Getting Started

Adding the NuGet package to your project

You need to pull BP.AdventureFramework into your project. The easiest way to do this is to add the NuGet package. The latest package and installation instructions are available here (https://github.com/benpollarduk/BP.AdventureFramework/pkgs/nuget/BP.AdventureFramework).


```
namespace BP.AdventureFramework.GettingStarted
{
    internal class Program
    {
        private static void Main(string[] args)
        {
        }
    }
}
```

Adding a PlayableCharacter

Every game requires a character to play as, lets add that next:

```
private static PlayableCharacter CreatePlayer()
{
    return new PlayableCharacter("Dave", "A young boy on a quest to find the meaning
of life.");
}
```

In this example whenever **CreatePlaye**r is called a new **PlayableCharacter** will be created. The character is called "Dave" and has a description that describes him as "A young boy on a quest to find the meaning of life.".

Creating the game world

The game world consists of a hierarchy of three tiers: **Overworld**, **Region** and **Room**. We will create a simple **Region** with two **Rooms**. We can do this directly in the **Main** function for simplicity. To start with lets make the **Rooms**:

```
private static void Main(string[] args)
{
    var cavern = new Room("Cavern", "A dark cavern set in to the base of the mountai
n", new Exit(Direction.North));

    var tunnel = new Room("Tunnel", "A dark tunnel leading inside the mountain.", ne
w Exit(Direction.South));
}
```

Although the **Rooms** haven't been added to a **Region** yet there are exits in place that will allow the player to move between them.

Game rooms are boring without **Items** to interact with, let's add an item to the tunnel:

```
var holyGrail = new Item("Holy Grail", "A dull golden cup, looks pretty old.", tru
e);

tunnel.AddItem(holyGrail);
```

Looking good, but the **Rooms** need to be contained within a **Region**. **RegionMaker** simplifies this process, but sometimes creating a **Region** directly may be more appropriate if more control is needed. Here we will use **RegionMaker**:

```
var regionMaker = new RegionMaker("Mountain", "An imposing volcano just East of tow
n.")
{
    [0, 0, 0] = cavern,
    [0, 1, 0] = tunnel
};
```

This needs more breaking down. The **RegionMaker** will create a region called "Mountain" with a description of "An imposing volcano just East of town.". The region will contain two rooms, the cavern and the tunnel. The cavern will be added at position *x* 0, *y* 0, *z* 0. The tunnel will be added at position *x* 0, *y* 1, *z* 0, north of the cavern.

The game world is nearly complete, but the **Region** needs to exist within an **Overworld** for it to be finished. We will use **OverworldMaker** to achieve this:

```
var overworldMaker = new OverworldMaker("Daves World", "An ancient kingdom.", region
Maker);
```

This will create an **Overworld** called "Daves World" which is described as "An ancient kingdom" and contains a single **Region**.

All together the code looks like this:

```
var cavern = new Room("Cavern", "A dark cavern set in to the base of the mountain.",
new Exit(Direction.North));

var tunnel = new Room("Tunnel", "A dark tunnel leading inside the mountain.", new Ex
it(Direction.South));

var holyGrail = new Item("Holy Grail", "A dull golden cup, looks pretty old.", tru
e);

tunnel.AddItem(holyGrail);

var regionMaker = new RegionMaker("Mountain", "An imposing volcano just East of tow
n.")
{
    [0, 0, 0] = cavern,
    [0, 1, 0] = tunnel
};

var overworldMaker = new OverworldMaker("Daves World", "An ancient kingdom.", region
Maker);
```


Checking if the game is complete

For a game to come to an end it needs to reach either a game over state or a completion state.

Firstly lets look at the logic that determines if the game is complete. An **EndCheck** is required, which returns an **EndCheckResult** that determines if the game is complete.

