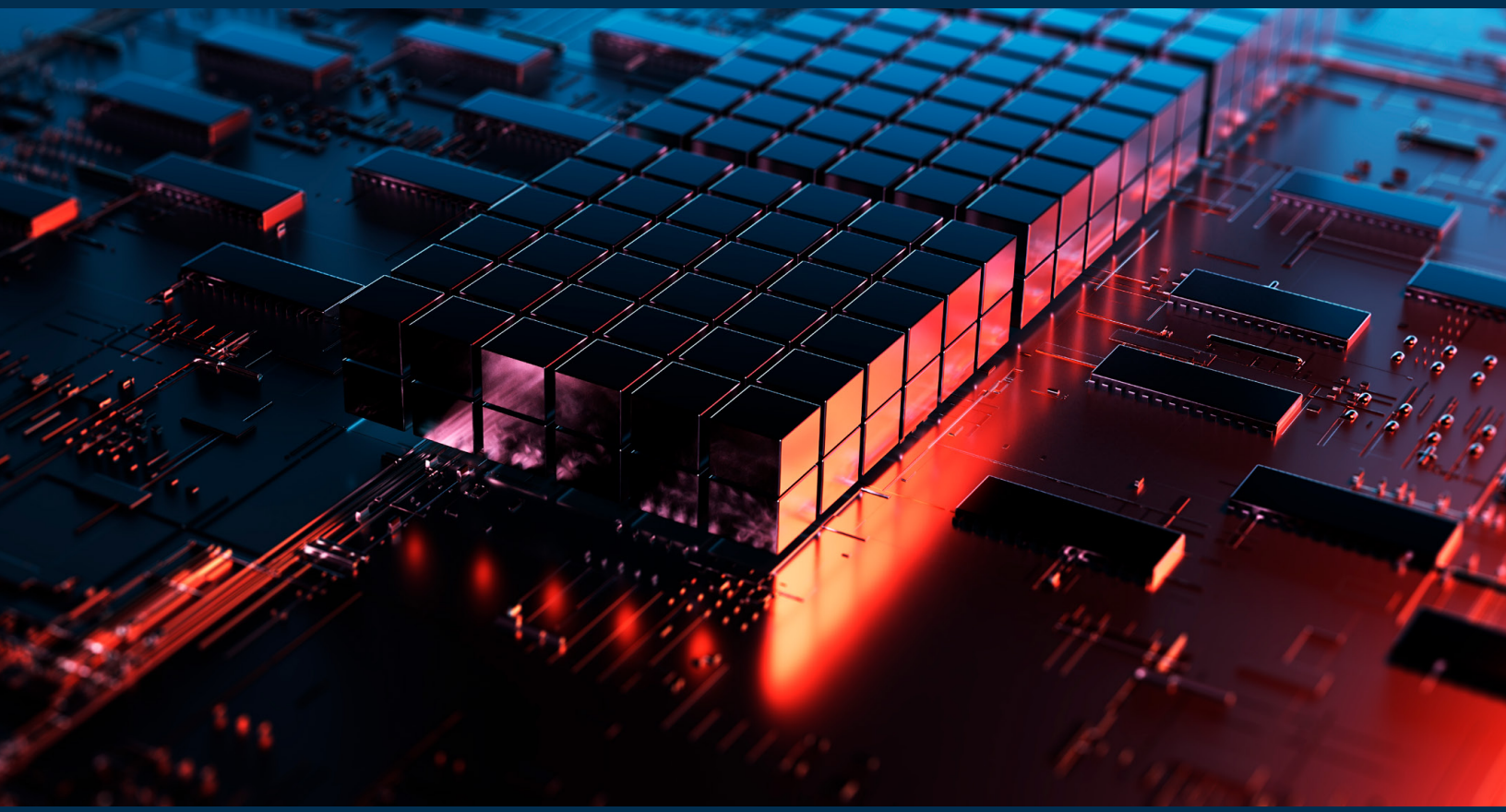
ЭЛЕКТРОННАЯ КНИГА

Создание модульной игровой архитектуры в Unity с помощью ScriptableObjects



Unity®

Содержание

Введение

Объекты ScriptableObject часто называют «контейнерами данных», однако такое определение не раскрывает всех их возможностей. При правильном применении объекты ScriptableObject помогают ускорить рабочий процесс в Unity, сократить расход памяти и упростить архитектуру кода.

В этом руководстве собраны советы профессиональных разработчиков по применению объектов ScriptableObject в реальных проектах. В частности, приведены примеры их использования в конкретных паттернах проектирования и способы избежать распространенных ошибок.

Объекты ScriptableObject помогают следовать принципам чистого кода, отделяя данные от логики. Благодаря этому изменения реже вызывают непредвиденные побочные эффекты, а код становится проще тестировать и развивать по модульному принципу.

С объектами ScriptableObject можно интерактивно работать в редакторе Unity, поэтому они особенно удобны для совместной работы программистов и специалистов без навыков программирования, например художников и дизайнеров. В этом руководстве мы рассмотрим несколько приемов эффективного применения ScriptableObject. Надеемся, они дополнят ваш рабочий процесс и упростят настройку проекта.

Познакомимся с незаметным героем игровой архитектуры - скромным ScriptableObject.

Что такое ScriptableObject?

ScriptableObject - это объект Unity, который не является частью экземпляра GameObject. На его основе можно создать пользовательский класс с собственными переменными и методами, но с меньшими накладными расходами, чем у MonoBehaviour.

У ScriptableObject нет Transform, и он находится вне иерархии сцены. Такой объект существует на уровне проекта в виде ассета, подобно материалу или 3D-модели.

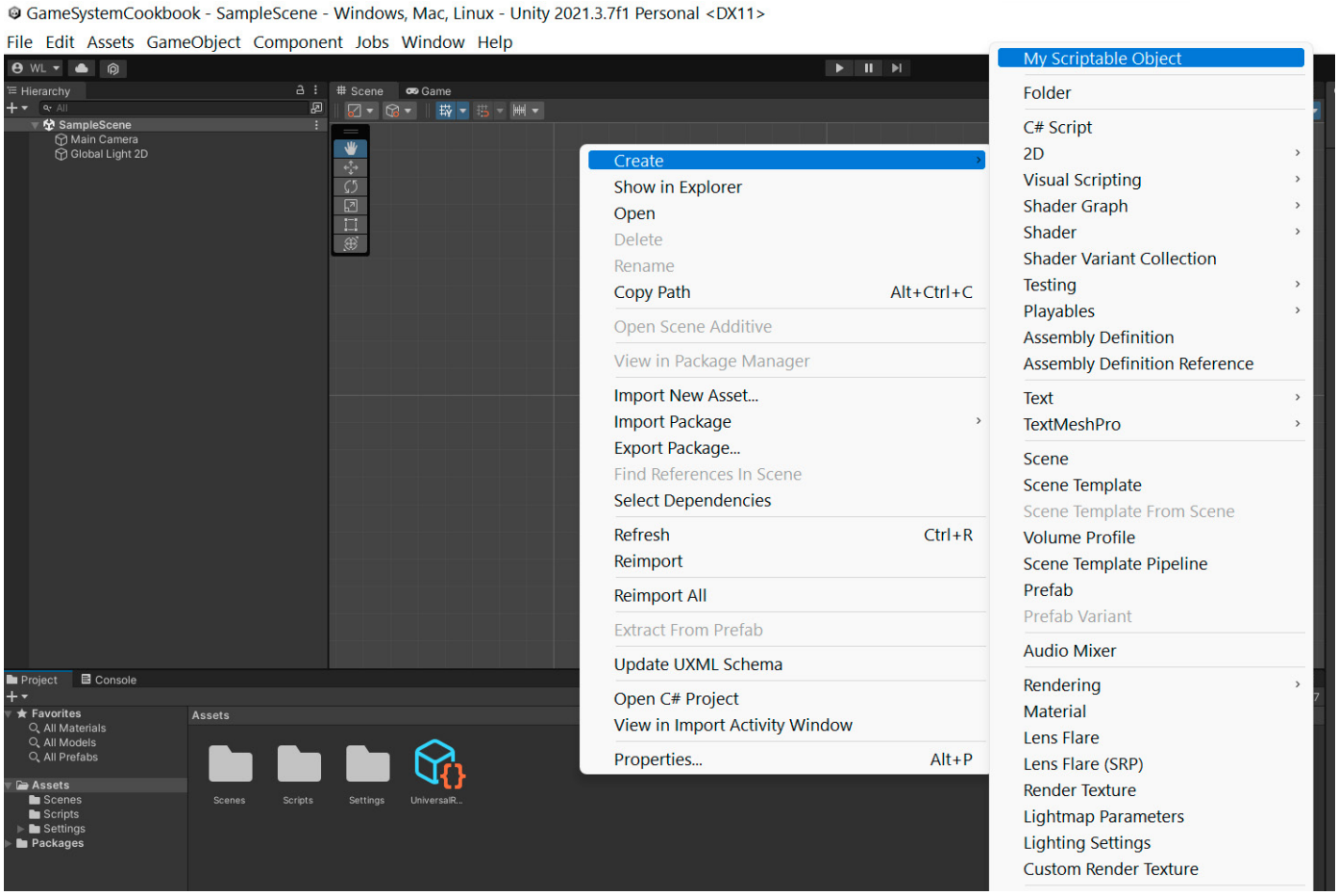
Объявить ScriptableObject можно следующим образом:

```
[CreateAssetMenu(fileName="MyScriptableObject")]
public class MyScriptableObject: ScriptableObject
{
	public int someVariable;
}
```

Как видно из приведенного кода, класс наследуется не от MonoBehaviour, а от ScriptableObject.

ScriptableObject нельзя напрямую прикрепить к GameObject. Вместо этого атрибут CreateAssetMenu добавляет соответствующую команду в меню.

Откройте Assets > Create > MyScriptableObject или щелкните правой кнопкой мыши в окне Project, чтобы создать пользовательский ассет на основе класса ScriptableObject.



Создание ScriptableObject

После этого любой другой скрипт сможет ссылаться на данный ассет ScriptableObject из любой сцены с помощью поля:

```
public class MyMonoBehaviour : MonoBehaviour
{
    public ScriptableObject soInstance;
}
```

Объекты ScriptableObject особенно полезны для данных, которые не требуется изменять во время выполнения.

Объекты ScriptableObject являются полноправными объектами Unity, поэтому их можно:

- Хранить в переменных

- Динамически создавать и уничтожать во время выполнения

- Передавать в качестве аргументов

- Возвращать из методов

- Включать в структуры данных

- Сериализовать и десериализовать

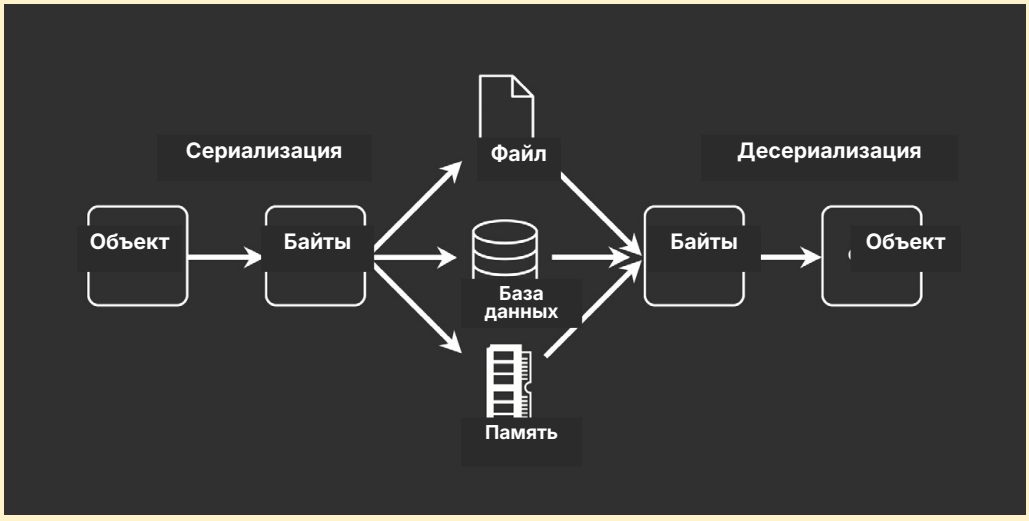
Последний пункт отражает одну из ключевых особенностей объектов ScriptableObject: они могут отображаться в окне Inspector. Благодаря этому их поля удобно просматривать и изменять в редакторе Unity.

Если изменить значения объекта ScriptableObject во время выполнения, игра обновится сразу. После выхода из режима Play эти значения сохранятся. Это удобно для гейм-дизайнеров, которым нужно настраивать баланс без написания кода. Кроме того, объекты ScriptableObject часто сохраняют изменения во время работы приложения, что дает им ряд преимуществ по сравнению с использованием только MonoBehaviour.

Сериализация

Сериализация - это автоматическое преобразование структур данных или состояний объектов в формат, который удобно хранить и позднее восстанавливать. Система сериализации Unity собирает разрозненные в памяти данные и располагает их последовательно.

Полученный поток данных можно сохранить в базе данных, файле или памяти. Десериализация выполняет обратное преобразование.



Сериализация преобразует объект в поток байтов, чтобы сохранить его либо передать в память, базу данных или файл.

При разработке на C# легко не задумываться об организации памяти, однако важно знать, какие встроенные возможности Unity используют сериализацию:

- Сохранение и загрузка: если открыть файл сцены .unity в текстовом редакторе и включить принудительную текстовую сериализацию, сериализатор будет использовать YAML в качестве внутреннего формата.

- Окно Inspector: этот интерфейс не обращается к API C#, чтобы получить значения проверяемого объекта. Вместо этого он просит объект сериализовать себя и отображает полученные данные.

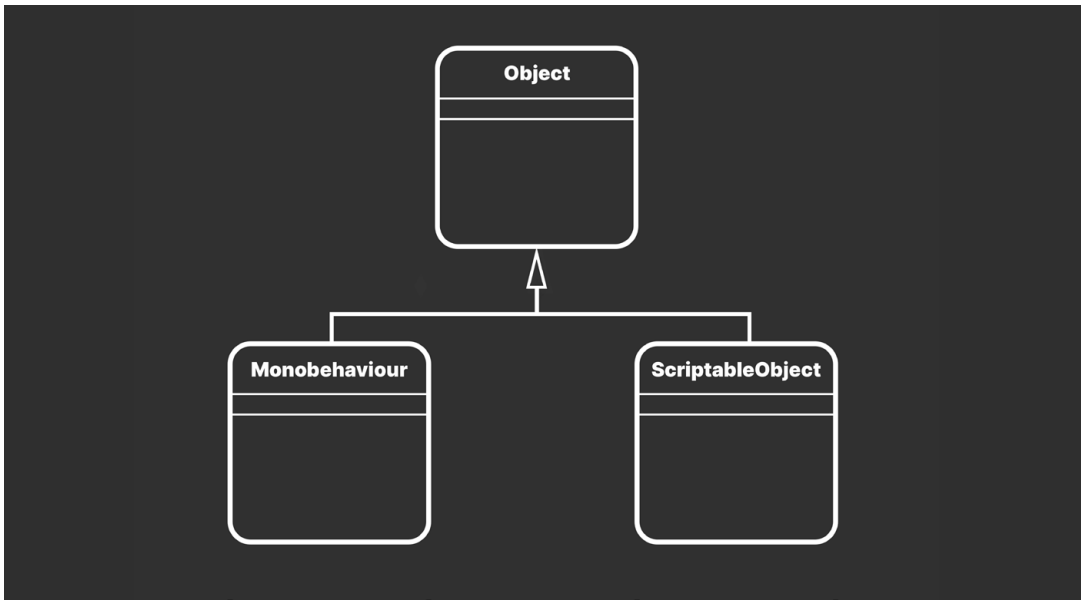
- Префабы: внутри префаб представляет собой сериализованный поток данных объектов GameObject и компонентов. Экземпляр префаба хранит список изменений, которые следует применить поверх сериализованных данных.

- Создание экземпляра: при создании экземпляра префаба или объекта GameObject на сцене Unity сериализует исходный объект, создает новый, а затем десериализует данные в новый объект.

Подробнее о сериализации в Unity см. в этой статье блога и видео «Как работает система сериализации Unity».

ScriptableObject и MonoBehaviour

На первый взгляд объекты ScriptableObject устроены просто: их API содержит всего несколько методов. В данном случае это преимущество, поскольку чем проще система, тем меньше вероятность ошибок.



UML-диаграмма объектов Unity

Как и MonoBehaviour, класс ScriptableObject наследуется от UnityEngine.Object.

