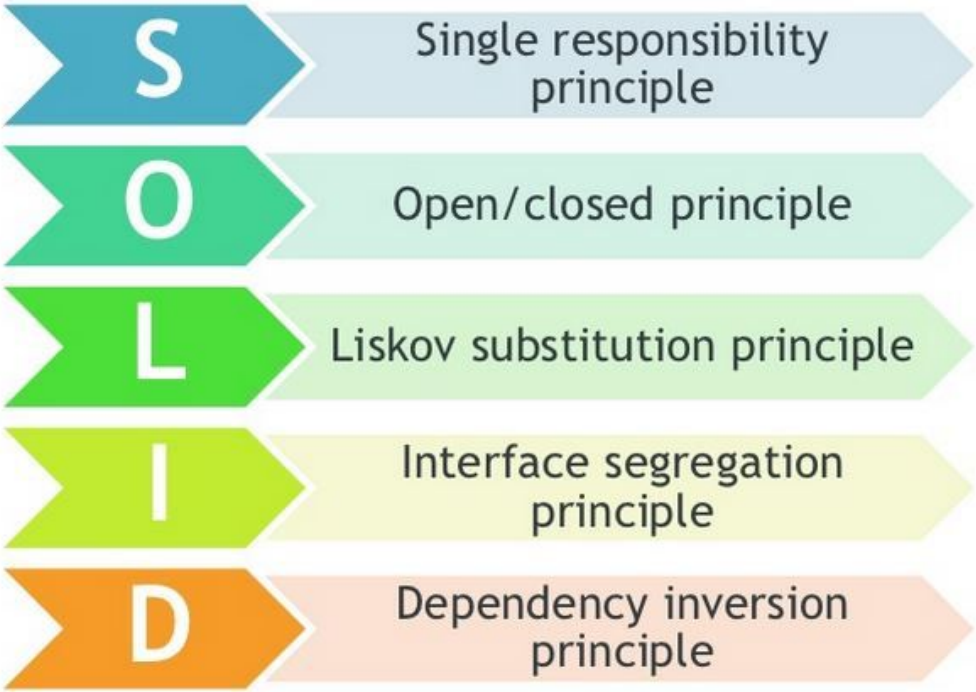
SOLID?

Architect better Unity applications that are:

- Modular → **easy to extend**
- Editable → **easy to change**
- Debuggable → **easy to test**

What is SOLID?

SOLID is an acronym for the first five object-oriented design principles by Robert C. Martin



Single Responsibility Principle

"A class should have only one reason to change."

I.e. a class should have only **one job**.

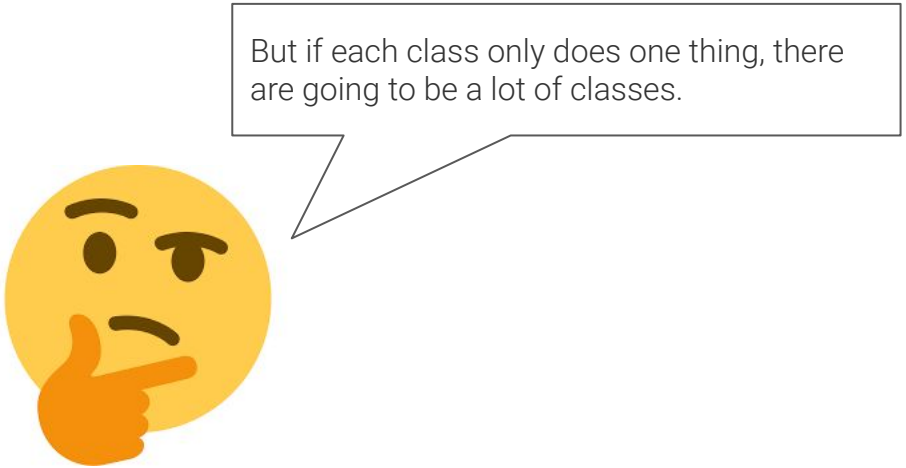
In Unity: a component should be responsible for a single piece of functionality

Video Demonstration Understanding SRP

Example

A weapon class should know nothing about the UI system.

Inversely, a WeaponAmmoUI class shouldn't need to know anything about how weapons work, and should instead ONLY work on the UI.



If you follow SRP, you'll end up with a <u>large number of very **small classes**</u>

The Alternative

Consider the alternative to SRP...

A very small number of **giant classes**.

… Or you could even go to an extreme and just have **one master class** that runs your entire game! (disclaimer: *don't* do this)

<u>Don't</u> create a GameManager God class!

If your classes are several hundreds lines of code long, warning bells should go off!

[The God Object anti-pattern]

Unity Does it!

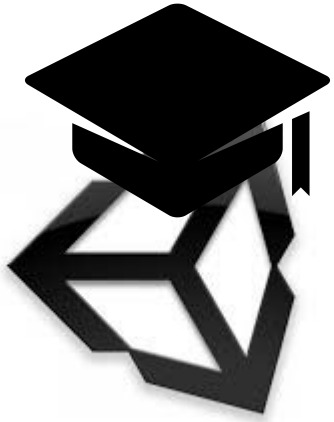
The Unity components already demonstrate the SRP principle!

AudioSource? **One responsibility** - to play audio.

=> Audio is NOT played through a more general entity.

The same goes for any other component (Renderer, Transform, BoxCollider, Rigidbody, etc..) - they all do **ONE THING** and they do it well.

Complex behaviour often involves interaction between components.



Benefits

Splitting up your logic into classes specifically responsible for one thing provides many great benefits:

Readability – Classes are easy to keep between 20-100 lines when they correctly follow SRP.

Extensibility – Small classes are easy to inherit from, modify, or replace.

Re-usability – If your class does one thing and does that thing well, it can do that thing for other parts of your game.

How to SRP

Single Responsibility Principle

- Take your class apart! Split, split, split!
- Make your class do one thing. If you can't describe it without "and" or "or" you are probably doing it wrong.
- Use events, polling or interfaces to communicate between the decoupled components.

Single Responsibility Principle

Open/Closed Principle

"Entities should be open for extension, but closed for modification."

When you **DON'T** follow the Open/Closed principle:

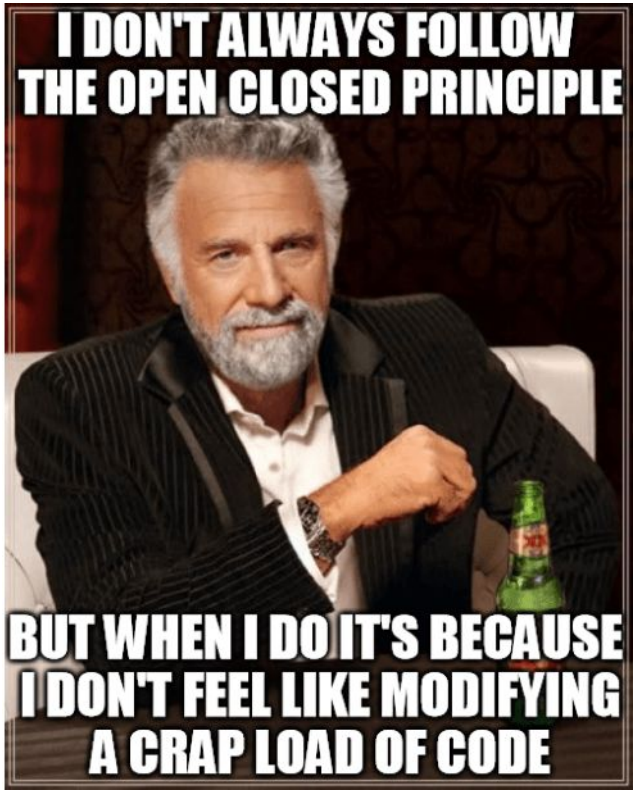
- Adding new functionality requires you to modify your existing classes.
- Your class is handling different inputs in different ways to the same method.
- You start to see if & else if statements cluttering your code.

Video Demonstration

How To Fix Your Code

How to fix your code and follow the Open/Closed principle:

- Have your classes act on interfaces, not discrete implementations.
- Use base classes and override their functionality.
- Add events to your class and have other components on the gameobject register for those events instead of the class calling them directly.



