

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

Clase 09: Principios SOLID (parte 1)

Francisco Gazitúa Requena (fco_gazitua_requena@uc.cl)

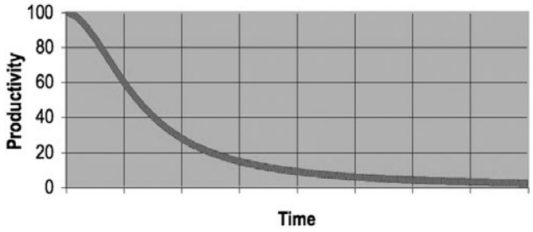
IIC2113 Diseño Detallado de Software

2 de Septiembre, 2025

SOLID

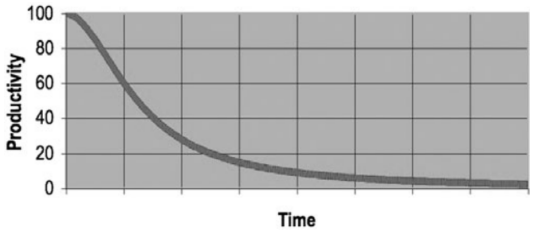
SOLID

En general, todos partimos con un buen diseño que se va deteriorando con el tiempo.



SOLID

En general, todos partimos con un buen diseño que se va deteriorando con el tiempo.



¿Qué síntomas indican que nuestro diseño se está deteriorando?

Síntomas de que tu diseño se está deteriorando

Rigidity: Tu software es difícil de cambiar. Incluso cambios chicos desencadenan una cascada de cambios que te obligan a cambiar módulo tras módulo. Un trabajo de dos días se transforma en semanas. Tu manager empieza a preferir que no se hagan cambios salvo que sean críticos. Esto no termina bien.

Síntomas de que tu diseño se está deteriorando

Rigidity: Tu software es difícil de cambiar. Incluso cambios chicos desencadenan una cascada de cambios que te obligan a cambiar módulo tras módulo. Un trabajo de dos días se transforma en semanas. Tu manager empieza a preferir que no se hagan cambios salvo que sean críticos. Esto no termina bien.

Fragility: Cada cambio rompe varios módulos en tu programa. En particular, se rompen módulos que no están conceptualmente relacionados con el cambio. Cada arreglo introduce más errores. Esto hace que el software sea imposible de mantener.

Síntomas de que tu diseño se está deteriorando

Rigidity: Tu software es difícil de cambiar. Incluso cambios chicos desencadenan una cascada de cambios que te obligan a cambiar módulo tras módulo. Un trabajo de dos días se transforma en semanas. Tu manager empieza a preferir que no se hagan cambios salvo que sean críticos. Esto no termina bien.

Fragility: Cada cambio rompe varios módulos en tu programa. En particular, se rompen módulos que no están conceptualmente relacionados con el cambio. Cada arreglo introduce más errores. Esto hace que el software sea imposible de mantener.

Immobility: Cuesta mucho reutilizar módulos en tu software. A veces querrás reutilizar un módulo, pero no puedes porque el módulo tiene muchas cosas y a ti solo te interesa una parte chica de él. Como resultado, terminas escribiendo un nuevo módulo.

Síntomas de que tu diseño se está deteriorando

Rigidity: Tu software es difícil de cambiar. Incluso cambios chicos desencadenan una cascada de cambios que te obligan a cambiar módulo tras módulo. Un trabajo de dos días se transforma en semanas. Tu manager empieza a preferir que no se hagan cambios salvo que sean críticos. Esto no termina bien.

Fragility: Cada cambio rompe varios módulos en tu programa. En particular, se rompen módulos que no están conceptualmente relacionados con el cambio. Cada arreglo introduce más errores. Esto hace que el software sea imposible de mantener.

Immobility: Cuesta mucho reutilizar módulos en tu software. A veces querrás reutilizar un módulo, pero no puedes porque el módulo tiene muchas cosas y a ti solo te interesa una parte chica de él. Como resultado, terminas escribiendo un nuevo módulo.

Viscosity: Siempre existen dos formas de hacer un cambio. Una forma preserva el diseño. La otra es un hack. Si el costo de hacer el cambio *bien* es mucho mayor que el hack, entonces tu sistema tiene una alta viscosidad.

Síntomas de que tu diseño se está deteriorando

Rigidity: Tu software es difícil de cambiar. Incluso cambios chicos desencadenan una cascada de cambios que te obligan a cambiar módulo tras módulo. Un trabajo de dos días se transforma en semanas. Tu manager empieza a preferir que no se hagan cambios salvo que sean críticos. Esto no termina bien.

Fragility: Cada cambio rompe varios módulos en tu programa. En particular, se rompen módulos que no están conceptualmente relacionados con el cambio. Cada arreglo introduce más errores. Esto hace que el software sea imposible de mantener.

Immobility: Cuesta mucho reutilizar módulos en tu software. A veces querrás reutilizar un módulo, pero no puedes porque el módulo tiene muchas cosas y a ti solo te interesa una parte chica de él. Como resultado, terminas escribiendo un nuevo módulo.

Viscosity: Siempre existen dos formas de hacer un cambio. Una forma preserva el diseño. La otra es un hack. Si el costo de hacer el cambio *bien* es mucho mayor que el hack, entonces tu sistema tiene una alta viscosidad.

¿Han experimentado estos síntomas?

¿Por qué los sistemas se deterioran?

¿Por qué los sistemas se deterioran?

Porque los requerimientos cambian en formas que nuestro diseño inicial no consideró.

¿Por qué los sistemas se deterioran?

Porque los requerimientos cambian en formas que nuestro diseño inicial no consideró. En general, estos cambios se deben hacer rápido y son hechos por ingenieros que no conocen del todo la filosofía del diseño original.

¿Por qué los sistemas se deterioran?

Porque los requerimientos cambian en formas que nuestro diseño inicial no consideró. En general, estos cambios se deben hacer rápido y son hechos por ingenieros que no conocen del todo la filosofía del diseño original. Con trabajo, se hacen los cambios

¿Por qué los sistemas se deterioran?

Porque los requerimientos cambian en formas que nuestro diseño inicial no consideró. En general, estos cambios se deben hacer rápido y son hechos por ingenieros que no conocen del todo la filosofía del diseño original. Con trabajo, se hacen los cambios a costa de romper (un poquito) el diseño original.

¿Por qué los sistemas se deterioran?

Porque los requerimientos cambian en formas que nuestro diseño inicial no consideró. En general, estos cambios se deben hacer rápido y son hechos por ingenieros que no conocen del todo la filosofía del diseño original. Con trabajo, se hacen los cambios a costa de romper (un poquito) el diseño original. Y así, poco a poco, tu software se transforma en un monstruo.

¿Por qué los sistemas se deterioran?

"However, we cannot blame the drifting of the requirements for the degradation of the design. We, as software engineers, know full well that requirements change. Indeed, most of us realize that the requirements document is the most volatile document in the project. If our designs are failing due to the constant rain of changing requirements, it is our designs that are at fault. We must somehow find a way to make our designs resilient to such changes and protect them from rotting."

