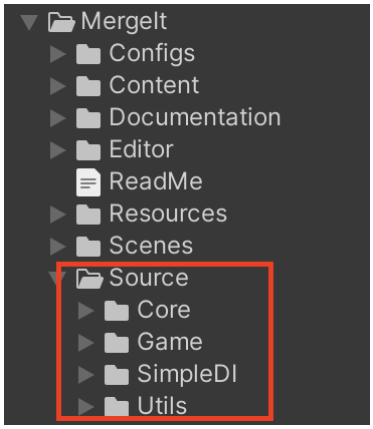
Merge Toolkit (MTK) - Manual

Introduction

Merge Toolkit has 4 different modules: **Core**, **Game**, **SimpleDI,** and **Utils**.



Utils module *(MergeIt/Source/Utils)*

This module contains useful helpers and utilities to make working with components easier.

1. Bindable

This class allows you to create reactive properties. For example, you can declare such property somewhere in your class and then subscribe to its changing in another class:

```
public class UserServiceModel
{
    public Bindable<int> Energy { get; } = new();
}
```

```
public class EnergyService
{
    private void Initialize()
    {
            _userServiceModel.Energy.Subscribe(OnEnergyChanged, true);
    }
}
```

But don’t forget to call the Unsubscribe method when your class is disposed of.

2. ResponsiveGridLayout

This class is used for adjusting cells in your GridLayouGroup. You can add this component to your game object, and it will calculate cell sizes according to the parameters you set in your grid to make those cells fit in the grid.

3. IMonoUpdateHandler and IMonoApplicationQuitHandler

Those interfaces are used for observing Unity Engine events, like an Update and OnApplicationQuit, in classes that are not MonoBehaviours. To use them you need to implement those interfaces in your own class and then add your class to the **MonoEventsListener** in the following way:

```
public class NonMonoBehaviourClass : IMonoUpdateHandler
{
   public NonMonoBehaviourClass()
   {
       MonoEventsListener.Instance.SubscribeOnUpdate(this);
   }

   public void Update()
   {
       // Your logic here.
   }
}
```

4. MonoEventsListener

This class is used to manage classes that implement **IMonoUpdateHandler** and **IMonoApplicationQuitHandler.** It's a singleton and is created automatically when the game starts.

5. MainThreadDispatcher

This class ensures that certain actions are executed on the main thread. This is necessary for Unity, as many of its APIs are not thread-safe and require operations to be performed on the main thread.

5.1. void Enqueue(Action action)

Adds actions to the execution queue if they are not null.

5.2. void RunOnMainThread(Action action)

Directly runs an action if already on the main thread; otherwise, it enqueues the action.

5.3. bool IsMainThread()

Checks if the current thread is the main thread.

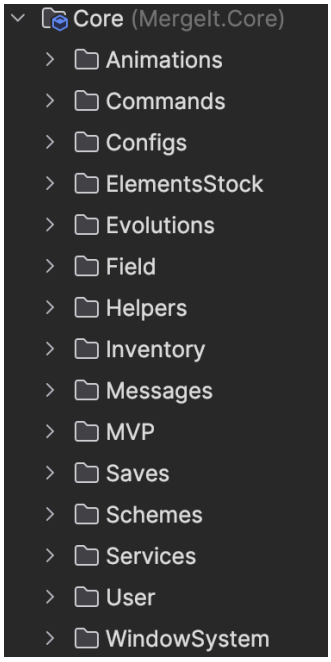
SimpleDI module *(MergeIt/Source/SimpleDI)*

This module contains a lightweight framework for working with dependency injections. You can use any other framework instead (Zenject, Reflex, etc.) or customize this framework on your own 🙂. All you need from it are the DiContainer, DiRoot, and IntroduceAttribute classes. We'll consider further examples of its usage.

Core module *(MergeIt/Source/Core)*

This module contains the core tools you will need to create a merge-like game, such as field creation, config processing, element merging logic, save handling, window system, and so on. We will review it alongside the **Game** module, which serves as an example of a game prototype.

So, the **Core module** has the following structure:



Let's consider each of those features.

1. Animations

Contains scripts and assets related to animations, including animation controllers, behaviors, and event handlers. It is used for handling FieldElement animations and Window animations. The main interfaces are:

a. IWindowAnimationListener

This interface is used for listening to window events invoked when the window opens or closes. All windows in the Merge Toolkit implement this interface. Interface has the following methods:

- **void OnOpenStarted()**

  This method should be called somewhere in your WindowAnimationController to indicate that the window starts opening.

- **void OnOpenFinished()**

  This method should be called somewhere in your WindowAnimationController to indicate that the window has opened.

- **void OnCloseStarted()**

  This method should be called somewhere in your WindowAnimationController to indicate that the window starts closing.

