

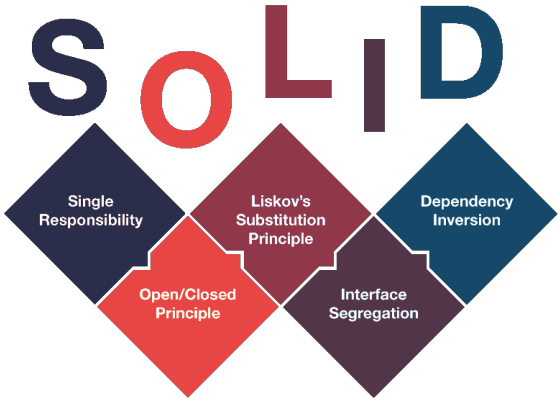
Pontificia Universidad Católica de Chile
Escuela de Ingeniería
Departamento de Ciencia de la Computación

Clase 11: Principios SOLID (parte 2)

Francisco Gazitúa Requena (fco_gazitua_requena@uc.cl)

IIC2113 Diseño Detallado de Software

09 de Septiembre, 2025


SOLID
Single Responsibility
Liskov's Substitution Principle
Dependency Inversion
Open/Closed Principle
Interface Segregation

The Open Closed Principle

The Open Closed Principle (OCP)

The Open Closed Principle (OCP)

"*A module should be open for extension but closed for modification.*"

The Open Closed Principle (OCP)

The Open Closed Principle (OCP)

"*A module should be open for extension but closed for modification.*"

Este es el principio más importante. Si puedes agregar nuevos features sin modificar tu código entonces estás en un escenario ideal. Tu sistema es robusto y flexible porque tus cambios no propagan otros cambios ni rompen módulos existentes.

The Open Closed Principle (OCP)

The Open Closed Principle (OCP)

"*A module should be open for extension but closed for modification.*"

Este es el principio más importante. Si puedes agregar nuevos features sin modificar tu código entonces estás en un escenario ideal. Tu sistema es robusto y flexible porque tus cambios no propagan otros cambios ni rompen módulos existentes.

Este principio nos dice lo siguiente:

1. Si una clase ya fue desarrollada y está en uso, alterar su código puede ser riesgoso

The Open Closed Principle (OCP)

The Open Closed Principle (OCP)

"*A module should be open for extension but closed for modification.*"

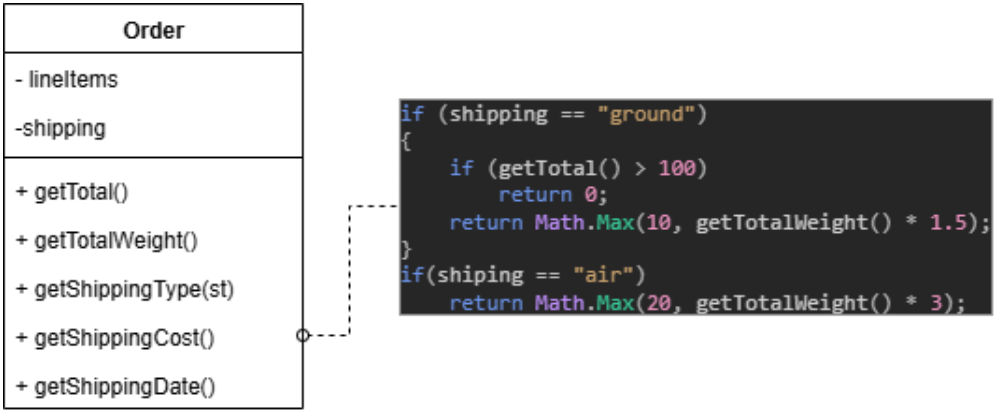
Este es el principio más importante. Si puedes agregar nuevos features sin modificar tu código entonces estás en un escenario ideal. Tu sistema es robusto y flexible porque tus cambios no propagan otros cambios ni rompen módulos existentes.

Este principio nos dice lo siguiente:

1. Si una clase ya fue desarrollada y está en uso, alterar su código puede ser riesgoso
2. En vez de cambiar su código directamente, podemos alterar su comportamiento creando una subclase y sobrescribiendo parte de su comportamiento original.

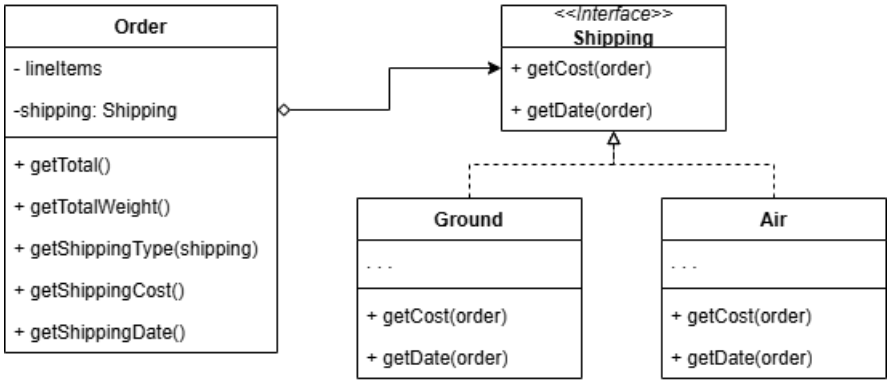
The Open Closed Principle (OCP)

Imaginemos que tenemos la siguiente clase ¿ven algún problema?



The Open Closed Principle (OCP)

Podemos solucionarlo extrayendo funcionalidades de la clase



The Open Closed Principle (OCP)

Sin embargo, el OCP es una MENTIRA!!

The Open Closed Principle (OCP)

Sin embargo, el OCP es una MENTIRA!! Es necesario crear un objeto para usarlo

The Open Closed Principle (OCP)

Sin embargo, el OCP es una MENTIRA!! Es necesario crear un objeto para usarlo

```
2  public static class ShipmentFactory
3  {
4      public static Shipping CreateShipmentMethod(ShipmentMethod method)
5      {
6          if(method == ShipmentMethod.Ground)
7              return new Ground()
8          if(method == ShipmentMethod.Air)
9              return new Air()
10         throw new NotImplementedException()
11     }
12 }
```

The Open Closed Principle (OCP)

Sin embargo, el OCP es una MENTIRA!! Es necesario crear un objeto para usarlo

```
2  public static class ShipmentFactory
3  {
4      public static Shipping CreateShipmentMethod(ShipmentMethod method)
5      {
6          if(method == ShipmentMethod.Ground)
7              return new Ground()
8          if(method == ShipmentMethod.Air)
9              return new Air()
10         throw new NotImplementedException()
11     }
12 }
```

Si solo modificamos nuestra Factory y nada más, estamos “cumpliendo” el OCP

The Open Closed Principle (OCP)

Recordemos la actividad `Letters`.

