Calculator Test

Serhii Smirnov

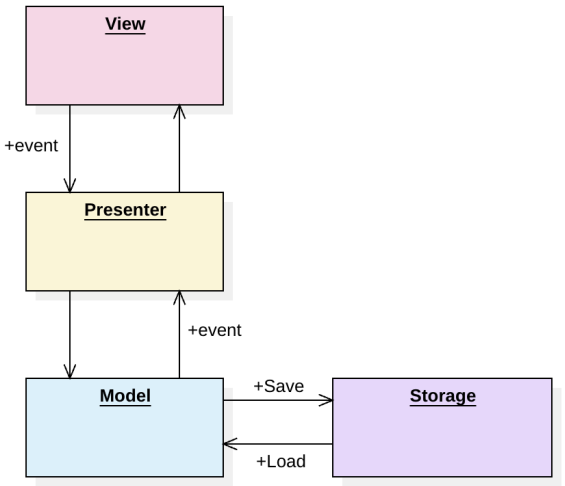
Ссылка на GitHub: https://github.com/Darkwing-Duck/calculator-app-test

Начну пожалуй с того, что опишу свою реализацию **MVP** паттерна, ведь как мы знаем реализация одного и того же архитектурного шаблона может быть множество и все, скорее, зависит от конкретного проекта и его потребностей.

В данной реализации есть 3 основные слоя модуля, что, собственно и говорится в названии паттерна.

1. Model - модель, которая содержит всю бизнес логику модуля. Модель ничего не знает ни о представлении, ни о презентере, что обеспечивает легкость тестирования независимо от других слоев модуля.
2. View - слой представления модуля. В данной реализации это пассивная вью, т.к. не содержит никакой логики относящейся к модулю, кроме визуальной и наследуется от **MonoBehaviour**. Также как и модель не имеет зависимостей на другие слои модуля.
3. Presenter - является медиатором между моделью и представлением и, соответственно, имеет зависимости на модель и представление.
4. Storage - если модулю нужно сохранять стейт между сессиями, то он может использовать хранилище для загрузки и сохранения стейта.

Обратная коммуникация модели и вью с презентером реализована через события.

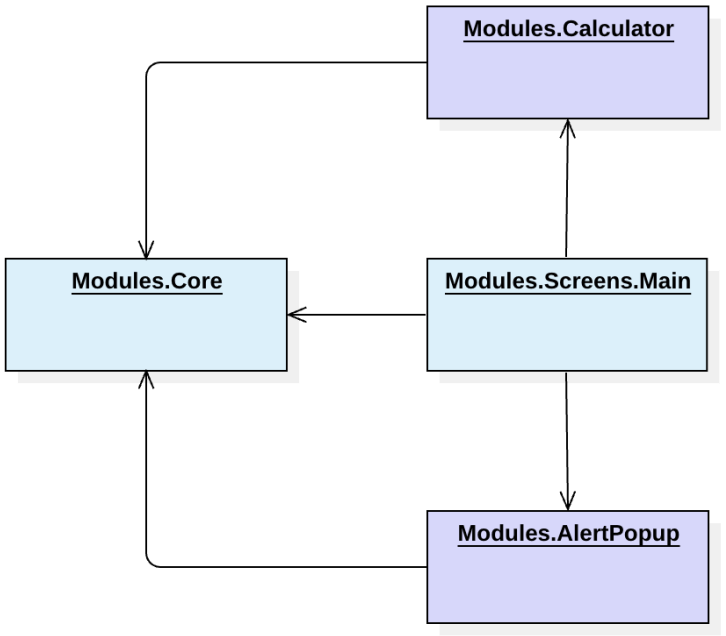


Сборки и их зависимости

В приложении на данный момент 4 сборки (Assembly Definition):

1. **Modules.Core** - основная сборка, которая содержит базовый код, на основе которого собираются остальные модули
2. **Modules.Screens.Main** - модуль основного экрана приложения. По сути пустой экран, внутри которого отображается модуль калькулятора.
3. **Modules.Calculator** - самостоятельные и легко переносимый модуль калькулятора (понятное дело, что если переносить в другой проект, то нужно переносить и, как минимум, модуль *Modules.Core*)
4. **Modules.AlertPopup** - самостоятельный и легко переносимый модуль вывода сообщений на экран.