¿Por qué los sistemas se deterioran?

"However, we cannot blame the drifting of the requirements for the degradation of the design. We, as software engineers, know full well that requirements change. Indeed, most of us realize that the requirements document is the most volatile document in the project. If our designs are failing due to the constant rain of changing requirements, it is our designs that are at fault. We must somehow find a way to make our designs resilient to such changes and protect them from rotting."

¿Qué hacemos para evitar que nuestro software se deteriore?

¿Por qué los sistemas se deterioran?

"However, we cannot blame the drifting of the requirements for the degradation of the design. We, as software engineers, know full well that requirements change. Indeed, most of us realize that the requirements document is the most volatile document in the project. If our designs are failing due to the constant rain of changing requirements, it is our designs that are at fault. We must somehow find a way to make our designs resilient to such changes and protect them from rotting."

¿Qué hacemos para evitar que nuestro software se deteriore?

- Tenemos que programar módulos que sean fáciles de reutilizar.

¿Por qué los sistemas se deterioran?

"However, we cannot blame the drifting of the requirements for the degradation of the design. We, as software engineers, know full well that requirements change. Indeed, most of us realize that the requirements document is the most volatile document in the project. If our designs are failing due to the constant rain of changing requirements, it is our designs that are at fault. We must somehow find a way to make our designs resilient to such changes and protect them from rotting."

¿Qué hacemos para evitar que nuestro software se deteriore?

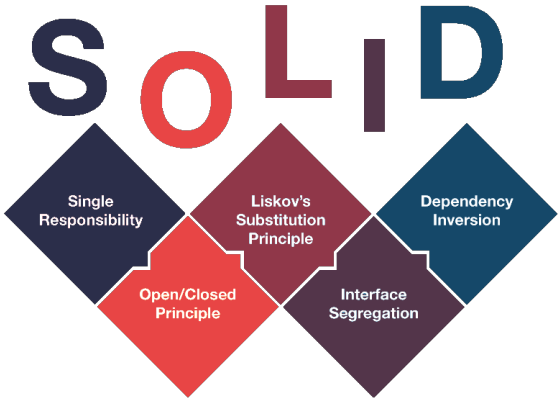
- Tenemos que programar módulos que sean fáciles de reutilizar.
- Tenemos que poner *cortafuegos* que limiten el impacto que tiene cambiar un módulo en tu software.

¿Por qué los sistemas se deterioran?

"However, we cannot blame the drifting of the requirements for the degradation of the design. We, as software engineers, know full well that requirements change. Indeed, most of us realize that the requirements document is the most volatile document in the project. If our designs are failing due to the constant rain of changing requirements, it is our designs that are at fault. We must somehow find a way to make our designs resilient to such changes and protect them from rotting."

¿Qué hacemos para evitar que nuestro software se deteriore?

- Tenemos que programar módulos que sean fáciles de reutilizar.
- Tenemos que poner *cortafuegos* que limiten el impacto que tiene cambiar un módulo en tu software.
- Hay 5 principios básicos de OOP que nos ayudan a lograr esto.


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

The Dependency Inversion Principle

The Dependency Inversion Principle (DIP)

The Dependency Inversion Principle (DIP)

"*Depend upon Abstractions. Do not depend upon concretions.*"

The Dependency Inversion Principle (DIP)

The Dependency Inversion Principle (DIP)

"*Depend upon Abstractions. Do not depend upon concretions.*"

Otra formulación es:

The Dependency Inversion Principle (DIP)

"*High-level classes should not depend on low-level classes. Both should depend on abstraction. Abstractions should not depend on detail. Details should depend on abstraction*"

The Dependency Inversion Principle (DIP)

The Dependency Inversion Principle (DIP)

"*Depend upon Abstractions. Do not depend upon concretions.*"

Otra formulación es:

The Dependency Inversion Principle (DIP)

"*High-level classes should not depend on low-level classes. Both should depend on abstraction. Abstractions should not depend on detail. Details should depend on abstraction*"

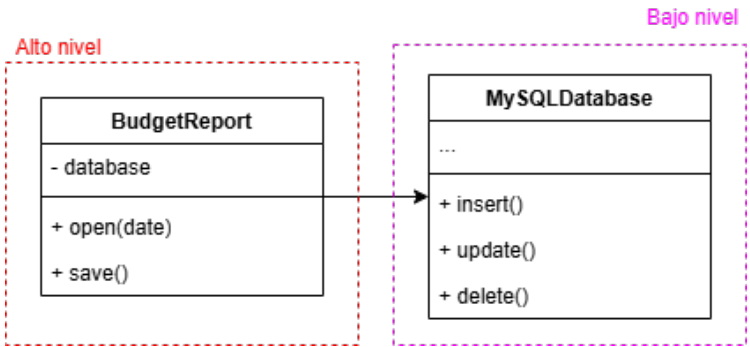
Invertir las dependencias nos permite crear programas más flexibles y extensibles.

The Dependency Inversion Principle (DIP)

Supongamos que trabajamos en un sistema que crea reportes a partir de los datos de la plataforma.

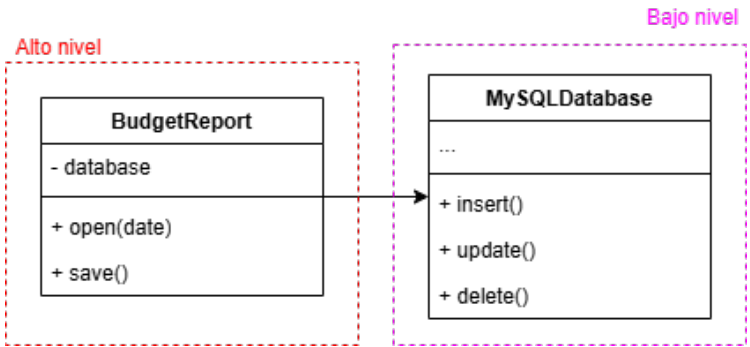
The Dependency Inversion Principle (DIP)

Supongamos que trabajamos en un sistema que crea reportes a partir de los datos de la plataforma.



The Dependency Inversion Principle (DIP)

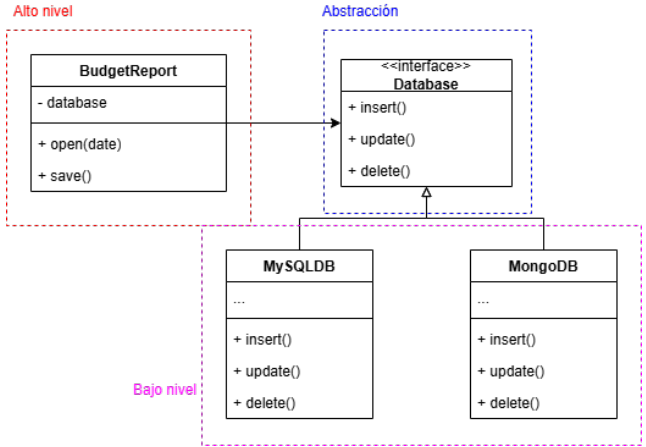
Supongamos que trabajamos en un sistema que crea reportes a partir de los datos de la plataforma.