Example

Given a rectangle class, compute the area

```
public class Rectangle
{
    public double Width { get; set; }
    public double Height { get; set; }
}
```

Easy!

```
public class AreaCalculator
{
    public double Area(Rectangle[] shapes)
    {
        double area = 0;
        foreach (var shape in shapes)
        {
            area += shape.Width*shape.Height;
        }

        return area;
    }
}
```

The Wrong Way

But what if we also wanted the area of a circle?

```
public double Area(object[] shapes)
{
    double area = 0;
    foreach (var shape in shapes)
    {
        if (shape is Rectangle)
        {
            Rectangle rectangle = (Rectangle) shape;
            area += rectangle.Width*rectangle.Height;
        }
        else
        {
            Circle circle = (Circle)shape;
            area += circle.Radius * circle.Radius * Math.PI;
        }
    }
    return area;
}
```

The Right Way

Open/Closed Principle

```
public class Rectangle : Shape
{
    public double Width { get; set; }
    public double Height { get; set; }
    public override double Area()
    {
        return Width*Height;
    }
}

public class Circle : Shape
{
    public double Radius { get; set; }
    public override double Area()
    {
        return Radius*Radius*Math.PI;
    }
}
```

```
public abstract class Shape
{
    public abstract double Area();
}

public double Area(Shape[] shapes)
{
    double area = 0;
    foreach (var shape in shapes)
    {
        area += shape.Area();
    }

    return area;
}
```

Liskov Substitution Principle

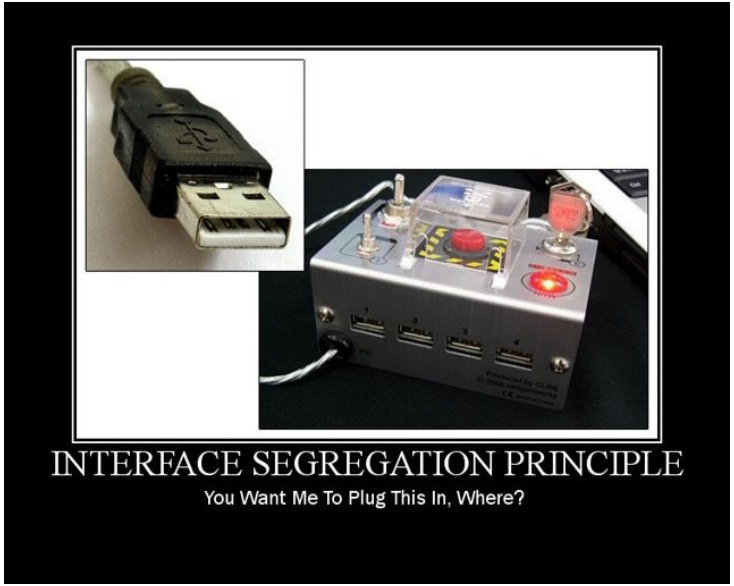
"Objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program."

Video Demonstration

Interface Segregation Principle

"Many client-specific interfaces are better than one general-purpose interface."

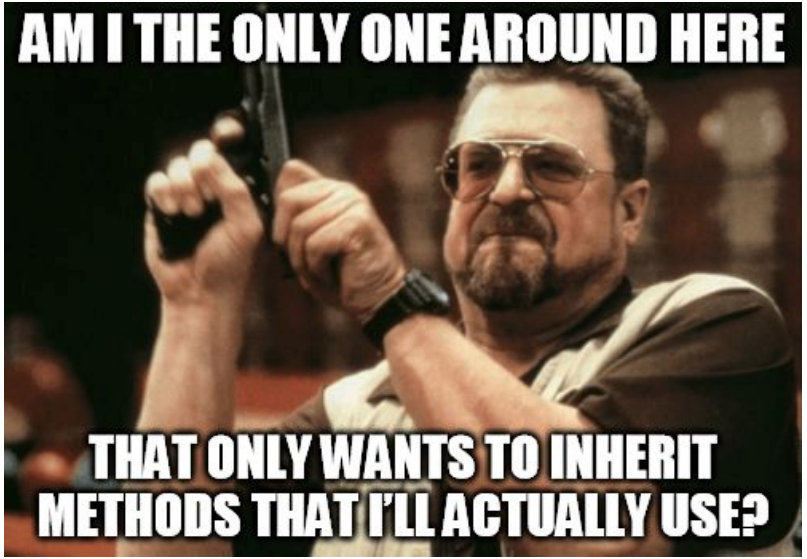
I.e. no client should be forced to depend on methods it does not use.



Video Demonstration

Interface Segregation Principle

Not following the ISP will lead to code that is tightly coupled which will make future code changes and maintenance a huge pain.



Interface Segregation Principle

```
public interface IUnitStats
{
    public float Health { get; set; }
    public int Defense { get; set; }

    public void Die();
    public void TakeDamage();
    public void RestoreHealth();

    public float MoveSpeed { get; set; }
    public float Acceleration { get; set; }

    public void GoForward();
    public void Reverse();
    public void TurnLeft();
    public void TurnRight();

    public int Strength { get; set; }
    public int Dexterity { get; set; }
    public int Endurance { get; set; }

}
```

Interface Segregation Principle

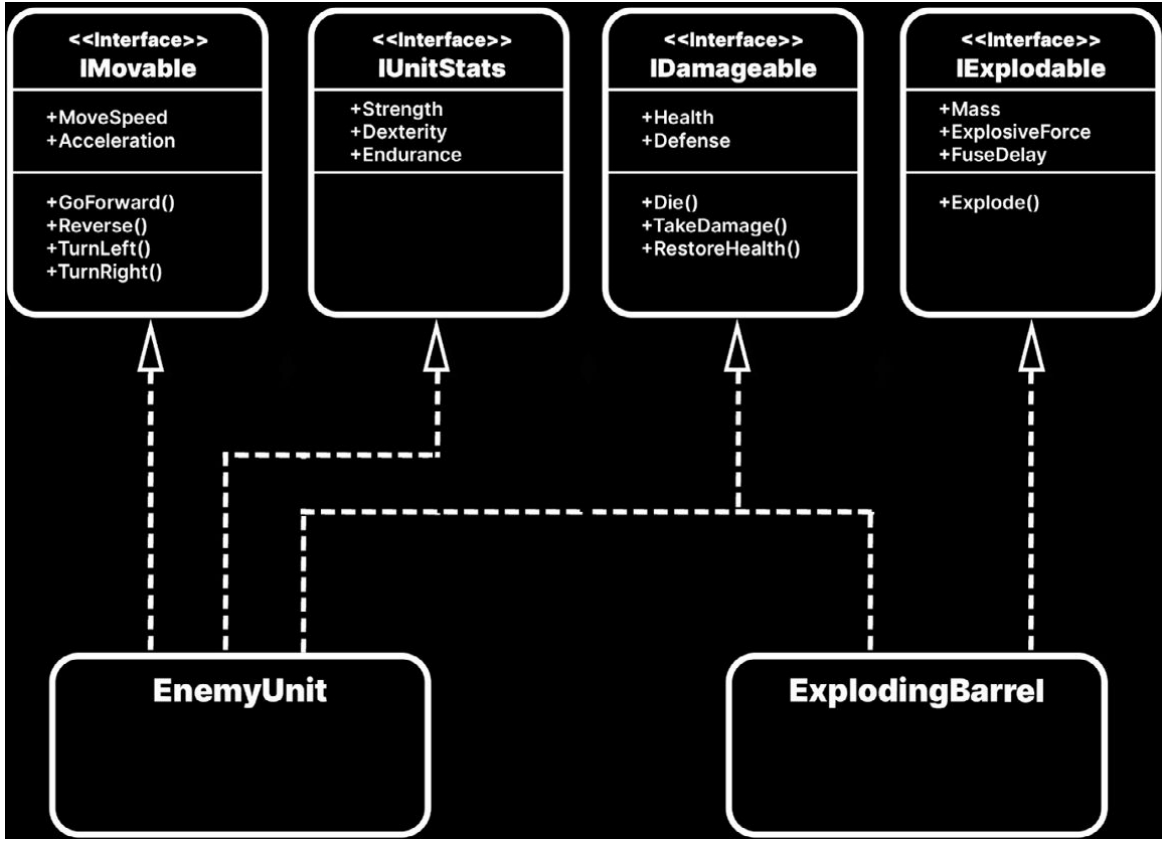
```
public interface IMovable
{
    public float MoveSpeed { get; set; }
    public float Acceleration { get; set; }

    public void GoForward();
    public void Reverse();
    public void TurnLeft();
    public void TurnRight();
}

public interface IDamageable
{
    public float Health { get; set; }
    public int Defense { get; set; }
    public void Die();
    public void TakeDamage();
    public void RestoreHealth();
}

public interface IUnitStats
{
    public int Strength { get; set; }
    public int Dexterity { get; set; }
    public int Endurance { get; set; }
}
```