Сравнение

Лучше всего понять ScriptableObject в сравнении с родственным ему классом MonoBehaviour. В таблице показаны их сходства и различия.

MonoBehaviour	ScriptableObject
И MonoBehaviour, и ScriptableObject являются типами скриптов. Классы MonoBehaviour и ScriptableObject наследуются от UnityEngine.Object.	
Компоненты MonoBehaviour получают обратные вызовы от Unity. Чтобы связать методы с циклом PlayerLoop игрового движка, назовите их в соответствии с функциями событий MonoBehaviour, например Update(). Например: Start, Awake, Update, OnEnable, OnDisable, OnCollisionEnter	**Объекты ScriptableObject не получают большинство обратных вызовов жизненного цикла Unity, таких как Update, Start и FixedUpdate.** Во время выполнения объекты ScriptableObject поддерживают ограниченный набор функций событий: Awake, OnEnable, OnDestroy и OnDisable. Редактор Unity также вызывает OnValidate и Reset при работе с окном Inspector. В ScriptableObject можно определять и другие методы, но PlayerLoop не вызывает их автоматически.
Во время выполнения компоненты MonoBehaviour должны быть прикреплены к объектам GameObject. Для создания MonoBehaviour во время выполнения используйте API AddComponent.	**Объекты ScriptableObject не привязаны к конкретному GameObject.** Сохраняйте объекты ScriptableObject в отдельных файлах ассетов на уровне проекта, а затем обращайтесь к ассету ScriptableObject из компонента MonoBehaviour или другого скрипта.
Данные MonoBehaviour сохраняются внутри сцен и префабов.	**Каждый экземпляр ScriptableObject можно сохранить в отдельном файле ассета на уровне проекта.**
В редакторе Unity изменения значений MonoBehaviour сбрасываются после выхода из режима Play.	**В редакторе Unity изменения значений ScriptableObject не сбрасываются после выхода из режима Play.** **В автономной сборке изменения значений ScriptableObject во время выполнения не сохраняются.**
MonoBehaviour и ScriptableObject поддерживают сериализацию и отображаются в окне Inspector.	

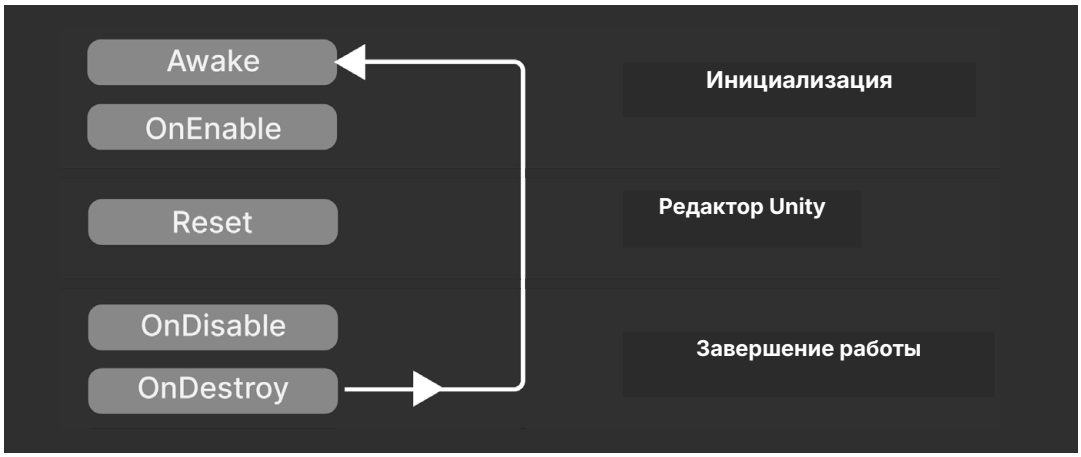
Обратные вызовы и сообщения

Объекты ScriptableObject поддерживают лишь часть функций событий, доступных MonoBehaviour. В них можно создавать собственные методы, но вызывать эти методы необходимо самостоятельно. В таблице ниже приведены только методы, которые PlayerLoop вызывает автоматически.

Функция события (во время выполнения)	**Когда вызывается**
Awake	Вызывается при запуске скрипта ScriptableObject, аналогично обратному вызову Awake у MonoBehaviour. Awake также выполняется при запуске игры или загрузке сцены, содержащей ссылку на ассет ScriptableObject.
OnEnable	Вызывается сразу после Awake при загрузке или создании экземпляра ScriptableObject. OnEnable выполняется при вызове ScriptableObject.CreateInstance или после успешной повторной компиляции скриптов.
OnDisable	Вызывается, когда ScriptableObject выходит из области использования: например, при загрузке сцены без ссылок на его ассет или непосредственно перед OnDestroy этого ScriptableObject. Unity также выполняет OnDisable перед повторной компиляцией скриптов. При входе в режим Play метод OnDisable вызывается непосредственно перед OnEnable.
OnDestroy	Вызывается при уничтожении ScriptableObject, например при его удалении в редакторе Unity или из кода. Если ScriptableObject создан во время выполнения, OnDestroy также вызывается при завершении приложения или выходе редактора Unity из режима Play. Примечание: уничтожается только нативная C++-часть объекта. Подробнее см. раздел «Создание и жизненный цикл».
Функции только для редактора Unity	**Когда вызывается**
OnValidate	Выполняется при загрузке скрипта или изменении значения в окне Inspector. Метод можно использовать, чтобы удерживать данные в допустимом диапазоне.
Reset	Вызывается при выборе команды Reset в контекстном меню окна Inspector.

Чтобы уничтожить ScriptableObject, удалите его в редакторе Unity либо вызовите Destroy или DestroyImmediate во время выполнения.

Ниже кратко показаны функции событий и жизненный цикл ScriptableObject. Сравните их с порядком выполнения функций событий MonoBehaviour, двигаясь сверху вниз.



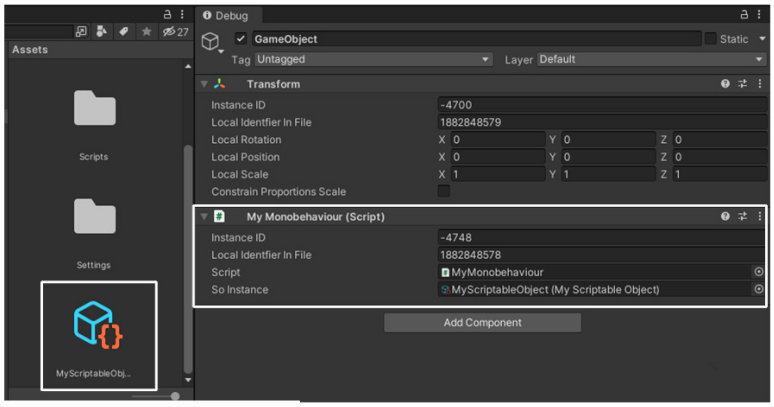
Функции событий ScriptableObject и порядок их выполнения

Файлы

Одно из главных различий между MonoBehaviour и ScriptableObject заключается в способе сохранения данных.

Unity сериализует компоненты MonoBehaviour внутри файлов сцен или префабов. Сохраняются следующие данные:

- Сам MonoBehaviour
- Связанный с ним GameObject
- Его Transform
- Остальные компоненты и MonoBehaviour на том же GameObject

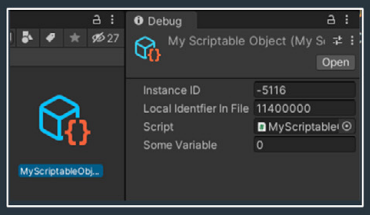


ScriptableObject и MonoBehaviour

В отличие от MonoBehaviour, Unity сохраняет объекты ScriptableObject в отдельных файлах ассетов. Такие файлы меньше, а данные в них лучше обособлены, чем в случае с MonoBehaviour.

Если в окне Project Settings > Asset Serialization выбрать Mode: Force Text, ассет ScriptableObject можно открыть в текстовом редакторе. Он будет выглядеть примерно так:

```
%YAML 1.1
%TAG !u! tag:unity3d.com,2011:
--- !u!114 &11400000
MonoBehaviour:
  m_ObjectHideFlags: 0
  m_CorrespondingSourceObject: {fileID: 0}
  m_PrefabInstance: {fileID: 0}
  m_PrefabAsset: {fileID: 0}
  m_GameObject: {fileID: 0}
  m_Enabled: 1
  m_EditorHideFlags: 0
  m_Script: {fileID: 11500000, guid: 3fc7a749f692b4f41923450f022d8428, type: 3}
  m_Name: MyScriptableObject
  m_EditorClassIdentifier:
  someVariable: 0

```



Экземпляр ScriptableObject, сериализованный в текстовом виде

YAML - не язык разметки

Unity использует высокопроизводительную библиотеку сериализации, реализующую подмножество спецификации YAML. Это легковесный и удобочитаемый язык, родственный XML и JSON.

В YAML данные организованы в иерархию вложенных элементов. У каждого объекта есть Class ID, File ID и тип объекта. Обратите внимание: для ScriptableObject используется тип «MonoBehaviour», а не отдельный собственный тип.

Заголовок объекта в YAML

Под каждым объектом перечислены его сериализованные свойства в виде пар «ключ - значение».

Подробнее см. статью «Разбираемся в языке сериализации YAML в Unity».

Создание и жизненный цикл

Жизненный цикл ScriptableObject похож на жизненный цикл любого другого ассета проекта, например материала или текстуры.

Как и в предыдущем примере, добавьте к скрипту атрибут [CreateAssetMenu], чтобы создать пользовательскую команду меню в редакторе Unity. При необходимости можно задать имя файла по умолчанию через fileName и порядок пункта меню через order. Ниже показан наиболее распространенный способ создания ассета ScriptableObject.

```
[CreateAssetMenu(fileName="MyScriptableObject"]
public class MyScriptableObject: ScriptableObject
{
	public int SomeVar;
}
```

Чтобы создать экземпляр ScriptableObject во время выполнения, вызовите статический метод CreateInstance:

```
ScriptableObject.CreateInstance<MyScriptableObjectClass>();
```

Уничтожение ScriptableObject

Как и другие объекты Unity, ScriptableObject состоит из нативной C++-части и управляемой C#-части. Нативную часть можно уничтожить напрямую, но управляемая останется до очистки сборщиком мусора (GC). Такая очистка выполняется при смене сцены или вызове Resources.UnloadUnusedAssets.

У ScriptableObject есть нативная и управляемая части.

Чтобы не задерживать сборку мусора, явно присвойте null всем ссылкам на ассет ScriptableObject.

Примечание: ссылки важно обнулить до вызова Destroy или DestroyImmediate. Иначе ссылка в редакторе Unity может отображаться как null, хотя управляемый объект все еще существует. GC очистит его только после удаления всех ссылок на ScriptableObject.

Освоив создание и уничтожение собственных ScriptableObject, можно перейти к более творческим способам их применения в игре.

Контейнеры данных

Чаще всего объекты ScriptableObject используют как контейнеры общих статических данных, особенно конфигурации игры, которая не изменяется во время выполнения.

Типичные варианты применения объектов ScriptableObject:

- Инвентарь: тип предмета, значок, редкость, эффекты
- Характеристики врагов, игроков и предметов: здоровье, урон, скорость, параметры ИИ
- Наборы аудио: группы клипов для шагов, звуков UI и зацикленных фоновых звуков
- Конфигурационные файлы: настройки сложности, кривые прогресса, таблицы появления объектов
- Данные диалогов

Во время выполнения эти данные можно хранить в компонентах MonoBehaviour, но это не всегда эффективно. Как показало сравнение выше, компоненты MonoBehaviour создают дополнительные накладные расходы, поскольку им нужен объект GameObject и, по умолчанию, Transform. В результате ради хранения одного значения приходится создавать большой объем неиспользуемых данных.

Чтобы убедиться в этом, создайте новый GameObject с пустым MonoBehaviour. Затем откройте сериализованный объект в текстовом редакторе. Он будет выглядеть примерно так:

```
GameObject:
  m_ObjectHideFlags: 0
  m_CorrespondingSourceObject: {fileID: 0}
  m_PrefabInstance: {fileID: 0}
  m_PrefabAsset: {fileID: 0}
  serializedVersion: 6
  m_Component:
  - component: {fileID: 7467558130563611466}
  - component: {fileID: 2006805663113952451}
  m_Layer: 0
  m_Name: GameObject
  m_TagString: Untagged
  m_Icon: {fileID: 0}
  m_NavMeshLayer: 0
  m_StaticEditorFlags: 0
  m_IsActive: 1
--- !u!4 &7467558130563611466
Transform:
  m_ObjectHideFlags: 0
  m_CorrespondingSourceObject: {fileID: 0}
  m_PrefabInstance: {fileID: 0}
  m_PrefabAsset: {fileID: 0}
  m_GameObject: {fileID: 338842853706088653}
  m_LocalRotation: {x: 0, y: 0, z: 0, w: 1}
  m_LocalPosition: {x: -1.1780801, y: -2.6573246, z: -0.01486098}
  m_LocalScale: {x: 1, y: 1, z: 1}
  m_ConstrainProportionsScale: 0
  m_Children: []
  m_Father: {fileID: 0}
  m_RootOrder: 0
  m_LocalEulerAnglesHint: {x: 0, y: 0, z: 0}
--- !u!114 &2006805663113952451
MonoBehaviour:
  m_ObjectHideFlags: 0
  m_CorrespondingSourceObject: {fileID: 0}
  m_PrefabInstance: {fileID: 0}
  m_PrefabAsset: {fileID: 0}
  m_GameObject: {fileID: 338842853706088653}
  m_Enabled: 1
  m_EditorHideFlags: 0
  m_Script: {fileID: 11500000, guid: 4d1457d233f8d479795a30f859537d5f,
  m_Name:
  m_EditorClassIdentifier:
```

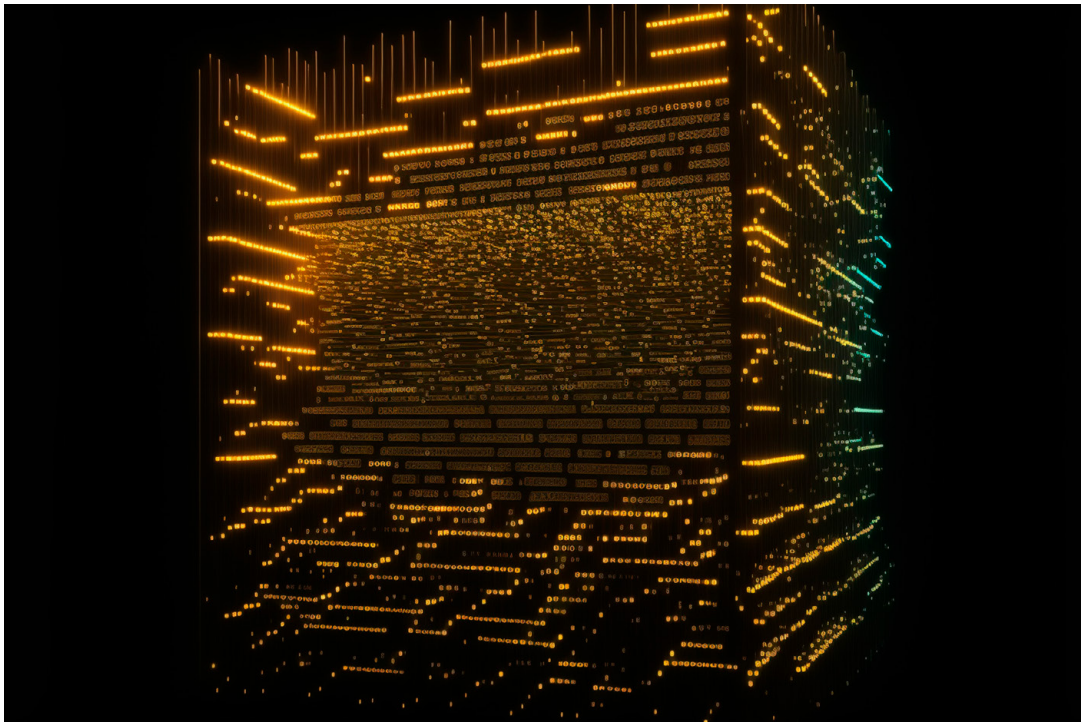
Новый GameObject с минимальным MonoBehaviour

Сравните его с пустым ScriptableObject: структура такого объекта заметно компактнее.

```
MonoBehaviour:
  m_ObjectHideFlags: 0
  m_CorrespondingSourceObject: {fileID: 0}
  m_PrefabInstance: {fileID: 0}
  m_PrefabAsset: {fileID: 0}
  m_GameObject: {fileID: 0}
  m_Enabled: 1
  m_EditorHideFlags: 0
  m_Script: {fileID: 11500000, guid: a2d85be56c1ae45d69604547c4544ba5,
  m_Name: PlayerID
  m_EditorClassIdentifier:
```

ScriptableObject сокращает накладные расходы на хранение данных.

