Pontificia Universidad Católica de Chile
Escuela de Ingeniería
Departamento de Ciencia de la Computación
IIC2113 Diseño Detallado de Software

Entrega 3: Shin Megami Tensei (Interfaz Gráfica)

Francisco Ignacio Gazitúa Requena
Cristian Andrés Hinostroza Espinoza

Introducción

La librería `Shin_Megami_Tensei_GUI` incluye una clase, 5 interfaces y una enumeración:

- `SMTGUI` es una clase que te permitirá abrir una ventana y cambiar lo que aparece en ella.
- `IUnit` es una interfaz con métodos para obtener la información de la unidad (e.j. su nombre, HP, etc).
- `IPlayer` es una interfaz con métodos para obtener la información de un jugador, como las unidades en el tablero y la reserva.
- `IState` es una interfaz con métodos para obtener la información del juego, como las opciones que tiene disponible para elegir el jugador, la cantidad de turnos, etc.
- `ITeamInfo` es una interfaz con métodos para obtener la información de un equipo, como el nombre del samurai y los demonios.
- `IClickedElement` es una interfaz con métodos para obtener la información del último elemento que se le hizo click en la ventana, como el texto y qué jugador estaba ligado.
- `ClickedElementType` es una enumeración que tiene los distintos tipos de elementos que puede ser `IClickedElement`.

Para utilizar la librería, puedes comenzar desde la nueva versión del código base o seguir las instrucciones descritas en el Apéndice A

1. Interfaces `IState`, `IPlayer` y `IUnit`

Cualquier clase que implemente la interfaz `IState` tendrá que implementar estos métodos:

```
public interface IState {
    public IPlayer Player1 { get; }
    public IPlayer Player2 { get; }
    public IEnumerable<string> Options { get; }
    public int Turns { get; }
    public int BlinkingTurns { get; }
    public IEnumerable<string> Order { get; }
}
```

El método `void Update(IState state)` de la clase `SMTGUI` recibe un objeto que implemente `IState`, por lo tanto, para utilizar este método, tendrás que tener una clase en tu código que la implemente.

Dado que `IState` tiene las properties `Player1` y `Player2` tendrás que tener una clase que implemente dicha interfaz, la cual expone los siguientes métodos:

```
public interface IPlayer
{
    IUnit?[] UnitsInBoard { get; }
    IEnumerable<IUnit> UnitsInReserve { get; }
}
```

De igual forma, tendrás que tener alguna clase que implemente la interfaz `IUnit`, la cual expone los siguientes métodos:

```
public interface IUnit
{
    string Name { get; }
    int HP { get; }
    int MP { get; }
    int MaxHP { get; }
    int MaxMP { get; }
}
```

2. Interfaz `ITeamInfo`

La interfaz `ITeamInfo` expone los siguientes métodos:

```
public interface ITeamInfo
{
    string SamuraiName { get; }
    string[] SkillNames { get; }
    string[] DemonNames { get; }
}
```

No es necesario que crees clases que implementen esta interfaz, ya que esta interfaz sólo es utilizada como retorno del método `ITeamInfo GetTeamInfo(int playerId)` de la clase `SMTGUI`.

3. Interfaz `IClickedElement`

La interfaz `IClickedElement` expone los siguientes métodos:

```
public interface IClickedElement
{
    string Text { get; }
    int? PlayerId { get; }
    ClickedElementType Type { get; }
}
```

No es necesario que crees clases que implementen esta interfaz, ya que esta interfaz sólo es utilizada como retorno del método `IClickedElement GetClickedElement()` de la clase `SMTGUI`.

Cabe destacara que `Type` retorna una instancia de tipo `ClickedElementType`, el cual es una enumeración que contiene los siguientes miembros:

```
public enum ClickedElementType
{
    UnitInBoard,
    UnitInReserve,
    Button,
}
```

4. Clase `SMTGUI`

Como se dijo anteriormente, esta clase permite abrir una ventana y actualizar su contenido. Para abrir una ventana, primero se crea un objeto del tipo `SMTGUI` y luego se llama el método `Start(...)`.