In this example lets make a method that determines if the game is complete. The game is complete if the player has the holy grail:

```
private static EndCheckResult IsGameComplete(Game game)
{
    if (!game.Player.FindItem("Holy Grail", out _))
        return EndCheckResult.NotEnded;

    return new EndCheckResult(true, "Game Complete", "You have the Holy Grail!");
}
```

If the player has the holy grail then the **EndCheckResult** will return that the game has ended, and have a title that will read "Game Complete" and a description that reads "You have the Holy Grail!".

A common game over state may be if the player dies:

```
private static EndCheckResult IsGameOver(Game game)
{
    if (game.Player.IsAlive)
        return EndCheckResult.NotEnded;

    return new EndCheckResult(true, "Game Over", "You died!");
}
```


Creating the game

The game now has all the required assets and logic it just needs some boilerplate to tie everything together before it is ready to play.

A **GameCreationCallback** is required to instantiate an instance of a **Game**. This is so that new instances of the **Game** can be created as required.

```
var gameCreator = Game.Create(
    "The Life Of Dave",
    "Dave finds himself in a cavern...",
    "A very low budget adventure.",
    x => OverworldMaker.Make(),
    PlayerMaker.Player,
    IsGameComplete,
    IsGameOver);
```

This requires some breaking down. The **Game** class has a **Create** method that can be used to create instances of **Game**. This method takes the following arguments:

- **Name** - the name of the game.
- **Introduction** - an introduction to the game.
- **Description** - a description of the game.
- **OverworldGenerator** - a callback for generating instances of the overworld.
- **PlayerGenerator** - a callback for generating instances of the player.
- **CompletionCondition** - a callback for determining if the game is complete.
- **GameOverCondition** - a callback for determining if the game is over.

Executing the game

The game is executed simply by calling the static **Execute** method on **Game** and passing in the game creation callback.

```
Game.Execute(gameCreator);
```

Bringing it all together

The full example code should look like this:

```
using BP.AdventureFramework.Assets;
using BP.AdventureFramework.Assets.Characters;
using BP.AdventureFramework.Assets.Locations;
using BP.AdventureFramework.Logic;
using BP.AdventureFramework.Utilities;

namespace BP.AdventureFramework.GettingStarted
{
    internal class Program
    {
        private static EndCheckResult IsGameComplete(Game game)
        {
            if (!game.Player.FindItem("Holy Grail", out _))
                return EndCheckResult.NotEnded;

            return new EndCheckResult(true, "Game Complete", "You have the Holy Grai
l!");
        }

        private static EndCheckResult IsGameOver(Game game)
        {
            if (game.Player.IsAlive)
                return EndCheckResult.NotEnded;

            return new EndCheckResult(true, "Game Over", "You died!");
        }

        private static PlayableCharacter CreatePlayer()
        {
            return new PlayableCharacter("Dave", "A young boy on a quest to find the
 meaning of life.");
        }

        private static void Main(string[] args)
        {
            var cavern = new Room("Cavern", "A dark cavern set in to the base of the
 mountain.", new Exit(Direction.North));

            var tunnel = new Room("Tunnel", "A dark tunnel leading inside the mounta
in.", new Exit(Direction.South));

            var holyGrail = new Item("Holy Grail", "A dull golden cup, looks pretty
 old.", true);

            tunnel.AddItem(holyGrail);

            var regionMaker = new RegionMaker("Mountain", "An imposing volcano just
 East of town.")
            {
                [0, 0, 0] = cavern,
                [0, 1, 0] = tunnel
            };
```

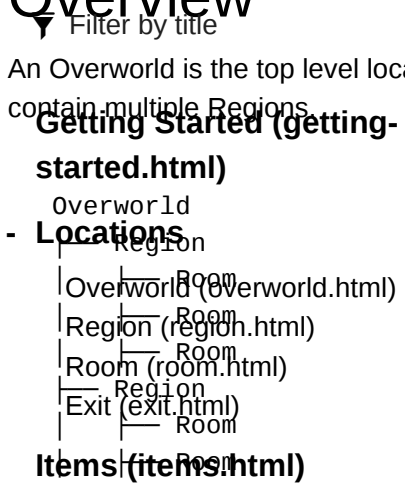
```
            var overworldMaker = new OverworldMaker("Daves World", "An ancient kingd
om.", regionMaker);

            var gameCreator = Game.Create(
                "The Life Of Dave",
                "Dave awakes to find himself in a cavern...",
                "A very low budget adventure.",
                x => overworldMaker.Make(),
                CreatePlayer,
                IsGameComplete,
                IsGameOver);

            Game.Execute(gameCreator);
        }
    }
}
```

Simply build and run the application and congratulations, you have a working BP.AdventureFramework game!