Interface Segregation Principle



Dependency Inversion Principle

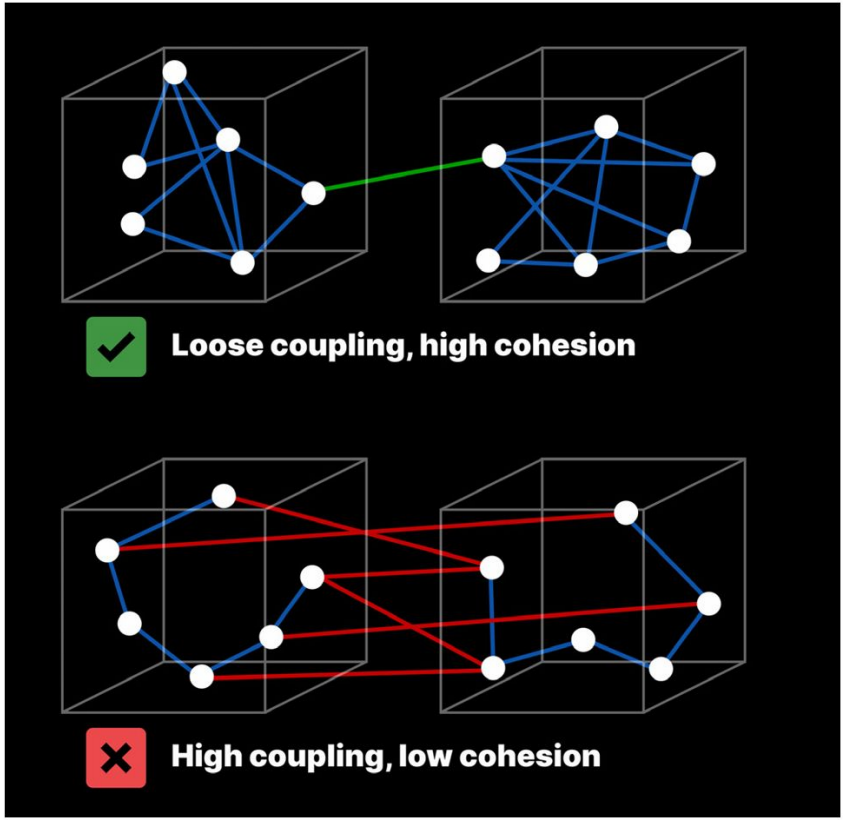
“Entities must depend on abstractions, not on concretions."

A. High-level modules should not depend on low-level modules. Both should depend on abstractions.

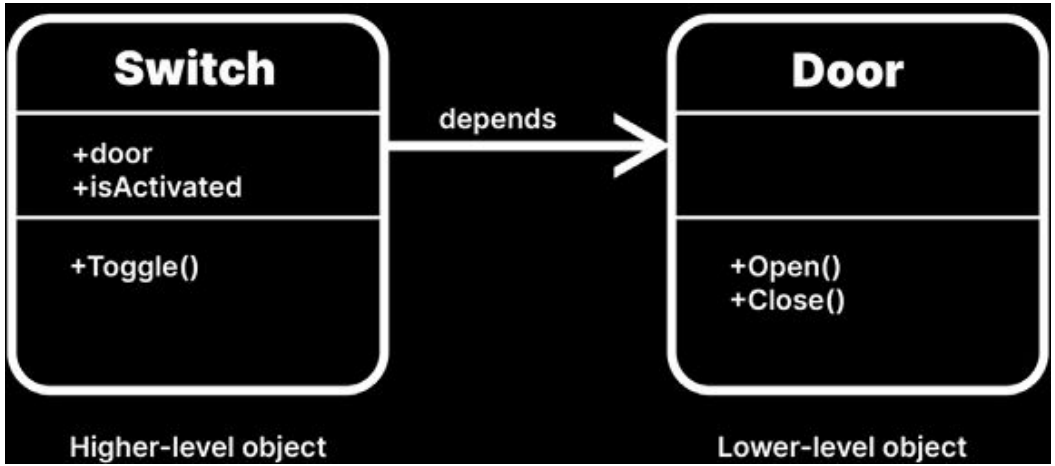
B. Abstractions should not depend on details. Details should depend on abstractions.