ScriptableObject уменьшает объем занимаемой памяти, поскольку обходится без GameObject и Transform. В крупных коммерческих проектах это может иметь существенное значение. Кроме того, данные хранятся на уровне проекта, что особенно удобно, если к ним требуется обращаться из нескольких сцен.



Поначалу дополнительные данные MonoBehaviour могут почти не влиять на производительность приложения, но по мере роста игры до коммерческого масштаба и увеличения числа объектов влияние станет заметным.

Данные ScriptableObject и постоянные данные

Когда ScriptableObject называют контейнерами данных, обычно имеют в виду статическую или общую конфигурацию, которую не требуется сохранять после изменений во время выполнения.

Изменения данных ScriptableObject сохраняются в редакторе Unity, подобно изменениям материала или префаба, но во время выполнения собранного приложения они не записываются. Изменения экземпляра ScriptableObject в ходе игры существуют только в памяти и теряются после завершения сеанса.

Постоянные данные, которые нужно сохранять между сеансами, обычно записывают в другом формате, например JSON, XML, MessagePack или Protocol Buffers. Подробнее см. раздел «Двойная сериализация» ниже.

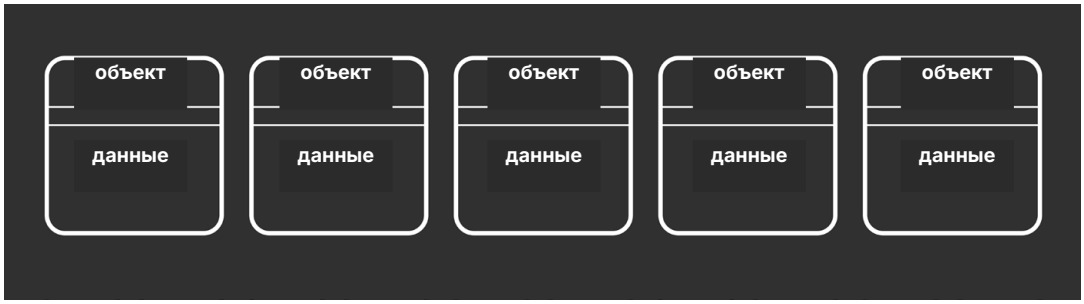
В сборке игры данные ScriptableObject можно изменять во время выполнения, например с помощью переменных ScriptableObject и Runtime Set, но такие изменения временны. При запуске нового сеанса данные вернутся к состоянию, которое было зафиксировано во время сборки.

Для задач постоянного хранения воспринимайте данные ScriptableObject как доступные только для чтения. Изменяемые постоянные данные следует хранить во внешнем хранилище с поддержкой чтения и записи.

Устранение дублирования данных

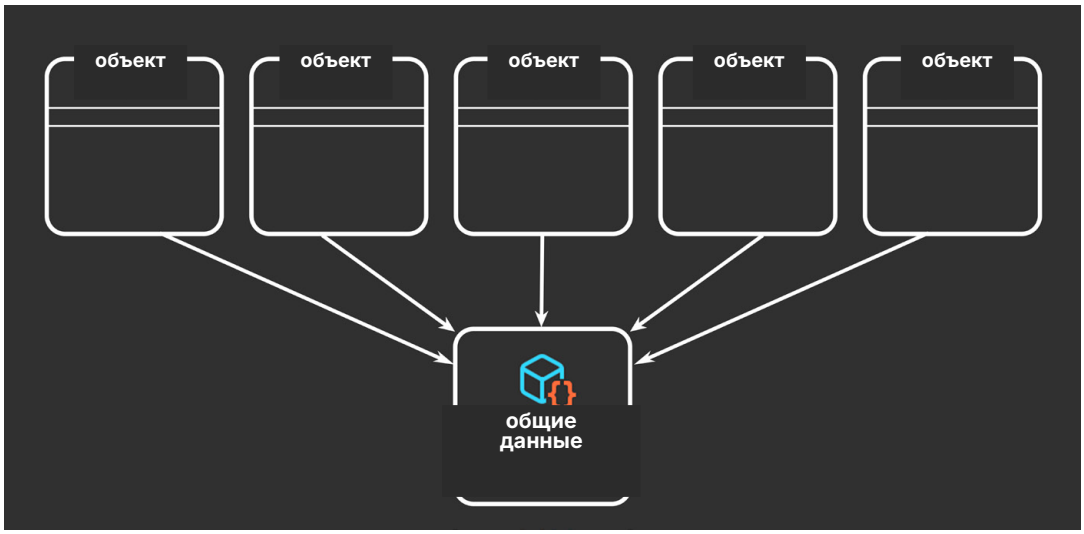
Представьте тысячу объектов GameObject, каждый с пользовательским компонентом MonoBehaviour, содержащим несколько полей.

Если каждый компонент хранит собственную копию одних и тех же значений, в памяти дублируется большой объем данных. Это неэффективно, особенно когда данные одинаковы для всех экземпляров и не меняются во время выполнения.



Множество объектов с одинаковыми локальными данными приводит к лишнему расходу памяти и снижению производительности.

Вместо дублирования статических данных поместите их в ScriptableObject. Тогда каждый из тысячи объектов сможет ссылаться на общий ассет с данными. Объекты будут хранить только ссылку, а не отдельную копию самих данных.



Совместное использование данных
множеством объектов через ScriptableObject

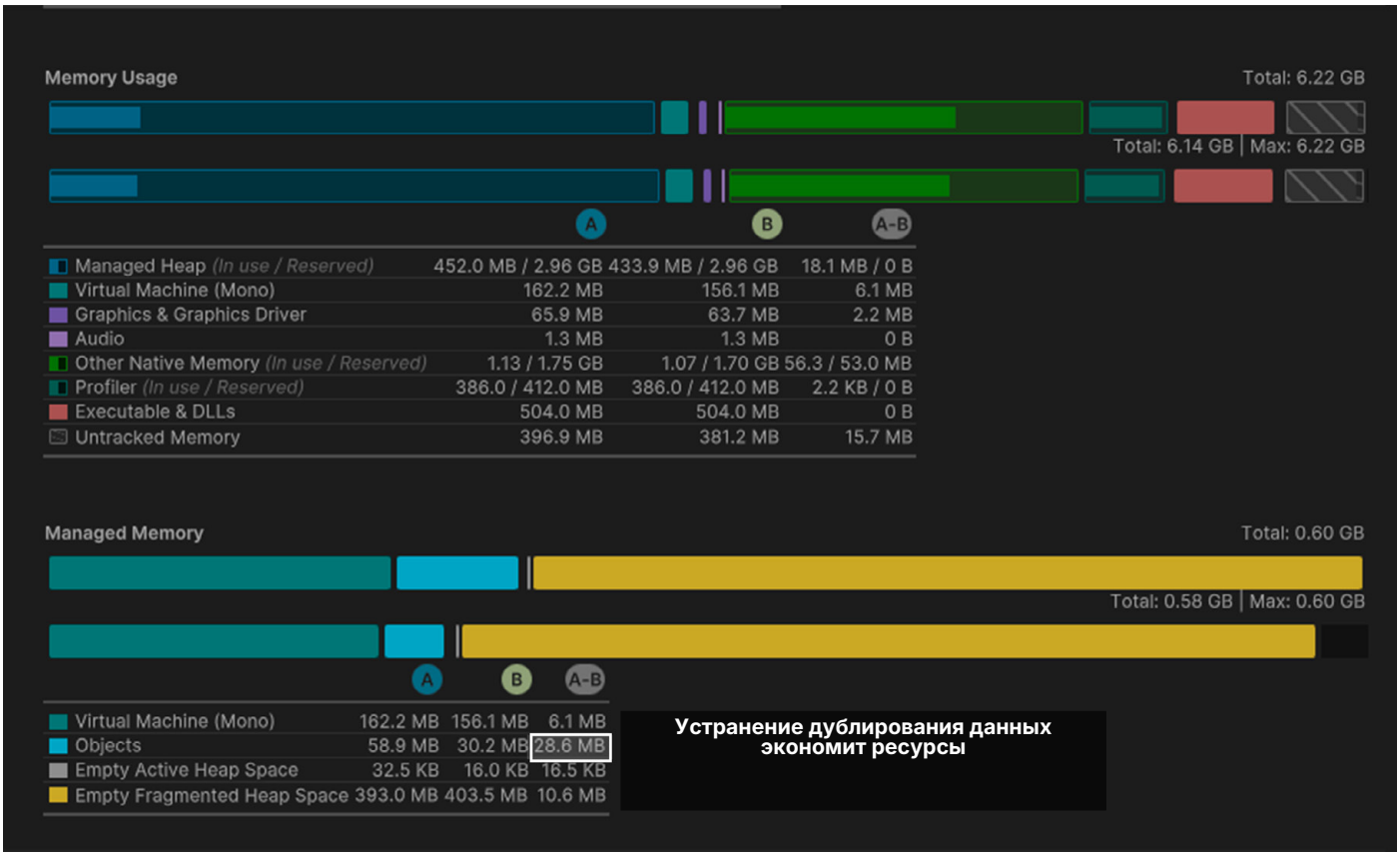
В проектировании программного обеспечения такая оптимизация известна как паттерн Flyweight, или «Приспособленец». Подобная реструктуризация кода устраняет множество копий одних и тех же значений и сокращает расход памяти.

Паттерны проектирования

Паттерны проектирования помогают создавать более гибкий и удобный для сопровождения код, что особенно важно в постоянно меняющихся игровых проектах.

Скачайте бесплатную электронную книгу «Повышаем уровень кода с паттернами проектирования и SOLID», чтобы подробнее узнать о принципах SOLID и паттернах проектирования.

Ссылка на ScriptableObject занимает гораздо меньше места, чем полная копия данных. По мере роста проекта экономия памяти благодаря устранению дубликатов может стать значительной.



Memory Profiler сравнивает расход памяти при дублировании данных (A) и совместном использовании данных (B).

Таким способом можно хранить большие объемы общих данных. Используйте ScriptableObject для следующих задач:

- Сохранение данных в течение сеанса работы в редакторе Unity

- Сохранение данных в виде ассета для использования во время выполнения

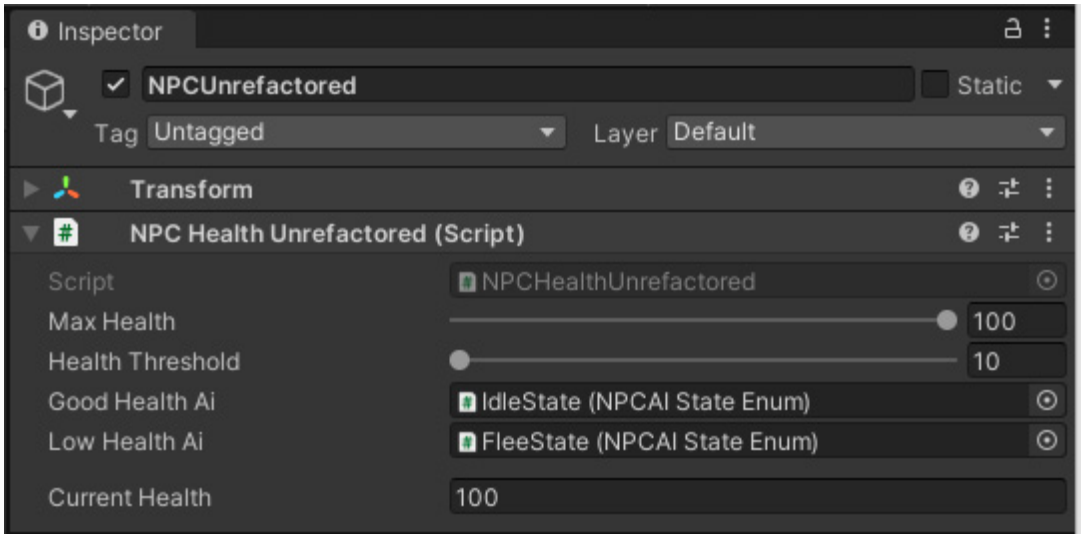
В отличие от компонентов MonoBehaviour, объекты ScriptableObject нельзя прикреплять к GameObject. Вместо этого их сохраняют как ассеты проекта. Это особенно удобно, если префаб использует неизменяемые данные в своих компонентах MonoBehaviour.

Пример рефакторинга

Рассмотрим компонент MonoBehaviour, который управляет здоровьем NPC. Его класс можно определить следующим образом:

```
public class NPCHealthUnrefactored : MonoBehaviour
{
    [Range(10, 100)]
    public int MaxHealth;
    [Range(10, 100)]
    public int HealthThreshold;


    public int CurrentHealth;
}
```

Такой код работает, но содержит данные, которые не должны меняться во время выполнения. Если компонент NPCHealthUnrefactored прикреплен ко множеству объектов, в памяти возникает большое количество ненужных копий.



MonoBehaviour NPCHealth до рефакторинга

Данные, которые не требуется изменять, можно перенести в ScriptableObject:

```
[CreateAssetMenu(fileName="NPCConfig")]
public class NPCConfigSO : ScriptableObject
{
    [Range(10, 100)]
    public int maxHealth;

    [Range(10, 100)]
    public int healthThreshold;


}
```

Атрибут CreateAssetMenu настраивает команду меню. При необходимости в нем можно указать имя файла по умолчанию через fileName и порядок пункта меню через order.

Соглашения по оформлению кода в этом руководстве

Для наглядности и простоты чтения многие примеры кода в этом руководстве упрощены, в частности используются открытые поля.

В рабочем коде применяйте закрытые поля и открытые свойства, чтобы улучшить инкапсуляцию и гибкость. Атрибут SerializeField позволяет отображать закрытые поля в окне Inspector редактора Unity.

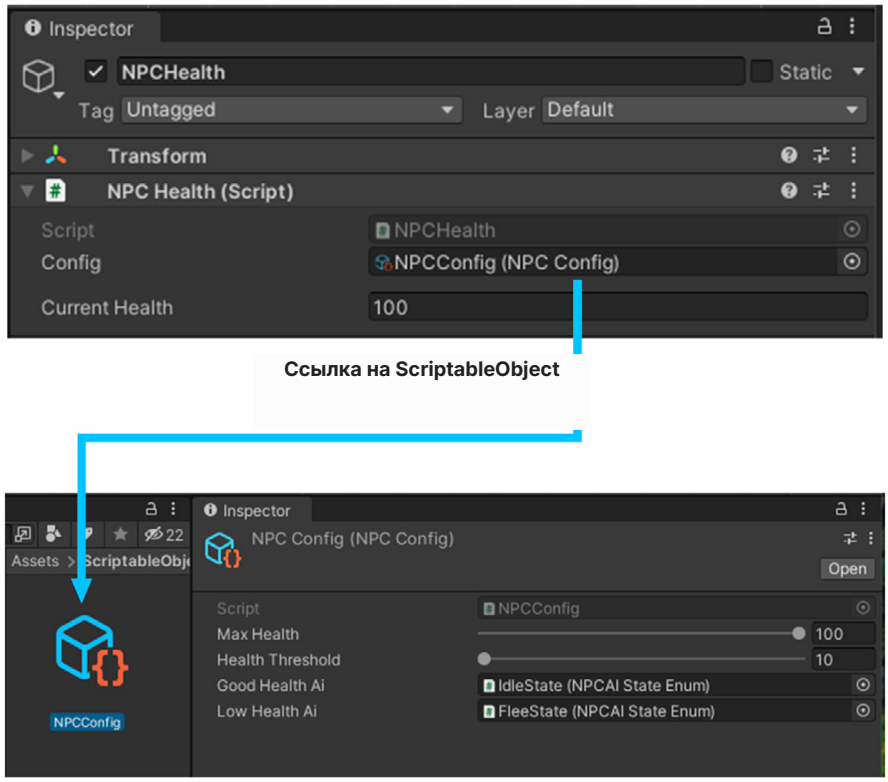
Соглашение об именовании также помогает отличать скрипты ScriptableObject от скриптов MonoBehaviour. Например, к имени класса можно добавлять суффикс Data или SO. Это необязательно, но позволяет лучше организовать проект и избежать неоднозначности.

По мере роста кодовой базы рекомендуется создать руководство по стилю и последовательно ему следовать. Подробнее см. электронную книгу «Руководство по стилю C# для Unity 6».

