SOFTWARE ARCHITECTURE

Think Big and Keep It Simple

DESIGN PATTERNS

In the game of programming, design patterns are like walkthroughs:

They are known solutions to known problems.

Very hard to come up with them by ourselves, but even harder to not use them once we know them.

You don’t have to follow them, but you do if you want to finish the game sooner.

They make you and your fellow programmers speak the same language.

DESIGN PATTERNS

- **Creational patterns** provide object creation mechanisms that increase flexibility and reuse of existing code.
- **Structural patterns** explain how to assemble objects and classes into larger structures, while keeping the structures flexible and efficient.
- **Behavioral patterns** take care of effective communication and the assignment of responsibilities between objects.

CREATIONAL PATTERNS

Provide object creation mechanisms that increase **flexibility** and **reuse** of existing code.

CREATIONAL PATTERNS

Simple factory
Factory method*
Abstract Factory*
Prototype*

One of the * patterns must be used in the assignment.

SIMPLE FACTORY

- Factory patterns are about encapsulation of instantiation of the object(s).
- Simple factory is not a real pattern but we use it often.

- Let's take a look at a real life(sort of) example:

AyeePhone is a world leading mobile brand. This was their method to produce phones:

```
public phone ProducePhone(){
        Screen screen = new Screen(QualityLevel.Low,  fragileMaterial, oddSize);
        Motherboard motherboard = new Motherboard(CPU.A18,);
        Case case = new Case();
        Battery battery = new Battery(Duration.Decreasing);
        Phone phone = Assemble(screen, motherboard, case, battery);
        //phone.QualityAssurance();
        return phone;
}
```

Of course the boss yelled at their developers for this:

```
public Phone ProducePhone(){
      Screen screen = new Screen(QualityLevel.Low,  fragileMaterial, oddSize);
      Motherboard motherboard = new Motherboard(CPU.A18,);
      Case case = new Case();
      Battery battery = new Battery(Duration.Decreasing);
      Phone phone = Assemble(screen, motherboard, case, battery);
      //phone.QualityAssurance();
      return phone;
}
```

Coupled with Screen class.
Any changes of any used classes would result in rewriting the whole function.
Not 'Pro' bro!

Factories saves the day!

```
ScreenFactory screenFactory;
MotherboardFactory motherBoardFactory;
CaseFactory caseFactory;
BatteryFactory batteryFactory;

void Init(ScreenFactory namelessChineseFactory1, MotherboardFactory namelessTaiwanFactory,
CaseFactory namelessChineseFactory2, BatteryFactory namelessChineseFactory3){
        screenFactory = namelessChineseFactory1;
        motherboardFactory = namelessTaiwanFactory;
        caseFactory = namelessChineseFactory2
        batteryFactory = namelessChineseFactory3;
}

public Phone ProducePhone(){
        Screen screen = screenFactory.ProduceScreen();
        Motherboard motherboard = motherboardFactory.ProduceMotherboard();
        Case case = caseFactory.ProduceCase();
        Battery battery = batteryFactory.ProduceBattery();
        Phone phone = Assemble(screen, motherboard, case, battery);
        //phone.QualityAssurance();
        return phone;
}
```

SIMPLE FACTORY

Concrete class instantiation is encapsulated to decouple the client code from the object creation.

- The client code doesn't have to worry about instantiating details.
- Main class remain untouched if other classes need to be modified.(If screen quality needs to be changed, only ScreenFactory class needs to be modified.)

SIMPLE FACTORY

We've actually seen examples of this pattern before…

Box2D v2.3.0 User Manual

1.8 Factories and Definitions

Fast memory management plays a central role in the design of the Box2D API. So when you create a b2Body or a b2Joint, you need to call the factory functions on b2World. You should never try to allocate these types in another manner.

There are creation functions:

```
b2Body* b2World::CreateBody(const b2BodyDef* def)
b2Joint* b2World::CreateJoint(const b2JointDef* def)
```

And there are corresponding destruction functions:

```
void b2World::DestroyBody(b2Body* body)
void b2World::DestroyJoint(b2Joint* joint)
```

FACTORY IN ACTION

It is easy to create a Box2D world. First, we define the gravity vector.

```
b2Vec2 gravity(0.0f, -10.0f);
```

Now we create the world object. Note that we are creating the world on the stack, so the world must remain in scope.

```
b2World world(gravity);
```

```
b2BodyDef groundBodyDef;
groundBodyDef.position.Set(0.0f, -10.0f);
```

```
b2Body* groundBody = world.CreateBody(&groundBodyDef);
```

FACTORY IN ACTION

What about Unity?