¿Cuál es el problema?

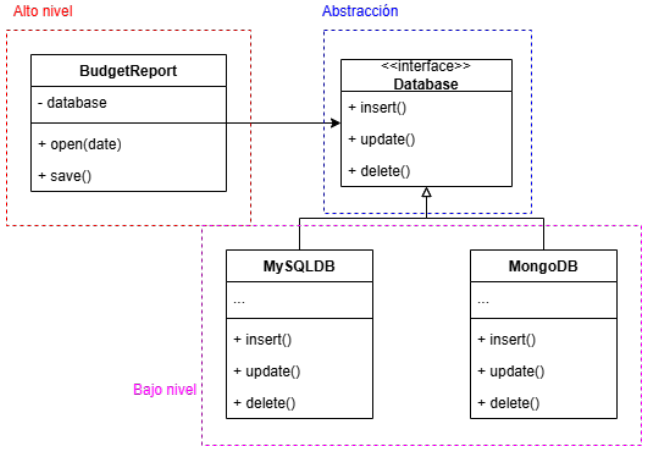
The Dependency Inversion Principle (DIP)

Invirtiendo la dependencia, podemos hacer nuestro sistema más extensible.



The Dependency Inversion Principle (DIP)

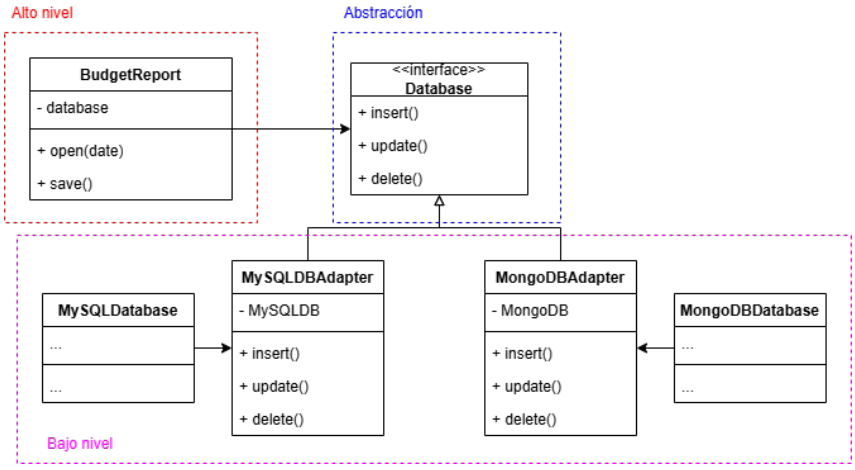
Invirtiendo la dependencia, podemos hacer nuestro sistema más extensible.



¿Y si no podemos editar la base de datos?

The Dependency Inversion Principle (DIP)

Podemos crear clases adaptadoras para respetar el DIP.



The Dependency Inversion Principle (DIP)

Veamos un ejemplo del proyecto. ¿Como se que equipos construir?

The Dependency Inversion Principle (DIP)

Veamos un ejemplo del proyecto. ¿Como se que equipos construir?

1. Debemos que indicarle al usuario que debe seleccionar un equipo

The Dependency Inversion Principle (DIP)

Veamos un ejemplo del proyecto. ¿Como se que equipos construir?

1. Debemos que indicarle al usuario que debe seleccionar un equipo
2. Debemos encontrar todos los archivos .txt de la carpeta con los equipos

The Dependency Inversion Principle (DIP)

Veamos un ejemplo del proyecto. ¿Como se que equipos construir?

1. Debemos que indicarle al usuario que debe seleccionar un equipo
2. Debemos encontrar todos los archivos .txt de la carpeta con los equipos
3. Debemos listar las opciones

The Dependency Inversion Principle (DIP)

Veamos un ejemplo del proyecto. ¿Como se que equipos construir?

1. Debemos que indicarle al usuario que debe seleccionar un equipo
2. Debemos encontrar todos los archivos .txt de la carpeta con los equipos
3. Debemos listar las opciones
4. Debemos pedirle al jugador que seleccione una opción

The Dependency Inversion Principle (DIP)

Veamos un ejemplo del proyecto. ¿Como se que equipos construir?

1. Debemos que indicarle al usuario que debe seleccionar un equipo
2. Debemos encontrar todos los archivos .txt de la carpeta con los equipos
3. Debemos listar las opciones
4. Debemos pedirle al jugador que seleccione una opción
5. Debo leer y procesar el contenido del .txt

The Dependency Inversion Principle (DIP)

Veamos un ejemplo del proyecto. ¿Como se que equipos construir?

1. Debemos que indicarle al usuario que debe seleccionar un equipo
2. Debemos encontrar todos los archivos .txt de la carpeta con los equipos
3. Debemos listar las opciones
4. Debemos pedirle al jugador que seleccione una opción
5. Debo leer y procesar el contenido del .txt

Luego de esto, recién puedo saber que equipos construir.

The Dependency Inversion Principle (DIP)

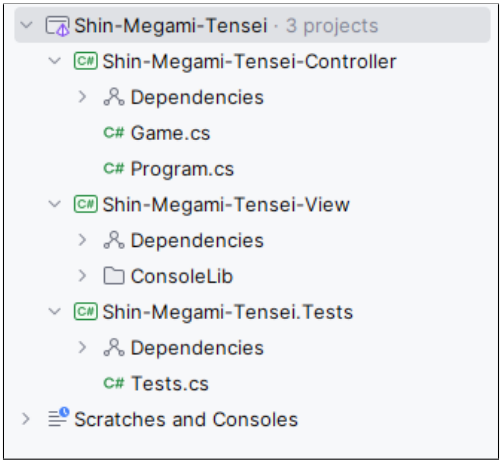
Veamos un ejemplo del proyecto. ¿Como se que equipos construir?

1. Debemos que indicarle al usuario que debe seleccionar un equipo
2. Debemos encontrar todos los archivos .txt de la carpeta con los equipos
3. Debemos listar las opciones
4. Debemos pedirle al jugador que seleccione una opción
5. Debo leer y procesar el contenido del .txt

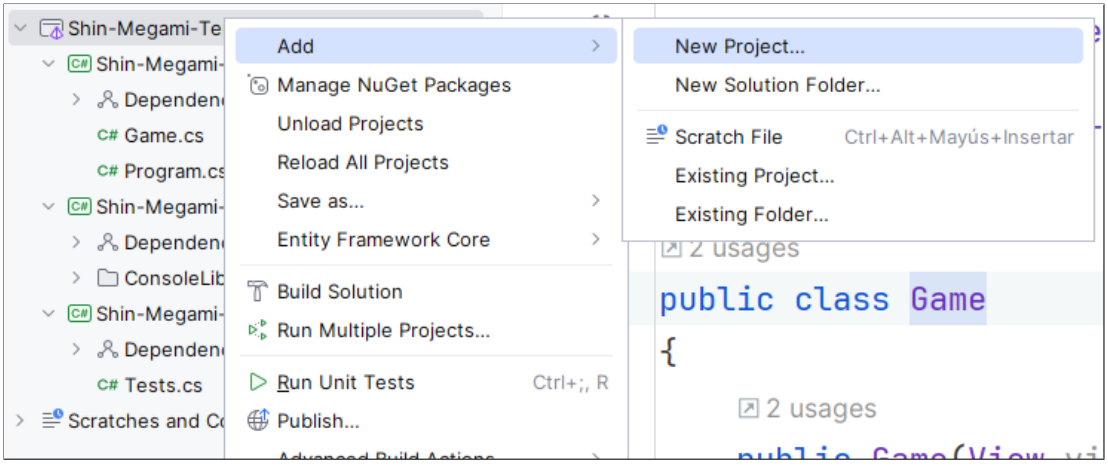
Luego de esto, recién puedo saber que equipos construir.