Video Demonstration

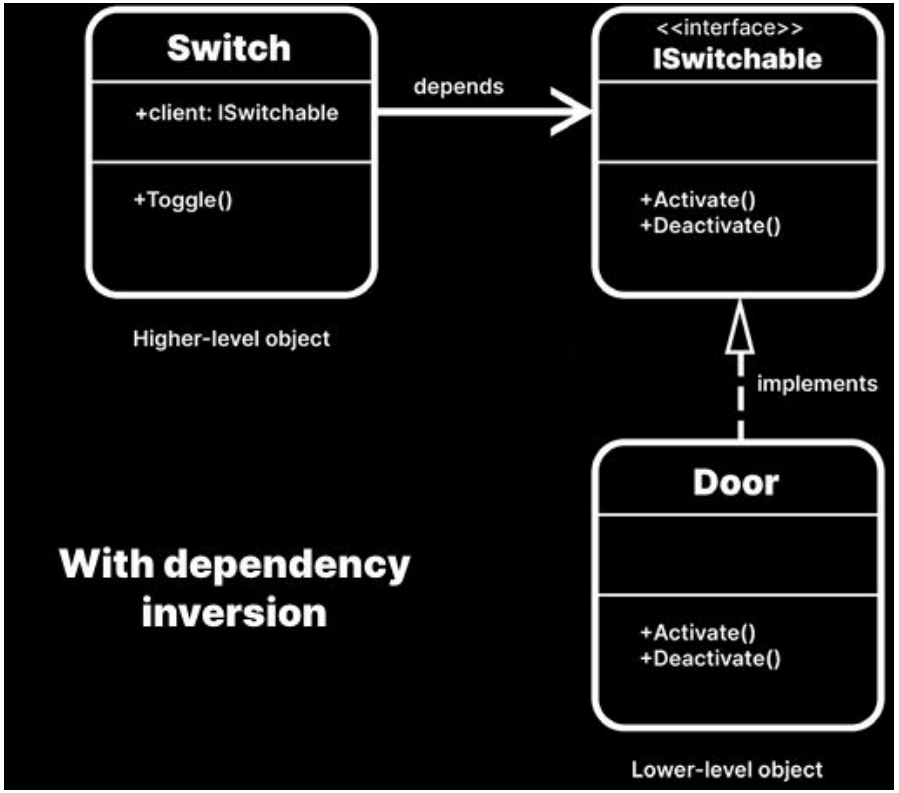
Dependency Inversion Principle



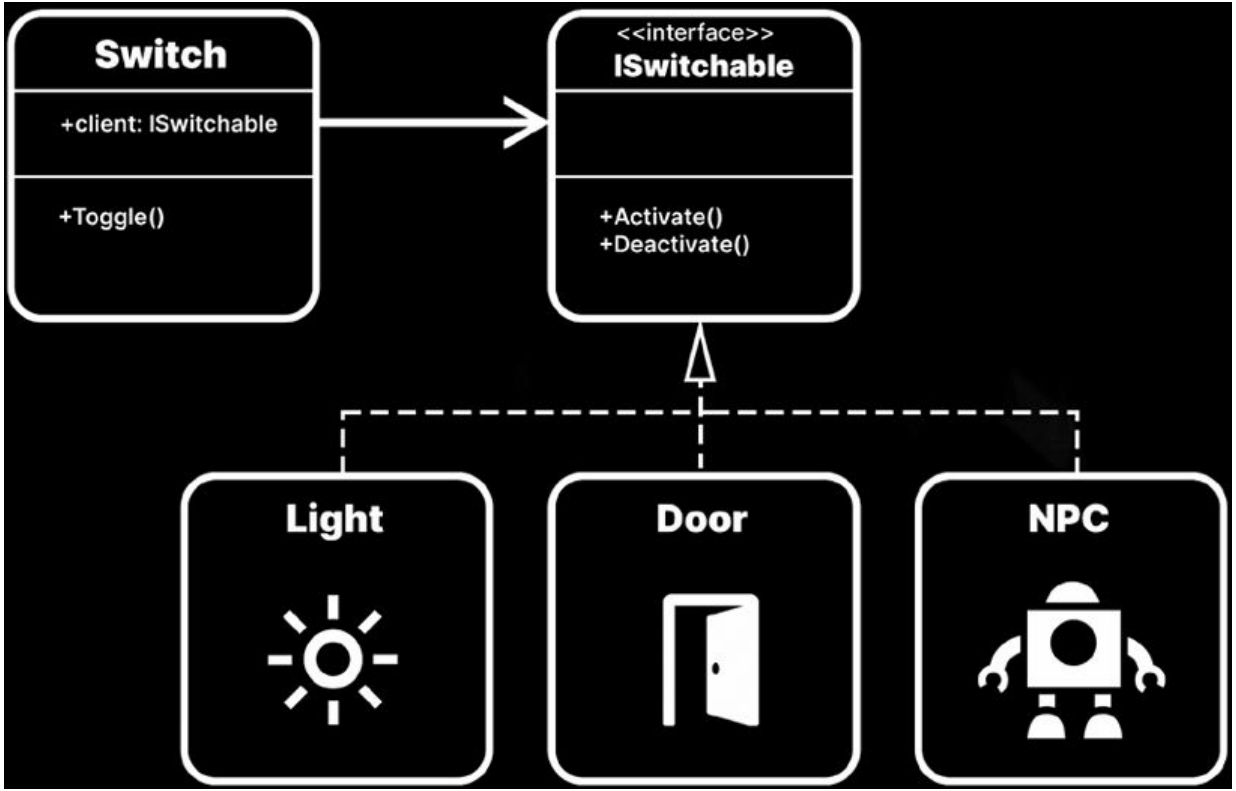
Dependency Inversion Principle



Dependency Inversion Principle



Dependency Inversion Principle



Design Patterns

Use patterns to solve actual problems

Keep it simple (simple does not mean easy)

Don’t add unnecessary complexity

Every pattern comes with tradeoffs

Singleton (Anti)Pattern

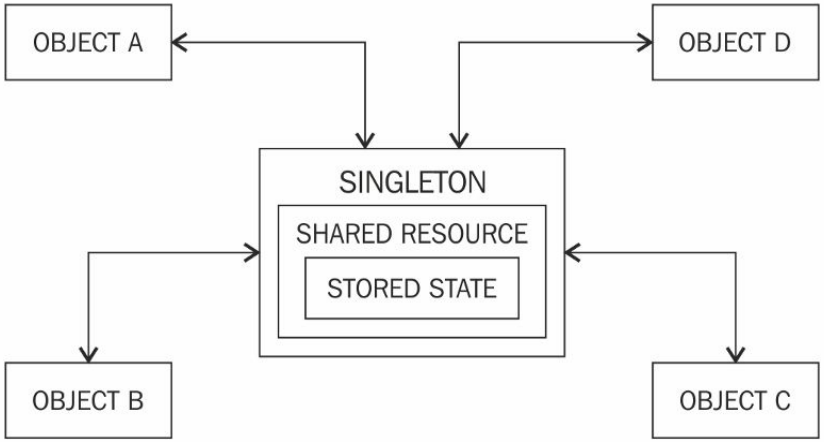
- Access anything from anywhere!
- Can enable persistent state across scenes
- Easy to understand
- Easy to start using
- Often seen with Game Managers

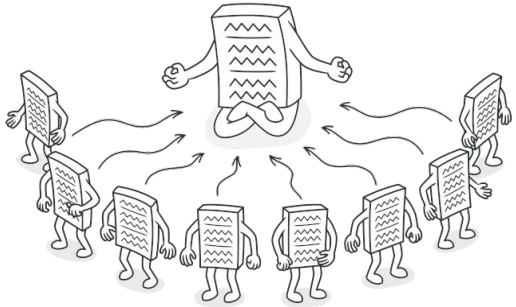
```
public class SingletonSample : MonoBehaviour
{
    private static SingletonSample instance;

    public static SingletonSample Instance { get { return instance; } }

    void Awake()
    {
        instance = this;
    }

    public void DoStuff()
    {
    }
}
```





Singleton & Game Managers

- Sometimes it is convenient to gather information and control in single object.
- Consider Pacman
  - Which GameObject is responsible for
    - Keeping score?
    - Respawning and letting out the ghosts?
    - Figuring out when the level is completed.

- Used to manage game state
- Control important, universal parameters/events/etc.
- Often created as empty GameObjects with scripts attached

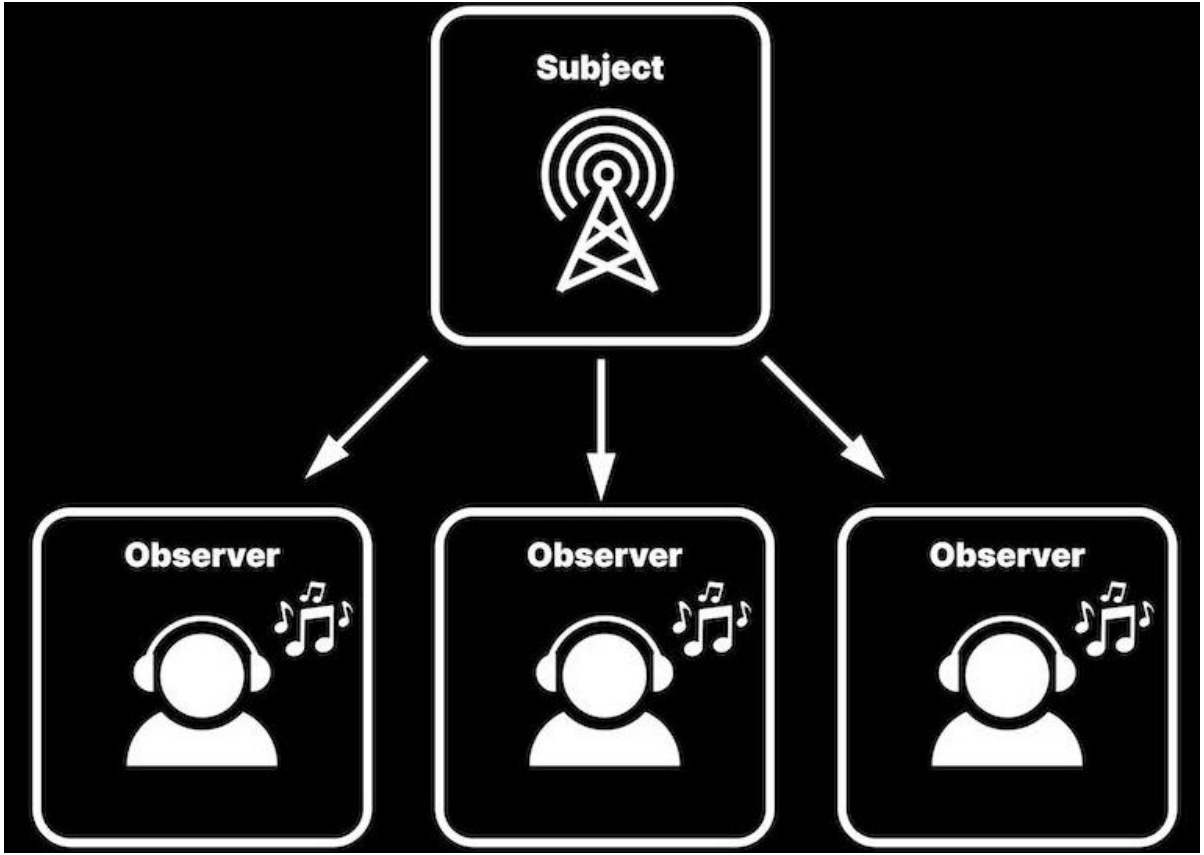
- It's just a design concept. Basically the Dungeon Master