- **void OnCloseFinished()**

  This method should be called somewhere in your WindowAnimationController to indicate that the window is just closed.

b. IWindowAnimationController

This interface is used for handling animations of your window. You should implement the logic of window opening and closing. You are free to implement this interface as a component or as a simple C# class. For instance, in the Merge Toolkit, there are two controllers that implement this interface: WindowDefaultAnimationController, which is a simple C# class, and WindowMecanimAnimationController, which is a Component. WindowDefaultAnimationController will be used by default if no other controllers are found. Interface has the following methods:

- **void Initialize(IWindowAnimationListener listener)**

  This method is used for the initialization of the controller. You should pass the IWindowAnimationListener into it.

- **void OpenWindow()**

  This method should contain the logic of window opening. It should be the Animator state changing, your own animation starting, etc. Here, IWindowAnimationListener.OnOpenStarted() could be called.

- **void CloseWindow()**

  This method should contain the logic of window closing. Here, IWindowAnimationListener.OnCloseStarted() could be called.

- **void OnOpenEnd()**

  This method could be called from your animation logic when the opening animation is finished. It could be invoked, for instance, from the AnimatorStateMachineBehaviour or when using DOTween or a similar plugin for animations upon completion of this animation. Here, IWindowAnimationListener.OnOpenFinished() could be called.

- **void OnCloseEnd()**

  Same as the previous method, but for closing. Here, IWindowAnimationListener.OnCloseFinished() could be called.

IAnimationListener and IAnimationController work in the same way, as interfaces described above, but not only for windows. You can customize and extend them to suit your needs. They are used for animating elements on the game field when a hint happens.

2. Commands

Contains CommandManager that provides functionality for executing some commands sequentially. For instance, you can use this manager on game start when you need to load configs, initialize user, prepare field, etc. Here is an example from the **Game module**:

```
private async void StartGameMessageHandler(StartGameMessage message)
{
   var manager = new CommandManager();

   manager.Add(new LoadConfigsCommand());
   manager.Add(new PrepareUserCommand());
   manager.Add(new PrepareEnergyCommand());
   manager.Add(new PrepareStockCommand());
   manager.Add(new PrepareInventoryCommand());
   manager.Add(new PrepareFieldCommand());
   manager.Add(new CheckEvolutionsProgressCommand());

   await manager.RunAsync();

   _messageBus.Fire<LoadedGameMessage>();
}
```

Each command here does some work, and the manager doesn't execute the next command until the previous command is finished. CommandManager has the following methods:

- ### void Add(ICommand command)

  Is used to add the new command to the command manager.

- ### void Run()

  Is used to run commands added to the command manager **synchronously** sequentially.

- **UniTask RunAsync()**

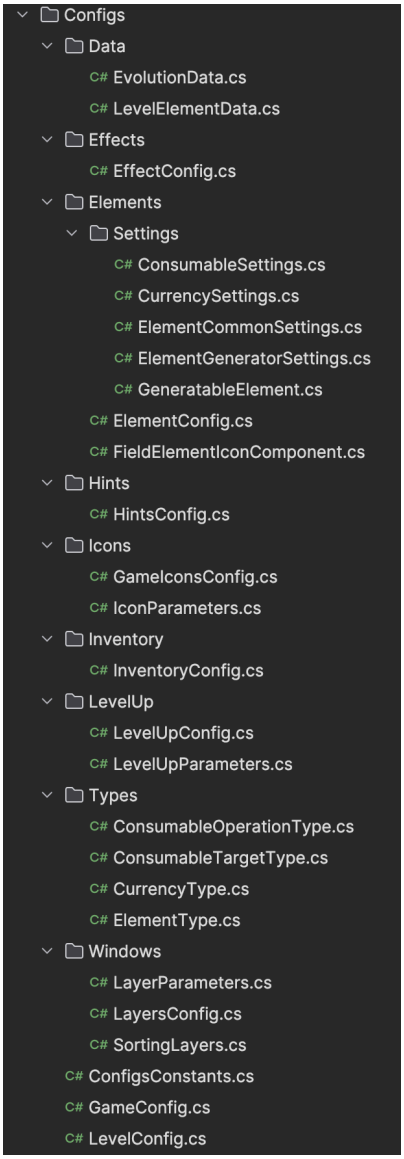
  Is used to run commands added to the command manager **asynchronously** sequentially.

- **void RunSimultaneously()**

  Is used to run commands simultaneously and waits while all of them are finished.