Overworld

Overview

An Overworld is the top level location in a game. A game can only contain a single Overworld. An Overworld can contain multiple Regions.



Use

An Overworld can be simply instantiated with a name and description.

```
var overworld = new Overworld("Name", "Description.");
```

Regions can be added to the Overworld with the **AddRegion** method.

```
overworld.AddRegion(region);
```

Regions can be removed from an Overworld with the **RemoveRegion** method.

```
overworld.RemoveRegion(region);
```

The Overworld can be traversed with the **Move** method.

```
overworld.Move(region);
```

OverworldMaker

The OverworldMaker simplifies the creation of the Overworld, when used in conjunction with RegionMakers.

```
var overworldMaker = new OverworldMaker("Name", "Description.", regionMakers);
```

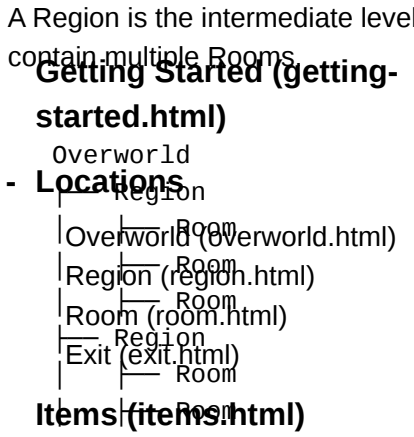
However, the main benefit of using an OverworldMaker is that it allows multiple instances of an Overworld to be created from a single definition of an Overworld.

```
var overworld = overworldMaker.Make();
```

Region

Overview

A Region is the intermediate level location in a game. An Overworld can contain multiple Regions. A Region can contain multiple Rooms.



A Region represents a 3D space.

- The **x** location always refers to the horizontal axis, with lower values being west and higher values being east.
- The **y** location always refers to the vertical axis, with lower values being south and higher values being north.
- The **z** location always refers to the depth axis, with lower values being down and higher values being up.

Use

A Region can be simply instantiated with a name and description.

```
var region = new Region("Name", "Description.");
```

Rooms can be added to the Region with the **AddRoom** method. The x, y and z location within the Region must be specified.

```
region.AddRoom(room, 0, 0, 0);
```

Rooms can be removed from a Region with the **RemoveRoom** method.

```
region.RemoveRoom(room);
```

The Region can be traversed with the **Move** method.

```
region.Move(Direction.North);
```

The start position, that is the position that the Player will start in when entering a Region, can be specified with **SetStartPosition**.

```
region.SetStartPosition(0, 0, 0);
```

The **UnlockDoorPair** method can be used to unlock an **Exit** in the current Room, which will also unlock the corresponding Exit in the adjoining **Room**.

```
region.UnlockDoorPair(Direction.East);
```

Like all Examinable objects, Regions can be assigned custom commands.

```
region.Commands =
[
    new CustomCommand(new CommandHelp("Warp", "Warp to the start."), true, (game, ar
gs) =>
    {
        region.JumpToRoom(0, 0, 0);
        return new Reaction(ReactionResult.OK, "You warped to the start.");
    })
];
```

RegionMaker

The RegionMaker simplifies the creation of a Region. Rooms are added to the Region with a specified **x**, **y** and **z** position within the Region.

```
var regionMaker = new RegionMaker("Region", "Description.")
{
    [0, 0, 0] = new Room("Room 1", "Description of room 1."),
    [1, 0, 0] = new Room("Room 2", "Description of room 2."),
};
```

The main benefit of using a RegionMaker is that it allows multiple instances of a Region to be created from a single definition of a Region.

```
var region = regionMaker.Make();
```

Room

Overview

A Room is the lowest level location in a game. A Region can contain multiple Rooms.