¿Cómo se vería esto con MVC?

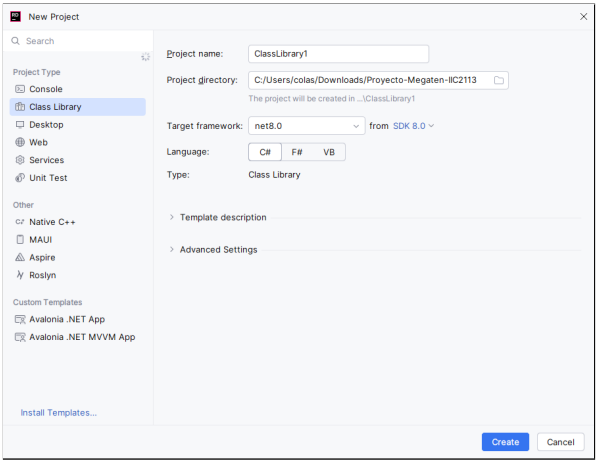
The Dependency Inversion Principle (DIP)



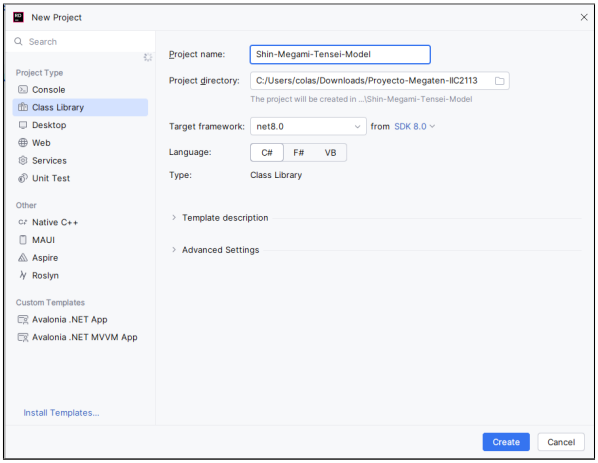
The Dependency Inversion Principle (DIP)



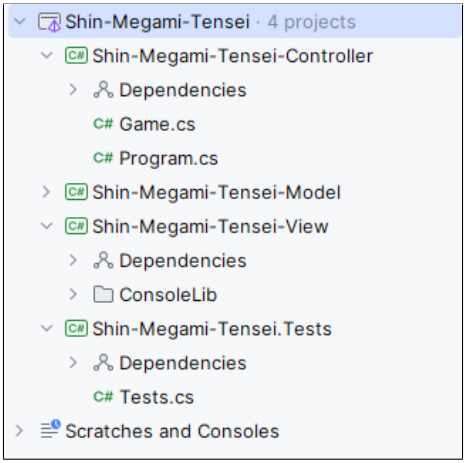
The Dependency Inversion Principle (DIP)



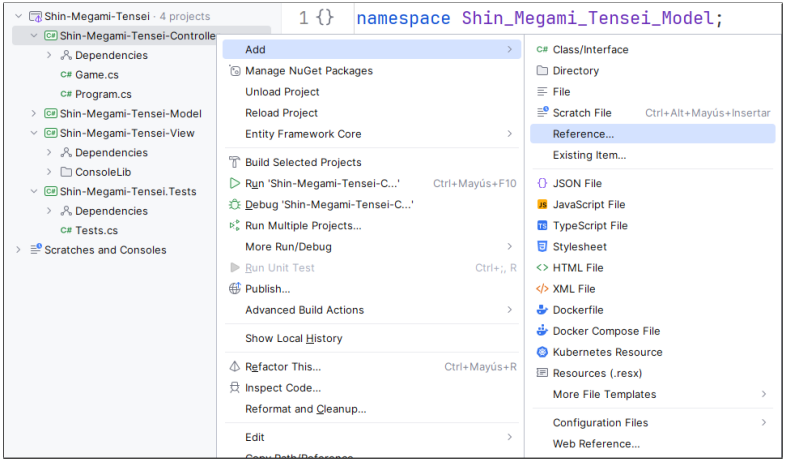
The Dependency Inversion Principle (DIP)



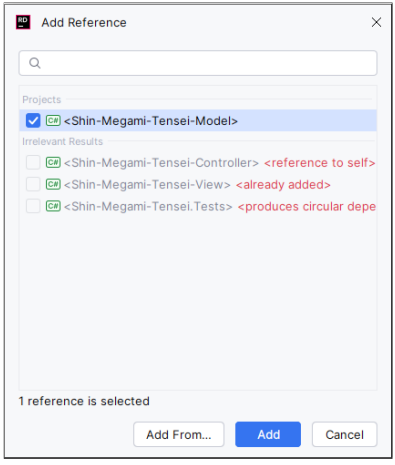
The Dependency Inversion Principle (DIP)



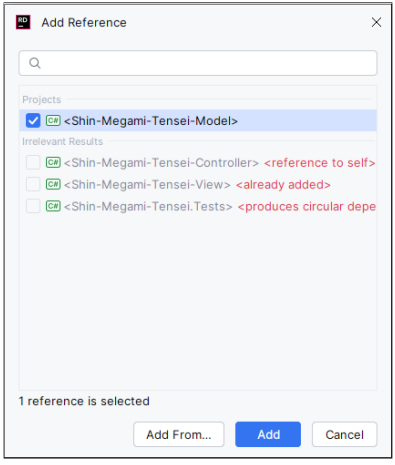
The Dependency Inversion Principle (DIP)



The Dependency Inversion Principle (DIP)



The Dependency Inversion Principle (DIP)



y hacemos lo mismo en la vista

The Dependency Inversion Principle (DIP)

En el código base nuestro controlador usa directamente la clase `View`.