The Open Closed Principle (OCP)

Recordemos la actividad `Letters`.

Básicamente, existen cuatro stats: `Atk`, `Def`, `Res` y `Spd`. También hay efectos que cambian sus valores. Cada efecto está asociado a una letra:

- a: Aumenta `Atk` en 5.
- b: Aumenta `Spd` en 2.
- ...
- g: Aumenta `Atk`, `Def`, `Res` y `Spd` en 1.

The Open Closed Principle (OCP)

Recordemos la actividad `Letters`.

Básicamente, existen cuatro stats: `Atk`, `Def`, `Res` y `Spd`. También hay efectos que cambian sus valores. Cada efecto está asociado a una letra:

- `a`: Aumenta `Atk` en 5.
- `b`: Aumenta `Spd` en 2.
- ...
- `g`: Aumenta `Atk`, `Def`, `Res` y `Spd` en 1.

Todas las stats parten en cero. El programa recibe un string (e.j., `"bda"`) y ejecuta los efectos de cada caracter en el string (e.j., primero `b`, luego `d` y luego `a`). Finalmente, se retornan los valores de las stats después de aplicados los efectos.

The Open Closed Principle (OCP)

```
1  public class Game
2  {
3    private int _atk = 0;
4    private int _spd = 0;
5    private int _res = 0;
6    private int _def = 0;
7    private string _actions;
8
9    public Game(string actions)
10     => _actions = actions;
11
12   public string Play()
13   {
14     foreach (var action in _actions)
15     {
16       if (action == 'a')
17         _atk += 5;
```

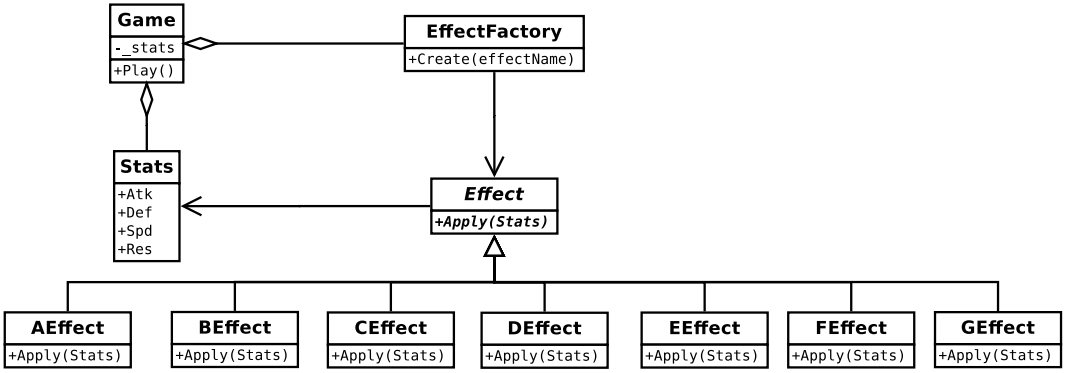
The Open Closed Principle (OCP)

```
18        if (action == 'b')
19          _spd += 2;
20        if (action == 'c')
21          _res += 7;
22        if (action == 'd')
23          _def += 3;
24        if (action == 'e')
25        {
26          _atk += 2;
27          _def += 2;
28          _res += 6;
29        }
30        if (action == 'f')
31        {
32          _atk += 2;
33          _res += 2;
34          _spd += 2;
```

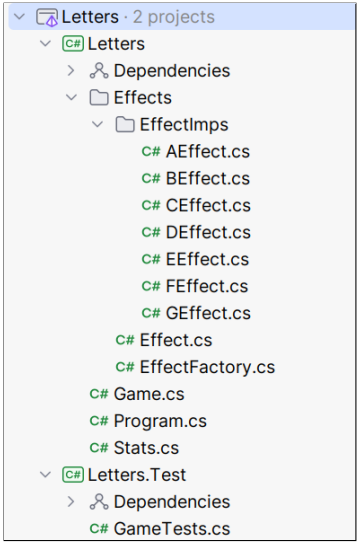
The Open Closed Principle (OCP)

```
35          }
36          if (action == 'g')
37          {
38              _atk += 1;
39              _def += 1;
40              _res += 1;
41              _spd += 1;
42          }
43      }
44
45      return $"{_atk}-{_spd}-{_res}-{_def}";
46    }
47  }
```

The Open Closed Principle (OCP)



The Open Closed Principle (OCP)



The Open Closed Principle (OCP)

```
1  public class Game
2  {
3    private readonly Stats _stats = new();
4    private readonly List<Effect> _effects = new();
5
6    public Game(string actions)
7    {
8      EffectFactory factory = new EffectFactory();
9      foreach (var action in actions)
10       _effects.Add(factory.Create(action));
11   }
12
13   public string Play()
14   {
15     foreach (var effect in _effects)
16       effect.Apply(_stats);
17     return $"{_stats.Atk}-{_stats.Spd}-{_stats.Res}-{_stats.Def}";
18   }
19 }
```

The Open Closed Principle (OCP)

Los stats son una estructura de datos con atributos públicos:

```
1 public class Stats
2 {
3   public int Atk = 0;
4   public int Spd = 0;
5   public int Res = 0;
6   public int Def = 0;
7 }
```

The Open Closed Principle (OCP)

Los efectos son una clase abstracta con un método para aplicar el efecto:

```
1 public abstract class Effect
2 {
3   public abstract void Apply(Stats stats);
4 }
```

The Open Closed Principle (OCP)

Los efectos son una clase abstracta con un método para aplicar el efecto:

```
1 public abstract class Effect
2 {
3   public abstract void Apply(Stats stats);
4 }
```

Luego cada efecto debe implementar el método `Apply(...)`:

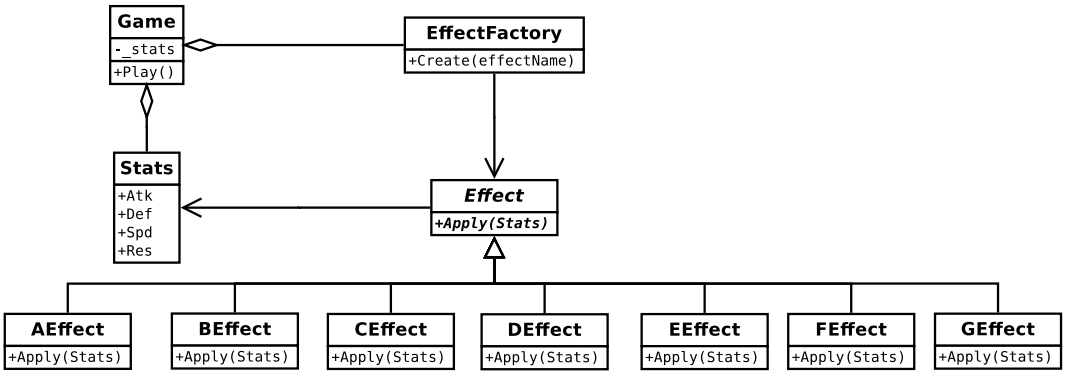
```
1 public class FEffect : Effect
2 {
3   public override void Apply(Stats stats)
4   {
5     stats.Atk += 2;
6     stats.Res += 2;
7     stats.Spd += 2;
8   }
9 }
```

Finalmente, la factory crea los objetos según el nombre del efecto:

```
public class EffectFactory {
  public Effect Create(char name) {
    if (name == 'a')
      return new AEffect();
    if (name == 'b')
      return new BEffect();
    if (name == 'c')
      return new CEffect();
    if (name == 'd')
      return new DEffect();
    if (name == 'e')
      return new EEffect();
    if (name == 'f')
      return new FEffect();
    if (name == 'g')
      return new GEffect();
    throw new ApplicationException();
  }
}
```

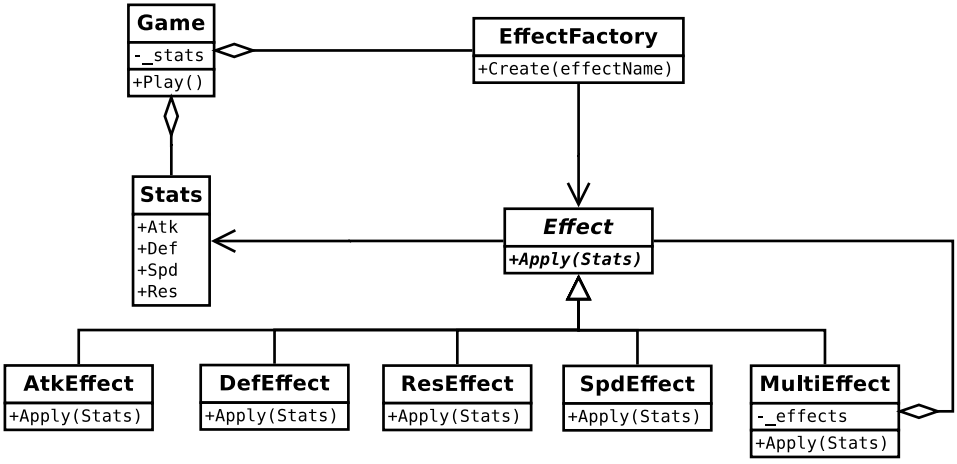
The Open Closed Principle (OCP)

Problema: Hay una explosión de clases en el modelo anterior.

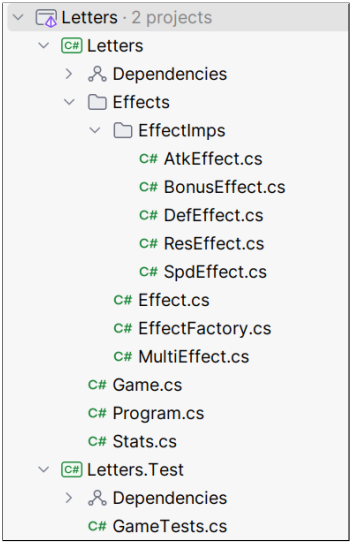


The Open Closed Principle (OCP)

Alternativa: Tener un conjunto acotado de efectos bases y combinarlos.



The Open Closed Principle (OCP)



The Open Closed Principle (OCP)

```
1  public class Game
2  {
3    private readonly Stats _stats = new();
4    private readonly List<Effect> _effects = new();
5
6    public Game(string actions)
7    {
8      EffectFactory factory = new EffectFactory();
9      foreach (var action in actions)
10       _effects.Add(factory.Create(action));
11   }
12
13   public string Play()
14   {
15     foreach (var effect in _effects)
16       effect.Apply(_stats);
17     return $"{_stats.Atk}-{_stats.Spd}-{_stats.Res}-{_stats.Def}";
18   }
19 }
```

The Open Closed Principle (OCP)

```
1 public abstract class Effect {
2   public abstract void Apply(Stats stats);
3 }
```

The Open Closed Principle (OCP)

```
public abstract class Effect {
  public abstract void Apply(Stats stats);
}
```

```
public abstract class BonusEffect : Effect {
  protected readonly int Bonus;

  protected BonusEffect(int bonus)
    => Bonus = bonus;
}
```

The Open Closed Principle (OCP)

```
public abstract class Effect {
  public abstract void Apply(Stats stats);
}
```

```
public abstract class BonusEffect : Effect {
  protected readonly int Bonus;

  protected BonusEffect(int bonus)
    => Bonus = bonus;
}
```

```
public class AtkEffect : BonusEffect {
  public AtkEffect(int bonus) : base(bonus) { }

  public override void Apply(Stats stats)
    => stats.Atk += Bonus;
}
```

The Open Closed Principle (OCP)

```
public abstract class Effect {
  public abstract void Apply(Stats stats);
}
```

```
public class MultiEffect : Effect
{
  private readonly Effect[] _effects;

  public MultiEffect(params Effect[] effects)
    => _effects = effects;

  public override void Apply(Stats stats)
  {
    foreach (var effect in _effects)
      effect.Apply(stats);
  }
}
```

The Open Closed Principle (OCP)

```
public class EffectFactory {
  public Effect Create(char name)
  {
    if (name == 'a')
      return new AtkEffect(5);
    if (name == 'b')
      return new SpdEffect(2);
    if (name == 'c')
      return new ResEffect(7);
    if (name == 'd')
      return new DefEffect(3);
    if (name == 'e')
      return new MultiEffect(
        new AtkEffect(2),
        new DefEffect(2),
        new ResEffect(6));
```

The Open Closed Principle (OCP)