FACTORY IN ACTION

In which of the following method(s) of GameObject can you see factory pattern?

Public methods

AddComponent

GetComponent

static methods

Find

CreatePrimitive

FACTORY IN ACTION

In which of the following method(s) of GameObject can you see factory pattern?

Public methods

AddComponent

GetComponent

static methods

Find

CreatePrimitive

FACTORY IN ACTION

```
BoxCollider bc = gameObject.AddComponent<BoxCollider>();
bc.size = Vector3.one;

GameObject cube = GameObject.CreatePrimitive(PrimitiveType.Cube);
cube.transform.position = Vector3.zero;
```

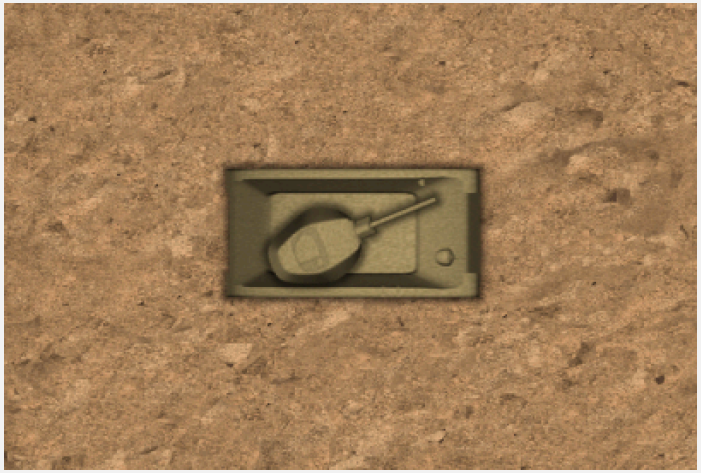
FACTORY METHOD

Simple factory is not an actual design pattern(just a good habit).

Factory method is a design pattern.

THE PROBLEM

Introducing another old friend:



THE PROBLEM

Let's use try to use a simple factory to take care of bullet creation:

```
void Fire()
{
    Bullet bullet = bulletFactory.CreateBullet();
    bullet.SetTarget(myTarget);
    bullet.SetSpeed(bulletSpeed);
    bullet.Fire();
}
```

THE PROBLEM

Years later… your tank game is doing so well that you decide to send the code to some Chinese game company to make a mobile game. Then this happened to your code:

```
void Fire()
{
    Bullet bullet = bulletFactory.CreateBullet();
    //bullet.SetTarget(myTarget);
    //bullet.SetSpeed(bulletSpeed);
    //We removed the code above because it's a mobile game and we made the bullet automatically
    //follow the nearest target, and bullet speed is now determined by how much money you pay
    //for your tank.
    bullet.Fire();
}
```

THE PROBLEM

Now your code is ruined and gamers hate you. What went wrong?

FACTORY METHOD

The pattern:
Provides an interface for creating objects, but delegate the actual creation to subclasses.

FACTORY METHOD

So you went back in time and this time you applied Factory Method Pattern:

```
//Made Tank into an abstract class
…
void Fire()
{
    Bullet bullet = CreateBullet();
    bullet.SetTarget(myTarget);
    bullet.SetSpeed(bulletSpeed);
    bullet.Fire();
}

public abstract Bullet CreateBullet();//Factory method.
```

FACTORY METHOD

You then created an interface/abstract class to make sure all the necessary methods for Bullet were implemented.

```
public interface IBullet
{
	void SetVelocity(Vector2 velocity);
}
```

FACTORY METHOD

Now the mobile developers just need to implement the factory method in sub classes of Tank.