A Room can contain up to six Exits, one for each of the directions **north**, **east**, **south**, **west**, **up** and **down**.

Uso

A Region can be simply instantiated with a name and description.

```
var room = new Room("Name", "Description.");
```

Exits can be added to the Room with the **AddExit** method.

```
room.AddExit(new Exit(Direction.East));
```

Exits can be removed from a Room with the **RemoveExit** method.

```
region.RemoveExit(exit);
```

Items can be added to the Room with the **AddItem** method.

```
room.AddItem(new Item("Name", "Description."));
```

Items can be removed from a Room with the **RemoveItem** method.

```
region.RemoveItem(item);
```

Characters can be added to the Room with the **AddCharacter** method.

```
room.AddCharacter(new NonPlayableCharacter("Name", "Description."));
```

Characters can be removed from a Room with the **RemoveCharacter** method.

```
region.RemoveCharacter(character);
```

Rooms can contain custom commands that allow the user to directly interact with the Room.

```
room.Commands =
[
    new CommandHelp("Pull lever", "Pull the lever."), true, (game, args) =>
    {
        room.FindExit(Direction.East, true, out var exit);
        exit.Unlock();
        return new Reaction(ReactionResult.OK, "The exit was unlocked.");
    })
];
```

Exit

Overview

Filter by title

An Exit is essentially a connector between to adjoining rooms.

- **Getting Started (getting-started.html)**
- **Locations**
  - Overworld (overworld.html)
  - Region (region.html)
  - Room (room.html)
  - Exit (exit.html)
- **Items (items.html)**
- **Characters**
- **Conditional Descriptions (conditional-descriptions.html)**
- **Attributes (attributes.html)**
- **Commands (commands.html)**
- **Frame Builders (frame-builders.html)**
- **End Conditions (end-conditions.html)**

Use

An Exit can be simply instantiated with a direction.

```
var exit = new Exit(Direction.North);
```

An Exit can be hidden from the player by setting its **IsPlayerVisible** property to false, this can be set in the constructor.

```
var exit = new Exit(Direction.North, false);
```

Or set explicitly.

```
exit.IsPlayerVisible = false;
```

Optionally, a description of the Exit can be specified.

```
var exit = new Exit(Direction.North, true, new Description("A door covered in iv
y", "));
```

This will be returned if the player examines the Exit.

Like all Examinable objects, an Exit can be assigned custom commands.

```
exit.Commands =
[
    new CustomCommand(new CommandHelp("Shove", "Shove the door."), true, (game, arg
s) =>
    {
        exit.Unlock();
        return new Reaction(ReactionResult.OK, "The door swung open.");
    })
];
```

Item

Overview

Items can be used to add interactivity with a game. Items can be something that a player can take with them, or they may be static in a Room.

Use

An Item can be simply instantiated with a name and description.

```
var sword = new Item("Sword", "A heroes sword.");
```

By default, an Item is not takeable and is tied to a Room. If it is takeable this can be specified in the constructor.

```
var sword = new Item("Sword", "A heroes sword.", true);
```

An Item can morph into another Item. This is useful in situations where the Item changes state. Morphing is invoked with the **Morph** method. The Item that Morph is invoked on takes on the properties of the Item being morphed into.

```
var brokenSword = new Item("Broken Sword", "A broken sword");
sword.Morph(brokenSword);
```

Like all Examinable objects, an Item can be assigned custom commands.

```
bomb.Commands =
[
    new CustomCommand(new CommandHelp("Cut wire", "Cut the red wire."), true, (game,
args) =>
    {
        game.Player.Kill();
        return new Reaction(ReactionResult.Fatal, "Boom!");
    })
];
```