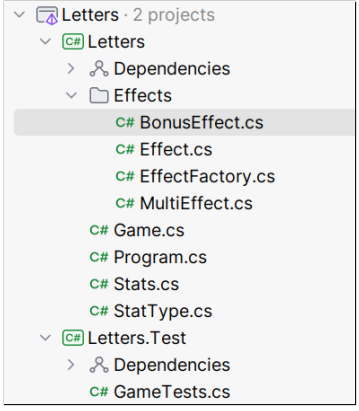
```
17      if (name == 'f')
18        return new MultiEffect(
19          new AtkEffect(2),
20          new ResEffect(2),
21          new SpdEffect(2));
22      if (name == 'g')
23        return new MultiEffect(
24          new AtkEffect(1),
25          new DefEffect(1),
26          new ResEffect(1),
27          new SpdEffect(1));
28      throw new ApplicationException();
29    }
30  }
```

The Open Closed Principle (OCP)

Importante: Esta actividad era con fines pedagógicos... y las soluciones anteriores asumían que los efectos base eran muy distintos entre sí.

The Open Closed Principle (OCP)

Importante: Esta actividad era con fines pedagógicos... y las soluciones anteriores asumían que los efectos base eran muy distintos entre sí.



The Open Closed Principle (OCP)

```
1 public enum StatType { Atk, Def, Res, Spd }
```

The Open Closed Principle (OCP)

```
public enum StatType { Atk, Def, Res, Spd }
```

```
public class BonusEffect : Effect {
  private readonly StatType _targetStat;
  private readonly int _bonus;

  public BonusEffect(StatType targetStat, int bonus) {
    _targetStat = targetStat;
    _bonus = bonus;
  }

  public override void Apply(Stats stats) {
    if (_targetStat == StatType.Atk) stats.Atk += _bonus;
    if (_targetStat == StatType.Def) stats.Def += _bonus;
    if (_targetStat == StatType.Res) stats.Res += _bonus;
    if (_targetStat == StatType.Spd) stats.Spd += _bonus;
  }
}
```

The Open Closed Principle (OCP)

```
1  public class EffectFactory {
2    public Effect Create(char name)
3    {
4      if (name == 'a')
5        return new BonusEffect(StatType.Atk, 5);
6      if (name == 'b')
7        return new BonusEffect(StatType.Spd, 2);
8      if (name == 'c')
9        return new BonusEffect(StatType.Res, 7);
10     if (name == 'd')
11       return new BonusEffect(StatType.Def, 3);
12     if (name == 'e')
13       return new MultiEffect(
14         new BonusEffect(StatType.Atk, 2),
15         new BonusEffect(StatType.Def, 2),
16         new BonusEffect(StatType.Res, 6));
```

The Open Closed Principle (OCP)

```
17      if (name == 'f')
18        return new MultiEffect(
19          new BonusEffect(StatType.Atk, 2),
20          new BonusEffect(StatType.Res, 2),
21          new BonusEffect(StatType.Spd, 2));
22      if (name == 'g')
23        return new MultiEffect(
24          new BonusEffect(StatType.Atk, 1),
25          new BonusEffect(StatType.Def, 1),
26          new BonusEffect(StatType.Res, 1),
27          new BonusEffect(StatType.Spd, 1));
28      throw new ApplicationException();
29    }
30  }
```

The Open Closed Principle (OCP)

Complejicemos un poco el problema

The Open Closed Principle (OCP)

Complejicemos un poco el problema

Agreguemos las siguientes condiciones para que un efecto sea aplicado:

- a: Siempre se aplica.
- b: Aplica solo si Atk ≥ 1.
- c: Aplica solo si Spd < 2.
- d: Aplica solo si Atk ≥ 4 y Res ≥ 4.
- e: Aplica solo si Atk ≥ 4 o Res ≥ 4.
- f: Aplica solo si Def ≥ 5.
- g: Aplica solo si Atk < 1, Def < 1, Spd < 1 y Res < 1.

The Open Closed Principle (OCP)

Complejicemos un poco el problema

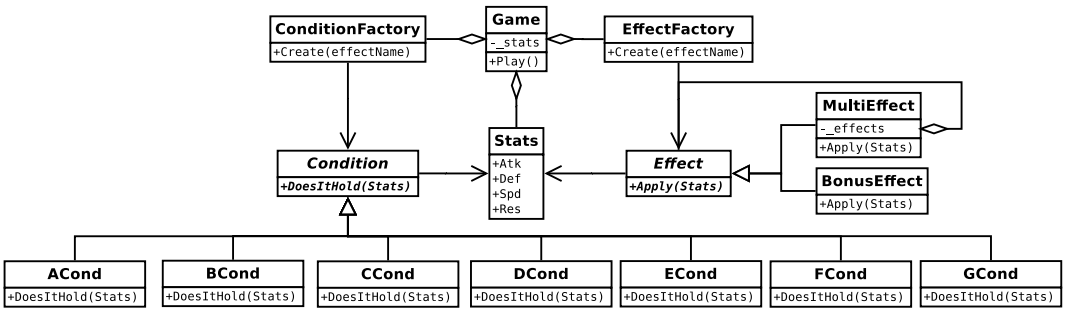
Agreguemos las siguientes condiciones para que un efecto sea aplicado:

- a: Siempre se aplica.
- b: Aplica solo si Atk ≥ 1.
- c: Aplica solo si Spd < 2.
- d: Aplica solo si Atk ≥ 4 y Res ≥ 4.
- e: Aplica solo si Atk ≥ 4 o Res ≥ 4.
- f: Aplica solo si Def ≥ 5.
- g: Aplica solo si Atk < 1, Def < 1, Spd < 1 y Res < 1.

¿Cómo podríamos lograr esto?