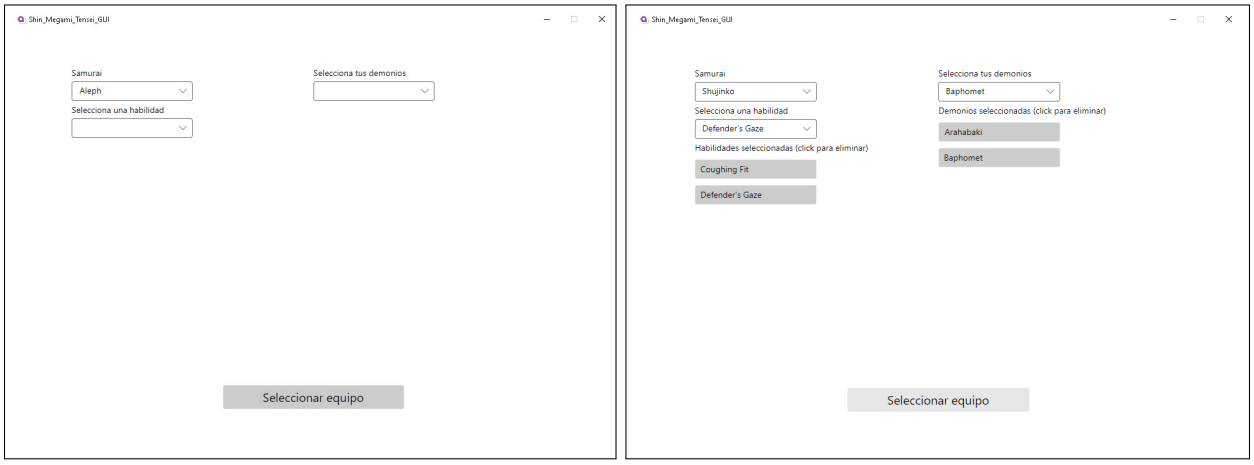
- `void Start(Action startProgramCallback)`: Este método inicia una ventana y luego invoca `startProgramCallback`. Notar que el parámetro `startProgramCallback` es de tipo `Action`. Este es un tipo de dato utilizado para encapsular funciones que no retornan nada y que no reciben argumentos.

Ocurre que MacOS no permite crear ventanas desde threads que no sean el principal del programa. Esto nos obliga a que, luego de creada la ventana, esa ventana se quede con el thread principal de la aplicación (y no lo suelta hasta que la ventana se cierra). Para que tu programa siga funcionando, nosotros invocamos `startProgramCallback` desde el thread secundario. Por lo mismo, considera que cuando llames a `Start(Action startProgramCallback)` tu código continuará desde el método encapsulado en `startProgramCallback`.

Por ejemplo, este `Program.cs` abre la ventana y luego llama a `Main()` - que no hace nada.

```
using Shin_Megami_Tensei_GUI;

SMTGUI gui = new SMTGUI();
gui.Start(Main);

void Main() {
    // Por ahora, no hace nada.
}
```

Si corres este programa se abre una ventana que permite elegir el equipo. Pero dado que el método `Main` está vacío, no ocurre nada al apretar el botón “Seleccionar equipo”.



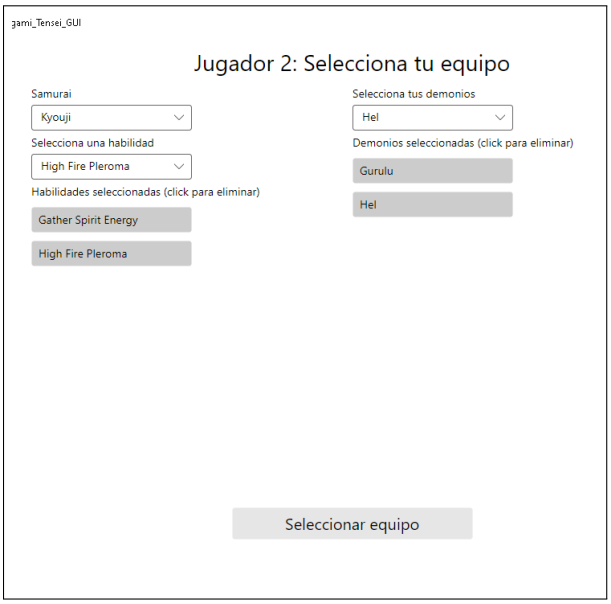
Para obtener el equipo ingresado por el usuario, puedes utilizar el método:

- `ITeamInfo GetTeamInfo(int playerId)`: Retorna un objeto de tipo `ITeamInfo` (ver Sección 3) con la información elegida por el usuario. El argumento `playerId` sirve para mostrar en la ventana cuál es el jugador que está seleccionando el equipo.