Interaction

Interactions can be set up between different assets in the game. The **InteractionResult** contains the result of the interaction, and allows the game to react to the interaction.

```
var dartsBoard = new Item("Darts board", "A darts board.");

var dart = new Item("Dart", "A dart")
{
    Interaction = item =>
    {
        if (item == dartsBoard)
            return new InteractionResult(InteractionEffect.SelfContained, item, "The
dart stuck in the darts board.");

        return new InteractionResult(InteractionEffect.NoEffect, item);
    }
};
```

PlayableCharacter

Overview

A PlayableCharacter represents the character that the player plays as throughout the game. Each game has only a single PlayableCharacter.

Use

A PlayableCharacter can be simply instantiated with a name and description.

```
var player = new PlayableCharacter("Ben", "A 39 year old man.");
```

A PlayableCharacter can be also be instantiated with a list of Items.

```
var player = new PlayableCharacter("Ben", "A 39 year old man.",
[
    new Item("Guitar", "A PRS Custom 22, in whale blue, of course."),
    new Item("Wallet", "An empty wallet, of course.")
]);
```

A PlayableCharacter can be given items with the **AcquireItem** method.

```
player.AcquireItem(new Item("Mallet", "A large mallet."));
```

A PlayableCharacter can lose an item with the **DequireItem** method.

```
player.DequireItem(mallet);
```

A PlayableCharacter can use an item on another asset:

```
var trapDoor = new Exit(Direction.Down);
var mallet = new Item("Mallet", "A large mallet.");
player.UseItem(mallet, trapDoor);
```

A PlayableCharacter can give an item to a non-playable character.

```
var goblin = new NonPlayableCharacter("Goblin", "A vile goblin.");
var daisy = new Item("Daisy", "A beautiful daisy that is sure to cheer up even the m
ost miserable creature.");
player.Give(daisy, goblin);
```

PlayableCharacters can contain custom commands that allow the user to directly interact with the character or other assets.

```
player.Commands =
[
    new CustomCommand(new CommandHelp("Punch wall", "Punch the wall."), true, (game,
args) =>
    {
        return new Reaction(ReactionResult.OK, "You punched the wall.");
    })
];
```

NonPlayableCharacter

Overview

A NonPlayableCharacter represents any character that the player may meet throughout the game.

Use

A NonPlayableCharacter can be simply instantiated with a name and description.

```
var goblin = new NonPlayableCharacter("Goblin", "A vile goblin.");
```

A NonPlayableCharacter can give an item to another NonPlayableCharacter.

```
var daisy = new Item("Daisy", "A beautiful daisy that is sure to cheer up even the m
ost miserable creature.");
npc.Give(daisy, goblin);
```

NonPlayableCharacters can contain custom commands that allow the user to directly interact with the character or other assets.

```
goblin.Commands =
[
    new CustomCommand(new CommandHelp("Smile", "Crack a smile."), true, (game, args)
=>
    {
        return new Reaction(ReactionResult.OK, "Well that felt weird.");
    })
];
```

Conversations

A NonPlayableCharacter can hold a conversation with the player.

- A Conversation contains **Paragraphs**.
- A Paragraph can contain one or more **Responses**.
- A **Response** can contain a delta or other implementation of **IEndOfPargraphInstruction** to shift the conversation by, which will cause the conversation to jump paragraphs by the specified value.
- A **Response** can also contain a callback to perform some action when the player selects that option.

```
goblin.Conversation = new Conversation(
    new Paragraph("This is a the first line."),
    new Paragraph("This is a question.")
    {
        Responses =
        [
            new Response("This is the first response.", new Jump(1)),
            new Response("This is the second response.", new Jump(2)),
            new Response("This is the third response.", new Jump(3))
        ]
    },
    new Paragraph("You picked first response, return to start of conversation.", new
GoTo(1)),
    new Paragraph("You picked second response, return to start of conversation.", ne
w GoTo(1)),
    new Paragraph("You picked third response, you are dead.", game => game.Player.Ki
ll())
);
```

Conditional Descriptions

Overview

Filter by title

Normally assets are assigned a **Description** during the constructor. This is what is returned when the asset is examined.