После рефакторинга компонент NPCHealth упрощается до следующего вида:

```
public class NPCHealth: MonoBehaviour
{
    // Ссылка на наш ScriptableObject
    public NPCConfigSO Config;

    public int CurrentHealth;
}
```

Теперь MonoBehaviour содержит ссылку на новый ScriptableObject. После рефакторинга содержимое окна Inspector выглядит почти так же, но данные разделены между объектами.

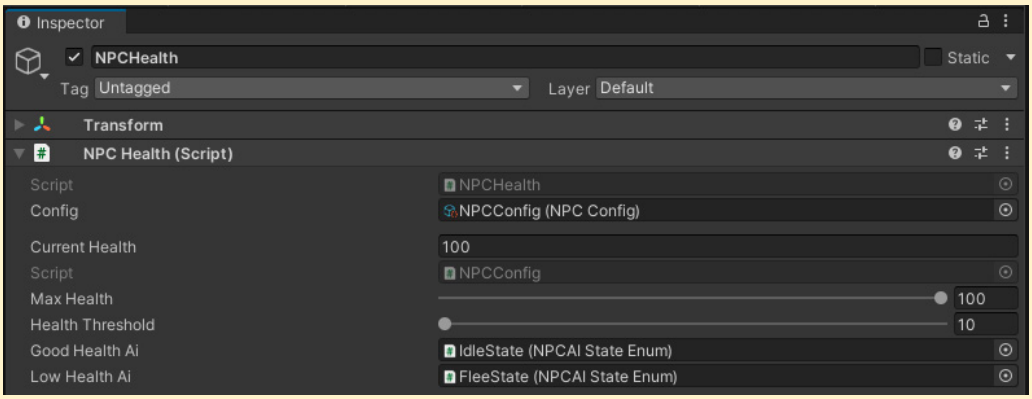


Рефакторинг разделяет данные между MonoBehaviour и ScriptableObject

Пользовательские окна Inspector

При переносе данных в ScriptableObject они оказываются в двух местах: в ассете ScriptableObject и в MonoBehaviour, который на него ссылается.

Чтобы упростить работу с компонентами MonoBehaviour, можно создать пользовательский редактор Unity. Ниже приведен пример для NPCHealth.



Пользовательское окно Inspector отображает переменные ScriptableObject внутри MonoBehaviour.

Так можно просматривать переменные NPCConfig вместе с остальными свойствами MonoBehaviour. Если выбрать исходный префаб NPCHealth, значения обоих объектов легко изменить в одном месте.

Для пользовательского редактора Unity достаточно нескольких строк кода:

- Создайте класс, производный от Editor, и сохраните его в папке Editor. Примените атрибут CustomEditor, указав тип NPCHealth.

- Подготовьте временный экземпляр Editor для ScriptableObject NPCConfig.

- В методе OnInspectorGUI создайте Editor для компонента NPCHealth.

- Отрисуйте содержимое окна Inspector базового класса и нового пользовательского окна Inspector.

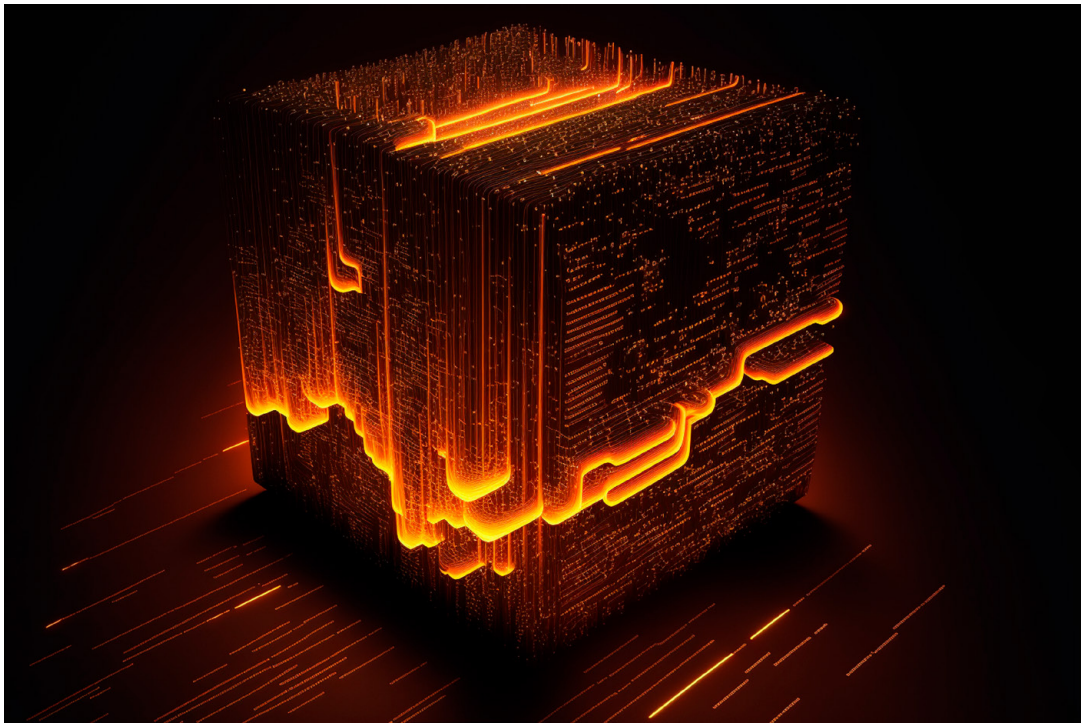
```
using UnityEditor;

[CustomEditor(typeof(NPCHealth))]
public class NPCHealthEditor : Editor
{
    private Editor editorInstance;

    private void OnEnable()
    {
        // Сбросить экземпляр Editor
        editorInstance = null;
    }

    public override void OnInspectorGUI()
    {
        // Проверяемый целевой компонент
        NPCHealth npcHealth = (NPCHealth)target;

        if (editorInstance == null)
            editorInstance = Editor.CreateEditor(npcHealth.config);

        // Показать переменные MonoBehaviour
        base.OnInspectorGUI();
```

```
        // Отрисовать окно Inspector для ScriptableObject
        editorInstance.DrawDefaultInspector();
    }
}
```

Этот пример можно развить с помощью пользовательских отрисовщиков свойств и атрибутов редактора. Это позволит сделать работу с объектами ScriptableObject еще удобнее.

Архитектурные преимущества

ScriptableObject позволяет четко разделить общие и индивидуальные данные. Все уникальные динамические данные конкретного экземпляра GameObject остаются в MonoBehaviour, а общие данные хранятся в ScriptableObject. Однако преимущества архитектуры на основе ScriptableObject не ограничиваются экономией памяти.



Такая реорганизация архитектуры кода дает несколько преимуществ:

- Дизайнеры могут меньше зависеть от разработчиков: если данные и логика хранятся в одном MonoBehaviour, разработчики и гейм-дизайнеры рискуют мешать работе друг друга. Когда два человека изменяют разные части одного префаба или сцены, возникает конфликт слияния, на устранение которого уходит время. Вынос общих данных в небольшие отдельные файлы и ассеты снижает риск таких проблем. Архитектура на основе ScriptableObject также позволяет дизайнерам создавать игровой процесс, не обращаясь постоянно к программисту.

При совместной работе с данными заранее определите понятный процесс взаимодействия между командами. Четкие договоренности и грамотно установленные границы ответственности помогут избежать проблем. Также могут потребоваться дополнительные проверки ошибок или валидация данных, например атрибут Range или метод OnValidate для ограничения допустимых значений.

- Общие данные редактируются быстрее и с меньшим риском ошибок: теперь все изменения вносятся централизованно. Например, чтобы изменить параметр всех NPC, достаточно скорректировать его в одном месте, после чего новое значение применится ко всем затронутым компонентам во всех сценах.

Это уменьшает вероятность ошибок, возникающих при ручном массовом редактировании множества отдельных объектов GameObject. Перенос данных в ScriptableObject также упрощает контроль версий и помогает предотвращать конфликты слияния, когда участники команды работают с одной сценой или префабом.

- Сохраняйте изменения игрового процесса, сделанные в режиме Play: режим Play в редакторе Unity позволяет дизайнерам экспериментировать с игровым процессом и настройками. Однако изменения в MonoBehaviour теряются после выхода из режима Play, поскольку Unity удаляет временную копию сцены.

ScriptableObject является ассетом, поэтому изменения его значений сохраняются независимо от того, находится ли Unity в режиме Play. Это удобно, если параметры требуется настраивать во время выполнения.

Однако это может стать недостатком, если изменения понадобится отменить. Используйте Unity Version Control или другую систему контроля версий, чтобы при необходимости всегда можно было восстановить прежнее состояние проекта. Подробнее см. руководство «Рекомендации по контролю версий и организации проекта (издание для Unity 6)».

- Сокращайте время загрузки сцен: при сохранении сцены или префаба Unity сериализует все их содержимое, включая каждый GameObject, все прикрепленные к нему компоненты и каждое открытое поле. При этом Unity не проверяет данные на дублирование.

Перенос данных в ScriptableObject может уменьшить размер сцен и префабов и заметно ускорить их загрузку и сохранение.

Переменные ScriptableObject

Общие контейнеры данных можно сделать еще более специализированными: один ScriptableObject будет представлять всего одно значение. Например, можно создать класс ScriptableObject с именем IntVariable и единственным открытым полем value:

```
using UnityEngine;

[CreateAssetMenu(menuName = "Variables/Int", order = 1)]

public class IntVariableSO : ScriptableObject

{

    public int value;

}
```

Затем IntVariable можно использовать в MonoBehaviour. Например, класс PlayerHealth будет выглядеть так:

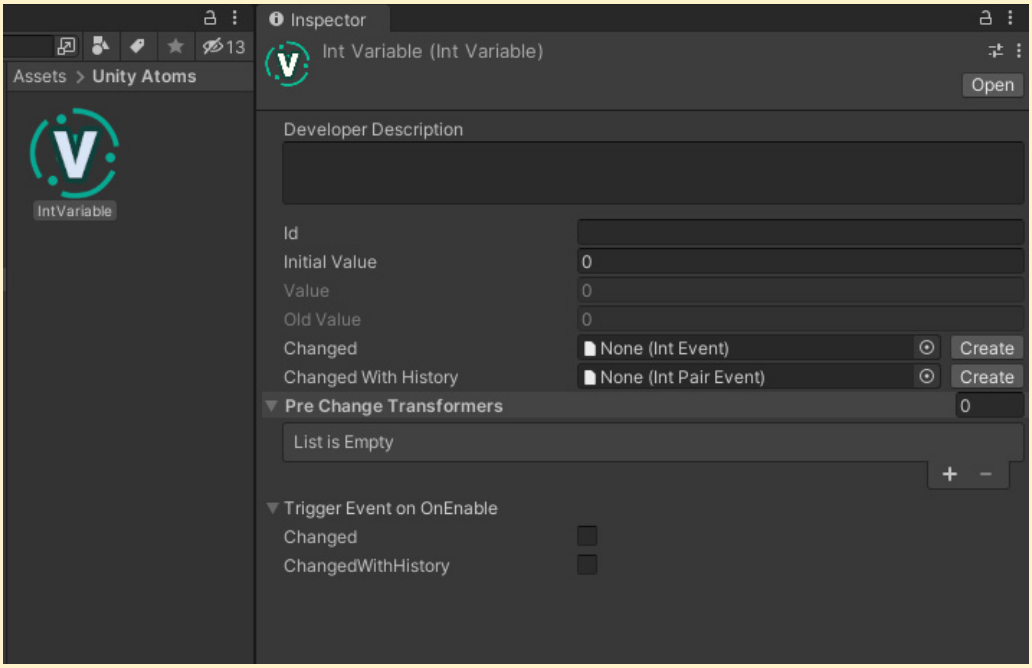
```
public class PlayerHealth : MonoBehaviour

{

    public IntVariableSO health;

}
```

Обычно ScriptableObject воспринимается как хранилище неизменяемых значений, однако в него можно добавить методы, которые обновляют данные во время выполнения и восстанавливают начальное значение после выхода из режима Play. Таким образом, ScriptableObject может фактически выполнять роль переменной и хранить целые числа, числа с плавающей точкой, логические значения и другие данные.

После этого дизайнеры смогут самостоятельно задавать данные для игровой логики, не обращаясь каждый раз к разработчику. Но такой подход требует предварительного планирования. Вместе с дизайнерами определите, как распределить ответственность за создание данных игрового процесса.

Главное - установить четкие границы совместной работы. Например, команда программистов может подготовить объекты ScriptableObject для системы инвентаря, а команда дизайнеров затем заполнит характеристики и поведение каждого игрового предмета.

Дополнительные скрипты редактора Unity могут сделать этот процесс почти бесшовным. Например, можно дать пользователю возможность переключать поля в окне Inspector между общим значением из ScriptableObject и константой. Тогда команда гейм-дизайна получит больше свободы и сможет переопределять данные ScriptableObject для отдельных экземпляров.



Пример IntVariable на основе ScriptableObject из проекта Unity Atoms с открытым исходным кодом

О реализации такого поведения в собственных проектах рассказывается в докладе «Игровая архитектура с ScriptableObject» с конференции Unite Austin. Также можно скачать проект Unity Atoms с открытым исходным кодом и изучить рабочую реализацию переменных ScriptableObject.

Двойная сериализация

В Unity можно сочетать разные способы сериализации. Например, работать с объектами ScriptableObject в редакторе Unity, а затем сохранять их данные в другом месте, например в файле JSON или XML. Такой подход позволяет использовать сильные стороны каждого формата.

Форматы JSON и XML подходят для постоянных данных, например сохранений игры или настроек. С ними не всегда удобно работать в редакторе Unity, зато их легко изменять вне Unity в любом текстовом редакторе.

ScriptableObject, напротив, удобен в редакторе Unity и легко заменяется обычным перетаскиванием. Однако изменять его вне Unity и распространять среди сообщества игроков сложнее.

Сочетание форматов сериализации открывает дополнительные возможности, например редактирование уровней и поддержку модификаций. Во время сборки скрипт может преобразовать внешние файлы в объекты ScriptableObject, которые загружаются быстрее обычного текста.

Конфиденциальные данные, например сведения о виртуальной валюте и учетных записях, следует надежно хранить на серверах. В то же время доступ сообщества к части игровых данных может обогатить игровой процесс и позволить пользователям экспериментировать с уровнями-песочницами.

Представьте ScriptableObject, который описывает структуру игрового уровня. Он может содержать набор объектов Transform, задающих расположение префабов, начальную конфигурацию и другие параметры. Игровые скрипты будут использовать эти данные для сборки каждого уровня.

Например, стены и начальные позиции игры можно хранить в ScriptableObject:

```
[CreateAssetMenu(fileName ="LevelLayout")]
public class LevelLayout : ScriptableObject
{
    public Vector3[] wallPositions = new Vector3[2];
    public Vector3[] playerPositions = new Vector3[2];
    public Vector3[] goalPositions = new Vector3[2];
    public Vector3 ballPosition;
}
```

Эти данные определяют компоновку уровня. Скрипты управления уровнем считывают сведения из объекта LevelLayout, а затем создают экземпляры префабов в нужных местах.