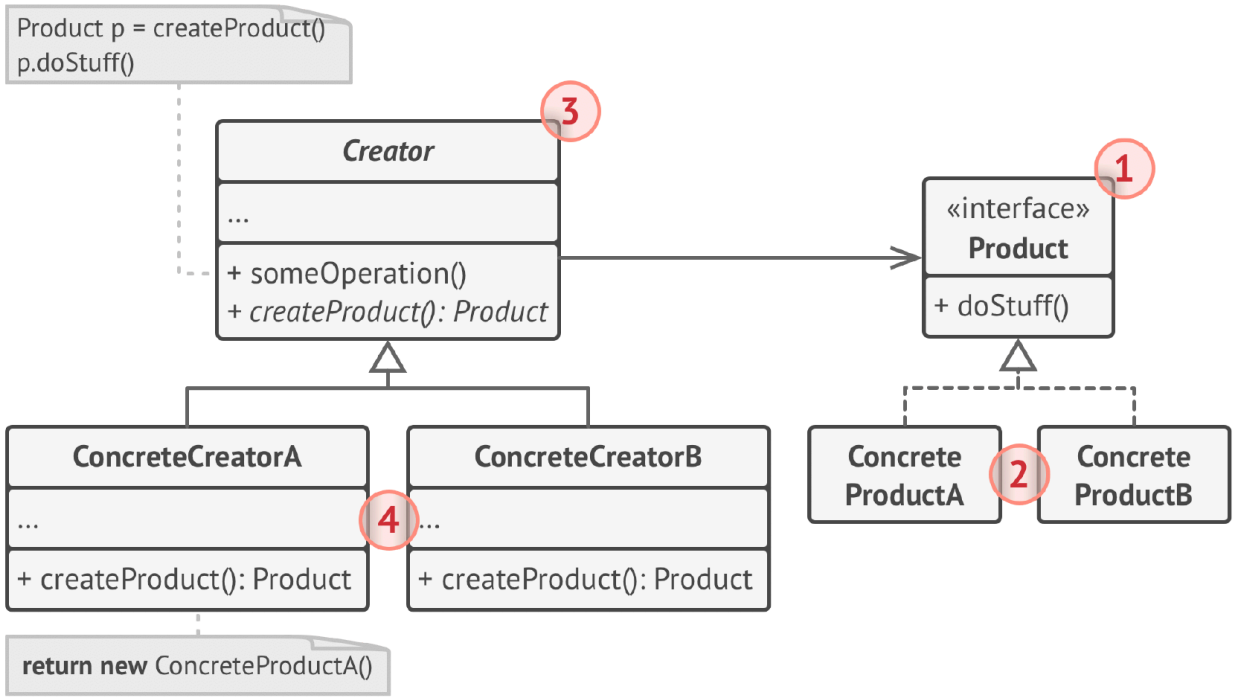
```
public class MobileTank :Tank{
    public override IBullet CreateBullet(){
        return new MobileBullet(moneyPaidInTotal);//MobileBullet class implements Bullet interface.
    }
}
```

Maybe even a Pad version:

```
public class PadTank :Tank{
    public override IBullet CreateBullet(){
        return new PadBullet(moneyPaidInTotal);//PadBullet class implements Bullet interface.
    }
}
```

Now gamers still hate you, but at least your code base is intact.

FACTORY METHOD



1) The product declares an interface for methods to be called by the creator.
2) Concrete products implement the product interface.
3) The creator uses factory method to get product objects. **Keep in mind that creating product objects is not the main responsibility of the creator.**
4) Concrete creators override the factory method to return different type of products.

FACTORY METHOD

Separate the construction details of the product from the creator and put them into a factory method.

```
IBullet bullet = CreateBullet();
VS.
Bullet bullet = new Bullet(int pType, float pSpeed);
```

The creator doesn't need to change with the product.

```
Adding a property(followingTarget : bool) to Bullet class:
IBullet bullet = CreateBullet();//No change here
VS.
Bullet bullet = new Bullet(int pType, float pSpeed, bool
pFollowingTarget);//This needs to change with Bullet class
```

The subclasses can override the factory method to create new concrete products.

```
IBullet bullet = CreateBullet();//Can be overridden by
subclasses to create new concrete bullet types.
VS.
Bullet bullet = new Bullet(int pType, float pSpeed, bool
pFollowingTarget);//Can't be extended.
```

ABSTRACT FACTORY

Factory method is a nice way to decouple creator from a concrete product, but what if we need to create multiple different products?

THE PROBLEM

We are making a monster spawner class for a rogue like game which can create normal, enhanced and rare monsters.

The monsters created need to be the same level, for example in the first dungeon we have: normal zombies, normal skeletons, normal goblins, in the second dungeon we have enhanced zombies, enhanced skeletons, enhanced goblins, etc.

We also don't want to change our spawner class each time we add a new monster level(e.g. super rare monsters) to the game.

One solution to this problem: **abstract factory**.

ABSTRACT FACTORY

The pattern:
Provides an interface for creating related objects without specifying their concrete classes.