Por ejemplo, este código abre una ventana y pide a ambos jugadores que seleccionen sus equipos y luego, muestra la información en consola.

```
1  using Shin_Megami_Tensei_GUI;
2
3  var gui = new SMTGUI();
4  gui.Start(Main);
5
6  void Main() {
7      ITeamInfo team1 = gui.GetTeamInfo(1);
8      ShowTeamInfo(team1);
9      ITeamInfo team2 = gui.GetTeamInfo(2);
10     ShowTeamInfo(team2);
11 }
12
13 void ShowTeamInfo(ITeamInfo team)
14 {
15     Console.WriteLine($"Samurai: {team.SamuraiName}");
16     Console.WriteLine("Skills:");
17     foreach (var skill in team.SkillNames)
18         Console.WriteLine(skill);
19     Console.WriteLine("Demons:");
20     foreach (var demon in team.DemonNames)
21         Console.WriteLine(demon);
22 }
```

Al ejecutar este código, el programa se quedará en la línea 7 esperando hasta que el primer jugador presione "Seleccionar equipo". En ese momento, `GetTeamInfo(...)` retornará un objeto del tipo `ITeamInfo` que contiene la información seleccionada por el jugador.



En este caso, presionar "Seleccionar equipo" hará que el método `ShowTeamInfo(...)` imprima esto en consola:

```
Samurai: Kyouji
Skills:
Gather Spirit Energy
High Fire Pleroma
Demons:
Gurulu
Hel
```

Una vez seleccionados los equipos, hay dos métodos que podrían ser de interés:

- `void ShowEndGameMessage(string message)`: Muestra `message` en la ventana. Este método puede servir para mostrar que la selección fue inválida, para mostrar que un jugador se rindió o para mostrar que un jugador ganó el juego.
- `void Update(IState state)`: Actualiza el contenido de la ventana en base a `state`, en particular, muestra la vista principal del juego (tablero, unidades en reserva, opcciones, etc.).

Como se mencionó anteriormente, para poder utilizar el método `Update(...)` es necesario crear clases que implementen las interfaces que utiliza este método. Supongamos que creamos las siguientes clases[1]:

```
public class Unit(string name, int hp, int mp, int maxHp, int maxMp)
    : IUnit
{
    public string Name { get; } = name;
    public int HP { get; set; } = hp;
    public int MP { get; set; } = mp;
    public int MaxHP { get; } = maxHp;
    public int MaxMP { get; } = maxMp;
}
```

```
public class Player(IUnit?[] unitsInBoard, IEnumerable<IUnit> unitsInReserve) : IPlayer
{
    public IUnit?[] UnitsInBoard { get; } = unitsInBoard;
    public IEnumerable<IUnit> UnitsInReserve { get; set; } = unitsInReserve;
}
```

```
public class State : IState
{
    public IPlayer Player1 { get; set; }
    public IPlayer Player2 { get; set; }

    public IEnumerable<string> Options { get; set; } =
    [
        "Atacar",
        "Disparar",
        "Usar Habilidad",
        "Invocar",
        "Rendirse"
    ];

    public int Turns { get; set; } = 2;
    public int BlinkingTurns { get; set; } = 7;

    public IEnumerable<string> Order { get; set; } =
    [
        "Flynn",
        "Joker",
        "Alice",
    ];
}
```