3. Configs

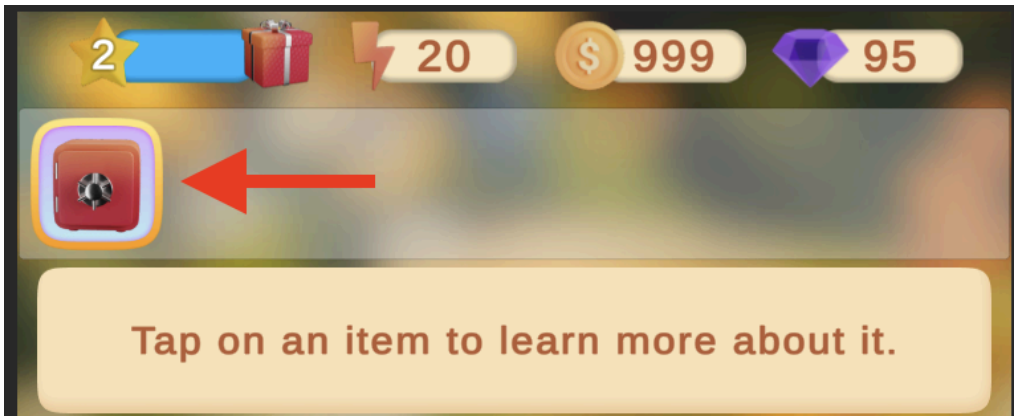
Contains scripts for managing different settings and parameters.



Each config is a ScriptableObject containing settings for different game parts. The **MTK-FAQ.pdf** document shows how they look. Here, it's worth paying attention to the window settings. Later, we will review them in the **WindowSystem** section.

4. ElementsStock

This folder contains scripts for managing the component shown below.



This component stores elements that the user claims after level-up. There are the following interfaces:

a. IElementsStockComponent

Represents a component assigned to the game object.

- **void SetupElement(ElementConfig elementConfig)**

  This method is used to set up the current element in the queue that is ready to be popped.

- **void PopElement()**

  This method is used to pop the element from the element stock. Here the logic of creating a new field element and notifying everyone about it should be implemented. In the **Game module** it is implemented using IGameFieldService and IMessageBus.

```
public void PopElement()
{
   if (_currentElement != null)
   {
       GridPoint? pointContainer = _fieldService.GetFreeCell();

       if (pointContainer != null)
       {
           GridPoint point = pointContainer.Value;

           IFieldElement newElement = _fieldService.CreateNewElement(_currentElement, point);

           var message = new CreateElementMessage
           {
               NewElement = newElement,
               FromPosition = _elementContainer.position,
               ToPoint = point,
           };
           _messageBus.Fire(message);

           if (_iconPrefab)
           {
               Destroy(_iconPrefab);
           }

           if (_stockService.Remove())
           {
               SetupElement(_stockService.GetNext());
           }
           else
           {
               Hide();
           }
       }
   }
```

b. IElementsStockData

This interface stores the element's stock state and saves it for further restoration after restarting the game. It contains only the array with elements' id.

```
public interface IElementsStockData : ISavable
{
   string[] Elements { get; set; }
}
```

5. Evolutions

This folder contains data classes related to elements' evolution progress. When the user merges elements, the new element from the evolution chain becomes available. We need to remember this and save this data for further display in the ElementInfo window.

6. FieldElements

This folder contains scripts that are related to the game field elements with their parameters.

IFieldElement is the basic logic element on the field in the Merge Toolkit. It is a composition of different parameters' sets:

```
public interface IFieldElement
{
   IConfigParameters ConfigParameters { get; set; }
   IInfoParameters InfoParameters { get; set; }
   IGeneratorParameters GeneratorParameters { get; set; }
   IGeneratorOpenParameters GeneratorOpenParameters { get; set; }
   IProduceParameters ProduceParameters { get; set; }
   IProducedByParameters ProducedByParameters { get; set; }
}
```

Each set is filled or null when the game starts, and a conversion from the saved data occurs.

- ### IConfigParameters

  contains information about the element config and evolution this element belongs to.

- **IInfoParameters**

  contains information about an element's name, description, type, position, and whether the element is blocked on the game field.

- **IGeneratorParameters**

  if an element is a generator, these parameters contain information about the available to drop elements, the count of charged elements, the time when charging was started, the time that remained to charge the one element, the time needed to full MinDrop charge, count of dropped elements and time remained to charge MinDrop.

- **IGeneratorOpenParameters**

  If an element is an openable generator, these parameters contain information on whether the generator is opening, when it started opening, and the time remaining to open the generator.

- **IProduceParameters**

  if an element is a generator, these parameters contain a list of elements that could be generated and their possibilities.

- **IProducedByParameters**

  contains information about the generators that could generate this element. It is a list of ElementConfig classes.

7. Helpers

This folder contains some helpers and extensions to simplify work with curves, lists, saves, RectTransform, and time.