Descriptions are usually specified as a string.

```
var item = new Item("The items name", "The items description.");
```

They can also be specified as a **Description**.

```
var item = new Item(new Identifier("The items name"), new Description("The items des
cription."));
```

However, sometimes it may be desirable to have a conditional description that can change based on the state of the asset.

Conditional descriptions can be specified with **ConditionalDescription** and contain a lambda which determines which one of two strings are returned when the asset is examined.

```
// the player, just for demo purposes
var player = new PlayableCharacter("Ben", "A man.");

// the description to use when the condition is true
var trueString = "A gleaming sword, owned by Ben.";

// the string to use when the condition is false
var falseString = "A gleaming sword, without an owner.";

// a lambda that determines which string is returned
Condition condition = () => player.FindItem("Sword", out _);

// the conditional description itself
var conditionalDescription = new ConditionalDescription(trueString, falseString, con
dition);

// create the item with the conditional description
var sword = new Item(new Identifier("Sword"), conditionalDescription);
```

Attributes

Overview

All examinable objects can have attributes. Attributes provide a way of adding a lot of depth to games. For example, attributes could be used to buy and sell items, contain a characters XP or HP or even provide a way to add durability to items.

Use

To add to an existing attribute or to create a new one use the **Add** method.

```
var player = new PlayableCharacter("Player", string.Empty);
player.Attributes.Add("$", 10);
```

To subtract from an existing attribute use the **Subtract** method.

```
player.Attributes.Subtract("$", 10);
```

Attributes values can be capped. In this example the $ attribute is limited to a range of 0 - 100. Adding or subtracting will not cause the value of the attribute to change outside of this range.

```
var cappedAttribute = new Attribute("$", "Dollars.", 0, 100);
player.Attributes.Add(cappedAttribute, 50);
```

An example - buying an Item from a NonPlayableCharacter.

The following is an example of buying an Item from NonPlayableCharacter. Here a trader has a spade. The player can only buy the spade if they have at least $5. The conversation will jump to the correct paragraph based on if they choose to buy the spade or not. If the player chooses to buy the spade and has enough $ the transaction is made and the spade changes hands.

```
const string currency = "$";

var player = new PlayableCharacter("Player", string.Empty);
player.Attributes.Add(currency, 10);

var trader = new NonPlayableCharacter("Trader", string.Empty);
var spade = new Item("Spade", string.Empty);
trader.AcquireItem(spade);
trader.Conversation = new Conversation(
    new Paragraph("What will you buy?")
    {
        Responses =
        [
            new Response("Spade", new ByCallback(() =>
                player.Attributes.GetValue(currency) >= 5
                    ? new ToName("BoughtSpade")
                    : new ToName("NotEnough"))),
            new Response("Nothing", new Last())
        ]
    },
    new Paragraph("Here it is.", _ =>
    {
        player.Attributes.Subtract(currency, 5);
        trader.Attributes.Add(currency, 5);
        trader.Give(spade, player);
    }, new First(), "BoughtSpade"),
    new Paragraph("You don't have enough money.", new First(), "NotEnough"),
    new Paragraph("Fine.")
);
```


This is just one example of using attributes to add depth to a game.

Commands

Overview

There are three main types of Command.

- **Game Commands** are used to interact with the game.
- **Meta Commands** are used to interact with the program running the game.
- **Custom Commands** allow developers to add custom commands to the game without having to worry about extended the games interpreters.

Game Commands

Drop

Allows players to drop an item. **R** can be used as a shortcut.

```
drop sword
```

The player can also drop **all** items.

```
drop all
```

Examine

Allows players to examine any asset. **X** can be used as a shortcut.

Examine will examine the current room.

```
examine
```

The player themselves can be examined with **me** or the players name.

```
examine me
```

or

```
examine ben
```

The same is true for Regions, Overworlds, Items and Exits.