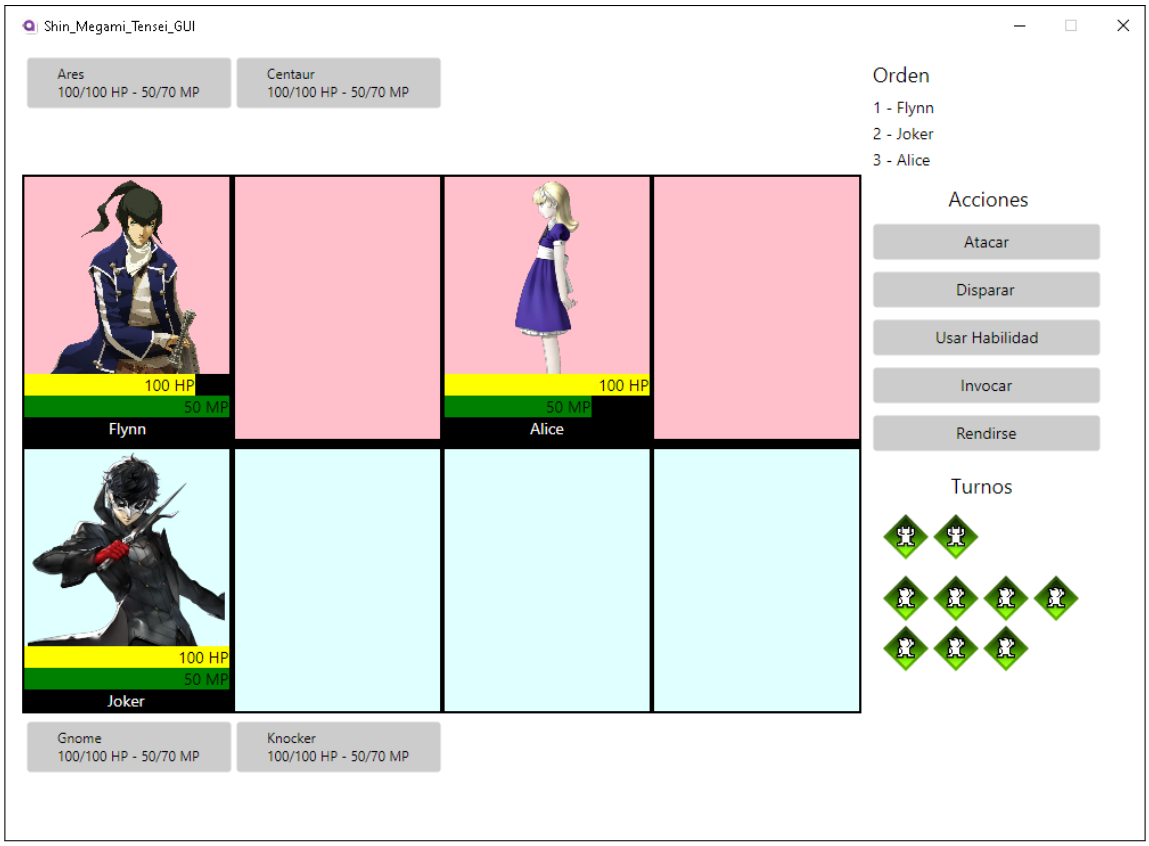
Ahora, en el `Program.cs` creamos una instancia de `State` cualquiera, ignorando los equipos seleccionado por el usuario:

```
using Shin_Megami_Tensei_GUI;

var gui = new SMTGUI();
gui.Start(Main);

void Main() {
    ITeamInfo team1 = gui.GetTeamInfo(1);
    ITeamInfo team2 = gui.GetTeamInfo(2);
    var flynn = new Unit("Flynn", 100, 50, 120, 50);
    var joker = new Unit("Joker", 100, 50, 100, 50);
    var alice = new Unit("Alice", 100, 50, 100, 70);
    var ares = new Unit("Ares", 100, 50, 100, 70);
    var centaur = new Unit("Centaur", 100, 50, 100, 70);
    var gnome = new Unit("Gnome", 100, 50, 100, 70);
    var knocker = new Unit("Knocker", 100, 50, 100, 70);
```

[1] En este ejemplo, el orden y las opciones están hardcodeados. En la práctica, no debería ser así

```
16
17      var player1 = new Player(
18          [
19              flynn,
20              null,
21              alice,
22          ],
23          [
24              ares,
25              centaur,
26          ]
27      );
28      var player2 = new Player(
29          [
30              joker,
31          ],
32          [
33              gnome,
34              knocker
35          ]
36      );
37
38      var state = new State
39      {
40          Player1 = player1,
41          Player2 = player2,
42      };
43
44      gui.Update(state);
45  }
```

Este código mostrará las ventanas para elegir los equipos y luego mostrará esta ventana:



Desde esta ventana podemos interactuar con todos los botones (`Button`), unidades en el tablero (`UnitInBoard`) y unidades en reserva (`UnitInReserve`).

Para poder obtener información del usuario puedes utilizar el método `GetClickedElement()`, el cual retorna una instancia de `IClickedElement`, el cual además de contener el tipo de elemento seleccionado (`Type`), retorna información como el contenido del elemento seleccionado (`Text`) y, en caso de que el elemento no sea `Button` contiene el `PlayerId` relacionado (para diferenciar unidades que tengan el mismo nombre).