8. Inventory

Contains the data class that is needed to save the state of the game inventory.

```
public interface IInventoryData : ISavable
```

```
{
    int InventorySize { get; set; }
    FieldElementData[] InventoryElements { get; set; }
}
```

9. Messages

This folder contains the functionality of MessageBus, which is needed to communicate between different parts of the project that are included in the dependency injection system.

10. MVP

This folder contains an implementation of the Model-View-Presenter pattern for organizing code, separating concerns, and managing UI logic. It is mostly used for the WindowSystem.

11. Saves

This folder contains the stuff needed to mark classes as saveable so that data can be serialized and saved between sessions.

12. Schemes

This folder contains the classes that are needed to work with the evolutions' schemes in the evolutions editor.

13. Services

This folder contains various service interfaces that should be implemented to provide core functionality across the game. The Game module provides examples of implementations of these services.

14. User

This folder contains an interface that should be implemented to provide information about the user. This data could be serialized to save the user's state between sessions.

```
public interface IUserData : ISavable
{
   string Name { get; set; }
   int Energy { get; set; }
   int SoftCurrency { get; set; }
   int HardCurrency { get; set; }
   int Splitters { get; set; }
   int Level { get; set; }
   int Experience { get; set; }
   long EnergyRestoringStartTime { get; set; }
}
```

15. WindowSystem

This folder contains the logic of the WindowSystem.

How it works

First, let's consider the main parts of our toolkit.

Dependency injections

As mentioned above, our Game template uses SimpleDI. This is a lightweight framework for working with dependency injections using property injection and field injection. First of all, you need to create some interface, for instance:

```
public interface IUserService
{
   void CreateUser();
   void SetupUser(IUserData userData);
}
```

and then create its implementation:

```
public class UserService : IUserService
{
   [Introduce] private IConfigsService _configsService;
   [Introduce] private IMessageBus _messageBus;
   [Introduce] private IGameSaveService _saveService;
   [Introduce] private UserServiceModel _userServiceModel;

   public void CreateUser()
   {
       GameConfig config = _configsService.GameConfig;
       var userData = new UserData
       {
           Name = $"User{new Random().Next(ushort.MinValue, ushort.MaxValue)}",
           Energy = config.EnergyCap,
           Experience = 0,
           SoftCurrency = config.InitialSoftCurrency,
           HardCurrency = config.InitialHardCurrency,
           Splitters = config.InitialSplittersCount,
           Level = 1
       };

       SetupUser(userData);

       _saveService.Save(GameSaveType.User);
   }

   public void SetupUser(IUserData userData)
   {
       _userServiceModel.Set(userData);
   }
}
```

Then you can bind your implementation in the composition root (in the GameModule this root is the GameRoot class) in the OnInstall method using DiContainer.RegisterInterfacesFor method:

```
DiContainer.RegisterInterfacesFor<UserService>().AsSingleton();
```

If you bind your class using the AsSingleton method, it is only one instance of this class that would be used in the game. Then it would be possible to inject your class in the other classes, registered with DiContainer using the IntroduceAttribute, for instance:

```
public class UserProgressService : IUserProgressService, IInitializable
{
   [Introduce]
   private IUserService _userService;
}
```

Also, you can get such classes directly from the DiContainer using the method Get:

```
public class PrepareUserCommand : Command
{
   private readonly IUserService _userService = DiContainer.Get<IUserService>();
}
```

IInitializable, IUpdatable

SimpleDI also has some special interfaces, like an IInitializable and IUpdatable.
When you need to do some work when your class is just created (e.g. you need to subscribe to some events or messages), you can implement the IInitializable interface and write your own logic in the Initialize method.
In case you want to make some work in your class for each frame like in the MonoBehaviour, you can implement the IUpdatable interface.

Messaging

This mechanism allows you to make communication between modules. It consists of MessageBus and messages that should implement the IMessage interface. To make the MessageBus work you need to register it in DiContainer. In the Game module, we make it in the GameRoot:

```
protected override void OnInstall()
{
   DiContainer.RegisterInterfacesFor<MessageBus>().AsSingleton();
}
```

Then you can create your own messages, for instance:

```
public class StartGameMessage : IMessage
{
}
```

After that, you can subscribe to this message in your classes (and don't forget to unsubscribe from it) and do some work when this message is fired:

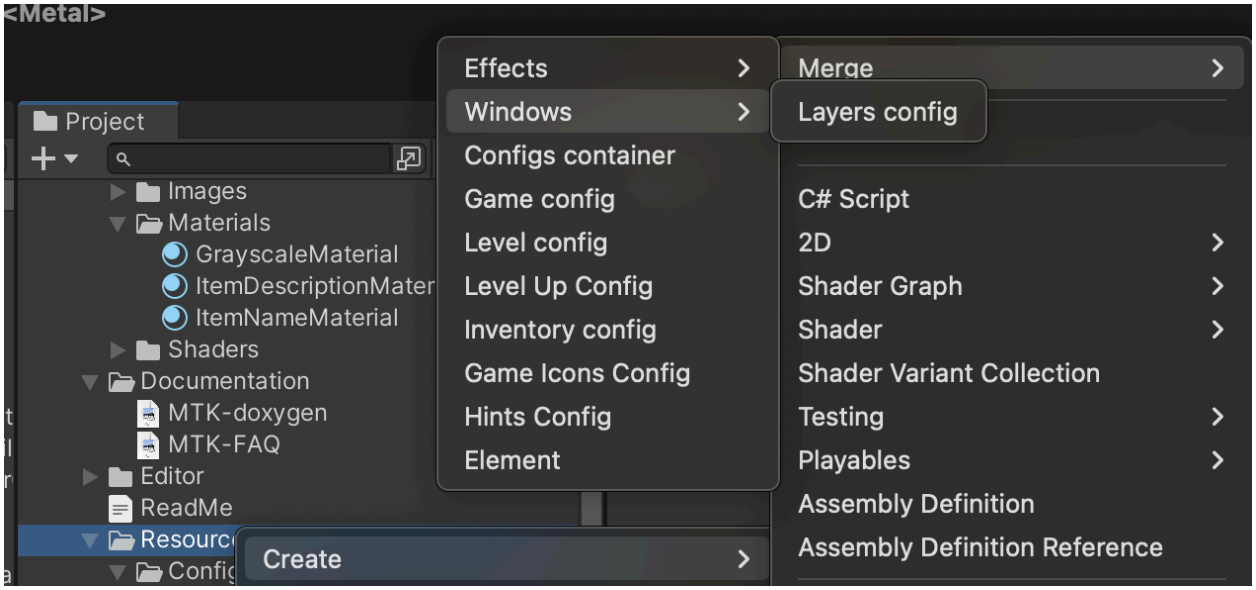
```
public class GameService : IGameService, IInitializable, IDisposable
{
   private GameServiceModel _gameServiceModel;

   [Introduce] public IMessageBus _messageBus;

   public void Dispose()
   {
       _messageBus.RemoveListener<StartGameMessage>(StartGameMessageHandler);
   }

   public void Initialize()
   {
       _messageBus.AddListener<StartGameMessage>(StartGameMessageHandler);
   }
```

To fire this message you can use the Fire<T> method from the MessageBus, for example:

```
protected override void Run()
{
   var messageBus = DiContainer.Get<IMessageBus>();
   messageBus.Fire<StartGameMessage>();
}
```

WindowSystem

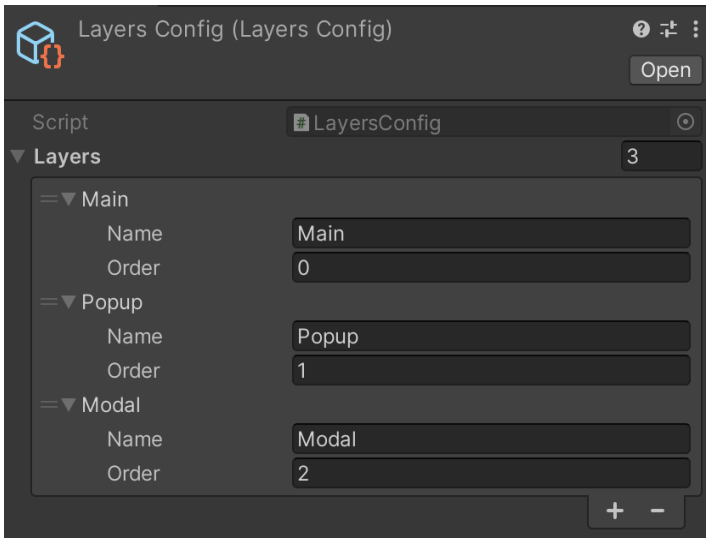
WindowSystem provides a basic functionality to work with windows. First, you need to setup layers that would be used in the WindowSystem, for that you need to create the LayersConfig



By default, WindowSystem expects the LayersConfig to be located in the *Resources/Configs/Windows/*. But you are free to change this location in the WindowSystem class:

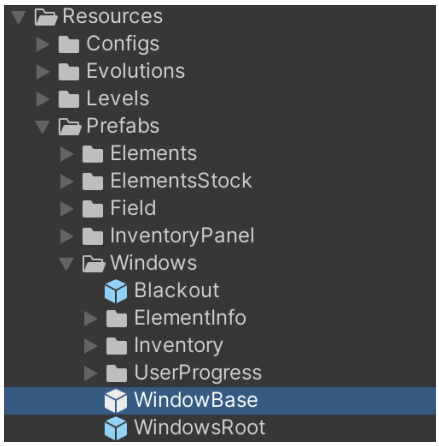
```
public class WindowsSystem : IWindowSystem, IInitializable, IDisposable
{
    private const string WindowsLayersConfigPath = "Configs/Windows/LayersConfig";
}
```

In the LayersConfig you can define the layers which control how the windows would be showing:



Windows with the high order would be shown over the windows with the low order. It is not necessary to place the layers by the order in the LayersConfig, because they would be sorted in the WindowSystem.