The Dependency Inversion Principle (DIP)

En el código base nuestro controlador usa directamente la clase `View`.

```
1 public class Game
2 {
3     public Game(View view, string teamsFolder) { }
4
5     public void Play()
6         => throw new NotImplementedException();
7 }
```

The Dependency Inversion Principle (DIP)

En el código base nuestro controlador usa directamente la clase `View`.

```
1 public class Game
2 {
3     public Game(View view, string teamsFolder) { }
4
5     public void Play()
6         => throw new NotImplementedException();
7 }
```

Sin embargo, la clase `View` solo tiene dos métodos:

- `ReadLine()`
- `WriteLine()`

Estos solo nos permiten mostrar cosas en consola y recibir inputs en consola

The Dependency Inversion Principle (DIP)

En el código base nuestro controlador usa directamente la clase `View`.

```
1 public class Game
2 {
3     public Game(View view, string teamsFolder) { }
4
5     public void Play()
6         => throw new NotImplementedException();
7 }
```

Sin embargo, la clase `View` solo tiene dos métodos:

- `ReadLine()`
- `WriteLine()`

Estos solo nos permiten mostrar cosas en consola y recibir inputs en consola

Esto rompe el MVC

The Dependency Inversion Principle (DIP)

Podemos crear una clase en nuestro proyecto **Vista** que utilice la clase `View`

The Dependency Inversion Principle (DIP)

Podemos crear una clase en nuestro proyecto **Vista** que utilice la clase `View`

```
public class SMTConsoleView(View view)
{
    private readonly View _view = view;

    public void AnnounceTeamSelection()
        => _view.WriteLine("Elige un archivo para cargar los equipos");

    public void ListAvailableTeams(string teamsFolder)
    {
        string[] files = Directory.GetFiles(teamsFolder, "*.txt");
        Array.Sort(files);
        for (int i = 0; i < files.Length; i++)
            _view.WriteLine($"{i}- {Path.GetFileName(files[i])}");
    }

    public string AskUserToSelectATeamFile()
    {
        string[] files = Directory.GetFiles("data", "*.txt");
        Array.Sort(files);
        int userOption = Int32.Parse(_view.ReadLine());
        return Path.GetFileName(files[userOption]);
    }
}
```

The Dependency Inversion Principle (DIP)

Luego, podemos usar la vista que acabamos de crear en nuestro controlador:

The Dependency Inversion Principle (DIP)

Luego, podemos usar la vista que acabamos de crear en nuestro controlador:

```
1  public class Game
2  {
3      private readonly SMTConsoleView _view;
4      private string _teamsFolder;
5      public Game(View view, string teamsFolder)
6      {
7          _view = new SMTConsoleView(view);
8          _teamsFolder = teamsFolder;
9      }
10
11     public void Play() { /* ... */ }
12
13     private string[] GetTeams()
14     {
15         _view.AnnounceTeamSelection();
16         _view.ListAvailableTeams(_teamsFolder);
17         string file = _view.AskUserToSelectATeamFile();
18         string[] fileContent = File.ReadAllLines(Path.Combine(_teamsFolder, file));
19         return GetUnitsFromFile(fileContent);
20     }
21
22     private string[] GetUnitsFromFile(string[] fileContent) { /* ... */}
23 }
```

The Dependency Inversion Principle (DIP)

Luego, podemos usar la vista que acabamos de crear en nuestro controlador:

```
1  public class Game
2  {
3      private readonly SMTConsoleView _view;
4      private string _teamsFolder;
5      public Game(View view, string teamsFolder)
6      {
7          _view = new SMTConsoleView(view);
8          _teamsFolder = teamsFolder;
9      }
10
11     public void Play() { /* ... */ }
12
13     private string[] GetTeams()
14     {
15         _view.AnnounceTeamSelection();
16         _view.ListAvailableTeams(_teamsFolder);
17         string file = _view.AskUserToSelectATeamFile();
18         string[] fileContent = File.ReadAllLines(Path.Combine(_teamsFolder, file));
19         return GetUnitsFromFile(fileContent);
20     }
21
22     private string[] GetUnitsFromFile(string[] fileContent) { /* ... */}
23 }
```

Pero esto rompe el DIP ¿Por qué?

The Dependency Inversion Principle (DIP)

Arreglemoslo creando una interfaz para la vista.

The Dependency Inversion Principle (DIP)

Arreglemoslo creando una interfaz para la vista.

```
1 public interface IView
2 {
3     void AnnounceTeamSelection();
4     void ListAvailableTeams(string teamsFolder);
5     string AskUserToSelectATeamFile();
6 }
```

The Dependency Inversion Principle (DIP)

Arreglemoslo creando una interfaz para la vista.

```
1 public interface IView
2 {
3     void AnnounceTeamSelection();
4     void ListAvailableTeams(string teamsFolder);
5     string AskUserToSelectATeamFile();
6 }
```

Luego, que nuestra vista por consola la implemente

The Dependency Inversion Principle (DIP)

```
public class SMTConsoleView(View view) : IView
{
    private readonly View _view = view;

    public void AnnounceTeamSelection()
        => _view.WriteLine("Elige un archivo para cargar los equipos");

    public void ListAvailableTeams(string teamsFolder)
    {
        string[] files = Directory.GetFiles(teamsFolder, "*.txt");
        Array.Sort(files);
        for (int i = 0; i < files.Length; i++)
            _view.WriteLine($"{i}- {Path.GetFileName(files[i])}");
    }

    public string AskUserToSelectATeamFile()
    {
        string[] files = Directory.GetFiles("data", "*.txt");
        Array.Sort(files);
        int userOption = Int32.Parse(_view.ReadLine());
        return Path.GetFileName(files[userOption]);
    }
}
```

The Dependency Inversion Principle (DIP)

```
public class SMTConsoleView(View view) : IView
{
    private readonly View _view = view;

    public void AnnounceTeamSelection()
        => _view.WriteLine("Elige un archivo para cargar los equipos");

    public void ListAvailableTeams(string teamsFolder)
    {
        string[] files = Directory.GetFiles(teamsFolder, "*.txt");
        Array.Sort(files);
        for (int i = 0; i < files.Length; i++)
            _view.WriteLine($"{i}- {Path.GetFileName(files[i])}");
    }

    public string AskUserToSelectATeamFile()
    {
        string[] files = Directory.GetFiles("data", "*.txt");
        Array.Sort(files);
        int userOption = Int32.Parse(_view.ReadLine());
        return Path.GetFileName(files[userOption]);
    }
}
```

Ahora la utilizamos en el controlador.

The Dependency Inversion Principle (DIP)

```
1  public class Game
2  {
3      private readonly IView _view;
4      private string _teamsFolder;
5      public Game(View view, string teamsFolder)
6      {
7          _view = new SMTConsoleView(view);
8          _teamsFolder = teamsFolder;
9      }
10
11     public void Play() { /* ... */ }
12
13     private string[] GetTeams()
14     {
15         _view.AnnounceTeamSelection();
16         _view.ListAvailableTeams(_teamsFolder);
17         string file = _view.AskUserToSelectATeamFile();
18         string[] fileContent = File.ReadAllLines(Path.Combine(_teamsFolder, file));
19         return GetUnitsFromFile(fileContent);
20     }
21
22     private string[] GetUnitsFromFile(string[] fileContent) { /* ... */}
23 }
```

The Dependency Inversion Principle (DIP)

```
1  public class Game
2  {
3      private readonly IView _view;
4      private string _teamsFolder;
5      public Game(View view, string teamsFolder)
6      {
7          _view = new SMTConsoleView(view);
8          _teamsFolder = teamsFolder;
9      }
10
11     public void Play() { /* ... */ }
12
13     private string[] GetTeams()
14     {
15         _view.AnnounceTeamSelection();
16         _view.ListAvailableTeams(_teamsFolder);
17         string file = _view.AskUserToSelectATeamFile();
18         string[] fileContent = File.ReadAllLines(Path.Combine(_teamsFolder, file));
19         return GetUnitsFromFile(fileContent);
20     }
21
22     private string[] GetUnitsFromFile(string[] fileContent) { /* ... */}
23 }
```

Pero esto tiene un gran problema ¿Cuál?