Cabe destacar que `GetClickedElement(...)` **espera** a que el usuario haga *click* en algún elemento y retorna. Por lo tanto, su retorno nunca será `null`. Por otro lado, dado que en ocasiones ustedes querrán que el usuario seleccione ciertos elementos, probablemente necesitarán algo de este estilo:

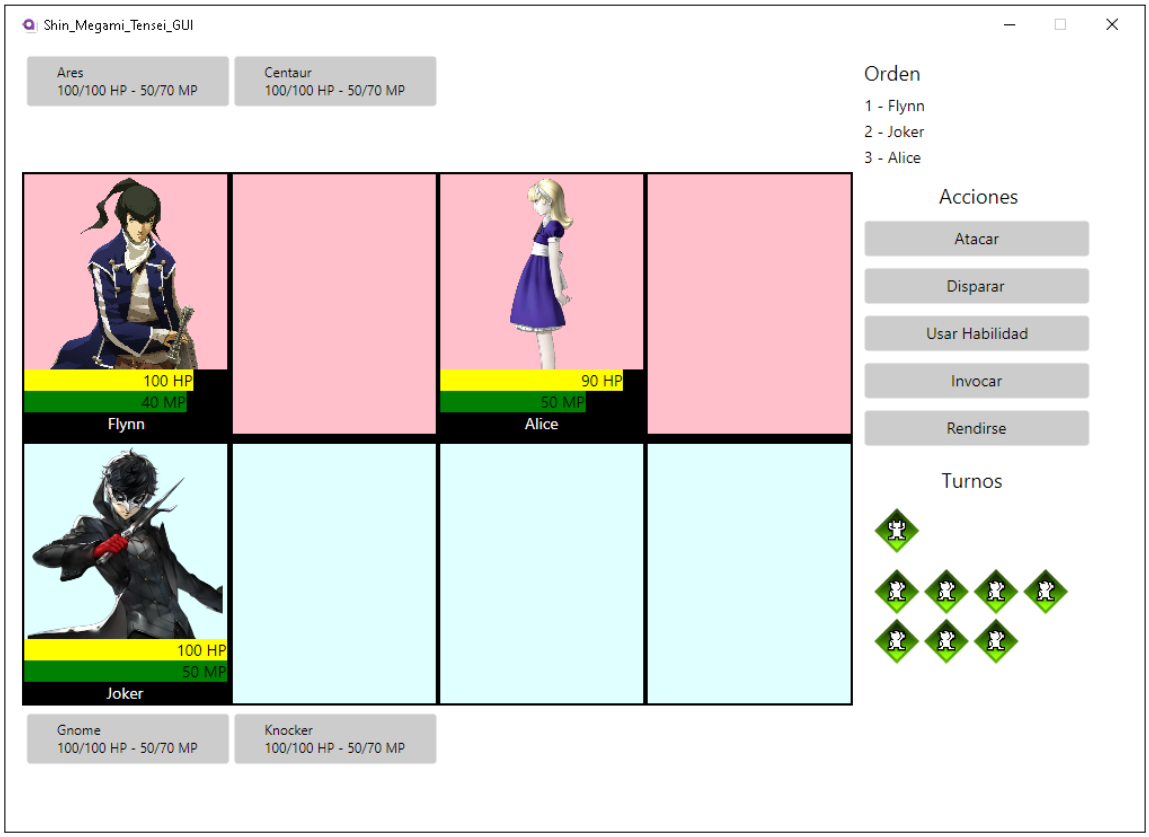
```
IClickedElement clickedElement;
do {
    clickedElement = gui.GetClickedElement();
} while (!(clickedElement.Type == ClickedElementType.Button && clickedElement.Text == "
    Atacar"));
```

Este bucle sólo se romperá cuando el elemento retornado por `GetClicledElement` cumpla las condiciones correspondientes.