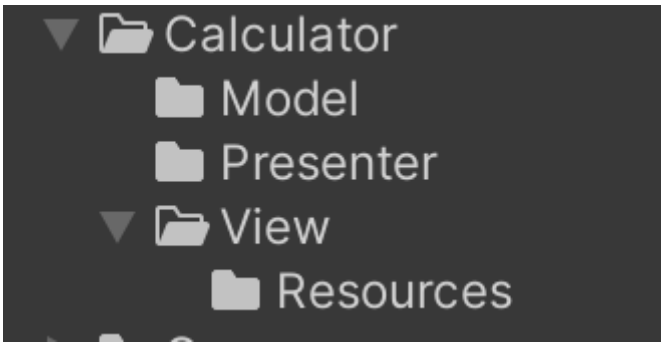
На диаграмме ниже показаны зависимости между всеми сборками в проекте:



Как видно из диаграммы, все конкретные модули зависят от Modules.Core, где сосредоточена базовая реализация.
Модули **Calculator** и **AlertPopup** выделены другим цветом, чтобы выделить те модули, которые были важны в реализации тестового задания.

Слой Представления

В данном проекте вью префабы загружаются из папки Resources. И каждый модуль имеет папку View, в которой находится папка Resources, относящаяся к данному модулю.



Для загрузки вью, презентеру нужно указать конкретную фабрику *IViewProvider<T>* из которой презентер будет доставать вью. В проекте, на данный момент, используются только фабрика *ResourcesViewProvider<T>*, которая скрывает реализацию згрузки вью из папки Resources. Но также есть имплементация *ChildViewProvider<T>*, которая позволяет не создавать новый инстанс префаба, а передать уже существующую вью (например можно было бы вынести InputField из модуля Calculator в отдельный модуль, который бы создавался внутри *CalculatorPresenter* и передавал бы вложенный GameObject из *CalculatorView*).
Имя префаба должно совпадать с именем класса, который описывает вью, и иметь в начале префикс "P_", это не обязательно, но в виду хорошей практики, я именую все префабы с префиксом "P_", материалы - с префиксом "MAT_" и тд. Соответственно, фабрика ищет префаб, используя имя класса, такая себе автоматизация:

```
var prefab = Resources.Load<TView>( path: $"P_{typeof(TView).Name}");
```

Презентер

По хорошему, для создания конкретного презентера, необходимо передать в конструктор фабрику IViewProvider<T>, соответственно презентеру не важно как будет создаваться вью, эта логика спрятана в конкретной реализации провайдера представления (например, ResourcesViewProvider или AddressablesViewProvider).

```
var presenter = new CalculatorPresenter(new ResourcesViewProvider<CalculatorView>());
```

Но в данном примере я указываю конкретную реализацию ViewProvider непосредственно в конкретном презентере, для простоты создания презентера с пустым конструктором.

```
1 usage  Serhii Smirnov
public CalculatorPresenter(IAlertPopupService alertPopupService) : base( viewProvider: new ResourcesViewProvider<CalculatorView>())
{
    _alertPopupService = alertPopupService;
    _historyItemViewProvider = new ResourcesViewProvider<CalculatorHistoryItemView>();
}
```

Хранилище

Для того, чтобы иметь возможность сохранять стейт модели между сессиями, нужно унаследовать модель от *PersistentModel<T>*, где T указывает на персистентный стейт, которым эта модель будет управлять и сохранять его в хранилище. А также презентер с персистентной моделью должен создать инстанс хранилища у себя внутри, потому что не каждый презентер должен иметь возможность сохранять, соответственно тот, кому нужно сохранение сам решает какое хранилище создавать. В данном проекте есть только одна реализация хранилища *IModuleStorage<TData>*, и это **PlayerPrefsModuleStorage**, который работает с дефолтным локальным хранилищем Unity - PlayerPrefs.

Например, модуль калькулятора использует, как раз, PlayerPrefsModuleStorage:

```
// module storage to save the persistent state
private IModuleStorage _storage = new PlayerPrefsModuleStorage(Name);
```

и при инициализации загружает стейт из хранилища:

```
protected override void InitializeModel(CalculatorModel model)
{
    Model.LoadFrom(_storage);
}
```

и при деактивации сохраняет стейт в хранилище:

```
protected override void OnDeactivate()
{
    // other code here ...

    // saving state only on presenter deactivation.
    // but it's possible to save it on any change model change
    Save();
}
```