The Dependency Inversion Principle (DIP)

Los principales problemas:

The Dependency Inversion Principle (DIP)

Los principales problemas:

- Solamente listamos archivos .txt porque la vista es de consola

The Dependency Inversion Principle (DIP)

Los principales problemas:

- Solamente listamos archivos .txt porque la vista es de consola
- Solamente accedemos a un directorio porque la vista es de consola

The Dependency Inversion Principle (DIP)

Los principales problemas:

- Solamente listamos archivos .txt porque la vista es de consola
- Solamente accedemos a un directorio porque la vista es de consola
- Solamente leemos los equipos desde un .txt porque la vista es de consola

The Dependency Inversion Principle (DIP)

Los principales problemas:

- Solamente listamos archivos .txt porque la vista es de consola
- Solamente accedemos a un directorio porque la vista es de consola
- Solamente leemos los equipos desde un .txt porque la vista es de consola
- Nuestro controlador asume que la vista lista archivos desde un directorio y que los equipos están en un .txt

The Dependency Inversion Principle (DIP)

Los principales problemas:

- Solamente listamos archivos .txt porque la vista es de consola
- Solamente accedemos a un directorio porque la vista es de consola
- Solamente leemos los equipos desde un .txt porque la vista es de consola
- Nuestro controlador asume que la vista lista archivos desde un directorio y que los equipos están en un .txt
- El controlador depende de la implementación de la vista de consola

The Dependency Inversion Principle (DIP)

Los principales problemas:

- Solamente listamos archivos .txt porque la vista es de consola
- Solamente accedemos a un directorio porque la vista es de consola
- Solamente leemos los equipos desde un .txt porque la vista es de consola
- Nuestro controlador asume que la vista lista archivos desde un directorio y que los equipos están en un .txt
- El controlador depende de la implementación de la vista de consola

En conclusión, estamos violando el DIP

The Dependency Inversion Principle (DIP)

Solución

Solución

Creemos una interfaz mucho más genérica y dejemos que la vista se encargue de leer los archivos .txt.

Solución

Creemos una interfaz mucho más genérica y dejemos que la vista se encargue de leer los archivos .txt.

```
1 public interface IView
2 {
3     string[] AskUsersForTeams();
4 }
```

Solución

Creemos una interfaz mucho más genérica y dejemos que la vista se encargue de leer los archivos .txt.

```
1 public interface IView
2 {
3     string[] AskUsersForTeams();
4 }
```

Luego, creemos nuestra vista por consola

The Dependency Inversion Principle (DIP)

```
1  public class SMTConsoleView(View view, string teamsFolder) : IView
2  {
3      private readonly View _view = view;
4      private string _teamsFolder = teamsFolder;
5
6      public string[] AskUsersForTeams()
7      {
8          _view.WriteLine("Elige un archivo para cargar los equipos");
9          ListAvailableTeams();
10         string file = AskUserToSelectATeamFile();
11         return GetTeamsFromFileContent(file);
12     }
13
14     private void ListAvailableTeams() { /* ... */}
15
16     private string AskUserToSelectATeamFile() { /* ... */}
17
18     private string[] GetTeamsFromFileContent(string file) {/* ... */}
19 }
```

The Dependency Inversion Principle (DIP)

```
public class SMTConsoleView(View view, string teamsFolder) : IView
{
    private readonly View _view = view;
    private string _teamsFolder = teamsFolder;

    public string[] AskUsersForTeams()
    {
        _view.WriteLine("Elige un archivo para cargar los equipos");
        ListAvailableTeams();
        string file = AskUserToSelectATeamFile();
        return GetTeamsFromFileContent(file);
    }

    private void ListAvailableTeams() { /* ... */}

    private string AskUserToSelectATeamFile() { /* ... */}

    private string[] GetTeamsFromFileContent(string file) {/* ... */}
}
```

Ahora la utilizamos en el controlador.

The Dependency Inversion Principle (DIP)

```
1  public class Game
2  {
3      private readonly IView _view;
4      private string _teamsFolder;
5      public Game(View view, string teamsFolder)
6      {
7          _view = new SMTConsoleView(view, teamsFolder);
8          _teamsFolder = teamsFolder;
9      }
10
11     public void Play()
12     {
13         string[] teams = _view.AskUsersForTeams();
14         /* El resto del código */
15     }
16 }
```

The Dependency Inversion Principle (DIP)

```
1  public class Game
2  {
3      private readonly IView _view;
4      private string _teamsFolder;
5      public Game(View view, string teamsFolder)
6      {
7          _view = new SMTConsoleView(view, teamsFolder);
8          _teamsFolder = teamsFolder;
9      }
10
11     public void Play()
12     {
13         string[] teams = _view.AskUsersForTeams();
14         /* El resto del código */
15     }
16 }
```

Ahora nuestra implementación respeta el DIP

The Interface Segregation Principle

The Interface Segregation Principle (ISP)

The Interface Segregation Principle (ISP)

"*Clients should not be forced to depend upon interfaces that they do not use.*"

The Interface Segregation Principle (ISP)

The Interface Segregation Principle (ISP)

"*Clients should not be forced to depend upon interfaces that they do not use.*"

Este principio nos dice dos cosas complementarias:

The Interface Segregation Principle (ISP)

The Interface Segregation Principle (ISP)

"*Clients should not be forced to depend upon interfaces that they do not use.*"

Este principio nos dice dos cosas complementarias:

1. El cliente debería implementar solo aquellos métodos que realmente necesitan

The Interface Segregation Principle (ISP)

The Interface Segregation Principle (ISP)

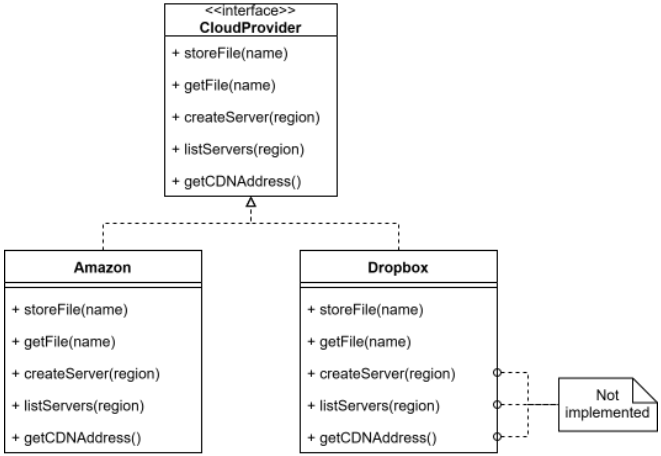
"*Clients should not be forced to depend upon interfaces that they do not use.*"

Este principio nos dice dos cosas complementarias:

1. El cliente debería implementar solo aquellos métodos que realmente necesitan
2. Debemos separar interfaces muy grandes en interfaces más pequeñas y especificas.