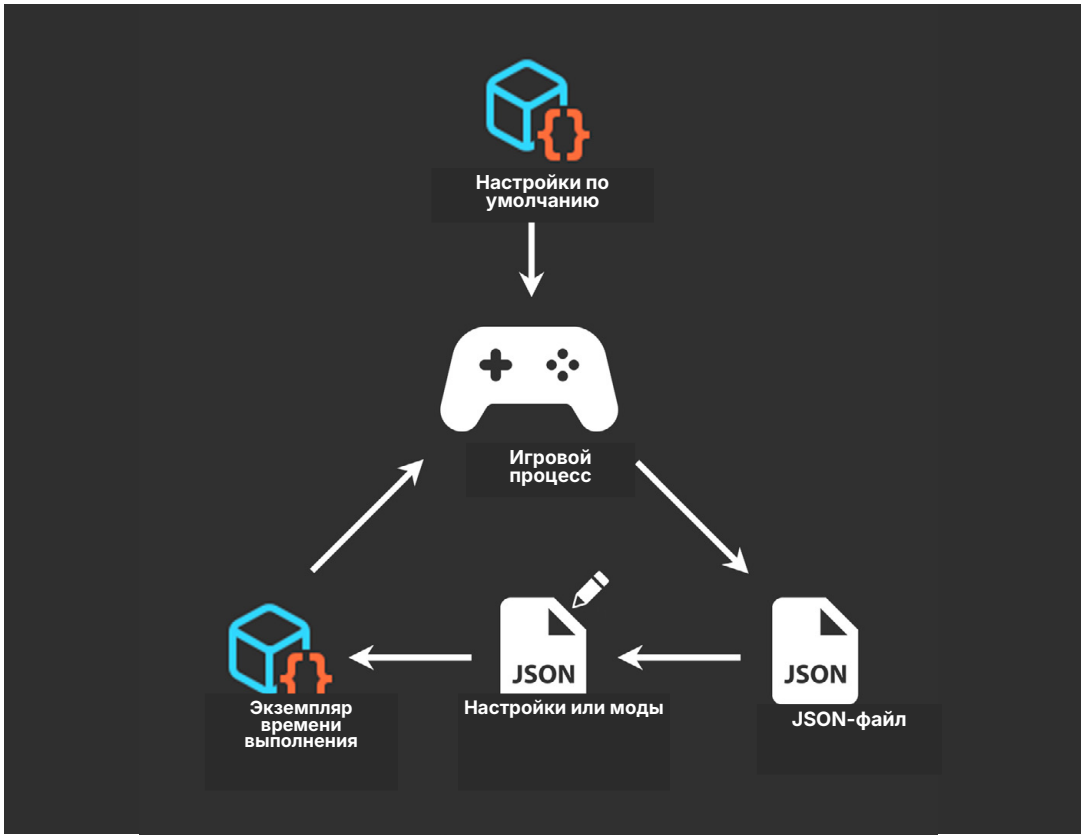
Пользовательский скрипт может экспортировать те же данные на диск с помощью JsonUtility. В результате за пределами редактора Unity создается текстовый файл, который пользователи смогут изменять сторонними инструментами.

Чтобы загрузить измененный пользователем уровень, метод ScriptableObject.CreateInstance может создать ScriptableObject во время выполнения. Затем данные из файла JSON считываются в этот объект. Ниже показан пример метода LoadLevelFromJson:

```
using System.IO;

public class LevelManager : MonoBehaviour
{
    public ScriptableObject levelLayout;

    public void LoadLevelFromJson(string jsonFile)
    {
        if (levelLayout == null)
        {
            levelLayout = ScriptableObject.
CreateInstance<LevelLayout>();
        }

        var importedFile = File.ReadAllText(jsonFile);
        JsonUtility.FromJsonOverwrite(importedFile, levelLayout);

    }
}
```

Пользовательские данные заменяют содержимое ScriptableObject, после чего измененный извне уровень можно использовать в игре как любой другой. Для самого приложения разницы нет.

Обязательно проверьте этот механизм в примере проекта. При загрузке измененного файла JSON пользовательский уровень переопределяет стандартные данные уровня в ScriptableObject.



Сочетайте форматы сериализации для большей гибкости

Примечание. При десериализации JSON в объекты ScriptableObject с помощью JsonUtility необходимо использовать метод FromJsonOverwrite.

JsonUtility не создает новый объект для загрузки данных JSON, а записывает их в уже существующий объект. Значения в классах и объектах обновляются без дополнительных выделений памяти.

Защита данных

Простой пример модификации выше показывает один из вариантов применения ScriptableObject. Однако, открывая игровые данные для изменений, соблюдайте осторожность, чтобы игроки не могли вмешаться в работу остальных частей приложения.

Распространенные способы защиты данных, которые не должны изменяться модификациями:

- Шифрование: зашифруйте файлы данных, чтобы их было сложнее прочитать или изменить. Это затруднит вмешательство пользователей в критически важные данные.

- Цифровые подписи: алгоритм вычисления цифрового отпечатка позволяет проверить, не были ли файлы данных подменены или изменены.

- Проверка на стороне сервера: если игра использует данные, хранящиеся на сервере, проверяйте их на сервере до передачи в игру и отклоняйте любые сведения с признаками вмешательства.

Ни один метод не обеспечивает абсолютной защиты, поэтому обычно стоит сочетать несколько подходов. Это поможет не допустить, чтобы изменения игроков привели к ошибкам или уязвимостям в игре.

Паттерн расширяемых перечислений

При разработке игр часто приходится решать повторяющиеся или похожие задачи. К счастью, можно опереться на коллективный опыт инженеров-программистов, которые уже сталкивались с такими задачами и воплотили свои знания в паттернах проектирования.

Паттерны проектирования предлагают универсальные решения для создания крупных масштабируемых приложений. Они повышают читаемость кода, делают кодовую базу чище и сокращают объем рефакторинга и время тестирования.

Паттерн проектирования можно рассматривать как шаблон для решения распространенных задач, например:

- эффективное хранение больших объемов данных;

- обмен данными между объектами из разных игровых систем;

- динамическая замена поведения во время выполнения.

ScriptableObject помогает реализовать некоторые из этих паттернов. Вы уже видели, как он работает в роли контейнера данных, но его возможности не ограничиваются хранением значений и настроек. В следующих разделах рассматриваются более сложные варианты применения ScriptableObject.

Категории, подобные enum

Чтобы приносить пользу, ScriptableObject вовсе не обязан содержать данные. Даже пустой ScriptableObject можно применять, например, для сравнения с другими объектами ScriptableObject.

Предположим, в игре создается несколько ассетов на основе пустого ScriptableObject GameItemSO:

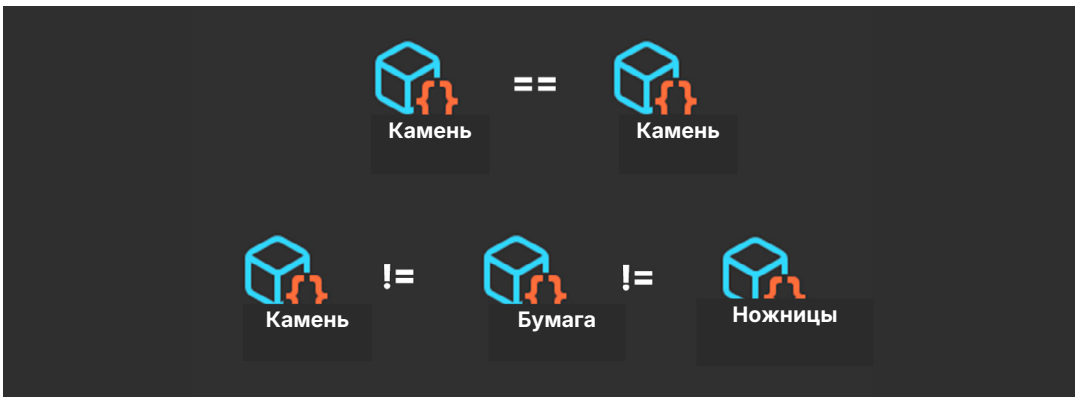
```
using UnityEngine;


[CreateAssetMenu(fileName="GameItem")]
public class GameItemSO : ScriptableObject
{


}
```

Пустые объекты ScriptableObject работают как значения enum.

Так можно создать в проекте любое количество ассетов. Даже если ScriptableObject не содержит данных, сам объект способен обозначать категорию или тип предмета подобно значению enum.

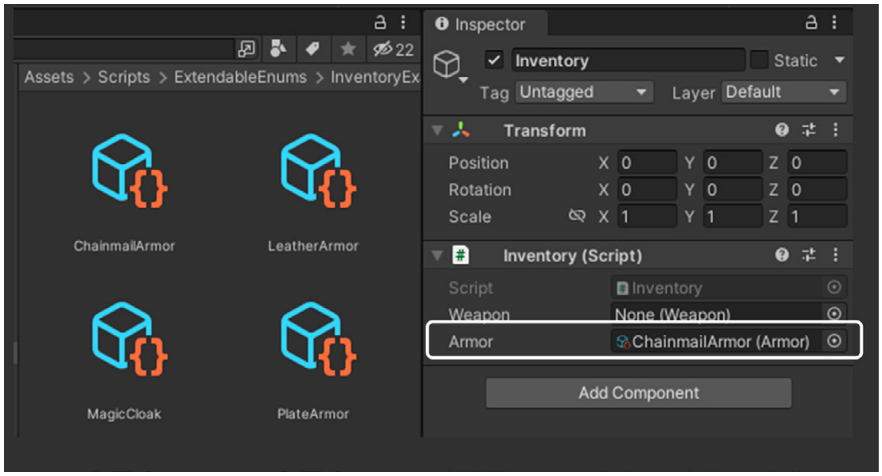
Если две переменные ссылаются на один ScriptableObject, они обозначают один и тот же тип предмета. Если ссылки различаются, различаются и типы.

Например, ScriptableObject может обозначать особый эффект урона, такой как холод, жар, электричество или магия, либо один из вариантов для игры с нулевой суммой «камень, ножницы, бумага».



Сравнение объектов ScriptableObject

Если приложению нужна система инвентаря для экипировки игровых предметов, объекты ScriptableObject могут представлять типы предметов или слоты оружия. Поля в окне Inspector при этом образуют интерфейс настройки с перетаскиванием элементов.



Ячейки инвентаря распределены по категориям ScriptableObject

Категории на основе ScriptableObject можно назначать перетаскиванием

Такой удобный для художников интерфейс позволяет дизайнерам изменять и расширять игровые данные без дополнительной помощи разработчика. Если команда дизайнеров получает инструменты и ответственность за поддержку этих данных, каждый участник проекта может сосредоточиться на своей специализации.

Расширение поведения

Применение ScriptableObject в роли enum становится еще интереснее, если расширить объект дополнительными данными. В отличие от обычного enum, ScriptableObject может содержать дополнительные поля и методы.

Ниже показана адаптированная версия GameItem для игры «камень, ножницы, бумага». Ассет ScriptableObject по-прежнему задает категорию, подобную enum, но теперь он уже не пуст.

```
public class GameItem : ScriptableObject
{
    public GameItem weakness;
    public bool IsWinner(GameItem other)
    {
        return other.weakness == this;
    }
}
```

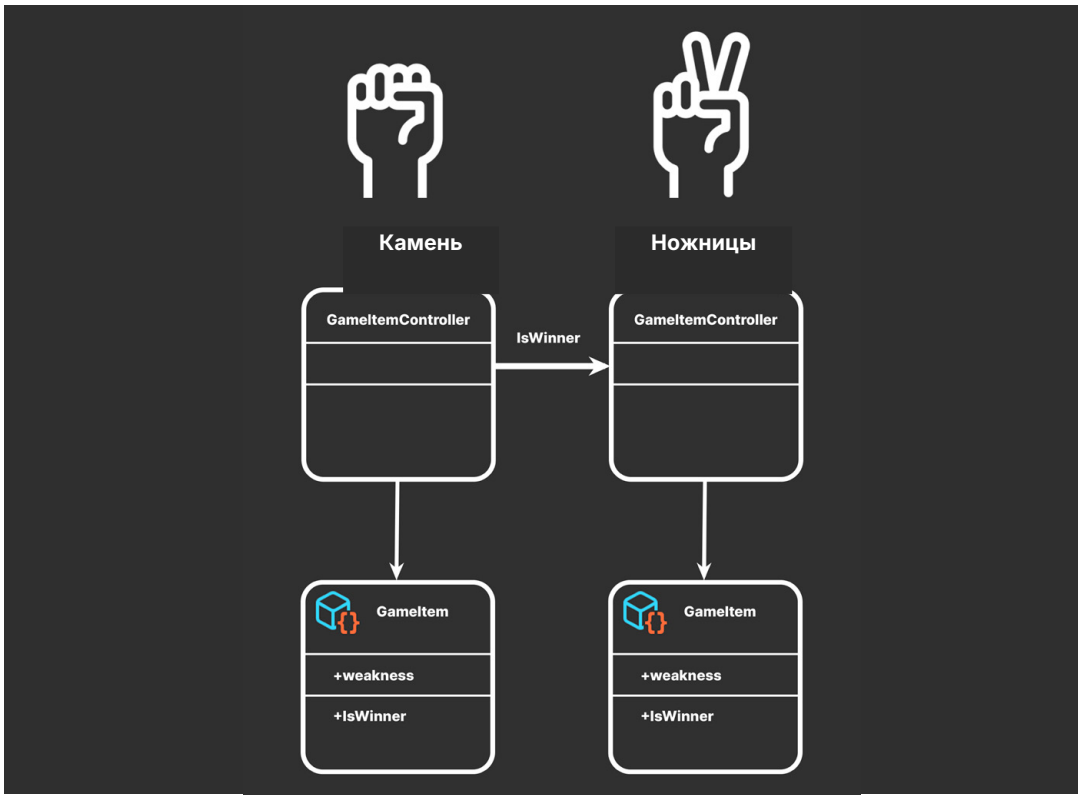
Теперь ScriptableObject содержит поле weakness, которое определяет, какой другой предмет победит при взаимодействии. Помимо данных, каждый ScriptableObject хранит простую логику сравнения в методе IsWinner.

Каждому игровому предмету требуется MonoBehaviour со ссылкой на определенный ассет ScriptableObject. В этом примере роль контроллера выполняет следующий скрипт:

```
public class GameItemController : MonoBehaviour
{
    // камень, ножницы, бумага
    public GameItem gameItem;

    private void OnTriggerEnter(Collider other)
    {
        GameItemController otherController =
            other.GetComponent<GameItemController>();
        GameItem otherGameItem = otherController.gameItem;

        if (gameItem.IsWinner(otherGameItem))
        {
            Debug.Log(gameItem.name + " побеждает " + otherGameItem.name);
        }
    }
}
```

Скрипт хранит ссылку на ScriptableObject в поле. В OnTriggerEnter метод IsWinner позволяет определить победителя, когда gameItem соприкасается с другим предметом. Так создается основа для столкновения в духе игры «камень, ножницы, бумага».

В отличие от enum, ScriptableObject легко расширять. Не нужно создавать отдельную таблицу поиска или связывать перечисление с новым массивом данных. Достаточно добавить поле и (или) метод с необходимой логикой.



Объекты ScriptableObject с логикой сравнения. Источник: Flaticon

Сравните это с поддержкой традиционного enum. Если длинный список его значений не имеет явной нумерации, вставка или удаление элемента может изменить порядок остальных. Такая перенумерация способна вызвать трудноуловимые ошибки или непредвиденное поведение.

У перечислений на основе ScriptableObject этой проблемы нет. Добавляйте в проект новые ассеты или удаляйте существующие - остальные элементы продолжат работать.

Предположим, предмет нужно сделать экипируемым в RPG. Для этого в ScriptableObject достаточно добавить логическое поле. Нужно запретить некоторым персонажам брать определенные предметы? Добавить магические свойства или особые способности? Перечисления на основе ScriptableObject поддерживают и такие сценарии.

Игровые данные могут развиваться по мере реализации дизайна игры. Команде потребуется согласовать первоначальную структуру полей, но затем дизайнеры смогут самостоятельно заполнять детали.

Паттерн: объекты-делегаты

В Unity объекты ScriptableObject предназначены не только для хранения данных. Они могут содержать методы, а значит, объединять сведения о том, что нужно сделать (логику), и о том, с чем работать (данные).

Делегаты и события

Делегаты и события в C# тесно связаны, но решают разные задачи. Делегат представляет собой тип, определяющий сигнатуру метода. Благодаря этому методы можно передавать другим методам в качестве аргументов. Делегат можно представить как переменную, которая вместо значения хранит ссылку на метод.

Событие, в свою очередь, представляет собой особый вид делегата, который позволяет классам взаимодействовать при слабой связанности. Подробнее события рассматриваются в главе о паттерне «Наблюдатель».

Общие сведения см. в материале «Различия между делегатами и событиями в C#».

Суть подхода в том, чтобы инкапсулировать алгоритмы выполнения отдельных задач в самостоятельных объектах. Авторы книги «Банда четырех» называют такое общее решение паттерном «Стратегия».

Предположим, объект поиска пути должен построить маршрут через лабиринт. Сам объект не содержит алгоритма поиска, а лишь хранит ссылку на другой объект, который выполняет вычисления.

Чтобы решить лабиринт конкретным алгоритмом поиска пути, например A* или алгоритмом Дейкстры, реализуйте это решение в отдельном объекте-стратегии. Во время выполнения алгоритм можно заменить, просто назначив другой объект.