Por ejemplo, si modificamos el método `Main()` mostrado anteriormente, y le agregamos las siguientes líneas:

```
IClickedElement clickedElement;
do {
    clickedElement = gui.GetClickedElement();
} while (!(clickedElement.Type == ClickedElementType.Button && clickedElement.Text == "
    Atacar"));

do {
    clickedElement = gui.GetClickedElement();
} while (clickedElement.Type != ClickedElementType.UnitInBoard);

if (clickedElement.Text == "Flynn")
    flynn.HP -= 10;
else if (clickedElement.Text == "Joker")
    joker.HP -= 10;
else if (clickedElement.Text == "Alice")
    alice.HP -= 10;
flynn.MP -= 10;
state.Turns--;

gui.Update(state);
```

Luego de dar *click* en "Atacar" y en alguna de las 3 unidades en el tablero, el tablero se actualizará. Suponiendo que le dimos *click* a Alice, la vista se verá así:



Cabe destacar que si se da *click* en algún elemento distinto al botón "Atacar", el programa se quedará en el primer bucle. Esto te servirá para garantizar que no pase nada si el usuario selecciona un elemento incorrecto.

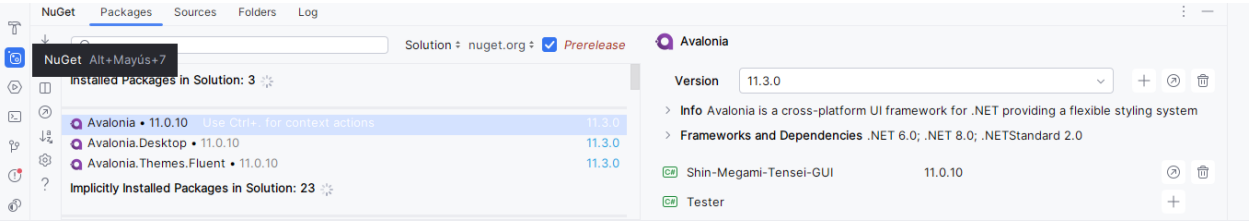
Consideraciones para la entrega

Cuando agregues la interfaz gráfica a tu entrega, debes cumplir con los siguientes requerimientos:

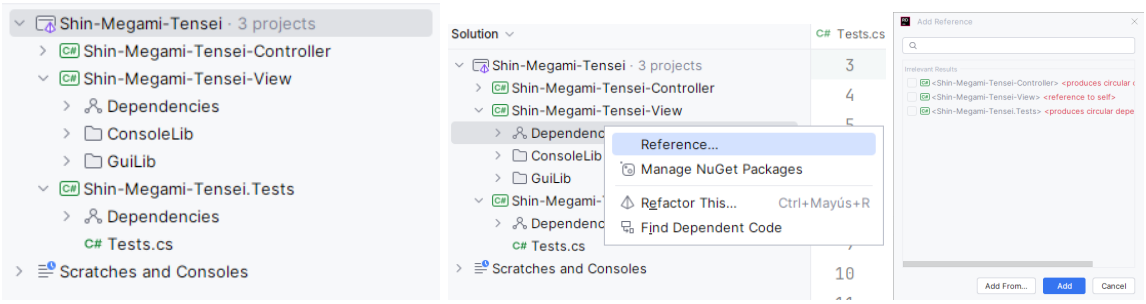
- Se puede elegir un equipo a ambos jugadores.
- Si alguno de los equipos es inválido, se muestra el mensaje de que el equipo es inválido.
- Si ambos equipos son válidos, se muestra la ventana con ambos equipos junto sus stats (HP y MP), el orden inicial, la cantidad de turnos correspondientes y las opciones correspondientes a la unidad inicial.
- Se permite realizar todas las acciones disponible para cada unidad, como atacar, disparar, etc.
- Durante el juego se actualiza correctamente la ventana en base a las acciones del usuario, como el movimiento de unidades entre el tablero y la reserva, la actualización de stats, turnos, etc.
- Al finalizar el juego, se felicita al equipo que gana.