Singleton Problems

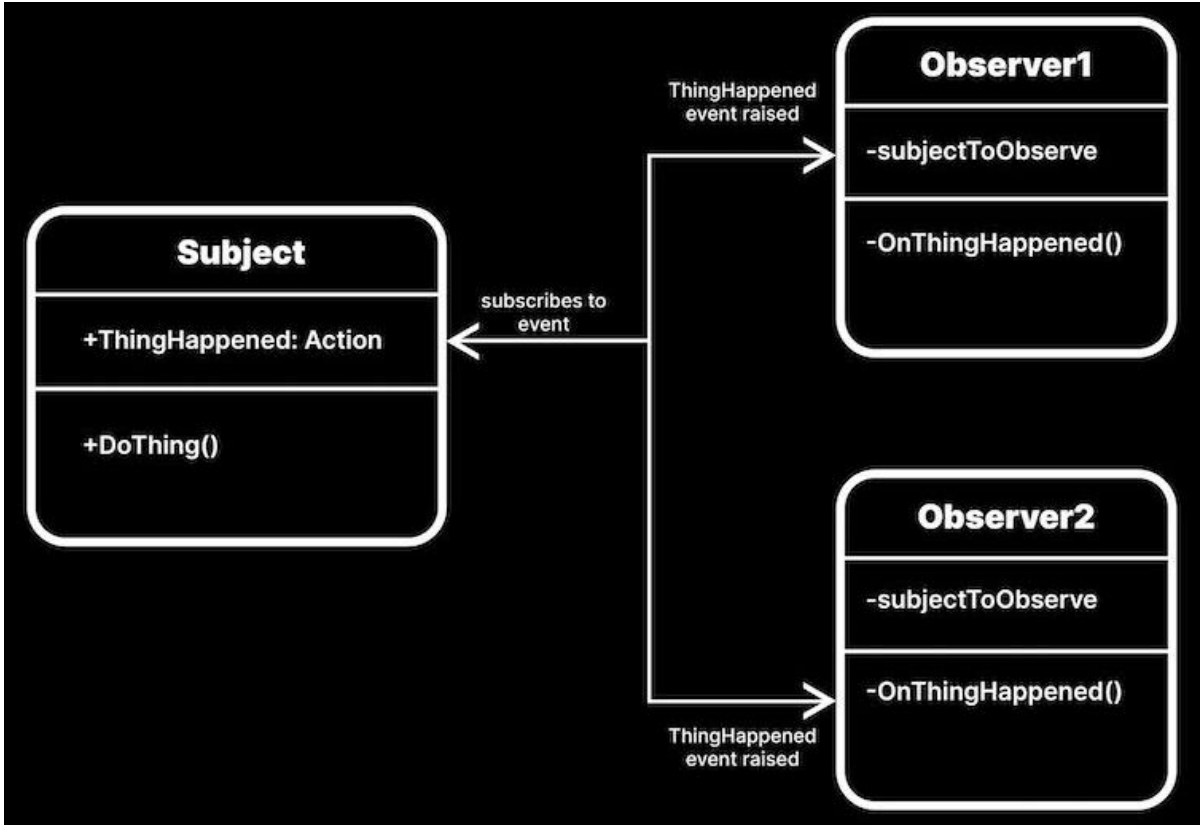
- Rigid connections - not modular!
- Breaks encapsulation
- No polymorphism
- Not testable
- Dependency spaghetti

- EnemyManager.Instance.Movespeed
  - Hard reference
  - Dependency on manager being loaded

Observer Pattern



Observer Pattern



Events

- Events are specialized delegates
- Useful for alerting other classes, that something has happened
- Functions very similarly to public, multicast delegates
- Broadcast system instead of polling
- Any class interested in an event can subscribe methods to it
- Observer pattern
- Remember to check for null!

```
public delegate void MyDelegate();
public static event MyDelegate myDelegate;
```

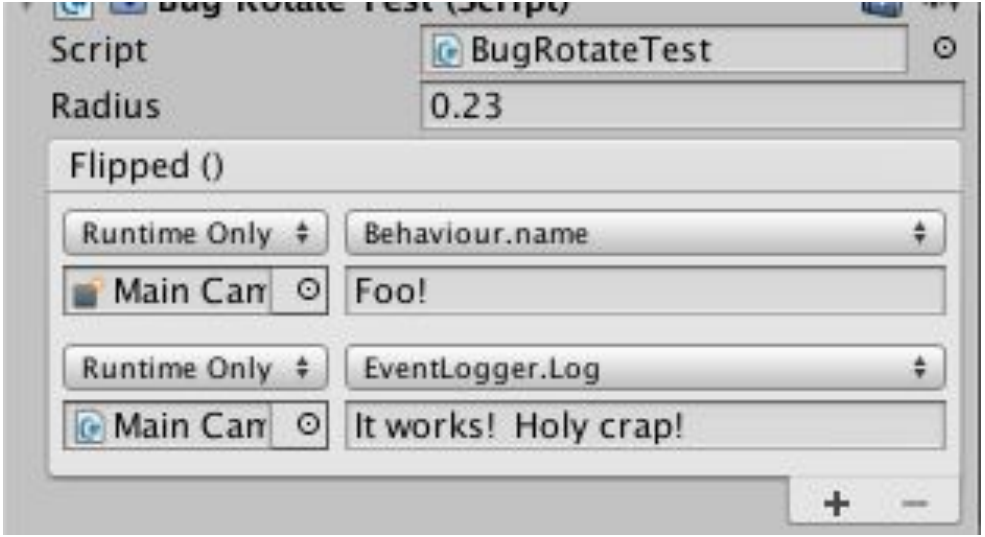
When an event occurs, like:

- Power-up
- Button click
- Player shoots
- New personal high score

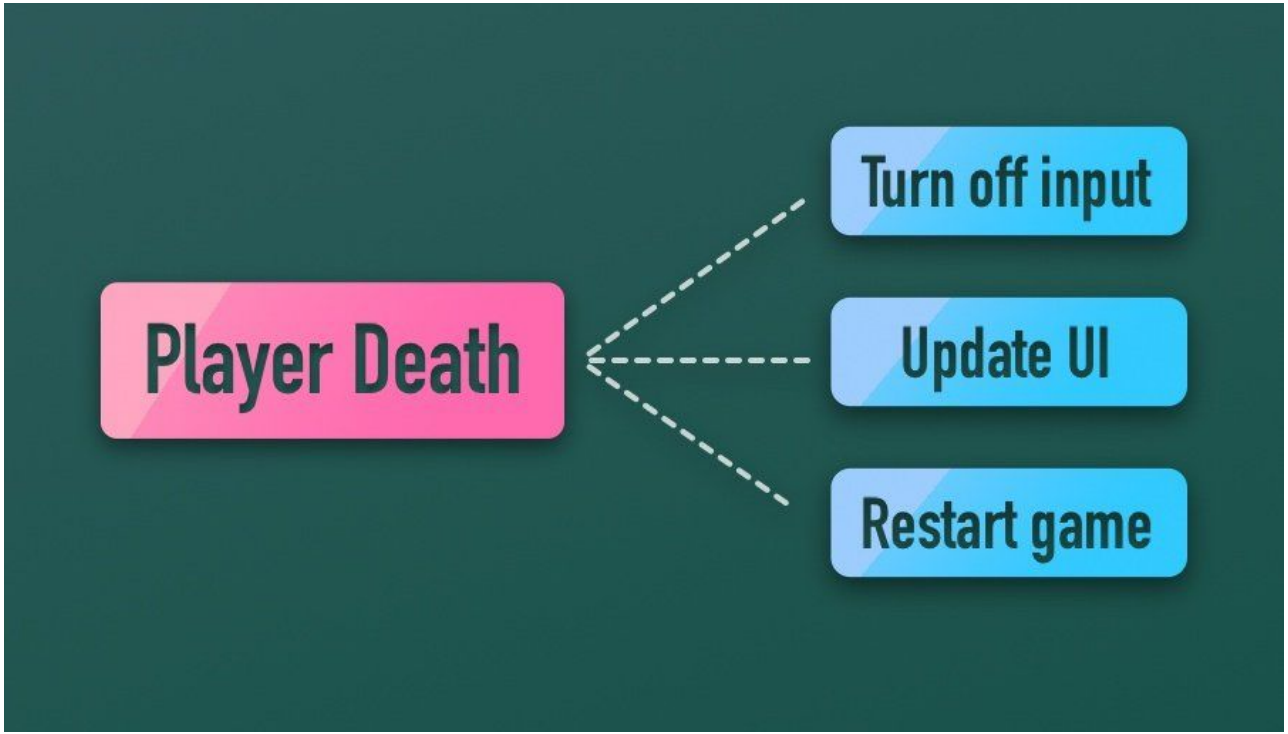
The event calls the methods of the subscribed classes.