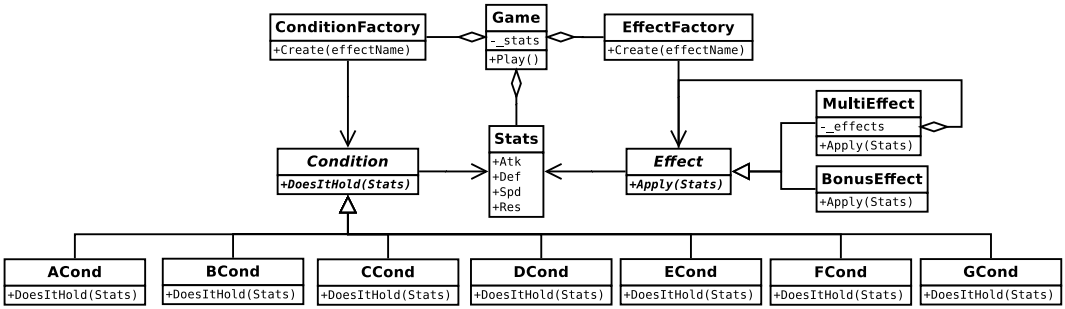
The Open Closed Principle (OCP)

Opción 1: Crear una clase por condición



The Open Closed Principle (OCP)

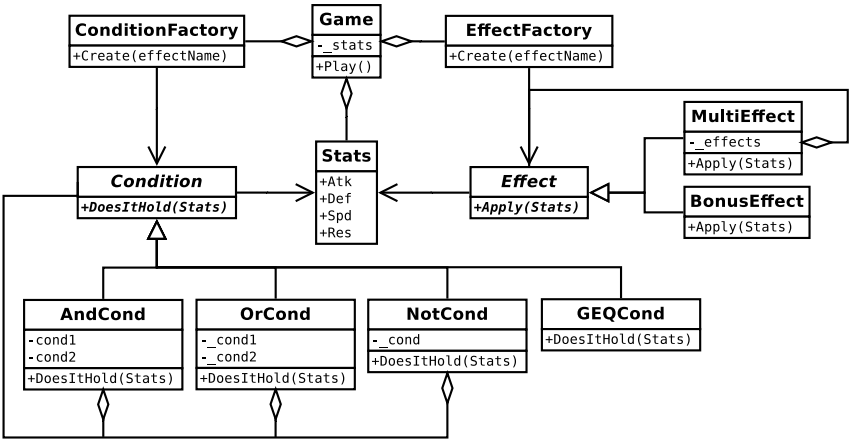
Opción 1: Crear una clase por condición



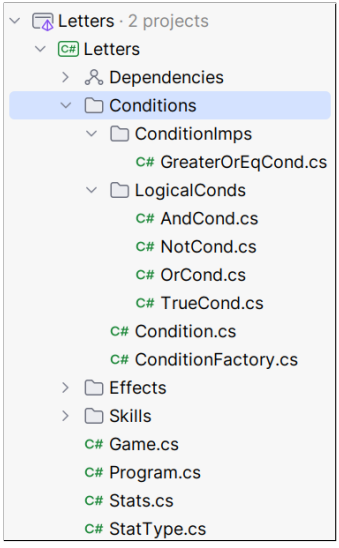
¿Cuál es el problema?

The Open Closed Principle (OCP)

Opción 2: Pocas clases y poco código duplicado



The Open Closed Principle (OCP)



The Open Closed Principle (OCP)

La definición de una condición queda igual:

```
1 public abstract class Condition {
2   public abstract bool DoesHold(Stats stats);
3 }
```

The Open Closed Principle (OCP)

La definición de una condición queda igual:

```
public abstract class Condition {
  public abstract bool DoesHold(Stats stats);
}
```

Ahora definimos una sola condición para mayor o igual:

```
public class GreaterOrEqCond : Condition {
  private readonly StatType _targetStat;
  private readonly int _threshold;

  public GreaterOrEqCond(StatType targetStat, int threshold) {
    _targetStat = targetStat;
    _threshold = threshold;
  }

  public override bool DoesHold(Stats stats)
    => stats.GetStat(_targetStat) >= _threshold;
}
```

The Open Closed Principle (OCP)

Después definimos condiciones lógicas:

```
public class TrueCond : Condition {
  public override bool DoesHold(Stats stats)
    => true;
}
```

The Open Closed Principle (OCP)

Después definimos condiciones lógicas:

```
public class TrueCond : Condition {
  public override bool DoesHold(Stats stats)
    => true;
}
```

Condición *not*:

```
public class NotCond : Condition {
  private readonly Condition _condition;

  public NotCond(Condition condition)
    => _condition = condition;

  public override bool DoesHold(Stats stats)
    => !_condition.DoesHold(stats);
}
```

The Open Closed Principle (OCP)

Condición *and*:

```
public class AndCond: Condition {
  private readonly Condition _firstCond;
  private readonly Condition _secondCond;

  public AndCond(Condition firstCond, Condition secondCond) {
    _firstCond = firstCond;
    _secondCond = secondCond;
  }

  public override bool DoesHold(Stats stats)
    => _firstCond.DoesHold(stats) && _secondCond.DoesHold(stats);
}
```

The Open Closed Principle (OCP)

Condición *or*:

```
public class OrCond: Condition {
  private readonly Condition _firstCond;
  private readonly Condition _secondCond;

  public OrCond(Condition firstCond, Condition secondCond) {
    _firstCond = firstCond;
    _secondCond = secondCond;
  }

  public override bool DoesHold(Stats stats)
    => _firstCond.DoesHold(stats) || _secondCond.DoesHold(stats);
}
```

The Open Closed Principle (OCP)

Y creamos un factory de condiciones:

```
public class ConditionFactory {
  public Condition Create(char name) {
    if (name == 'a')
      return new TrueCond();
    if (name == 'b')
      return new GreaterOrEqCond(StatType.Atk, 1);
    if (name == 'c')
      return new NotCond(new GreaterOrEqCond(StatType.Spd, 2));
    if (name == 'd')
      return new AndCond(
        new GreaterOrEqCond(StatType.Atk, 4),
        new GreaterOrEqCond(StatType.Res, 4));
    if (name == 'e')
      return new OrCond(
        new GreaterOrEqCond(StatType.Atk, 4),
        new GreaterOrEqCond(StatType.Res, 4));
```

The Open Closed Principle (OCP)

Y creamos un factory de condiciones:

```
17      if (name == 'f')
18        return new GreaterOrEqCond(StatType.Def, 5);
19      if (name == 'g')
20        return new NotCond(
21          new OrCond(
22            new OrCond(
23              new GreaterOrEqCond(StatType.Atk, 1),
24              new GreaterOrEqCond(StatType.Res, 1)),
25            new OrCond(
26              new GreaterOrEqCond(StatType.Def, 1),
27              new GreaterOrEqCond(StatType.Spd, 1))));
28      throw new ApplicationException();
29    }
30  }
```

The Open Closed Principle (OCP)

Y creamos un factory de condiciones:

```
17      if (name == 'f')
18        return new GreaterOrEqCond(StatType.Def, 5);
19      if (name == 'g')
20        return new NotCond(
21          new OrCond(
22            new OrCond(
23              new GreaterOrEqCond(StatType.Atk, 1),
24              new GreaterOrEqCond(StatType.Res, 1)),
25            new OrCond(
26              new GreaterOrEqCond(StatType.Def, 1),
27              new GreaterOrEqCond(StatType.Spd, 1))));
28      throw new ApplicationException();
29    }
30  }
```

... ese `OrCond` de 4 condiciones se ve feo.