Методы ScriptableObject

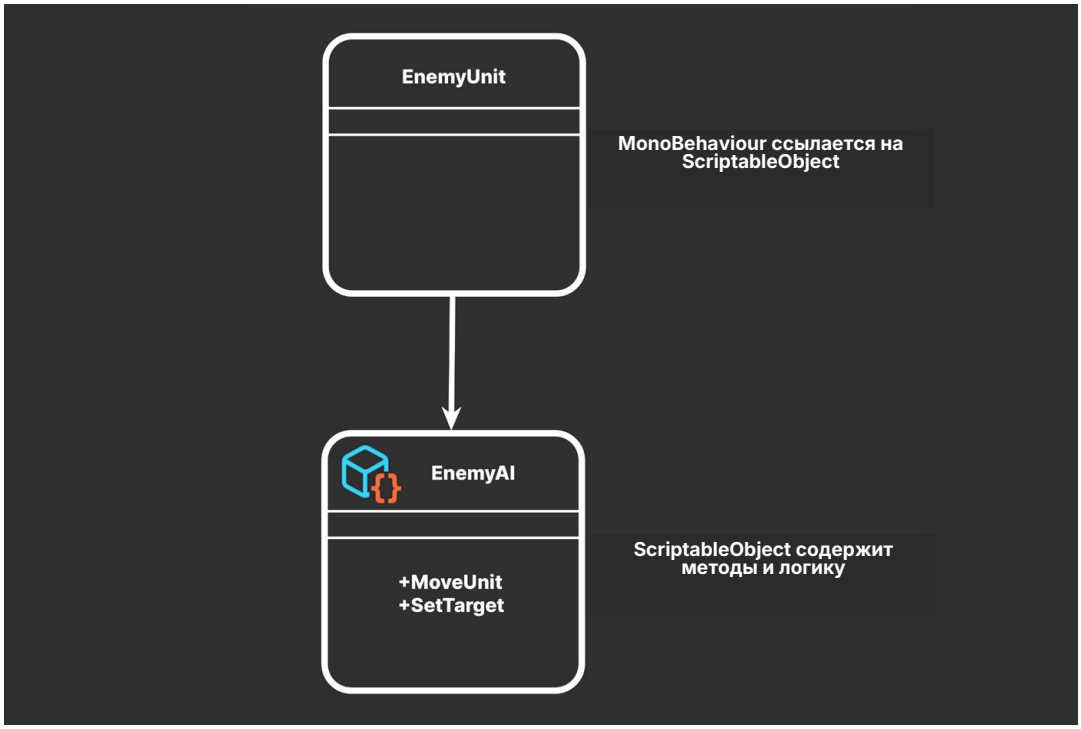
В Unity этот паттерн можно реализовать с помощью MonoBehaviour, который ссылается на ScriptableObject с необходимой логикой. Выполняя задачу, MonoBehaviour вызывает внешние методы ScriptableObject вместо собственных.

Однако у такого подхода есть несколько ограничений:

- Методы ScriptableObject не вызываются автоматически из цикла PlayerLoop компонента MonoBehaviour, как Start(), Update() и OnCollisionEnter(). Их необходимо вызывать самостоятельно.

- Как и префабы, объекты ScriptableObject не могут напрямую ссылаться на объекты сцены. Если ScriptableObject должен работать с объектом сцены, этот объект необходимо передать в качестве параметра.

> При вызове метода ScriptableObject компонент MonoBehaviour часто может передать в качестве аргумента самого себя или другие зависимости. Это позволяет выполнять логику, запускать корутины и решать другие задачи, хотя ScriptableObject существует на уровне проекта.



ScriptableObject может содержать взаимозаменяемые реализации поведения или логики.

Например, в игре можно создать несколько типов противников с разным поведением при перемещении. Одни будут патрулировать территорию, другие стоять на месте, а третьи убегать от игрока.

Один MonoBehaviour EnemyUnit может ссылаться на ScriptableObject EnemyAI с методом MoveUnit. Сам скрипт EnemyUnit не содержит логики перемещения или поведения, а лишь вызывает MoveUnit объекта ScriptableObject в подходящий момент.

Если методу нужны данные сцены, EnemyUnit может передать ссылку на самого себя в качестве параметра. Таким же образом передаются и другие необходимые зависимости из сцены.

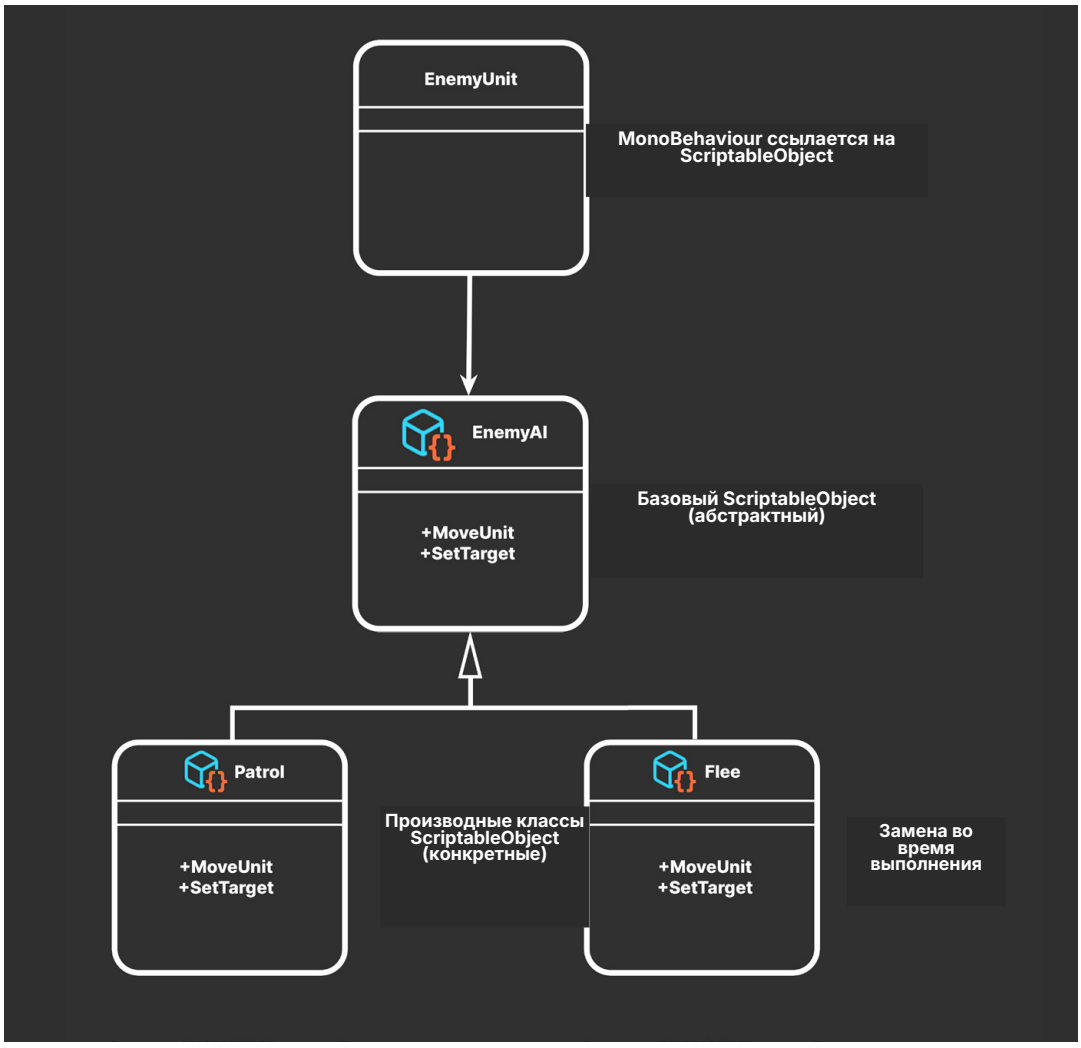
Изменение данных ScriptableObject

Данные ScriptableObject можно изменять во время выполнения, но делать это следует осторожно. Если несколько компонентов MonoBehaviour совместно используют один ScriptableObject и изменяют одни и те же данные, могут возникнуть проблемы.

Чтобы избежать этой ситуации, во время выполнения можно создать отдельный экземпляр ScriptableObject. Исходный ScriptableObject послужит шаблоном со всей логикой и данными, а каждый MonoBehaviour создаст собственный экземпляр, который можно свободно изменять.

Подключаемое поведение

Этот паттерн станет полезнее, если определить ScriptableObject EnemyAI как абстрактный класс. Тогда он послужит шаблоном для разных объектов ScriptableObject, совместимых с MonoBehaviour EnemyUnit, а один абстрактный ScriptableObject сможет представлять несколько алгоритмов.



Подключаемое поведение можно менять во время выполнения или в редакторе Unity.

Например, от базового EnemyAI можно унаследовать конкретные классы ScriptableObject с поведением Patrol, Idle и Flee. Все они реализуют один метод MoveUnit, но результаты его работы могут существенно различаться.

В редакторе Unity каждый такой ассет можно заменить другим. Достаточно перетащить нужный ScriptableObject в поле EnemyAI. Любой совместимый ScriptableObject подключается таким способом.

EnemyUnit или другой компонент может играть роль «мозга»: отслеживать момент смены ScriptableObject и заменять поведение во время выполнения. Так EnemyUnit способен реагировать на игровые события, например переходить от патрулирования к бегству. При каждой смене состояния достаточно назначить другой ScriptableObject EnemyAI.

В рабочем проекте второй разработчик или дизайнер может реализовать перемещение или логику ИИ непосредственно в ScriptableObject. При добавлении новых вариантов перемещения и поведения, например DuckAndCover или Chase, исходный скрипт EnemyUnit останется без изменений. Такой паттерн повышает расширяемость кодовой базы и соответствует принципу открытости/закрытости из SOLID. Поскольку система уже разделена на небольшие объекты, проект легче масштабировать при расширении команды или изменении дизайна игры.

Игровой ИИ на основе ScriptableObject

Более подробный пример управления поведением с помощью ScriptableObject представлен в серии видео «Подключаемый ИИ с ScriptableObject». В этих записях прямых эфиров показана система ИИ на основе конечного автомата, где состояния, действия и переходы между состояниями настраиваются с помощью объектов ScriptableObject.

Пример: аудиоделегаты

Поведение в ScriptableObject не обязательно должно быть сложным. Это может быть даже обычное воспроизведение настроенного звука.

Например, ScriptableObject в роли «звукового делегата» позволяет разнообразить звучание объектов AudioClip. AudioDelegateSO определяет абстрактный класс с методом Play, который принимает AudioSource в качестве параметра.

```
using UnityEngine;
using Random = UnityEngine.Random;
using System;

[Serializable]
public struct RangedFloat
{
	public float MinValue;
	public float MaxValue;
}

public abstract class AudioDelegateSO: ScriptableObject
{
	public abstract void Play(AudioSource source);
}
```

Конкретный ScriptableObject SimpleAudioDelegate может выбирать случайный клип из доступных вариантов и изменять при воспроизведении его громкость и высоту тона. Это делает повторяющиеся звуки менее однообразными.

```
[CreateAssetMenu(fileName ="AudioDelegate")]
public class SimpleAudioDelegateSO : AudioDelegateSO
{
	public AudioClip[] Clips;
	public RangedFloat Volume;
	public RangedFloat Pitch;

	public void Play(AudioSource source)
	{
		if (clips.Length == 0 || source == null)
			return;

		source.clip = clips[Random.Range(0, Clips.Length)];
		source.volume = Random.Range(Volume.minValue, Volume.maxValue);
		source.pitch = Random.Range(Pitch.minValue, Pitch.maxValue);
		source.Play();
	}
}
```

После этого любой MonoBehaviour сможет использовать экземпляр ScriptableObject, унаследованный от AudioDelegateSO. Для разных звуковых эффектов можно создавать и другие варианты AudioDelegate.

Методы ScriptableObject открывают множество возможностей. Они могут не только выполнять действия, но и отправлять сообщения любому объекту сцены.

Теперь рассмотрим систему событий на основе ScriptableObject и паттерна «Наблюдатель».

Славная революция ScriptableObject

Доклад Ричарда Файна «Свержение тирании MonoBehaviour в славной революции ScriptableObject» на Unite 2016 заложил основу значительной части этой электронной книги. Фрагмент демонстрационного проекта, который в оригинале назывался AudioEvent, был изменен для данного примера.

Подробности реализации и пример применения ScriptableObject см. в примере проекта.

Паттерн «Наблюдатель»

При разработке игры нескольким объектам GameObject часто требуется обмениваться данными или состояниями. В небольшой игре между ними можно создать прямые ссылки, но при масштабировании этот подход быстро становится неудобным. Управление такими зависимостями требует значительных усилий и нередко приводит к ошибкам.

По мере роста приложения потребуется более эффективное решение.

Отказ от синглтонов

Многие разработчики выбирают синглтон, то есть единственный глобальный экземпляр класса, который сохраняется при загрузке сцен. Однако синглтоны создают глобальное состояние и затрудняют модульное тестирование.

Если префаб ссылается на синглтон, для проверки одной изолированной функции придется подключить все зависимости этого синглтона. Это снижает модульность, а отладка кода усложняется.

В качестве альтернативы для обмена данными между объектами можно использовать события на основе ScriptableObject.

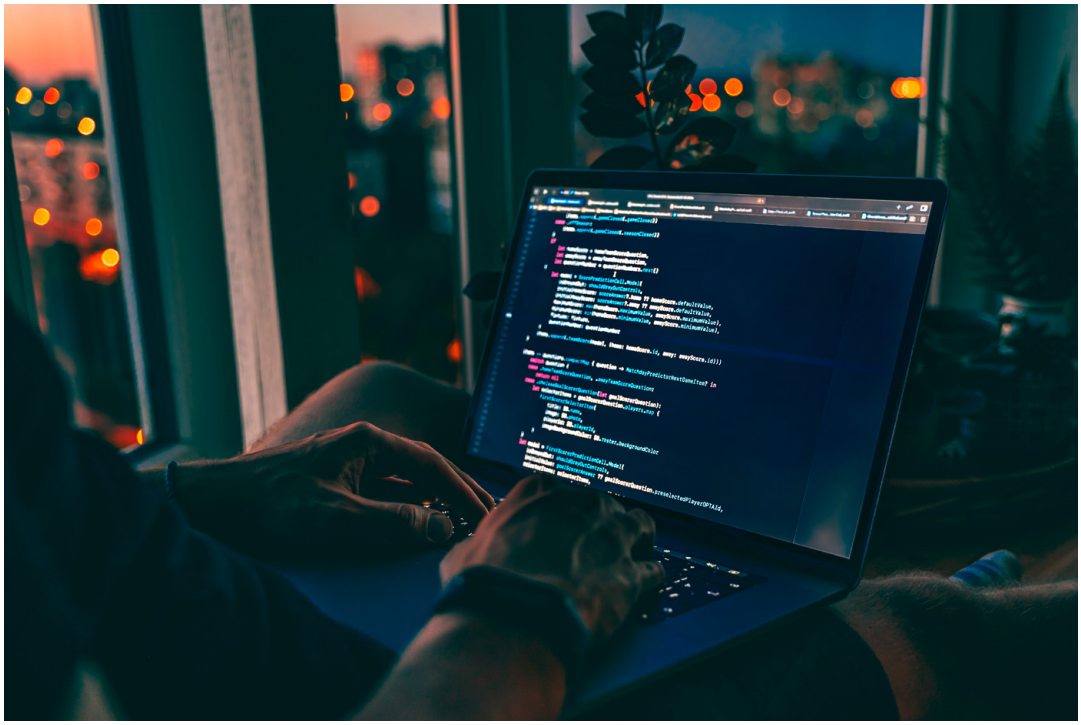
Подробнее о синглтонах

Синглтоны в разработке игр на Unity часто вызывают споры. Они могут быть подходящим решением для небольших проектов или прототипов, но в крупных приложениях недостатки обычно перевешивают преимущества. Поэтому многие разработчики считают синглтон антипаттерном.

Синглтоны легко изучить и понять, однако при неправильном применении они создают проблемы. Большинство описанных здесь паттернов помогает обходиться без синглтонов.

Если нужен простой доступ к общим данным, рассмотрите Runtime Set на основе ScriptableObject, описанный далее. Для обмена сообщениями между объектами подойдет канал событий на основе ScriptableObject. Переход от синглтонов к такой архитектуре может повысить масштабируемость и тестируемость проекта.

Подробнее о преимуществах и недостатках синглтонов см. в электронной книге «Повышаем уровень кода с паттернами проектирования и SOLID».