Unity Events

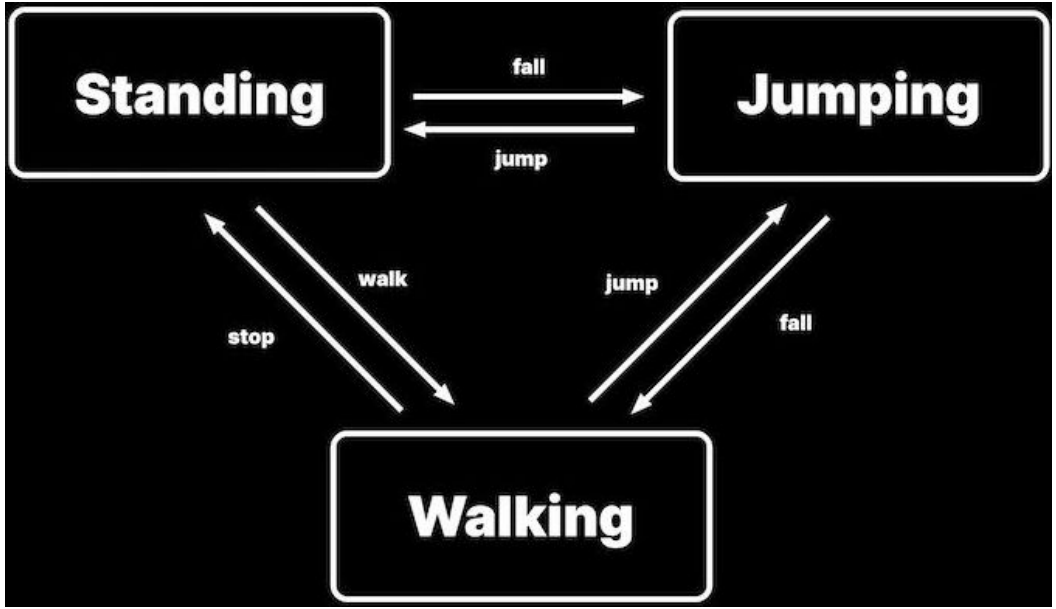
Serialized function calls

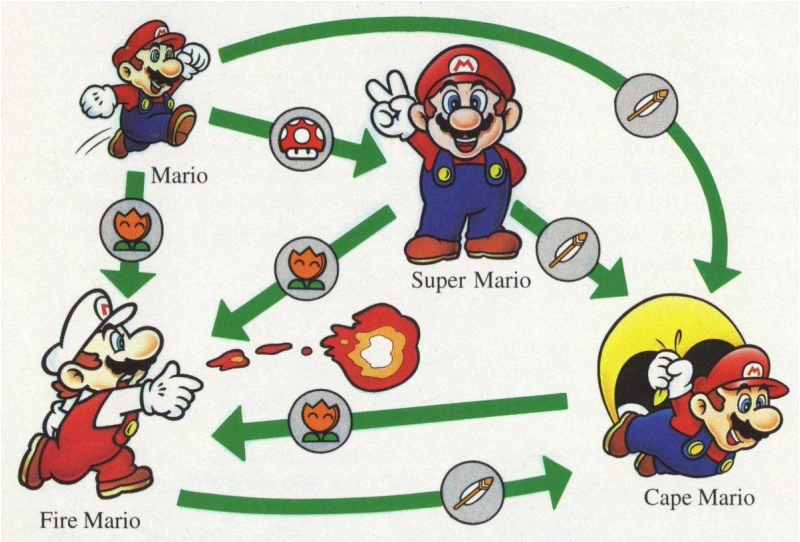


Event Architecture

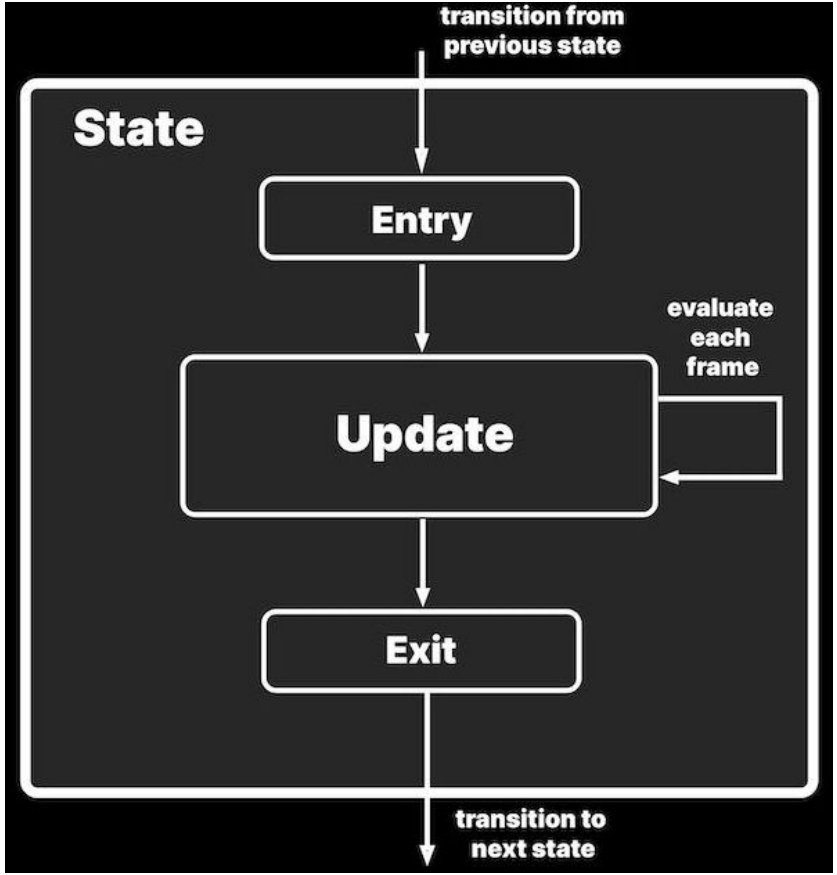


State Pattern

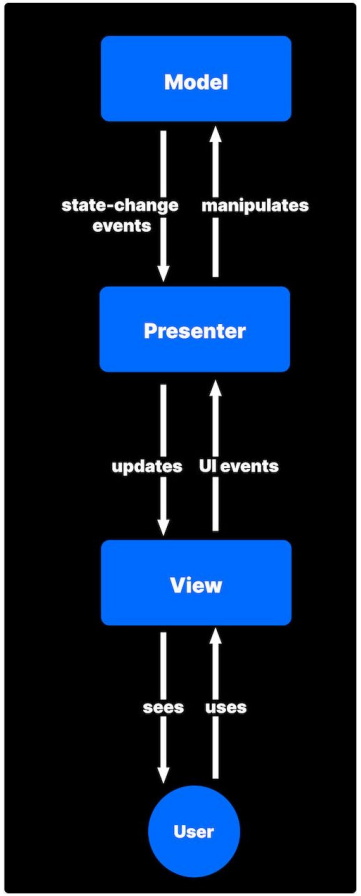


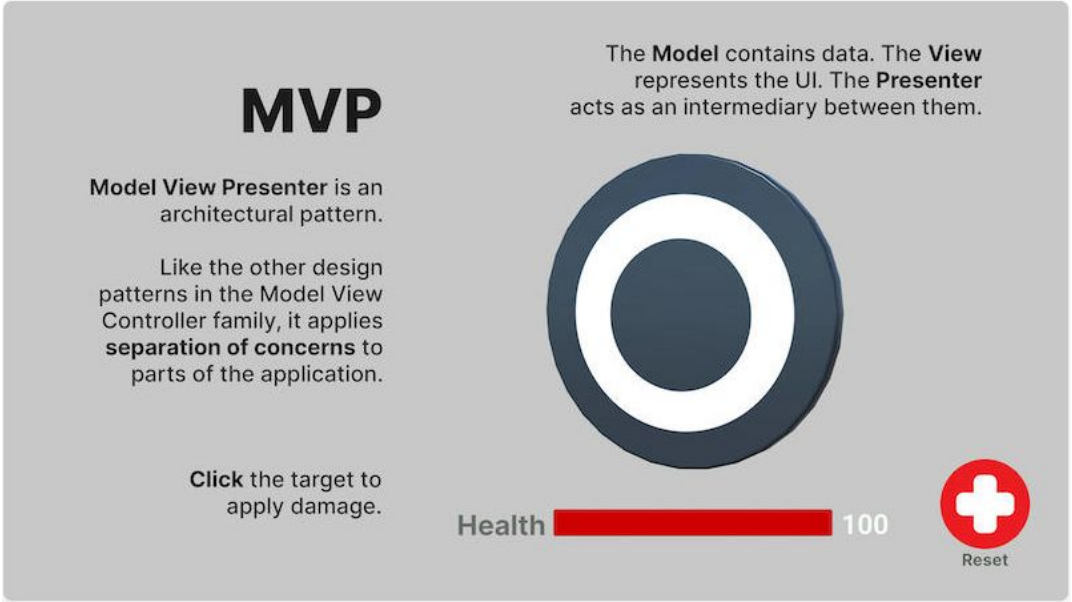


State Pattern

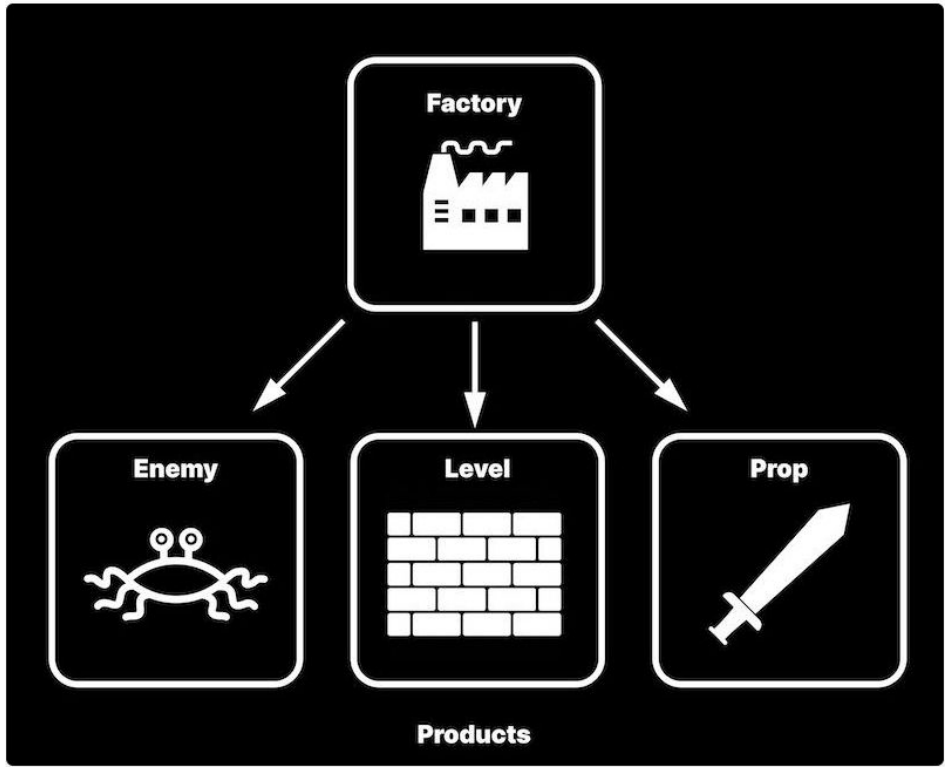


Model View Presenter

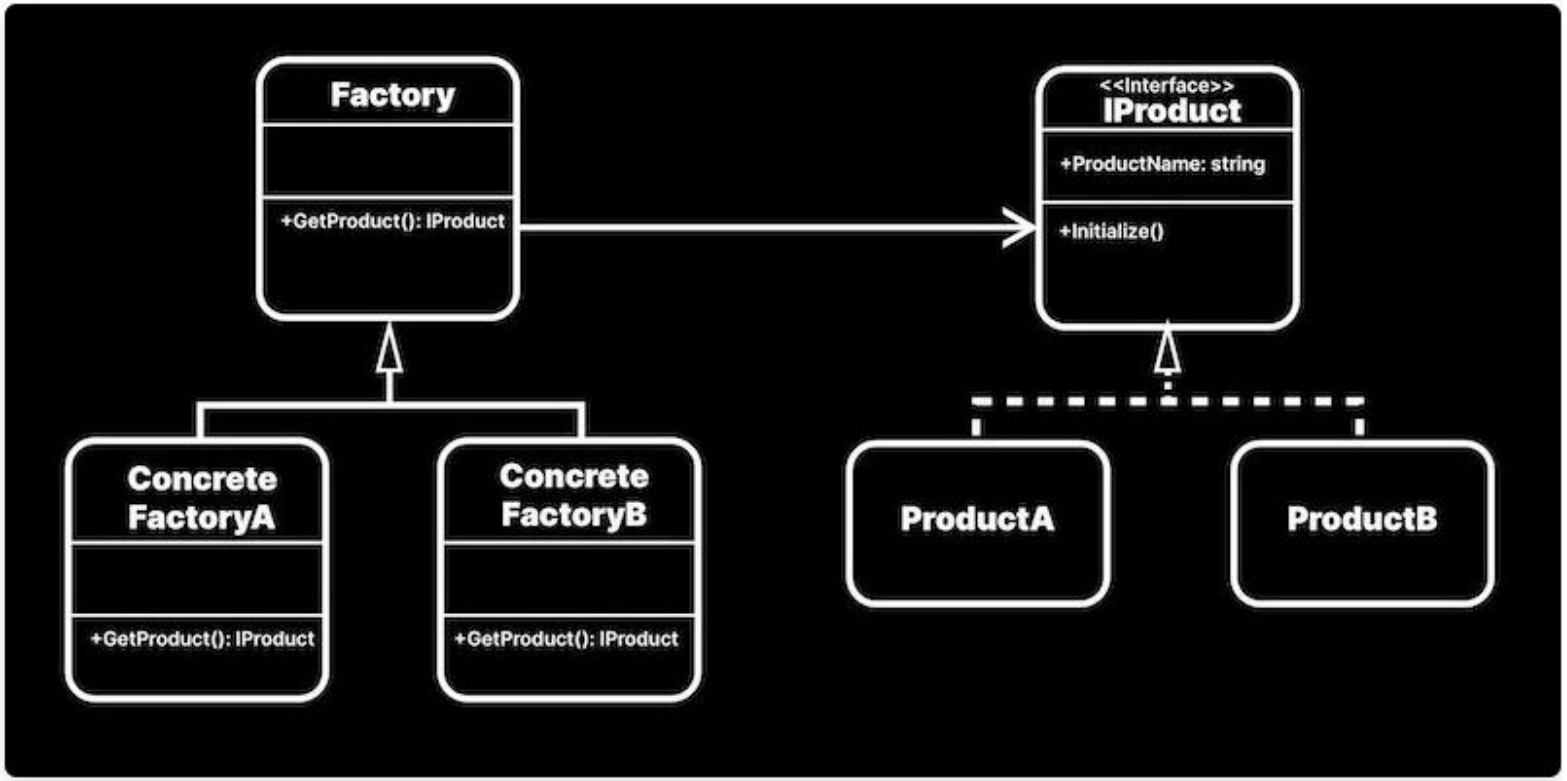




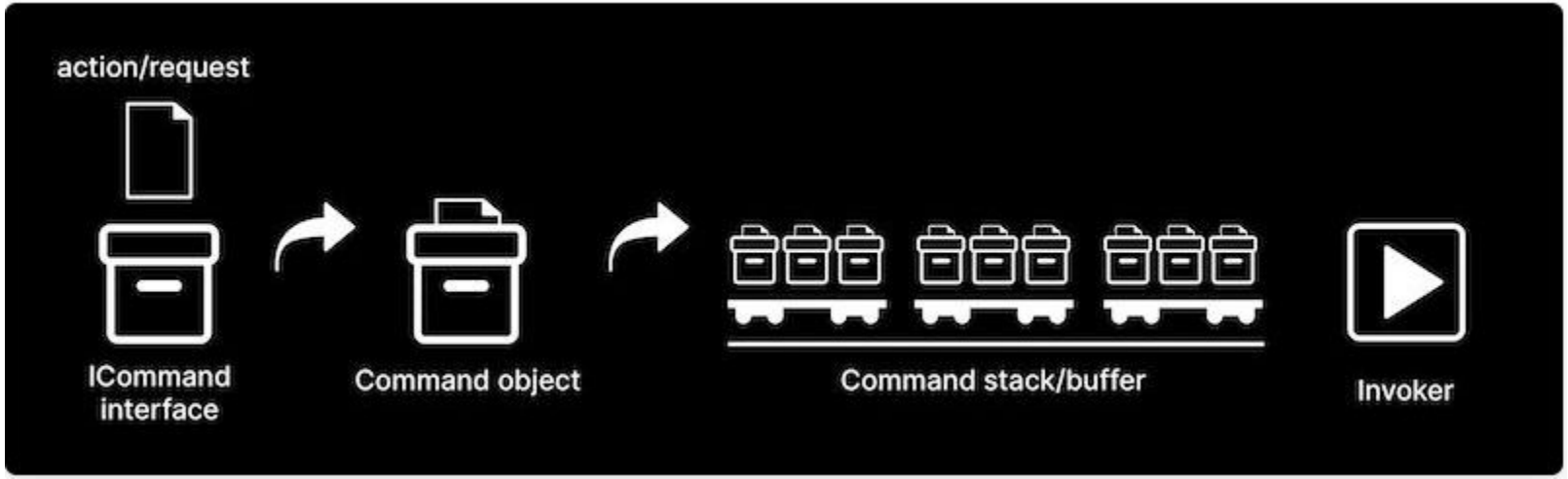
Factory Pattern



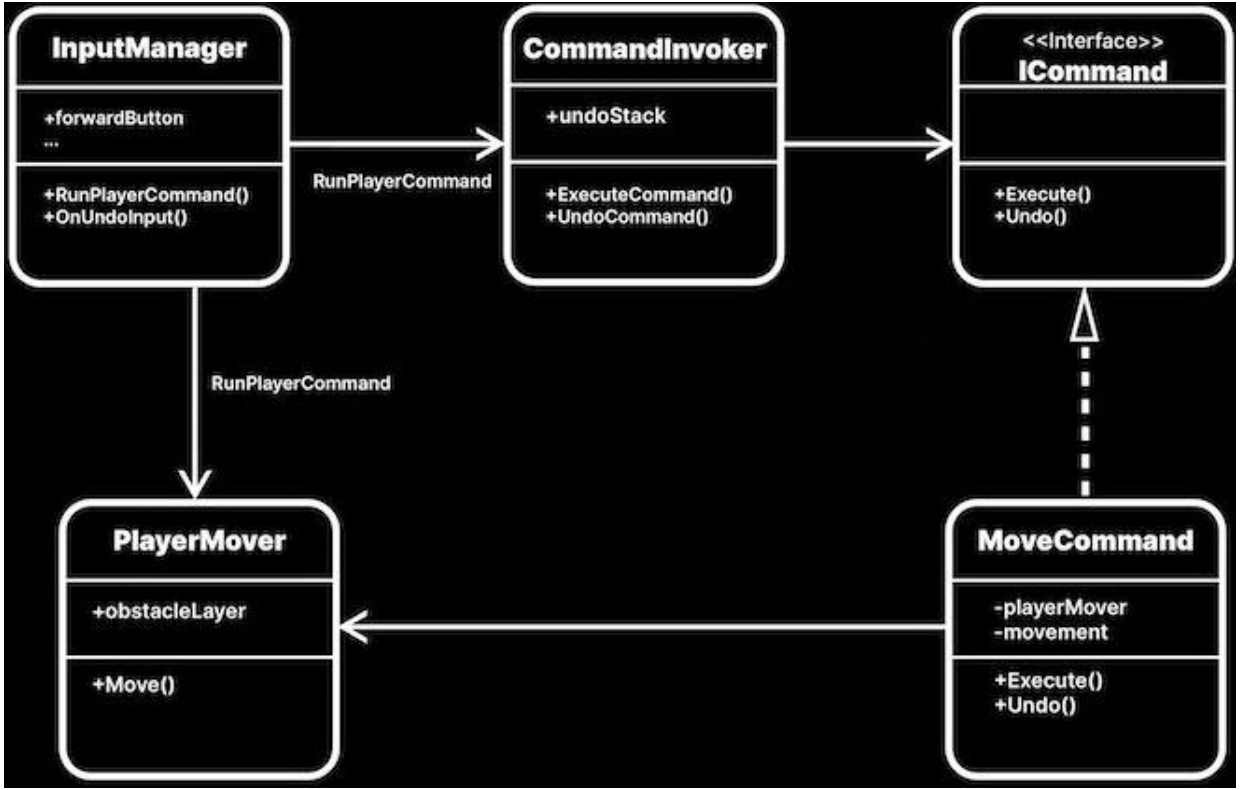
Factory Pattern



Command Pattern



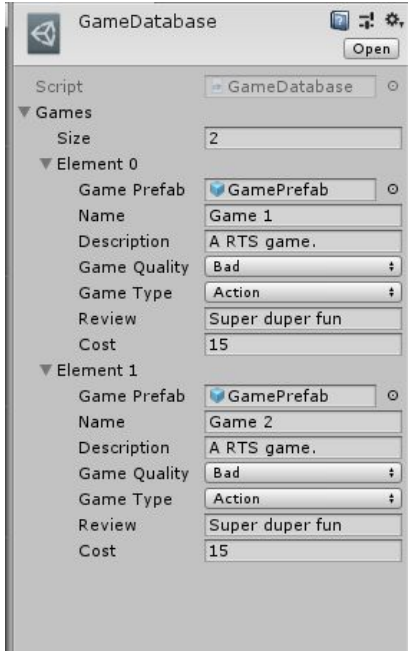
Command Pattern



ScriptableObjects

- Serializable Unity class that you can derive from if you want to create objects that don't need to be attached to game objects.
- This is most useful for assets which are only meant to store data.
- Can be used for runtime data editing
- Useful for shared state (without statics or singletons)

```
[CreateAssetMenu(fileName = "New Card", menuName = "Card")]
Unity Script | 3 references
public class CardModel : ScriptableObject
{
    public string Title;
    public string Description;
    public string Exhibit;
    public Sprite AnimalImage;
}
```