The Open Closed Principle (OCP)

Este `OrCond` permite recibir cualquier número de condiciones:

```
public class OrCond: Condition {
  private readonly Condition[] _conditions;

  public OrCond(params Condition[] conditions)
    => _conditions = conditions;

  public override bool DoesHold(Stats stats)
  {
    foreach(Condition condition in _conditions)
      if(condition.DoesHold(stats))
        return true;
    return false;
  }
}
```

The Open Closed Principle (OCP)

Así el factory queda un poquito mejor :)

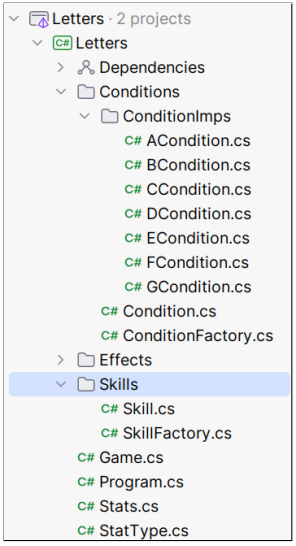
```
17      if (name == 'f')
18        return new GreaterOrEqCond(StatType.Def, 5);
19      if (name == 'g')
20        return new NotCond(
21          new OrCond(
22            new OrCond(
23              new GreaterOrEqCond(StatType.Atk, 1),
24              new GreaterOrEqCond(StatType.Res, 1)),
25            new OrCond(
26              new GreaterOrEqCond(StatType.Def, 1),
27              new GreaterOrEqCond(StatType.Spd, 1))));
28      throw new ApplicationException();
29    }
30  }
```

The Open Closed Principle (OCP)

Así el factory queda un poquito mejor :)

```
17      if (name == 'f')
18        return new GreaterOrEqCond(StatType.Def, 5);
19      if (name == 'g')
20        return new NotCond(
21          new OrCond(
22            new GreaterOrEqCond(StatType.Atk, 1),
23            new GreaterOrEqCond(StatType.Res, 1),
24            new GreaterOrEqCond(StatType.Def, 1),
25            new GreaterOrEqCond(StatType.Spd, 1)));
26      throw new ApplicationException();
27    }
28  }
```

The Open Closed Principle (OCP)



The Open Closed Principle (OCP)

Cada `Skill` es una condición con un efecto:

```
1  public class Skill {
2    private readonly Condition _condition;
3    private readonly Effect _effect;
4
5    public Skill(Condition condition, Effect effect) {
6      _condition = condition;
7      _effect = effect;
8    }
9
10   public void ApplyIfApplicable(Stats stats) {
11     if(_condition.DoesHold(stats))
12       _effect.Apply(stats);
13   }
14 }
```

The Open Closed Principle (OCP)

Para crear *skills* usamos las factories de condiciones y efectos:

```
1  public class SkillFactory {
2    private readonly ConditionFactory _conditionFactory = new ();
3    private readonly EffectFactory _effectFactory = new ();
4
5    public Skill Create(char name) {
6      Condition condition = _conditionFactory.Create(name);
7      Effect effect = _effectFactory.Create(name);
8      return new Skill(condition, effect);
9    }
10 }
```

The Open Closed Principle (OCP)

Finalmente, `Game.cs` nos queda casi igual que cuando eran solo efectos:

```
1  public class Game {
2    private readonly Stats _stats = new();
3    private readonly List<Skill> _skills = new();
4
5    public Game(string actions) {
6      SkillFactory factory = new SkillFactory();
7      foreach (var action in actions)
8        _skills.Add(factory.Create(action));
9    }
10
11   public string Play() {
12     foreach (var skill in _skills)
13       skill.ApplyIfApplicable(_stats);
14     return $"{_stats.Atk}-{_stats.Spd}-{_stats.Res}-{_stats.Def}";
15   }
16 }
```

The Open Closed Principle (OCP)

Existen muchas violaciones típicas del OCP:

1. Utilizar estructuras de datos cuando queremos crear más tipos a futuro (por ejemplo, habilidades)

The Open Closed Principle (OCP)

Existen muchas violaciones típicas del OCP:

1. Utilizar estructuras de datos cuando queremos crear más tipos a futuro (por ejemplo, habilidades)
2. Hacer abstracciones muy grandes (Ej. Habilidad en vez de Condición + Efecto)

The Open Closed Principle (OCP)

Existen muchas violaciones típicas del OCP:

1. Utilizar estructuras de datos cuando queremos crear más tipos a futuro (por ejemplo, habilidades)
2. Hacer abstracciones muy grandes (Ej. Habilidad en vez de Condición + Efecto)
3. Tener `ifs` sobre el tipo de habilidad fuera de la factory

The Open Closed Principle (OCP)

Existen muchas violaciones típicas del OCP:

1. Utilizar estructuras de datos cuando queremos crear más tipos a futuro (por ejemplo, habilidades)
2. Hacer abstracciones muy grandes (Ej. Habilidad en vez de Condición + Efecto)
3. Tener `ifs` sobre el tipo de habilidad fuera de la factory
4. Usar implementaciones concretas y no abstracciones

The Open Closed Principle (OCP)

Existen muchas violaciones típicas del OCP:

1. Utilizar estructuras de datos cuando queremos crear más tipos a futuro (por ejemplo, habilidades)
2. Hacer abstracciones muy grandes (Ej. Habilidad en vez de Condición + Efecto)
3. Tener `ifs` sobre el tipo de habilidad fuera de la factory
4. Usar implementaciones concretas y no abstracciones
5. Agregar un método a una clase por cada tipo de habilidad

The Open Closed Principle (OCP)

Existen muchas violaciones típicas del OCP:

1. Utilizar estructuras de datos cuando queremos crear más tipos a futuro (por ejemplo, habilidades)
2. Hacer abstracciones muy grandes (Ej. Habilidad en vez de Condición + Efecto)
3. Tener `ifs` sobre el tipo de habilidad fuera de la factory
4. Usar implementaciones concretas y no abstracciones
5. Agregar un método a una clase por cada tipo de habilidad

Prefiere abstracción y polimorfismo cuando tengas que agregar nuevos tipos de cosas entrega a entrega:

The Open Closed Principle (OCP)

Existen muchas violaciones típicas del OCP:

1. Utilizar estructuras de datos cuando queremos crear más tipos a futuro (por ejemplo, habilidades)
2. Hacer abstracciones muy grandes (Ej. Habilidad en vez de Condición + Efecto)
3. Tener `ifs` sobre el tipo de habilidad fuera de la factory
4. Usar implementaciones concretas y no abstracciones
5. Agregar un método a una clase por cada tipo de habilidad

Prefiere abstracción y polimorfismo cuando tengas que agregar nuevos tipos de cosas entrega a entrega:

1. Tenemos que agregar nuevos tipos de afinidades

The Open Closed Principle (OCP)

Existen muchas violaciones típicas del OCP:

1. Utilizar estructuras de datos cuando queremos crear más tipos a futuro (por ejemplo, habilidades)
2. Hacer abstracciones muy grandes (Ej. Habilidad en vez de Condición + Efecto)
3. Tener `ifs` sobre el tipo de habilidad fuera de la factory
4. Usar implementaciones concretas y no abstracciones
5. Agregar un método a una clase por cada tipo de habilidad

Prefiere abstracción y polimorfismo cuando tengas que agregar nuevos tipos de cosas entrega a entrega:

1. Tenemos que agregar nuevos tipos de afinidades
2. Tenemos que agregar nuevos mensajes para anunciar los efectos del juego

The Open Closed Principle (OCP)

Existen muchas violaciones típicas del OCP:

1. Utilizar estructuras de datos cuando queremos crear más tipos a futuro (por ejemplo, habilidades)
2. Hacer abstracciones muy grandes (Ej. Habilidad en vez de Condición + Efecto)
3. Tener `ifs` sobre el tipo de habilidad fuera de la factory
4. Usar implementaciones concretas y no abstracciones
5. Agregar un método a una clase por cada tipo de habilidad

Prefiere abstracción y polimorfismo cuando tengas que agregar nuevos tipos de cosas entrega a entrega:

1. Tenemos que agregar nuevos tipos de afinidades
2. Tenemos que agregar nuevos mensajes para anunciar los efectos del juego
3. Tenemos que agregar nuevas habilidades... pero algunas varían muy poco.

The Liskov Substitution Principle

The Liskov Substitution Principle (LSP)



Barbara Liskov
Programming Methodology Group
MIT

The Liskov Substitution Principle (LSP)



Barbara Liskov
Programming Methodology Group
MIT

- Pionera en metodologías de abstracción de datos
- Desarrollo CLU, uno de los primeros lenguajes de programación en utilizar abstracción de tipos de datos
- Introdujo `The Liskov Substitution Principle (LSP)`

The Liskov Substitution Principle (LSP)



Barbara Liskov
Programming Methodology Group
MIT

The key to building reliable software is to understand that complexity is the enemy.

The Liskov Substitution Principle (LSP)



The Liskov Substitution Principle (LSP)



Nadie sabe como construir software:

The major cause of the software crisis is that the machines have become several orders of magnitude more powerful! To put it quite bluntly: as long as there were no machines, programming was no problem at all; when we had a few weak computers, programming became a mild problem, and now we have gigantic computers, programming has become an equally gigantic problem.

The Liskov Substitution Principle (LSP)



Se manifiesta de varias formas:

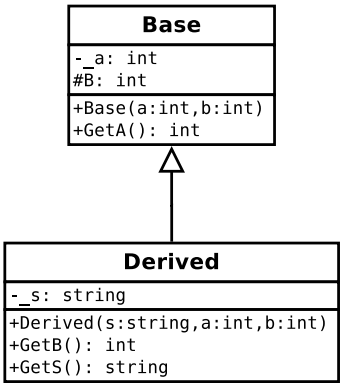
- Proyectos que exceden su presupuesto
- Proyectos que se retrasan
- Software que no cumple los requerimientos
- Código imposible de entender y mantener
- Vaporware

The Liskov Substitution Principle (LSP)

Solución: Abstracción

The Liskov Substitution Principle (LSP)

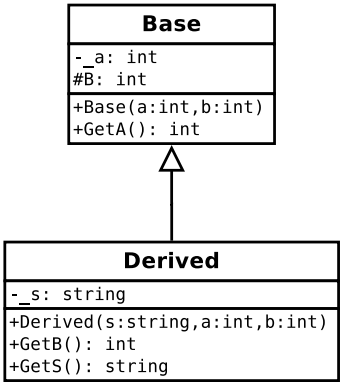
Solución: Abstracción



La abstracción nos permite definir muchas formas de hacer la misma cosa de forma sencilla y extensible.

The Liskov Substitution Principle (LSP)

Solución: Abstracción



La abstracción nos permite definir muchas formas de hacer la misma cosa de forma sencilla y extensible.

No me interesa que clase estoy usando siempre que herede de `Base`

The Liskov Substitution Principle (LSP)

The Liskov Substitution Principle (LSP)

"*Subclasses should be substitutable for their base classes.*"

The Liskov Substitution Principle (LSP)

The Liskov Substitution Principle (LSP)

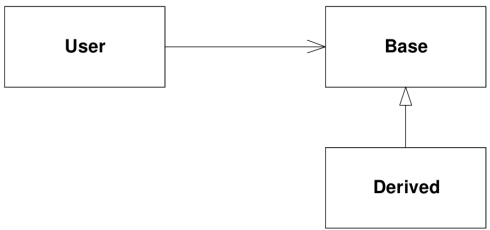
"*Subclasses should be substitutable for their base classes.*"

Existe otra formulación:

The Liskov Substitution Principle (LSP)

"*You should be able to pass objects of a subclass in place of objects of the parent class without braking the client code.*"

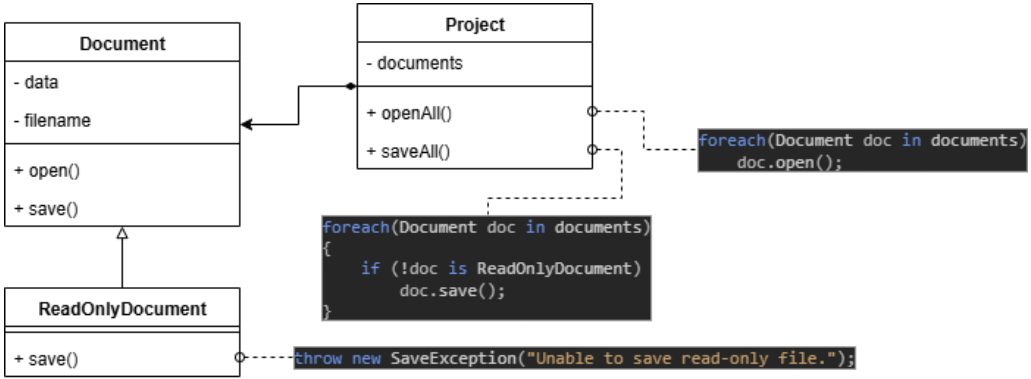
The Liskov Substitution Principle (LSP)



El código de la clase `User` debe funcionar independiente de que clase `Derived` le den.