So, for now, we can create windows. All windows in the game should be created as prefab variants of WindowBase prefab which is located at *MergeIt/Resources/Prefabs/Windows.*



For example, let's examine the UserInfoWindow in the MergeIt/Resources/Prefabs/Windows/UserProgress/ folder. When we create a new window in the code, we should inherit it from the WindowBase class.

```
public class UserInfoWindow : WindowBase
```

Then we need to create a model of our window:

```
public class UserInfoModel : WindowModel
```

and after that, we need to create a presenter which would contain a logic of our window:

```
public class UserInfoPresenter : WindowPresenter<UserInfoWindow, UserInfoModel>
```

Then, we need to register our window in the GameRoot in the following way:

```
WindowsStorage.Register<UserInfoPresenter, UserInfoWindow>(WindowName.UserInfo,
SortingLayers.Main);
```

The WindowsStorage.Register method has the following signature:

```
void Register<TPresenter, TWindow>(string prefabName, string layerName)
```

It means we need to pass the path to the prefab of our window and the name of the layer on which this window would be shown. In our Game template this prefab would be searched in the directory *Prefabs/Windows* under the Resources folder. You can change this path in the WindowsStorage class:

```
public static class WindowsStorage
{
   private const string WindowsBasePath = "Prefabs/Windows";
}
```

After that, we can open our window using the WindowSystem class. For example:

```
   public class ProgressComponent : HudPanelBase
   {
       [SerializeField]
       private Button _progressButton;

       private IWindowSystem _windowSystem;

       protected override void Start()
       {
           base.Start();
             _progressButton.onClick.AddListener(OnProgressButtonClick);
       }

       public void OnDestroy()
       {
           _progressButton.onClick.RemoveListener(OnProgressButtonClick);
       }

       public void Initialize(IWindowSystem windowSystem)
       {
           _windowSystem = windowSystem;
       }

       private void OnProgressButtonClick()
       {
           _windowSystem.OpenWindow<UserInfoPresenter>(enableBlackout: true);
       }
}
```

The OpenWindow method has the following signature:

```
public void OpenWindow<TPresenter>(bool closePrevious = false, bool enableBlackout = false,
IWindowArgs args = null)
   where TPresenter : class, IWindowPresenter, new()
```

Let's consider it's parameters:

- **bool closePrevious** - if true, the existing window on the same layer would be closed after opening the new window; false means the existing window would be just hidden and restored after closing the next window.
- **bool enableBlackout** - if true, the semi-transparent blackout would appear behind the window.
- **IWindowArgs args** - this is additional arguments for window. They will be available in the WindowPresenter in the OnInitialize(IWindowArgs args = null) method.

So, how do you make all of this stuff work?

1. First, you need to prepare all the configs you saw in the **MTK-FAQ.pdf**.
2. Then you need to create the new class inherited from DiRoot from the SimpleDI module. This class would be the entry point of your game. Let's call it GameRoot.
3. Then create the empty game object on the scene and add the GameRoot component to it.
4. Then, you need to implement the services mentioned above. Let's consider what services that are implemented in the Game module do.

4.1. CurrencyService

This service manages operations when the user needs to pay for certain actions in the game, e.g., buying more cells in the inventory, skipping the generator charging/opening, or unlocking an element in the field. Also, this service operates with item selling when the user decides to sell an item located on the game field.

There are four currency types:

```
public enum CurrencyType
{
   Soft,
   Hard,
```

```
  Splitter,
   Energy
}
```

CurrencyService has the following methods:

- **bool TryPay(CurrencySettings currencySettings)**

  This method contains logic when the user is trying to pay for some action.

  a. **currencySettings** - is a structure that contains currency type and amount. It could be obtained from the ElementConfig.

- **void Sell(CurrencySettings currencySettings)**

  This method contains logic when the user is trying to sell an element from the game field.

4.2. ElementService

This service in general deals with work that should be done before element unlock/sell/split. It implements the following methods:

- **void TrySell(IFieldElement fieldElement)**

  This method contains logic when the user is trying to sell an element.

- **void TryUnlock(IFieldElement fieldElement)**

  This method contains logic when the user is trying to unlock an element.