ABSTRACT FACTORY

First, define an interface to create different monsters:

```
public interface MonsterFactory
{
    Zombie CreateZombie();
    Skeleton CreateSkeleton();
    Goblin CreateGoblin();
}
```

Then implement the interface to create factories:

```
public class NormalFactory : MonsterFactory
{
    public Zombie CreateZombie(){
        return new Zombile(MonsterLevel.Normal);
    }
    public Skeleton CreateSkeleton(){
        return new Skeleton(MonsterLevel.Normal);
    }
    public Goblin CreateGoblin(){
        return new Gobline(MonsterLevel.Normal);
    }
}

public class EnhancedFactory : MonsterFactory{…}
```

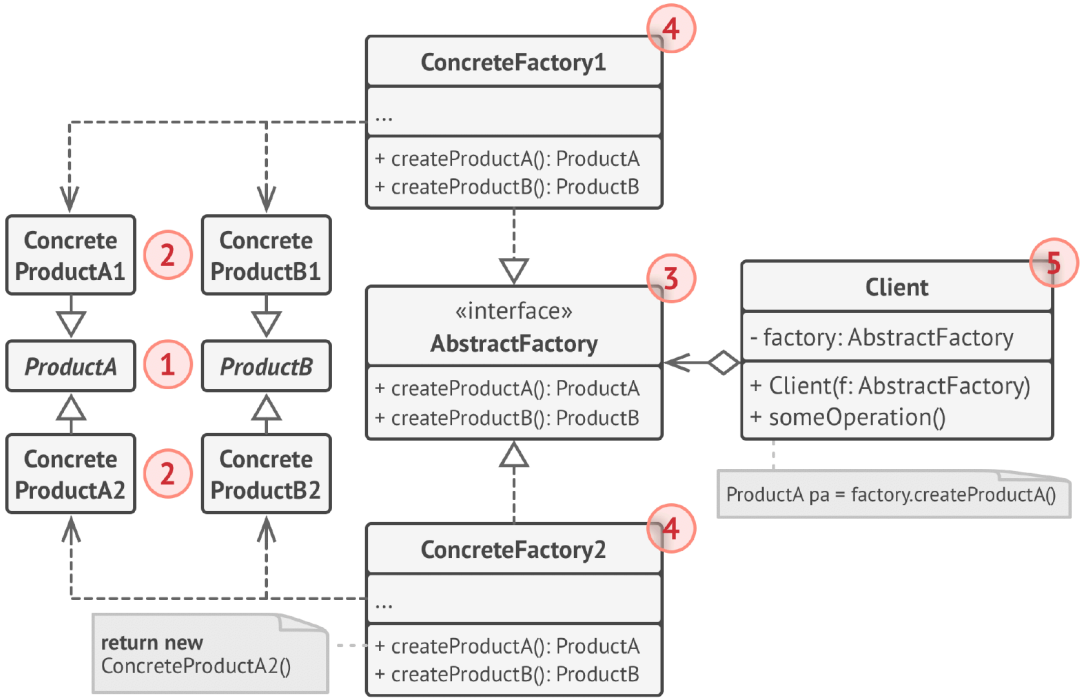
The spawner class uses a factory to create monsters:

```
public class Spawner
{
    MonsterFactory monsterFactory;
    public Spawner(MonsterFactory pMonsterFactory){
       monsterFactory = pMonsterFactory;
    }
    //A simplified version of spawn method
    public void SpawnMonsters(){
       Zombie zombie = monsterFactory.CreateZombie();
       zombie.SetActive(true);
       Skeleton skeleton = monsterFactory.CreateSkeleton();
       skeleton.SetActive(true);
       Goblin goblin = monsterFactory.CreateGoblin();
       goblin.SetActive(true);
    }
}
```

Create a normal level spawner, simply do:

```
Spawner normalSpawner = new Spawner(new NormalFactory());
normalSpawner.SpawnMonsters();
```

ABSTRACT FACTORY



1) Abstract products declare a common interface for each type of product.
2) Concrete products implement the product interface. Each product must be implemented in all variants(normal, enhanced, rare, etc.)
3) The abstract factory declare a method to create each product.
4) Concrete factories implement AbstractFactory interface to create one specific variant of products.
5) Client code can work with any concrete factory or product via abstract interfaces.

FACTORY METHOD VS. ABSTRACT FACTORY

Factory Method:
Usually, the base class's main purpose is not to create products.(In our tank example, tank's main purpose is not to create bullets)

Use inheritance and relies on its subclasses to implement the factory ***method.***

Abstract Factory:
The abstract factory class's main purpose is to create products.

