Small source code overview

Source code is organized into following namespaces:

PDG.LayoutStructures – all classes that represent or work with layout. Layout, Room, Connector, RoomBorder are defined there.

PDG.LayoutGeneration – components and classes for layout generation.

PDG.DungeonBuilders – components and classes for dungeon building.

PDG.DungeonBuilders.TileBased – TileBasedBuilder(base for TilemapDungeonBuilder and TiledGameObjectDungeonBuilder) and related classes.

PDG.LayoutGeneration.Initializers – layout initializers (classes that implement ILayoutInitializer).

PDG.LayoutGeneration.Selectors – room selectors (classes that Implement IRoomSelector).

PDG.LayoutGeneration.Modifiers – modifiers (classes that implement ILayoutModifier)

PDG.Tilemaps – abstract tilemap classes, and methods for their generation and modification.

PDG.Tags – classes that work with tags or tagged objects.

PDG.DataStructures – helpful datastructures.

PDG.MeshGeneration – classes for mesh generation o rmodification.

PDG.Misc – additional helping classes.

To define your own Dungeon Builder or Layout Generator that are compatible with BuildersCoordinator you need to inherit DungeonBuilder class or implement ILayoutGenerator interface.

```
public abstract class DungeonBuilder : MonoBehaviour
{
    3 references
    public abstract void Prepare(Layout layout, IRandom random);
    4 references
    public abstract void Build();
    7 references
    public abstract void Clear();
}
```

Prepare method is called before building. If BuildAsync is enabled – it will be called in background thread, so make sure that it doesn't use Unity API.

If you there is no need for any background work – just store layout and random objects

Build and Clear methods are called in main thread. use them to build or destroy the dungeon.

```
public class SomeBuilder : DungeonBuilder
{
    private Layout _layout;
    private IRandom _random;
    4 references
    public override void Prepare(Layout layout, IRandom random)
    {
        _layout = layout;
        _random = random;
        //some work for background thread
    }
    5 references
    public override void Build()
    {
        //actual building
    }

    8 references
    public override void Clear()
    {
        //destroying the dungeon
    }
}
```

To implement layout generator you need to implement ILayoutGenerator and IValidatable interfaces

```
interface ILayoutGenerator : IValidatable
{
    5 references
    public Layout Generate(IRandom random);
    5 references
    public List<Func<Layout, Layout>> IterativeGenerationList(IRandom random);
}
```

You can use GenerationSequence class to help with generation setup:

```
public class SomeLayoutGenerator : MonoBehaviour, ILayoutGenerator
{
    2 references
    private GenerationSequence SetupSequence()
    {
        GenerationSequence sequence = new GenerationSequence();
        RoomSpecification redRooms = new RoomSpecification()
            .AddMustIncludeTag("Red");
        sequence.SetInitializer(new SingleRoom(30, 30));
        sequence.AddStep(
            new BinarySpacePartitioning()
                .WidthRange(4, 9)
                .HeightRange(4, 9),
            new RandomSelector());
        sequence.AddStep(
            new AddTag("Red"),
            new CentralSelector()
                .PercentageRange(0.4f, 0.5f));
        sequence.AddStep(
            new AddTag("Green"),
            new RandomSelector()
                .CountRange(3, 5)
                .RoomParams(redRooms));
        return sequence;
    }
    6 references
    public Layout Generate(IRandom random)
    {
        return SetupSequence().Generate(random);
    }

    6 references
    public List<Func<Layout, Layout>> IterativeGenerationList(IRandom random)
    {
        return SetupSequence().IterativeGenerationList(random);
    }

    66 references
    public bool ValidateParameters(out string message)
    {
        //parameters check, fi you need it
        message = "";
        return true;
    }
}
```