A. ¿Cómo agregar la librería a mi proyecto?

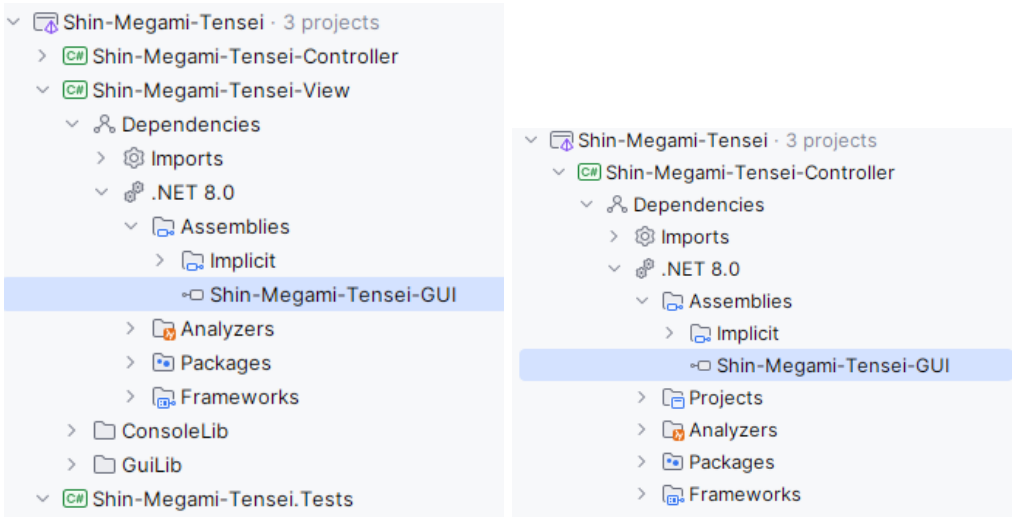
Para agregar la librería a un proyecto existente debes realizar los siguientes pasos. Primero, abre `NuGet` desde `Rider` e instala las librerías `Avalonia`, `Avalonia.Desktop` y `Avalonia.Themes.Fluent` en el proyecto `Shin-Megami-Tesei-View`:



Pon la carpeta `GuiLib` dentro del proyecto `Shin-Megami-Tensei-View`. Luego en *Dependencies* selecciona que quieres agregar una referencia mediante "Add From..."



Aparecerá un menú con las carpetas de tu computador. Anda hasta la carpeta `GuiLib` dentro de tu proyecto `Shin-Megami-Tensei-View` y selecciona `Shin-Megami-Tensei-GUI.dll`. **Repite el proceso para el proyecto** `Shin-Megami-Tensei-Controller`. Si seguiste los pasos correctamente, podrás ver que la librería `Shin-Megami-Tensei-GUI` fue agregada a las dependencias de los proyectos `Controller` y `View`.



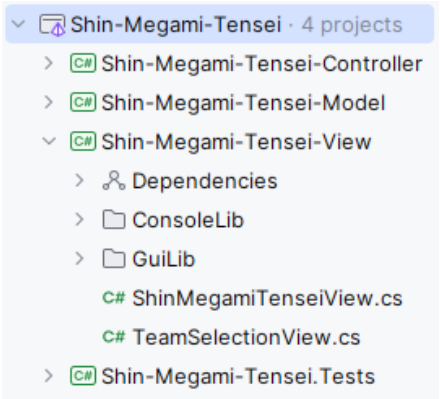
Finalmente, puedes verificar que la librería fue importada utilizando el siguiente código (en tu `Program.cs`):

```
using Shin_Megami_Tensei_View;
using Shin_Megami_Tensei;
using Shin_Megami_Tensei_GUI;


var gui = new SMTGUI();
gui.Start(Main);

void Main() {}
```

Al correr ese código debería aparecer la siguiente ventana:

B. ¿Cómo incluyo la interfaz sin echarme los test cases?

Comencemos desde una entrega que implementa MVC. Como ejemplo, utilicemos una entrega que permite elegir el archivo de equipos y luego, arbitrariamente, dice que el equipo es inválido.




```
public class Game {
    private readonly ShinMegamiTenseiView _view;
    private readonly string _teamsFolder;

    public Game(View view, string teamsFolder) {
        _view = new FireEmblemView(view);
        _teamsFolder = teamsFolder;
    }

    public void Play() {
        string teamFile = _view.SelectTeamFile(
    _teamsFolder);
        _view.AnnounceThatTeamIsInvalid();
    }
}
```

El objetivo es agregar la interfaz gráfica sin echar a perder los test cases. Lo primero es notar que `Game` usa como vista `ShinMegamiTenseiView`. Esta clase se encuentra en el proyecto de la vista, e incluye el siguiente código:

```
public class ShinMegamiTenseiView(View view){
    public void AnnounceThatTeamIsInvalid()
        => view.WriteLine("Archivo de equipos inválido");

    public string SelectTeamFile(string teamsFolder){
        TeamSelectionView teamSelectionView = new (view, teamsFolder);
        return teamSelectionView.SelectTeamFile();
    }
}
```

Básicamente, tiene un método para anunciar que el equipo es inválido y otro para mostrar el menú que permite seleccionar al equipo. Notar que la clase `ShinMegamiTenseiView` es nuestra vista consola. Contiene métodos que, mediante la consola, muestran mensajes y reciben inputs. Al mismo tiempo, esta clase permite que funcionen los test cases pues interactúa con el `View` utilizado para testear el proyecto.