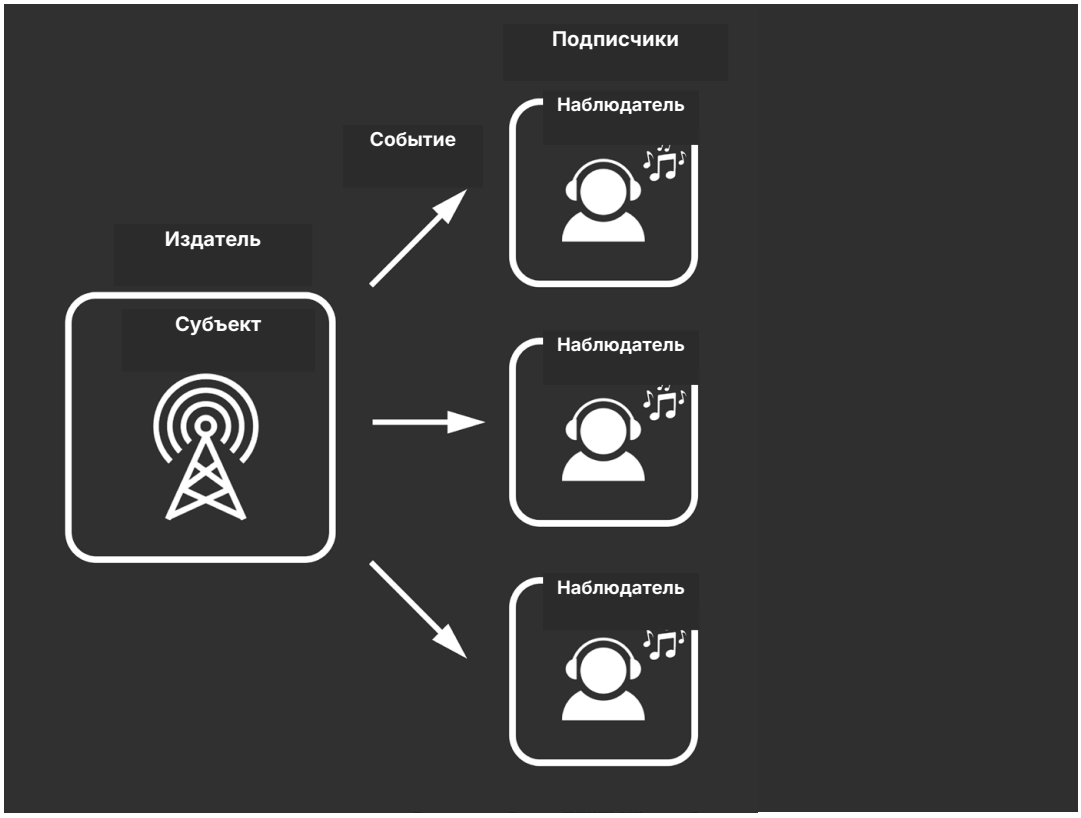


События на основе ScriptableObject

Как вы уже видели, ScriptableObject предназначен не только для работы с данными. Как и любой другой скрипт, он может содержать методы, через которые объекты будут обмениваться сообщениями.

В паттерне «Наблюдатель» субъект передает сообщение одному или нескольким слабо связанным наблюдателям. Каждый наблюдатель реагирует независимо от субъекта и ничего не знает об остальных наблюдателях. Субъект также называют издателем или отправителем, а наблюдателей - подписчиками или слушателями.

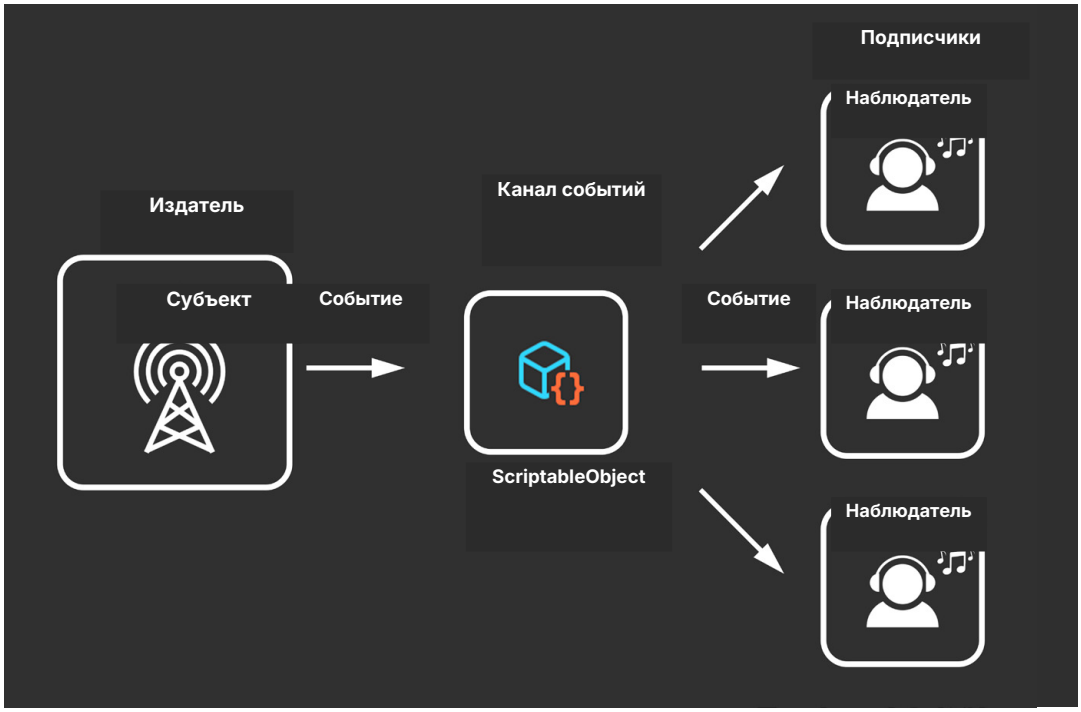
Архитектура на основе событий выполняет код только при необходимости, а не в каждом кадре. Поэтому она часто эффективнее логики, добавленной в методы обновления MonoBehaviour.



Базовый паттерн «Наблюдатель»

Паттерн «Наблюдатель» можно реализовать с помощью MonoBehaviour или объектов C#. Такой подход широко применяется в Unity, но при использовании одних скриптов дизайнеры будут обращаться к программистам для создания каждого события игрового процесса.

Альтернативой служат события на основе ScriptableObject. Это удобный для дизайнеров способ настроить паттерн «Наблюдатель»: ScriptableObject выступает посредником между субъектом и наблюдателем и предоставляет графический интерфейс в редакторе Unity.



Канал событий на основе ScriptableObject

На первый взгляд кажется, что в паттерне «Наблюдатель» появился лишь дополнительный служебный слой, однако такая структура дает ряд преимуществ. ScriptableObject является ассетом, поэтому доступен всем объектам в иерархии сцены и не исчезает при загрузке другой сцены.

Именно ради простого постоянного доступа к определенным ресурсам разработчики часто используют синглтоны. ScriptableObject нередко дает те же преимущества, не создавая столько лишних зависимостей.

В событиях на основе ScriptableObject издателем, который отправляет событие, и подписчиком, который его принимает, может быть любой объект. ScriptableObject находится между ними, передает сигнал и выступает единым посредником.

Такую структуру удобно представить как канал событий. ScriptableObject подобен радиовышке, сигналы которой принимает любое количество объектов. Заинтересованный MonoBehaviour подписывается на канал событий и реагирует, когда что-либо происходит.

Пример: каналы событий

Любой ScriptableObject со следующими элементами может работать как канал событий:

- Делегат UnityAction или System.Action: он уведомляет подписчиков и передает нужные данные в параметрах. UnityAction удобнее для художников и дизайнеров; в остальных случаях хорошо подходит System.Action.

> Ключевое слово event ограничивает делегат: вызвать его можно только из класса ScriptableObject, в котором он объявлен, или из производного класса.

- Метод отправки события: этот открытый метод вызывает делегат.

Вот и все.

Можно создать любое количество каналов событий для управления разными аспектами игрового процесса. Объекты ScriptableObject существуют на уровне проекта, поэтому могут отправлять глобально доступные события. Так можно связывать даже не связанные напрямую объекты сцены без ущерба для масштабируемости.

System.Action или UnityAction

System.Action - универсальный тип делегата, определенный в пространстве имен System платформы .NET Framework. Его можно использовать в проектах Unity без объявления собственного делегата. Ключевое слово event делает делегат доступным только для чтения: другие объекты могут подписываться на него, регистрируя свои методы, но не могут вызывать делегат напрямую.

UnityAction - тип делегата, определенный специально в пространстве имен UnityEngine.Events. Обычно он применяется вместе с классом UnityEvent, который предоставляет еще один способ создавать события в Unity. UnityEvent и UnityAction отображаются в окне Inspector, поэтому с их помощью часто удобнее реализовывать паттерн «Наблюдатель» для пользователей, художников и дизайнеров.

В зависимости от задачи можно использовать System.Action, UnityAction или оба типа в одном проекте.

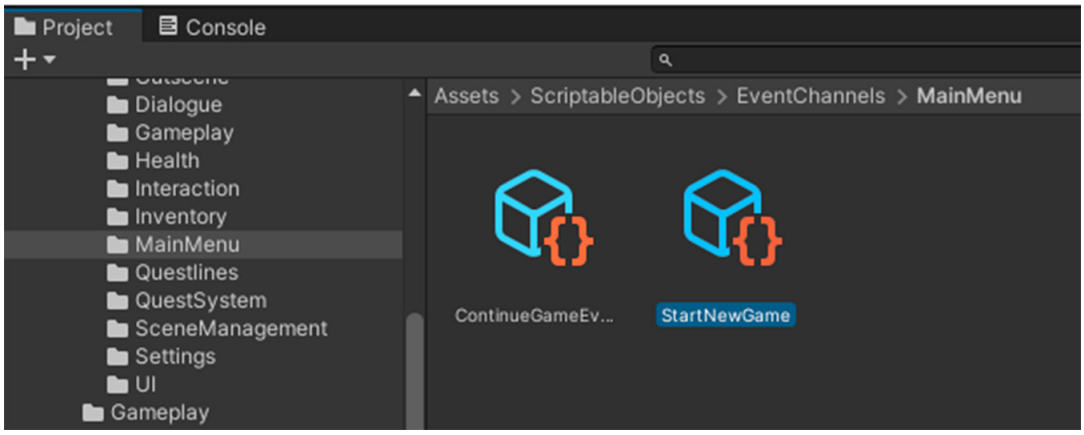
Если требуется универсальный делегат, не привязанный к игровому движку Unity, используйте System.Action. Если делегат предназначен специально для UnityEvent, выберите UnityAction.

В этом примере VoidEventChannelSO отправляет событие без параметров. Он содержит UnityAction с именем OnEventRaised.

```
[CreateAssetMenu(menuName = "Events/Void Event Channel")]
public class VoidEventChannelSO : ScriptableObject
{
	public event UnityAction OnEventRaised;
	public void RaiseEvent()
	{
		if (OnEventRaised != null)
			OnEventRaised.Invoke();
	}
}
```

После создания ScriptableObject типа VoidEventChannelSO любой MonoBehaviour сможет подписаться на OnEventRaised. Например, можно создать ScriptableObject StartNewGame типа VoidEventChannelSO.

Другой объект может вызвать открытый метод RaiseEvent и тем самым отправить событие.



Событие на основе ScriptableObject: пример канала событий

Другой MonoBehaviour может сослаться на ScriptableObject канала событий в окне Inspector, а затем подписаться на OnEventRaised или отменить подписку.

Когда канал событий вызывает OnEventRaised, в ответ выполняется StartNewGame:

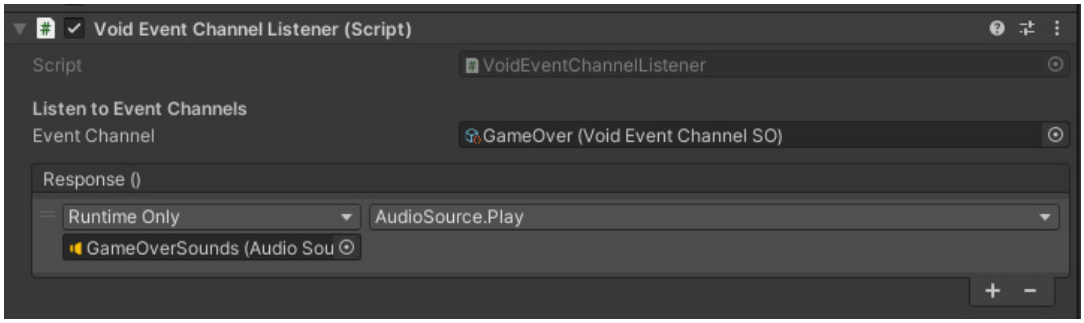
```
public class StartGame : MonoBehaviour
{
	[SerializeField] private VoidEventChannelSO m_onNewGameButton =
default;

	private void Start()
	{
		m_onNewGameButton.OnEventRaised += StartNewGame;
	}

	private void OnDestroy()
	{
		m_onNewGameButton.OnEventRaised -= StartNewGame;
	}

	private void StartNewGame()
	{
	    // здесь размещается логика загрузки уровня...
	}
}
```

Чтобы сделать компоненты-слушатели удобнее для художников и дизайнеров, можно создать MonoBehaviour, не требующий дополнительной настройки скриптом. Класс VoidEventListener не добавляет новых функций, но предоставляет поля, доступные в окне Inspector:

```
public class VoidEventListener : MonoBehaviour
{
	[SerializeField] private VoidEventChannelSO m_channel = default;

	public UnityEvent OnEventRaised;

	private void OnEnable()
	{
		if (m_channel != null)
			m_channel.OnEventRaised += Respond;
	}
```

```
	private void OnDisable()
	{
		if (m_channel != null)
			m_channel.OnEventRaised -= Respond;
	}

	private void Respond()
	{
		if (OnEventRaised != null)
			OnEventRaised.Invoke();
	}
}
```

Добавьте VoidEventListener на GameObject, затем перетащите ScriptableObject канала событий в поле _channel окна Inspector. Чтобы реагировать на событие, добавьте в OnRaisedEvent нужные обработчики UnityAction.



Слушатель событий позволяет специалисту без навыков программирования настраивать действия, запускаемые событиями.

Независимо от того, какой компонент выбран в качестве слушателя, каналы событий обеспечивают взаимодействие между объектами во время выполнения. Игрок выполнил задание или заработал очко? Игра завершилась? Событие может уведомить любой GameObject в сцене, которому нужна эта информация.

Поскольку такие события на основе ScriptableObject являются ассетами уровня проекта, они могут управлять значительной частью инфраструктуры приложения. Это особенно удобно для обмена сообщениями между системами, лежащими в основе архитектуры игры.

Распространенные системы управления:

- Управление звуком: самые разные события в игре могут запускать звуки. Эта система может воспроизводить объекты AudioClip или изменять параметры AudioMixer в ответ на события приложения.

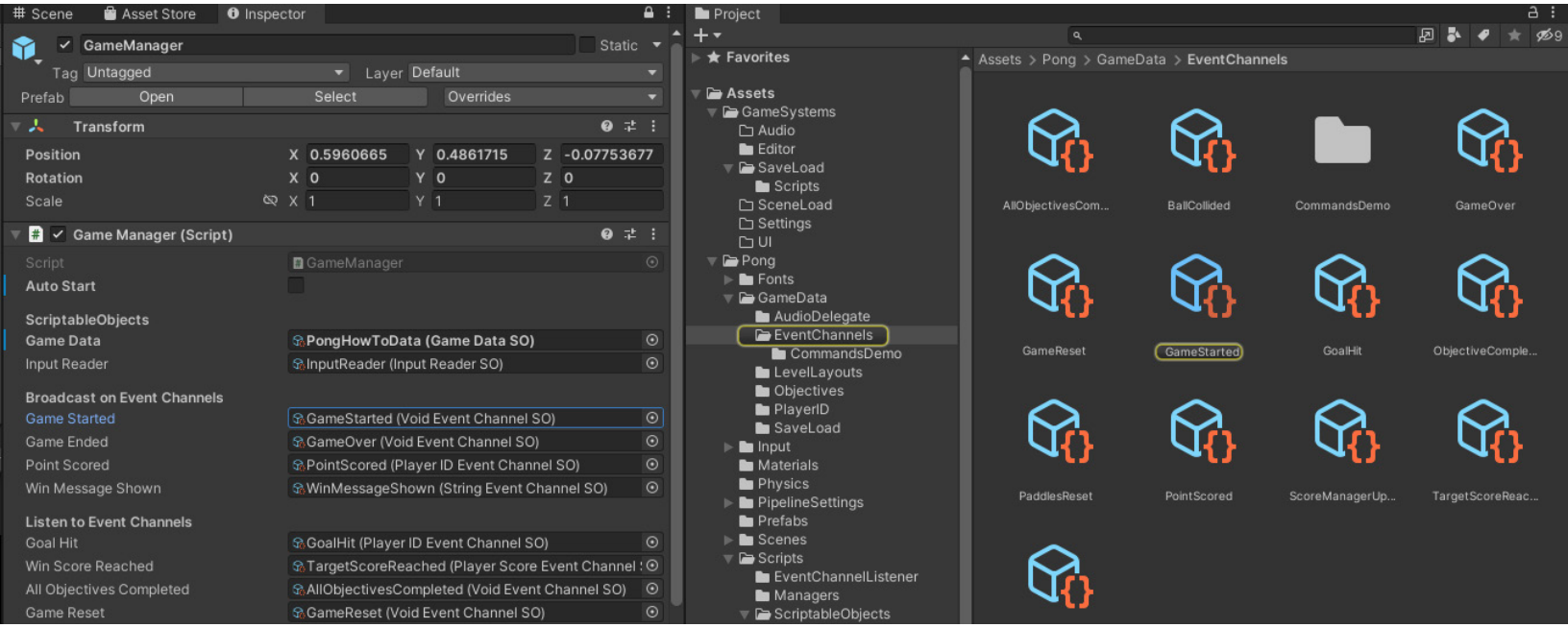
- Управление сценами: эта система отвечает за загрузку и выгрузку сцен Unity и игровых уровней.