- **void TrySplit(IFieldElement fieldElement)**

  This method contains logic when the user is trying to split an element.

4.3. ElementsStockService

This service deals with the operations related to element stock. It has the following methods:

- **void CreateStock()**

  This method contains the logic of stock creation. It creates the new stock and calls the SetupStock method. If the stock has not been created yet, this method should be called.

- **void SetupStock(IElementsStockData stockData)**

  This method contains the logic of the stock setup. It is called when the stock has been already created or restored from the save file.

- **void Add(ElementConfig elementConfig)**

  This method contains the logic of enquiring a new element to the element stock.

- **bool Remove()**

  This method contains the logic of dequeuing an element from the element stock.

- **ElementConfig GetNext()**

  This method contains the logic of getting the next element from the element stock.

- **ElementConfig GetCurrent()**

  This method contains the logic of getting the current element from the element stock.

- **IElementsStockData GetData()**

  This method contains the logic of retrieving elements stock data that could be serialized.

4.4. EnergyService

This service contains energy logic. It starts the energy restoration timer, checks this time, and restores energy.

4.5. EvolutionsService

This service deals with evolutions. It updates evolution's progress, allows getting the evolution chain for the element, allows getting the generators that could produce the passed element, and allows getting which elements could be generated by the passed element in case that element is the generator.

4.6. GameFieldActionsService

This service deals with actions related to the interactions with the field element like clicking and dragging. Every click on the element or dragging the element would be preceded by this service and appropriate processors. In the Game module there are 3 types of processors: FieldMergeProcessor, FieldGenerationProcessor, and FieldConsumableProcessor:

```
public void Initialize()
{
   _messageBus.AddListener<ClickElementMessage>(OnClickElementMessageHandler);
   _messageBus.AddListener<EndDragElementMessage>(OnEndDragElementMessageHandler);

   _fieldActionProcessors.Add(new FieldMergeProcessor());
   _fieldActionProcessors.Add(new FieldGenerationProcessor());
   _fieldActionProcessors.Add(new FieldConsumableProcessor());
}
```

FIeldMergeProcessor checks if the element was clicked or dragged and checks the possibility of merging this element with another one.

FIeldGenerationProcessor checks if the selected element is a generator and checks if it is able to generate field elements.

FieldConsumableProcessor checks if the selected field element is consumable (it could be soft currency/hard currency/energy/splitter/time/experience) and applies it if so. Consumable parameters could be set in the element config (scriptable object).

4.7. GameFieldService

This service deals with the game field. It creates and initializes the game field, allows us to find the free cell on the field, and allows us to create a new element on the field through the IFieldElementFactory.

4.8. GameService

This service is used for the initial setup of the game after setting up all the stuff in our composition root (GameRoot). When all dependencies are installed and all windows are registered in the GameRoot it sends the message StartGameMessage. GameService listens to this message and starts to set up our game:

```
private async void StartGameMessageHandler(StartGameMessage message)
{
   var manager = new CommandManager();
   manager.Add(new LoadConfigsCommand());
   manager.Add(new PrepareUserCommand());
   manager.Add(new PrepareEnergyCommand());
   manager.Add(new PrepareStockCommand());
   manager.Add(new PrepareInventoryCommand());
   manager.Add(new PrepareFieldCommand());
   manager.Add(new CheckEvolutionsProgressCommand());

   await manager.RunAsync();

   _messageBus.Fire<LoadedGameMessage>();
}
```

4.9. GeneratorService

This service deals with the field elements which are generators. It controls the generators' opening and charging, skipping the time of generators' charging and opening. Also, it monitors the addition and removal of generators on the game field.

4.10. InfoPanelService

This service deals with the actions that could be triggered in the InfoPanel and tightly works with the GeneratorService and ElementService.

In fact, it checks which action was triggered in this panel with some button clicking and calls the appropriate method from the corresponding service.

4.11. InventoryService

This service deals with the game inventory's logical representation. It creates and sets up the inventory and manages the addition and removal of its elements.

4.12. ResourcesLoaderService

This service loads the different game resources. In the Game module, it is implemented using Unity's Resources API. However, you can improve it by using Addressables, for instance.

4.13. UserProgressService

This service deals with the user's game progress. It checks whether the user's level can be raised, updates the user's level, and allows us to get the prizes list for reaching the new level.