The Liskov Substitution Principle (LSP)

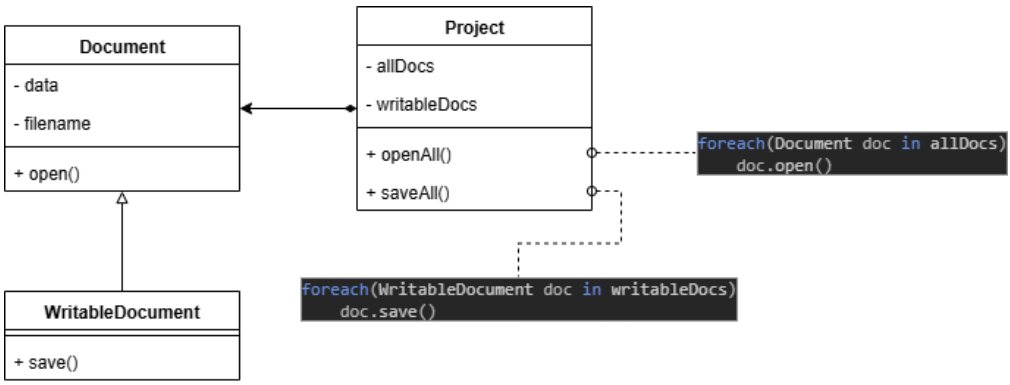
Imaginemos que tenemos las siguientes clases:



¿Ven el problema?

The Liskov Substitution Principle (LSP)

Podemos solucionarlo reordenando nuestra abstracción



The Liskov Substitution Principle (LSP)

En el proyecto, las violaciones del LSP son simples, pero importantes.

The Liskov Substitution Principle (LSP)

En el proyecto, las violaciones del LSP son simples, pero importantes.
Normalmente una unidad puede usar una habilidad si su MP actual es mayor al costo de la habilidad.

The Liskov Substitution Principle (LSP)

En el proyecto, las violaciones del LSP son simples, pero importantes.
Normalmente una unidad puede usar una habilidad si su MP actual es mayor al costo de la habilidad. Sin embargo, hay algunas habilidades con más condiciones

The Liskov Substitution Principle (LSP)

En el proyecto, las violaciones del LSP son simples, pero importantes.
Normalmente una unidad puede usar una habilidad si su MP actual es mayor al costo de la habilidad. Sin embargo, hay algunas habilidades con más condiciones

```
public Skill[] GetUsableSkills(int MP, int HP) {
    skill[] usableSkills = new();
    foreach (Skill skill in _unit.Skills) {
        if (skill.Cost <= MP){
            if (skill.Name == "Blood Ritual"){
                if(HP>1)
                    usableSkills.Add(skill);
            }
            else
                usableSkills.Add(skill);
        }
    }
    return usableSkills;
}
```

The Liskov Substitution Principle (LSP)

Si es que tenemos un if que chequea el nombre de la habilidad (o algun otro valor muy especifico) fuera de la factory estamos rompiendo el LSP

The Liskov Substitution Principle (LSP)

Si es que tenemos un if que chequea el nombre de la habilidad (o algun otro valor muy especifico) fuera de la factory estamos rompiendo el LSP

Solución

```
public Skill[] GetUsableSkills() {
    skill[] usableSkills = new();
    foreach (Skill skill in _unit.Skills)
    {
        if(skill.UserMeetsUseCondition(_unit))
            usableSkills.Add(skill);
    }
    return usableSkills;
}
```

The Liskov Substitution Principle (LSP)



Barbara Liskov
Programming Methodology Group
MIT

Modularity based on abstraction is the way things are done. This had become so embedded in the programming culture that my own graduate students didn't realize there was a before [...] they just thought this was the way things were.

The Liskov Substitution Principle (LSP)



Barbara Liskov
Programming Methodology Group
MIT

Modularity based on abstraction is the way things are done. This had become so embedded in the programming culture that my own graduate students didn't realize there was a before [...] they just thought this was the way things were.
And my husband was on the Internet every day when i got the turing award [...] and one day he saw a quote by someone who said "Why did she get the award? Everybody already knows this"

Consejos para la E2 y E3

Consejos

- Existen múltiples tipos de habilidades, que hacen cosas muy distintas. El código para usar tus habilidades no debería depender del tipo de la habilidad

Consejos

- Existen múltiples tipos de habilidades, que hacen cosas muy distintas. El código para usar tus habilidades no debería depender del tipo de la habilidad
- A medida que avances en la entrega, deberás agregar nuevos tipos de habilidades con nuevos mensajes de anuncio. Piensa en formas de que esto no rompa el OCP

Consejos

- Existen múltiples tipos de habilidades, que hacen cosas muy distintas. El código para usar tus habilidades no debería depender del tipo de la habilidad
- A medida que avances en la entrega, deberás agregar nuevos tipos de habilidades con nuevos mensajes de anuncio. Piensa en formas de que esto no rompa el OCP
- Piensa en que partes de una habilidad varía de habilidad a habilidad ¿Es toda la habilidad, o solo una parte específica?

Consejos

- Existen múltiples tipos de habilidades, que hacen cosas muy distintas. El código para usar tus habilidades no debería depender del tipo de la habilidad
- A medida que avances en la entrega, deberás agregar nuevos tipos de habilidades con nuevos mensajes de anuncio. Piensa en formas de que esto no rompa el OCP
- Piensa en que partes de una habilidad varía de habilidad a habilidad ¿Es toda la habilidad, o solo una parte específica?
- Estas respetando el OCP si agregar un nuevo tipo de de dato solo requiere crear una nueva clase y modificar tu factory. Si debes modificar más que eso, cuestiona si estas rompiendo el OCP o no
- Intenta que la relación entre tus objetos sea lo más abstracta posible.

Actividad