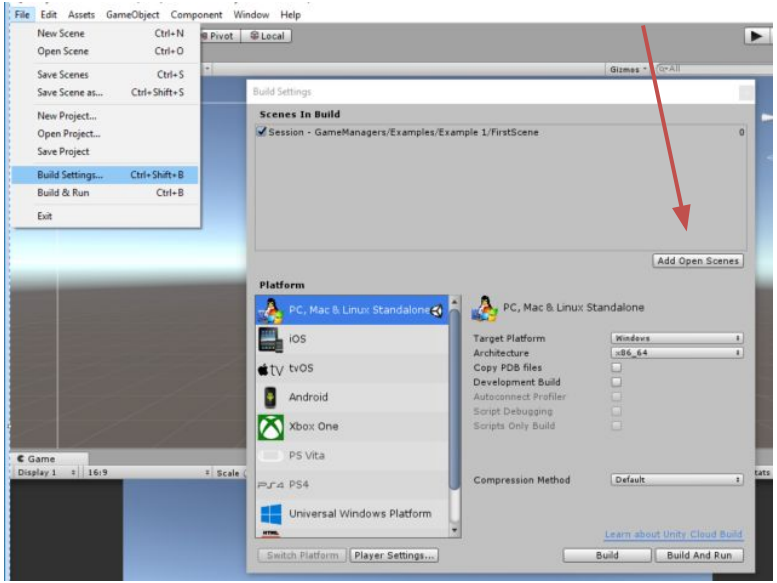
Example use cases:

- Game config files
- Inventory
- Enemy stats
- Audio Collections

Introduction to SOs

Scene Manager

- Usually levels and main menus are in different scenes.
- How do we change from the main menu to the different levels in our game?
- We use SceneManager to change between scenes.




```
using UnityEngine.SceneManagement;

public class SceneChanger : MonoBehaviour {

    public void changeScene() {
        SceneManager.LoadScene ("SecondScene");
    }

}
```

Persistence Between Scenes

By default, all GameObjects are only contained within one scene, they do not live across scene-changes.

Sometimes it's useful to preserve information

E.g. character selection

Persistence Between Scenes

- We can store information in a GameObject
- Then we can call: `DontDestroyOnLoad (gameObject);`
- This will make the gameObject persist between scenes.

- These kind of information objects are often created in the splash-scene, where game logos and such are shown, i.e. before the main menu.

- I.e. put a tag on your game manager and use “GameObject.FindGameObjectWithTag” to locate it in code.

- … Or use LoadSceneMode.Additive

Persistence Between Sessions

E.g.:

- Custom settings in the options menu
- Player profiles
- Progress in the game

```
PlayerPrefs.Set
```

UnityEngine.PlayerPrefs
- SetFloat
- SetInt
- SetString

```
public static void SetString (
  string key,
  string value
)
```

Summary
Sets the value of the preference identified by key.

We use the PlayerPrefs class.

It's basically a map, where you store key-value pairs.

```
PlayerPrefs.SetString ("PlayerAccount", "Troels, trmo@via.dk, 31");

string account = PlayerPrefs.GetString ("PlayerAccount");
```

Key Points

- Avoid “mega classes” and giant prefabs
  - Make sure your components only have one reason to change
  - Reduce need for global managers
  - Have prefabs work in isolation
  - Don’t have game logic in your UI
- Extending functionality shouldn’t always require modification of existing code
  - Handle behaviour locally and act on interfaces
- Use Inversion of Control
  - Give objects what they need, don’t make them “go out and get it”
- Use events in your architecture to avoid tightly coupled code
- Consider using ScriptableObjects for shared state instead of singletons
- KISS! Only add complexity if necessary

Clean code vs progress