Other client classes use an abstract factory ***object*** to create desired products.

PROTOTYPE

Using abstract factory to create a monster spawner sure needs a lot of setup!

If only we were using Unity… we need that ‘prefab’ for this!

PROTOTYPE

From Unity documentation:

Unity's **Prefab** system allows you to create, configure, and store a GameObject complete with all its components, property values, and child **GameObjects** as a reusable Asset.

In other words: Clone.

PROTOTYPE

Introducing prototype pattern, a.k.a. clone.

Clone an existing object without depending on its class.

PROTOTYPE

Back to our spawner, the problem can also be solved by using an existing monster object as a prototype, and spawn monsters by cloning it.

Then of course we need an interface as simple as:

```
public interface IPrototype
{
	IPrototype Clone();
}
```

Implementing the clone method:

```
public class Zombie : IPrototype
{
        private int hp;
        private int attack;

        public Zombie(int pHP, int pAttack){
                hp = pHP;
                attack = pAttack;
        }

        public Zombie(Zombie zombie){
                hp = zombie.hp;
                attack = zombie.attack;
        }

        public IPrototype Clone(){
                return new Zombie(this);
        }
}
```

Question 1, hp is a private field, is this assignment possible?

Question 2, why not just: return this?

Spawner class using prototype pattern:

```
public class Spawner<T> where T: IPrototype
{
	private Prototype prototype;
	public Spawner(T pPrototype){
		prototype = pPrototype;
	}

	public T Spawn(){
		return (T)prototype.Clone();
	}
}
```

To use it:

```
Zombie prototypeZombie = new Zombie(3, 3);
Spawner<Zombie> zombieSpawner = new Spawner<Zombie>(prototypeZombie);
Zombie zombie = zombieSpawner.Spawn();
…
```

prototypeZombie is never in the game, its sole purpose is to be used as a prototype to clone other zombies. Sounds familiar?

PROTOTYPE

It's handy to have two constructors, one is a regular constructor, the other one uses an object of the same class, the second constructor takes care of the clone method.

You can copy all the fields including private ones because private members are accessible within the body of the class in which they are declared.

If you are using entity component system pattern, you need to take care of the clone method of the attached components too.(Let's talk about it later)

PROTOTYPE

Exercise:

Draw a class diagram for the prototype pattern.

BACK TO OUR MAIN QUEST

Shop Model version 1:

Apply factory method/abstract factory/prototype pattern to the shop to procedurally generate items and upgrade them.

DESIGN PRINCIPLES

DRY principle.

KISS principle.

SOLID principles.

DRY

DRY stands for: Don't Repeat Yourself

Anti-DRY is WET.

Write every time.
Write everything twice.
We enjoy typing.
Waste everyone's time.

DRY

Typical WET code:

```
if (Input.GetKeyDown(KeyCode.A))
{
        player.Move(-5f, 0f);
}
else if (Input.GetKeyDown(KeyCode.D))
{
        player.Move(5f, 0f);
}
if (Input.GetKeyDown(KeyCode.W))
{
        player.Move(0f, 5f);
}
else if (Input.GetKeyDown(KeyCode.S))
{
        player.Move(0f, -5f);
}
```

DRY

To avoid WET:

Use modular design in code, divide system into reusable units.

Avoid lengthy methods, if it's long it probably can be divided.

Whenever you start typing the same code again, rethink your architecture. (Copy/Paste == Type, if not worse)

Code review questions:

- Is there any repeated part in the code?
- Is there any method in the class that is longer than one screen? If so can it be divided?

KISS

KISS stands for: Keep It Simple, Stupid

KISS

We talked about this in the beginning:

```
Move();
HandleCollisions(enemies);
UpdatePlayerState();
UpdateGameState();
```

KISS

Use descriptive variable names. This also applies to GameObject (1), GameObject (2) and GameObject (3) in your Unity Editor.

Don’t use magic values. itemAmount is always better than 16.

Divide lengthy code into multiple methods and give the methods understandable names.

Code review questions:

- Class names clear? Method names clear? Variable names clear?
- Are there any magic values?
- Are there any comments and are they clear?

SOLID

- **S**ingle Responsibility
- **O**pen / Closed
- **L**iskovSubstitution Principle (Out of scope)
- **I**nterface Segregation Principle
- **D**ependency Inversion Principle (Out of scope)

SINGLE RESPONSIBILITY

- **A class should only have one reason to change.**
- In Unity: monobehaviour. Components should have only one responsibility(think of built in components like RigidBody, Transform, Renderer, Collider, etc.)
- One giant Player class vs. Controler + Renderer + Animator + Killable + Collider…
- Code review question:

  Does a class have a single responsibility?