Para poder cambiar al modo interfaz gráfica tenemos que crear una nueva vista. La nueva vista debería tener los mismos métodos que `ShinMegamiTenseiView`, pero al elegir equipos y mostrar mensajes se debería hacer mediante la interfaz gráfica. De esa forma, el mismo controlador podrá funcionar con ambas vistas.

Partamos creando una interfaz con los métodos comunes a todas las vistas que el controlador podrá utilizar.

```
public interface IShinMegamiTenseiView{
    void AnnounceThatTeamIsInvalid();
    string SelectTeamFile(string teamsFolder);
}
```

Ahora hacemos que nuestra vista actual implemente la interfaz (y aprovechamos de cambiarle el nombre):

```
public class ShinMegamiTenseiView(View view) : IShinMegamiTenseiView {
    public void AnnounceThatTeamIsInvalid()
        => view.WriteLine("Archivo de equipos inválido");

    public string SelectTeamFile(string teamsFolder){
        TeamSelectionView teamSelectionView = new (view, teamsFolder);
        return teamSelectionView.SelectTeamFile();
    }
}
```

Finalmente, hacemos que `Game` ahora funcione con cualquier vista que implemente nuestra interfaz:

```
public class Game {
    private readonly IShinMegamiTenseiView _view;
    private readonly string _teamsFolder;

    public Game(View view, string teamsFolder) {
        _view = new FireEmblemView(view);
        _teamsFolder = teamsFolder;
    }

    public void Play() {
        string teamFile = _view.SelectTeamFile(_teamsFolder);
        _view.AnnounceThatTeamIsInvalid();
    }
}
```

En el proyecto `Shin-Megami-Tensei-View`, agregamos una nueva vista para la interfaz gráfica:

```
public class ShinMegamiTenseiGuiView : IShinMegamiTenseiView {
    private SMTGUI _window = new ();

    public void Start(Action startProgram)
        => _window.Start(startProgram);
    public void AnnounceThatTeamIsInvalid()
        => _window.ShowEndGameMessage("Al menos un equipo es inválido");

    public string SelectTeamFile(string teamsFolder){
        ITeamInfo team1 = _window.GetTeamInfo(1);
        ITeamInfo team2 = _window.GetTeamInfo(2);
        return TeamInfoFormatter.FormatTeamInfo(team1, team2);
    }
}
```

Gracias a que esta vista también implementa `ShinMegamiTenseiView`, `Game` podrá usarla sin problemas. Aunque, para que ello ocurra, tenemos que crear un objeto `ShinMegamiTenseiGuiView`. La forma más simple de hacer esto, sin tener que cambiar los test cases, es creando un segundo constructor para `Game`.

```
public class Game {
    private readonly IShinMegamiTenseiView _view;
    private readonly string _teamsFolder;

    public Game(View view, string teamsFolder) {
        _view = new FireEmblemView(view);
        _teamsFolder = teamsFolder;
    }

    public Game(ShinMegamiTenseiGuiView view){
        _view = view;
        _teamsFolder = "";
    }

    public void Play() {
        string teamFile = _view.SelectTeamFile(_teamsFolder);
        _view.AnnounceThatTeamIsInvalid();
    }
}
```

Finalmente, en el `Program.cs` podemos decidir si comenzamos en modo consola o en modo interfaz gráfica dependiendo del constructor que usemos.

```
using Shin_Megami_Tensei_GUI;
using Shin_Megami_Tensei_View;
using Shin_Megami_Tensei;

bool useGui = true;
if (useGui)
{
    ShinMegamiTenseiGuiView view = new ();
    Game game = new Game(view);
    view.Start(game.Play);
}
else
{
    string testFolder = SelectTestFolder();
    string test = SelectTest(testFolder);
    string teamsFolder = testFolder.Replace("-Tests","");
    AnnounceTestCase(test);

    var view = View.BuildManualTestingView(test);
    var game = new Game(view, teamsFolder);
    game.Play();
}
```