Take

Allows the player to take an Item. **T** can be used as a shortcut.

```
take sword
```

Take **all** allows the player to take all takeables Items in the current Room.

```
take all
```


Talk

Talk allows the player to start a conversation with a NonPlayableCharacter. **L** can be used as a shortcut.

If only a single NonPlayableCharacter is in the current Room no argument needs to be specified.

```
talk
```

However, if the current Room contains two or more NonPlayableCharacters then **to** and the NonPlayableCharacter's name must be specified.

```
talk to dave
```

Use

Use allows the player to use the Items that the player has or that are in the current Room.

```
use sword
```

Items can be used on the Player, the Room, an Exit, a NonPlayableCharacter or another Item. The target must be specified with the **on** keyword.

```
use sword on me
```

Or

```
use sword on bush
```

Move

Regions are traversed with direction commands.

- **North** or **N** moves north.
- **East** or **E** moves east.
- **South** or **S** moves south.
- **West** or **W** moves west.
- **Down** or **D** moves down.
- **Up** or **U** moves up.

End

Only valid during a conversation with a NonPlayableCharacter, the End command will end the conversation.

```
end
```

Global Commands

About

Displays a screen containing information about the game.

```
about
```

CommandsOn / CommandsOff

Toggles the display of the contextual commands on the screen on and off.

```
commandson
```

Or

```
commandsoff
```

Exit

Exit the current game.

```
exit
```

Help

Displays a Help screen listing all available commands.

```
help
```

KeyOn / KeyOff

Toggles the display of the map key on and off.

```
keyon
```

Or

```
keyoff
```

Map

Displays the Region Map screen.

```
map
```

New

Starts a new game.

```
new
```

Custom Commands

Custom commands can be added to many of the assets, including Room, PlayableCharacter, NonPlayableCharacter, Item and Exit.

Overview

In BP.AdventureFramework output is handled using the **FrameBuilders**. A FrameBuilder is essentially a class that builds a **Frame** that can render a specific state in the game. This **Frame** can then be rendered on a **TextWriter** by calling its **Render** method. Think of the **FrameBuilder** as the instructions that build the output display and the **Frame** as the output itself.

There are a few types of **FrameBuilder**, each responsible for rendering a specific game state.

- **SceneFrameBuilder** is responsible for building frames that render the scenes in a game.
- **TitleFrameBuilder** is responsible for building the title screen frame.
- **RegionMapFrameBuilder** is responsible for building a frame that displays a map of a Region.
- **TransitionFrameBuilder** is responsible for building frames that display transitions.
- **AboutFrameBuilder** is responsible for building a frame to display the about information.
- **HelpFrameBuilder** is responsible for building frames to display the help.
- **GameOverFrameBuilder** is responsible for building a frame to display the game over screen.
- **CompletionFrameBuilder** is responsible for building a frame to display the completion screen.
- **ConversationFrameBuilder** is responsible for building a frame that can render a conversation.

A game accepts a **FrameBuilderCollection**. A **FrameBuilderCollection** is a collection of all the different **FrameBuilders** required to render a game. All **FrameBuilders** are extensible, so the output for all parts of the game can be fully customised.

End Conditions

Overview

The **EndCheck** class allows the game to determine if it has come to an end. Each game has two end conditions

- **GameOverCondition** when the game is over, but has not been won.
- **CompletionCondition** when the game is over because it has been won.

Use

When an **EndCheck** is invoked it returns an **EndCheckResult**. The **EndCheckResult** details the result of the check to see if the game has ended.

```
private static EndCheckResult IsGameOver(Game game)
{
    if (game.Player.IsAlive)
        return EndCheckResult.NotEnded;

    return new EndCheckResult(true, "Game Over", "You died!");
}
```

This method can be used as an **EndCheck**:

```
EndCheck gameOverCheck = IsGameOver;
```

The **GameOverCondition** and **CompletionCondition** are passed in to the game as arguments when a game is created.