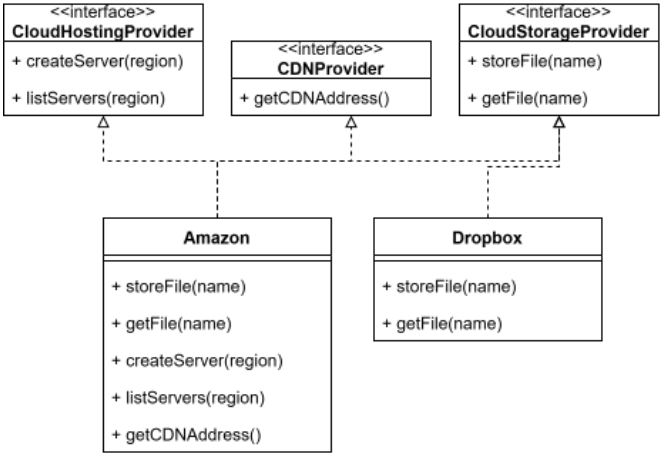
The Interface Segregation Principle (ISP)

Supongamos que tenemos los siguientes servicios ¿ven el problema?



The Interface Segregation Principle (ISP)

Podemos solucionarlo separando la interfaz original en otras más pequeñas



The Interface Segregation Principle (ISP)

A medida que avancemos en el proyecto, nuestra vista irá creciendo

The Interface Segregation Principle (ISP)

A medida que avancemos en el proyecto, nuestra vista irá creciendo

```
1  public interface IView
2  {
3      string[] AskUsersForTeams();
4      void AnnounceInvalidTeams();
5      void DisplayBoardState(Board board);
6      UserAction AskUserForAction();
7      void AnnounceActionResults(ActionResult[] results);
8      void AnnounceWinner();
9      void AnnounceTurnConsumption();
10 }
```

The Interface Segregation Principle (ISP)

A medida que avancemos en el proyecto, nuestra vista irá creciendo

```
public class SMTConsoleView(View view, string teamsFolder) : IView
{
    private readonly View _view = view;
    private string _teamsFolder = teamsFolder;

    public string[] AskUsersForTeams() { /* ... */ }
    public void AnnounceInvalidTeams() { /* ... */ }

    public void AnnounceWinner() { /* ... */ }
    public void AnnounceActionResults(ActionResult[] results) { /* ... */ }
    public void DisplayBoardState(Board board) { /* ... */ }
    public UserAction AskUserForAction() { /* ... */ }
    public void AnnounceTurnConsumption() { /* ... */ }
}
```

The Interface Segregation Principle (ISP)

Además, la utilizaremos en varios de nuestros controladores

The Interface Segregation Principle (ISP)

Además, la utilizaremos en varios de nuestros controladores

```
public class Game
{
    private readonly IView _view;

    public Game(View view, string teamsFolder)
        => _view = new SMTConsoleView(view, teamsFolder);

    public void Play()
    {
        string[] teams = _view.AskUsersForTeams();
        /* ... */
        if(TeamsAreInvalid())
            _view.AnnounceInvalidTeams();
    }
    /* ... */
}
```

The Interface Segregation Principle (ISP)

Además, la utilizaremos en varios de nuestros controladores

```
1  public class BattleController(IView view)
2  {
3      /* ... */
4      public void HandleBattle()
5      {
6          UserAction action = view.AskUserForAction();
7          /* ... */
8          view.AnnounceActionResults(results);
9          /* ... */
10         view.AnnounceTurnConsumption();
11     }
12     /* ... */
13 }
```

The Interface Segregation Principle (ISP)

Además, la utilizaremos en varios de nuestros controladores

```
public class RoundController(IView view)
{
    /* ... */
    public void HandleRound()
    {
        view.DisplayBoardState(Board);
        /* ... */
        if(BoardIsEmpty())
            view.AnnounceWinner();
    }
    /* ... */
}
```

The Interface Segregation Principle (ISP)

Además, la utilizaremos en varios de nuestros controladores

```
1  public class RoundController(IView view)
2  {
3      /* ... */
4      public void HandleRound()
5      {
6          view.DisplayBoardState(Board);
7          /* ... */
8          if(BoardIsEmpty())
9              view.AnnounceWinner();
10     }
11     /* ... */
12 }
```

Esto rompe el ISP

The Interface Segregation Principle (ISP)

Los principales problemas:

The Interface Segregation Principle (ISP)

Los principales problemas:

- No todas las clases necesitan usar todos los métodos de `Iview`

The Interface Segregation Principle (ISP)

Los principales problemas:

- No todas las clases necesitan usar todos los métodos de `Iview`
- No queremos que todas las clases tengan la posibilidad de usar los métodos de `IView`

The Interface Segregation Principle (ISP)

Los principales problemas:

- No todas las clases necesitan usar todos los métodos de `Iview`
- No queremos que todas las clases tengan la posibilidad de usar los métodos de `IView`
- Ahora mismo, cualquier clase que use `IView` puede usar todos sus métodos

The Interface Segregation Principle (ISP)

Los principales problemas:

- No todas las clases necesitan usar todos los métodos de `Iview`
- No queremos que todas las clases tengan la posibilidad de usar los métodos de `IView`
- Ahora mismo, cualquier clase que use `IView` puede usar todos sus métodos
- Cualquier clase que implemente `IView` tiene que implementar todos sus métodos

The Interface Segregation Principle (ISP)

Los principales problemas:

- No todas las clases necesitan usar todos los métodos de `Iview`
- No queremos que todas las clases tengan la posibilidad de usar los métodos de `IView`
- Ahora mismo, cualquier clase que use `IView` puede usar todos sus métodos
- Cualquier clase que implemente `IView` tiene que implementar todos sus métodos

En conclusión, estamos violando el ISP

The Interface Segregation Principle (ISP)

Solución

The Interface Segregation Principle (ISP)

Solución

Hacer interfaces más pequeñas

The Interface Segregation Principle (ISP)

Solución

Hacer interfaces más pequeñas

```
public interface ITeamView
{
    string[] AskUsersForTeams();
    void AnnounceInvalidTeams();
}
```

```
public interface IRoundView
{
    void DisplayBoardState(Board board);
    void AnnounceWinner();
}
```

```
public interface IBattleView
{
    UserAction AskUserForAction();
    void AnnounceActionResults(ActionResult[] results);
    void AnnounceTurnConsumption();
}
```

The Interface Segregation Principle (ISP)

Solución

Hacer interfaces más pequeñas

```
public class SMTConsoleView(View view, string teamsFolder) : ITeamView,
    IBattleView, IRoundView
{
    private readonly View _view = view;
    private string _teamsFolder = teamsFolder;

    public string[] AskUsersForTeams() { /* ... */ }
    public void AnnounceInvalidTeams() { /* ... */ }
    public void AnnounceWinner() { /* ... */ }
    public void AnnounceActionResults(ActionResult[] results) { /* ... */ }
    public void DisplayBoardState(Board board) { /* ... */ }
    public UserAction AskUserForAction() { /* ... */ }
    public void AnnounceTurnConsumption() { /* ... */ }
}
```