4.14. UserService

This service works with the serializable user's data. It creates and sets up the user. The user has the following information:

```
[Serializable, Savable("user", "dat")]
public class UserData : IUserData
{
   public string Name { get; set; }
   public int Energy { get; set; }
   public int SoftCurrency { get; set; }
   public int HardCurrency { get; set; }
   public int Splitters { get; set; }
   public int Level { get; set; }
   public int Experience { get; set; }
   public long EnergyRestoringStartTime { get; set; }
}
```

5. Then, you must implement other classes that would be registered further in the dependency injection container. It could be some services' models, systems, factories, and so on. You can see how it is set up in the Game module in the GameRoot class.
6. Then you need to set up WindowsSystem. We already considered how to make it above.
7. Finally, you can add all of these to the GameRoot: register injectable classes in the DI container and register all your windows in the WindowStorage.

```
protected override void OnInstall()
{
   DiContainer.RegisterInterfacesFor<MessageBus>().AsSingleton();

   DiContainer.RegisterInterfacesFor<GameService>().AsSingleton();
   DiContainer.RegisterInterfacesFor<GameFieldService>().AsSingleton();
   DiContainer.RegisterInterfacesFor<GameFieldActionsService>().AsSingleton();
   DiContainer.RegisterInterfacesFor<GameSaveEveryIntervalService>().AsSingleton();
   DiContainer.RegisterInterfacesFor<UserService>().AsSingleton();
   DiContainer.RegisterInterfacesFor<UserProgressService>().AsSingleton();
   DiContainer.RegisterInterfacesFor<InventoryService>().AsSingleton();
   DiContainer.RegisterInterfacesFor<EnergyService>().AsSingleton();
   DiContainer.RegisterInterfacesFor<GeneratorService>().AsSingleton();
   DiContainer.RegisterInterfacesFor<InfoPanelService>().AsSingleton();
   DiContainer.RegisterInterfacesFor<CurrencyService>().AsSingleton();
   DiContainer.RegisterInterfacesFor<ElementService>().AsSingleton();
   DiContainer.RegisterInterfacesFor<ElementsStockService>().AsSingleton();
   DiContainer.RegisterInterfacesFor<EvolutionsService>().AsSingleton();

   DiContainer.RegisterInterfacesFor<FieldLogic>().AsSingleton();
```

```
    DiContainer.RegisterInterfacesFor<EffectsFactory>().AsSingleton();
    DiContainer.RegisterInterfacesFor<EffectsManager>().AsSingleton();

    DiContainer.RegisterInterfacesFor<ConfigsService>().AsSingleton();
    DiContainer.RegisterInterfacesFor<ConfigProcessor>().AsSingleton();
    DiContainer.RegisterInterfacesFor<GameLoadService>().AsSingleton();
    DiContainer.RegisterInterfacesFor<ResourcesLoaderService>().AsSingleton();

    DiContainer.RegisterInterfacesFor<FieldElementVisualFactory>().AsSingleton();
    DiContainer.RegisterInterfacesFor<FieldElementFactory>().AsSingleton();
    DiContainer.RegisterInterfacesFor<FieldFactory>().AsSingleton();
    DiContainer.RegisterInterfacesFor<InventoryFactory>().AsSingleton();
    DiContainer.RegisterInterfacesFor<ElementInfoFactory>().AsSingleton();
    DiContainer.RegisterInterfacesFor<IconFactory>().AsSingleton();

    DiContainer.RegisterInterfacesFor<HintsManager>().AsSingleton();

    DiContainer.Register<FieldLogicModel>().AsSingleton();
    DiContainer.Register<GameServiceModel>().AsSingleton();
    DiContainer.Register<UserServiceModel>().AsSingleton();
    DiContainer.Register<InventoryServiceModel>().AsSingleton();
    DiContainer.Register<ElementsStockServiceModel>().AsSingleton();

    DiContainer.RegisterInterfacesFor<WindowsSystem>().AsSingleton();
    DiContainer.RegisterInterfacesFor<WindowFactory>().AsSingleton();

    DiContainer.RegisterInterfacesFor<JsonSerializeStrategy>().AsSingleton();
}

protected override void Run()
{
    RegisterWindows();

    var messageBus = DiContainer.Get<IMessageBus>();
    messageBus.Fire<StartGameMessage>();
}

private void RegisterWindows()
{
```

```
   WindowsStorage.Register<InventoryPresenter, InventoryWindow>(WindowName.Inventory,
SortingLayers.Main);
   WindowsStorage.Register<UserInfoPresenter, UserInfoWindow>(WindowName.UserInfo,
SortingLayers.Main);
   WindowsStorage.Register<ElementInfoPresenter, ElementInfoWindow>(WindowName.ElementInfo,
SortingLayers.Popup);
   WindowsStorage.Register<ElementInfoPresenter, ElementInfoWindow>(WindowName.ElementInfo,
SortingLayers.Popup);
}

private void OnDestroy()
{
   DiContainer.Clear();
}
```

And that's it! The full game example in the Game module provides more details of services implementation, messaging, working with the WindowSystem, UI and so on. Also, you are free to customize or modify all of this stuff on your own. Enjoy!