OPEN / CLOSED

- **Classes should be open for extension but closed for modification.**
- Which is why we usually have a base GameObject class in a game engine, we never modify the GameObject class, only create subclasses of it.
- In factory method pattern, the class itself is closed, subclasses override the factory method to achieve required functions.
- Of course if you are sure that there is a bug in the class, then just fix it.
- **With this principle in mind, think carefully about your base Item class.**
- Code review question:

  Is the class open for extension?

  Is the class closed for modification?

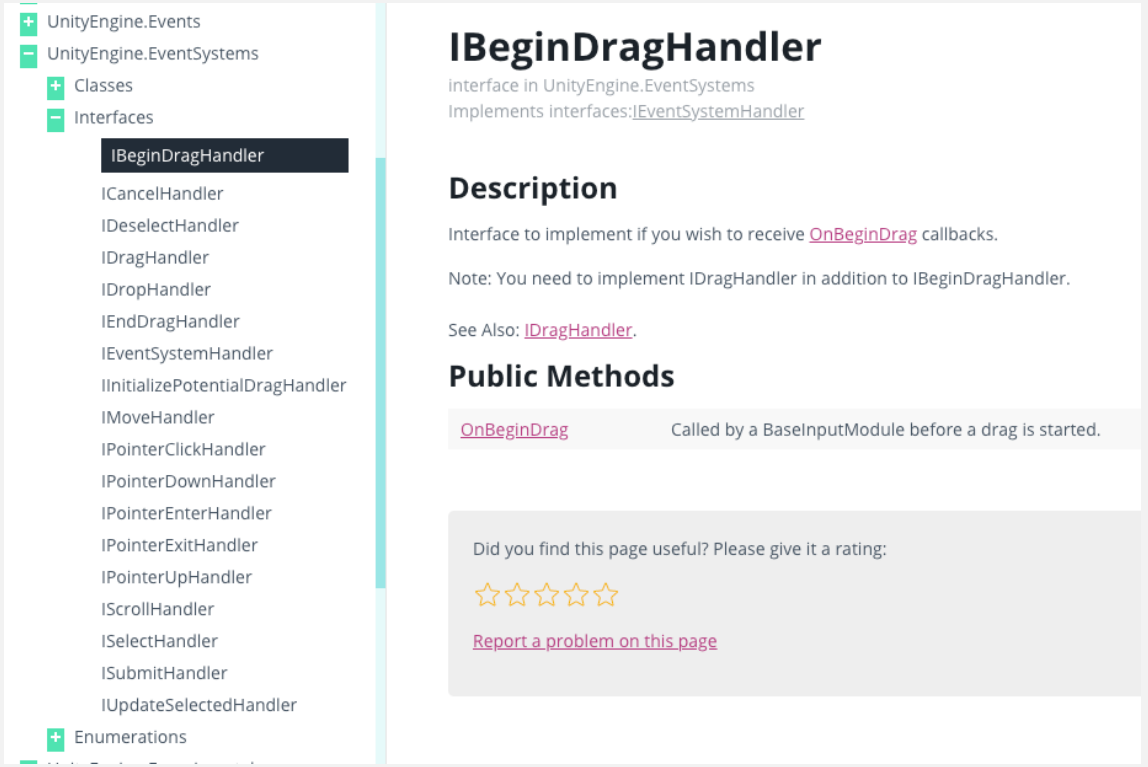
INTERFACE SEGREGATION PRINCIPLE

- **Clients shouldn't be forced to depend on methods they do not use.**

Make your interfaces narrow enough so that the client classes don't have to implement behaviors they don't use.

Think of this principle when you are more comfortable with using interfaces.

Don't overdo it.



CODE REVIEW QUESTIONS SO FAR:

- From the slides above:
- Class names clear? Method names clear? Variable names clear?
- Are there any magic values?
- Are there any comments and are they clear?
- Is there any repeated part in the code?
- Is there any method in the class that is longer than one screen? If so can it be divided?
- Does a class have a single responsibility?
- Is the class open for extension?
- Is the class closed for modification?

LAB 2 ASSIGNMENT

- Shop Model version 1:

- Apply factory method/abstract factory/prototype pattern to the shop
- to procedurally generate items and upgrade them.

QUESTIONS?