The Interface Segregation Principle (ISP)

Ahora nuestros controladores solo dependen de interfaces que usan

The Interface Segregation Principle (ISP)

Ahora nuestros controladores solo dependen de interfaces que usan

```
1  public class Game
2  {
3      private readonly ITeamView _view;
4
5      public Game(View view, string teamsFolder)
6          => _view = new SMTConsoleView(view, teamsFolder);
7
8      public void Play()
9      {
10         string[] teams = _view.AskUsersForTeams();
11         /* ... */
12         if(TeamsAreInvalid())
13             _view.AnnounceInvalidTeams();
14     }
15     /* ... */
16 }
```

The Interface Segregation Principle (ISP)

Ahora nuestros controladores solo dependen de interfaces que usan

```
public class RoundController(IRoundView view)
{
    /* ... */
    public void HandleRound()
    {
        view.DisplayBoardState(Board);
        /* ... */
        if(BoardIsEmpty())
            view.AnnounceWinner();
    }
    /* ... */
}
```

The Interface Segregation Principle (ISP)

Ahora nuestros controladores solo dependen de interfaces que usan

```
public class BattleController(IBattleView view)
{
    /* ... */
    public void HandleBattle()
    {
        UserAction action = view.AskUserForAction();
        /* ... */
        view.AnnounceActionResults(results);
        /* ... */
        view.AnnounceTurnConsumption();
    }
    /* ... */
}
```

The Interface Segregation Principle (ISP)

Ahora nuestros controladores solo dependen de interfaces que usan

```
1  public class BattleController(IBattleView view)
2  {
3      /* ... */
4      public void HandleBattle()
5      {
6          UserAction action = view.AskUserForAction();
7          /* ... */
8          view.AnnounceActionResults(results);
9          /* ... */
10         view.AnnounceTurnConsumption();
11     }
12     /* ... */
13 }
```

Pero, esto podría lograrse de otra forma

The Interface Segregation Principle (ISP)

En vez de tener una clase que implemente todas las interfaces, tengamos varias clases

The Interface Segregation Principle (ISP)

En vez de tener una clase que implemente todas las interfaces, tengamos varias clases

```
1 public interface IViewFactory
2 {
3     ITeamView CreateTeamView();
4     IBattleView CreateBattleView();
5     IRoundView CreateRoundView();
6 }
```

The Interface Segregation Principle (ISP)

En vez de tener una clase que implemente todas las interfaces, tengamos varias clases

```
public interface IViewFactory
{
    ITeamView CreateTeamView();
    IBattleView CreateBattleView();
    IRoundView CreateRoundView();
}
```

```
public class ConsoleViewFactory(View view, string teamsFolder) : IViewFactory
{
    private View _view = view;
    private string _teamsFolder = teamsFolder;

    public IBattleView CreateBattleView()
        => new ConsoleBattleView(_view);
    public IRoundView CreateRoundView()
        => new ConsoleRoundView(_view);
    public ITeamView CreateTeamView()
        => new ConsoleTeamView(_view, _teamsFolder);
}
```

The Interface Segregation Principle (ISP)

En vez de tener una clase que implemente todas las interfaces, tengamos varias clases

```
1  public class Game
2  {
3      private readonly IViewFactory _viewFactory;
4
5      public Game(View view, string teamsFolder)
6          => _viewFactory = new ConsoleViewFactory(view, teamsFolder);
7
8      public void Play()
9      {
10         ITeamView teamView = _viewFactory.CreateTeamView();
11         string[] teams = teamView.AskUsersForTeams();
12         /* ... */
13         if(TeamsAreInvalid())
14             teamView.AnnounceInvalidTeams();
15     }
16     /* ... */
17 }
```

The Interface Segregation Principle (ISP)

En vez de tener una clase que implemente todas las interfaces, tengamos varias clases

```
1  public class BattleController(IViewFactory viewFactory)
2  {
3      /* ... */
4      public void HandleBattle()
5      {
6          IBattleView view = viewFactory.CreateBattleView();
7          UserAction action = view.AskUserForAction();
8          /* ... */
9          view.AnnounceActionResults(results);
10         /* ... */
11         view.AnnounceTurnConsumption();
12     }
13     /* ... */
14 }
```

The Interface Segregation Principle (ISP)

En vez de tener una clase que implemente todas las interfaces, tengamos varias clases

```
public class RoundController(IViewFactory viewFactory)
{
    /* ... */
    public void HandleRound()
    {
        IRoundView view = viewFactory.CreateRoundView();
        view.DisplayBoardState(Board);
        /* ... */
        if(BoardIsEmpty())
            view.AnnounceWinner();
    }
    /* ... */
}
```

Consejos para la E2 y E3

Consejos

- Si tu código depende muy fuertemente de la vista de consola (por ejemplo, usas indiscriminadamente View.WriteLine() en tu programa) te será muy difícil agregar la GUI en la E3

Consejos

- Si tu código depende muy fuertemente de la vista de consola (por ejemplo, usas indiscriminadamente View.WriteLine() en tu programa) te será muy difícil agregar la GUI en la E3
- Intenta que la relación entre tus objetos sea lo más abstracta posible. Esto nos ayuda tanto con el DIP como con el ISP

Consejos

- Si tu código depende muy fuertemente de la vista de consola (por ejemplo, usas indiscriminadamente View.WriteLine() en tu programa) te será muy difícil agregar la GUI en la E3
- Intenta que la relación entre tus objetos sea lo más abstracta posible. Esto nos ayuda tanto con el DIP como con el ISP
- Intenta que los elementos del juego no dependan de la vista (Ej. No guardes strings que se van a imprimir en el modelo)

Consejos

- Si tu código depende muy fuertemente de la vista de consola (por ejemplo, usas indiscriminadamente View.WriteLine() en tu programa) te será muy difícil agregar la GUI en la E3
- Intenta que la relación entre tus objetos sea lo más abstracta posible. Esto nos ayuda tanto con el DIP como con el ISP
- Intenta que los elementos del juego no dependan de la vista (Ej. No guardes strings que se van a imprimir en el modelo)
- Una buena forma de que respetar el ISP es haciendo clases pequeñas y especializadas

Consejos

- Si tu código depende muy fuertemente de la vista de consola (por ejemplo, usas indiscriminadamente View.WriteLine() en tu programa) te será muy difícil agregar la GUI en la E3
- Intenta que la relación entre tus objetos sea lo más abstracta posible. Esto nos ayuda tanto con el DIP como con el ISP
- Intenta que los elementos del juego no dependan de la vista (Ej. No guardes strings que se van a imprimir en el modelo)
- Una buena forma de que respetar el ISP es haciendo clases pequeñas y especializadas
- Es mejor tener muchas clase chicas que hagan una sola cosa que tener una sola clase grande