- Управление UI: эта система отвечает за экраны меню до начала игры, во время нее и после завершения.

- Управление сохранениями: эта система отвечает за сохранение игровых данных и настроек в файловой системе, а также за их загрузку.

Все эти системы решают разные задачи, но им необходимо взаимодействовать. События служат связующим звеном между ними.

Дополнительные примеры реализации собственных событий на основе ScriptableObject приведены в сопутствующем примере проекта.



GameManager из примера проекта использует каналы событий.

События с разной полезной нагрузкой позволяют передавать данные разных типов. Например, среди событий на основе ScriptableObject есть IntEventChannelSO, Vector2EventChannelSO, VoidEventChannelsSO и другие. Выбор события зависит от контекста.

Создавайте дополнительные типы событий с учетом игровой логики. Например, событию нанесения урона могут понадобиться сведения о том, кто нанес урон и какова его величина.

Варианты применения каналов событий ограничены только вашей фантазией. Помимо перечисленных основных систем, события нередко помогают связать совершенно разные игровые подсистемы и наладить их взаимодействие:

- Камеры: позволяют создавать выразительные и кинематографичные эффекты, например тряску камеры или переход к другому ракурсу.

- Задания: задачи или цели, которые игрок должен выполнить, чтобы продвинуться по игре или получить награду. Задания часто объединяют разные элементы игрового процесса, например сбор предметов, победу над противниками или решение головоломок.

- Здоровье: этот важный элемент многих игр связывает игрока, противников, а также любые объекты и действия, способные нанести игроку урон.

- Достижения: как и задания, это особые награды, которые игроки открывают за выполнение определенных задач или целей. Достижения могут охватывать разные элементы игрового процесса, например достижение заданного уровня или накопление определенного количества очков.

В свою очередь, эти элементы игрового процесса взаимодействуют через события с другими системами управления, включая звук, UI и сохранения. Такой подход обеспечивает модульность и независимость компонентов архитектуры, сохраняя при этом возможность обмениваться данными с другими системами.

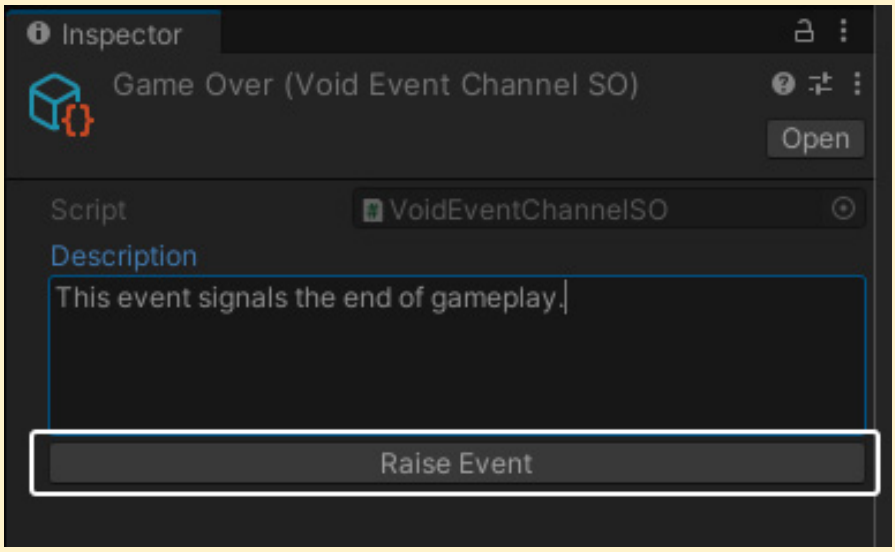
Отладка каналов событий

Пользовательский Editor или PropertyDrawer может добавить в окно Inspector кнопку «Вызвать событие». С ее помощью событие можно запускать вручную при отладке.

Ниже приведен простой скрипт Editor, который добавляет в окно Inspector специальную кнопку для VoidEventChannelSO:

```
[CustomEditor(typeof(VoidEventChannelSO))]
public class VoidEventChannelSOEditor : Editor
{
    public override void OnInspectorGUI()
    {
        DrawDefaultInspector();

        VoidEventChannelSO eventChannel = (VoidEventChannelSO)target;

        if (GUILayout.Button("Raise Event"))
        {
            eventChannel.RaiseEvent();
        }
    }
}
```

В результате появляется кнопка, с помощью которой событие можно вызвать в любой момент, что упрощает поиск проблем во время выполнения.



Кнопка, добавленная пользовательским Editor, помогает тестировать каналы событий.

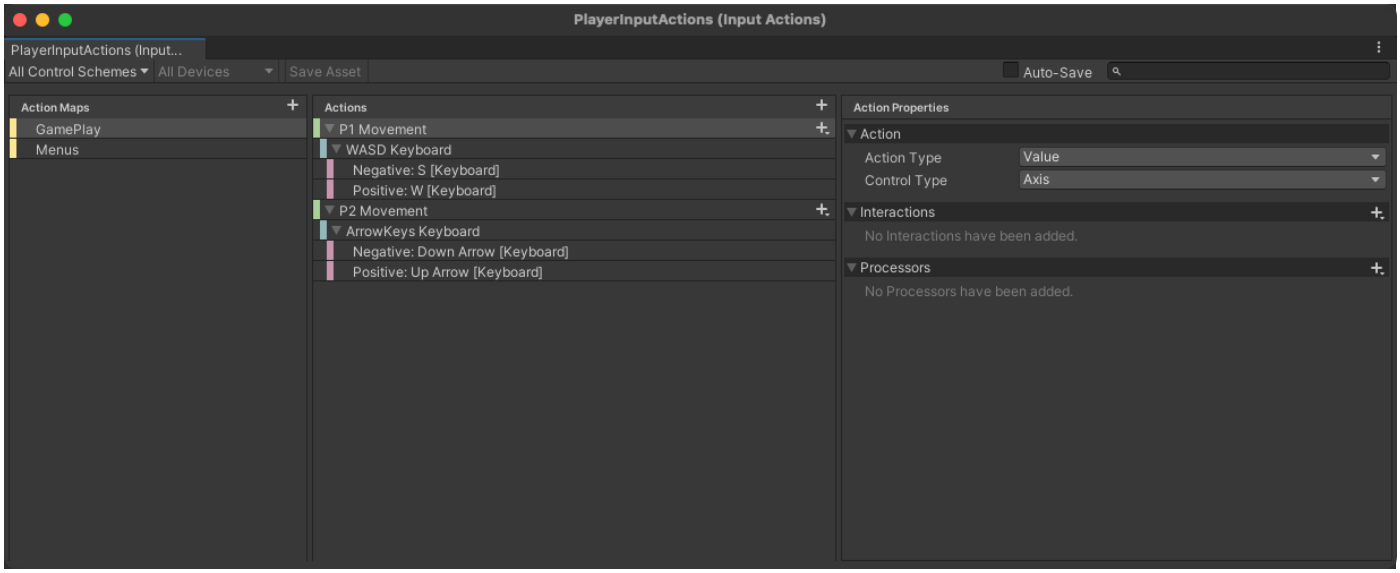
Добавив немного кода, можно создать кнопки и для каналов событий, передающих данные. Предусмотрите в самом канале поле для отладочного значения, а затем передайте его в скрипт Editor. Примеры такой реализации есть в проектах SOAP и Unity Atoms.

По мере использования каналов событий для разделения зависимостей между объектами стоит развивать средства отладки, например вести список всех слушателей каждого события. В класс канала событий можно добавить методы регистрации и отмены регистрации объектов, чтобы во время выполнения было проще определить, какие события вызывают конкретное поведение.

Пример: InputReader

Объектам, которые обрабатывают пользовательский ввод, нужен специализированный тип канала событий. Input System в Unity использует InputAction, чтобы представлять необработанные входные данные в виде логических действий, например прыжка или ходьбы.

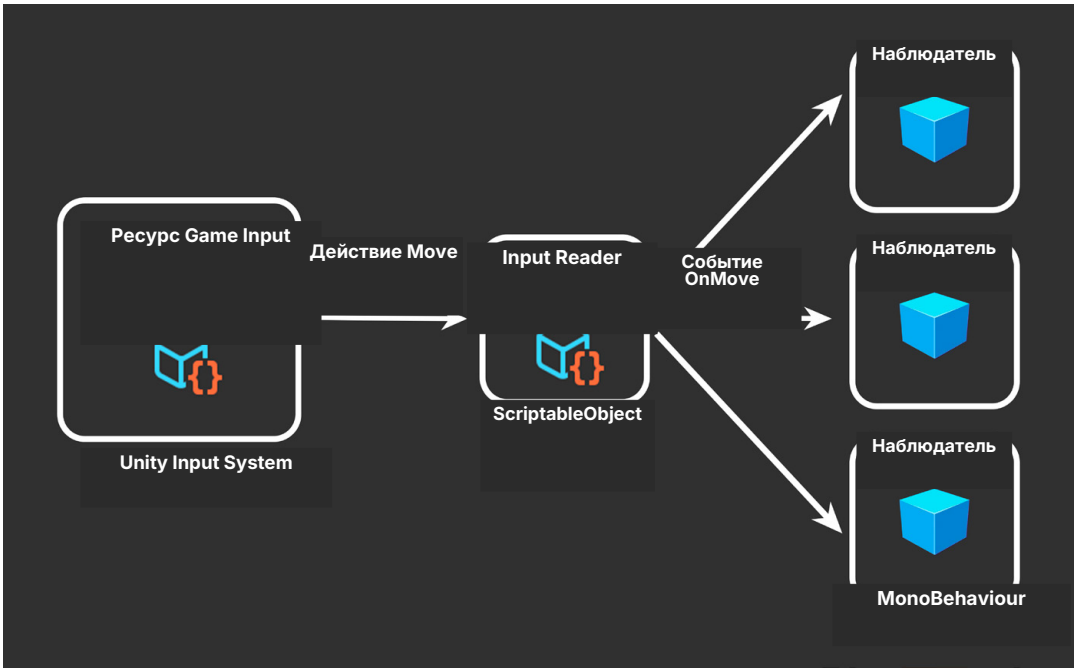
Каждый InputAction, в свою очередь, содержит собственные события started, performed и canceled.



Настройка действий (Actions) и карт действий (Action Maps) в редакторе Input System

Для обработки привязок InputAction можно создать специальный ScriptableObject InputReader. Он снова выступает посредником между субъектом и наблюдателями. Однако в этом случае компоненты MonoBehaviour не вызывают события напрямую.

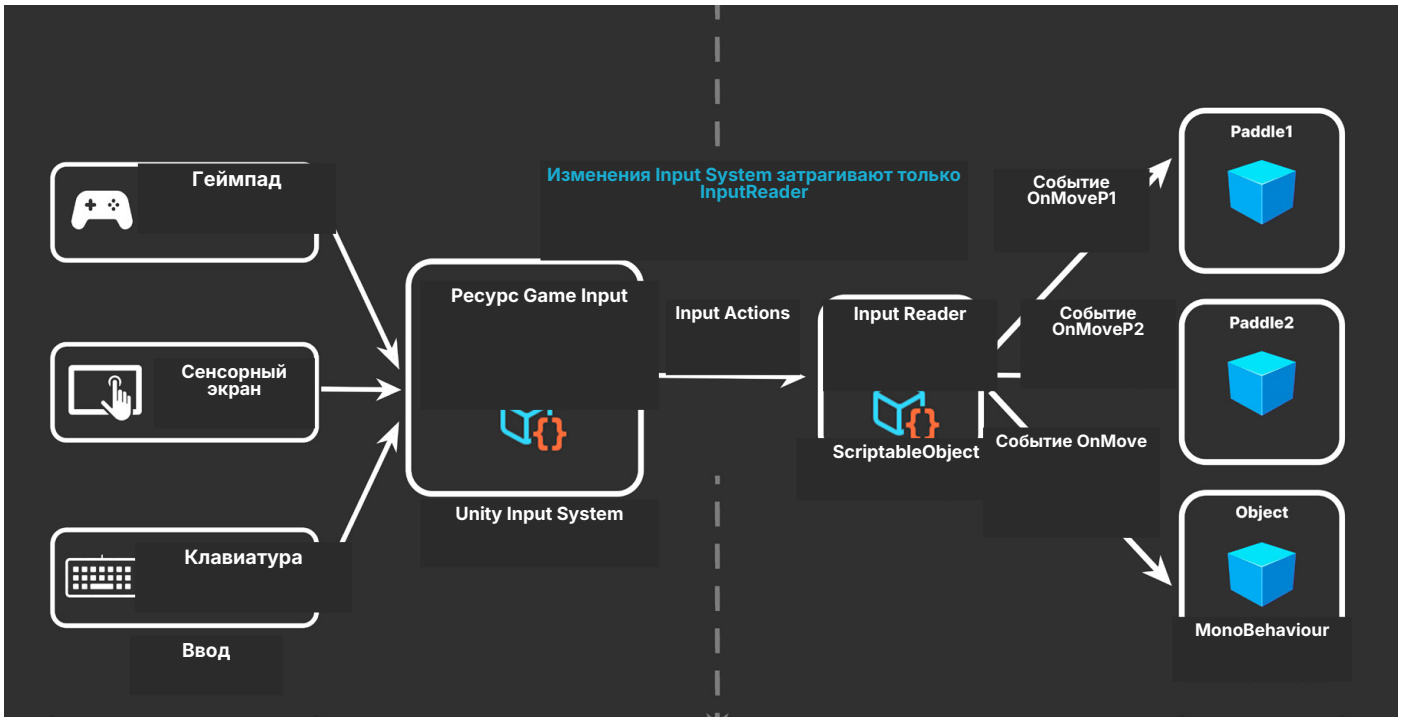
Вместо этого роль субъекта, то есть отправителя событий, выполняет Input System:



ScriptableObject InputReader ретранслирует события от объектов InputAction.

Здесь в Input System настраиваются действия (Actions) и карты действий (Action Maps). Каждый InputAction описывает отдельную ось ввода и связывается с клавиатурой, геймпадом или любым другим устройством ввода.

Вместо прямой подписки на InputAction контроллеры ракеток подписываются соответственно на события OnMoveP1.performed и OnMoveP2.performed.



InputReader изолирует объекты от прямой зависимости от системы ввода.

В результате InputReader задает единый способ обработки действий с геймпада и клавиатуры. Каждый GameObject, которому нужен ввод:

- хранит ссылку на ScriptableObject InputReader;

- подписывается на нужные события и привязывает к ним свои методы-обработчики.

Для небольшой игры такой паттерн может показаться избыточным, но этот пример помогает лучше понять принцип его работы.

Преимущества станут очевидны, когда проект разрастется и в нем появится гораздо больше компонентов. Отделение ввода от использующих его GameObject делает систему гибче и упрощает повторное использование компонентов.

Если во время разработки потребуется изменить InputAction, достаточно сопровождать только InputReader. Если сами события менять не требуется, объекты-слушатели не затрагиваются. Поэтому поддерживать связь между вводом и наблюдателями становится проще, особенно если наблюдателей много.

Статические и нестатические события

Можно использовать статические события, чтобы слушающим объектам не приходилось искать ScriptableObject.

Например, в методе OnEnable компонент MonoBehaviour может подписаться на статические события MoveP1Event и MoveP2Event класса InputReader:

```
InputReader.MoveP1Event += OnMoveP1;

InputReader.MoveP2Event += OnMoveP2;
```

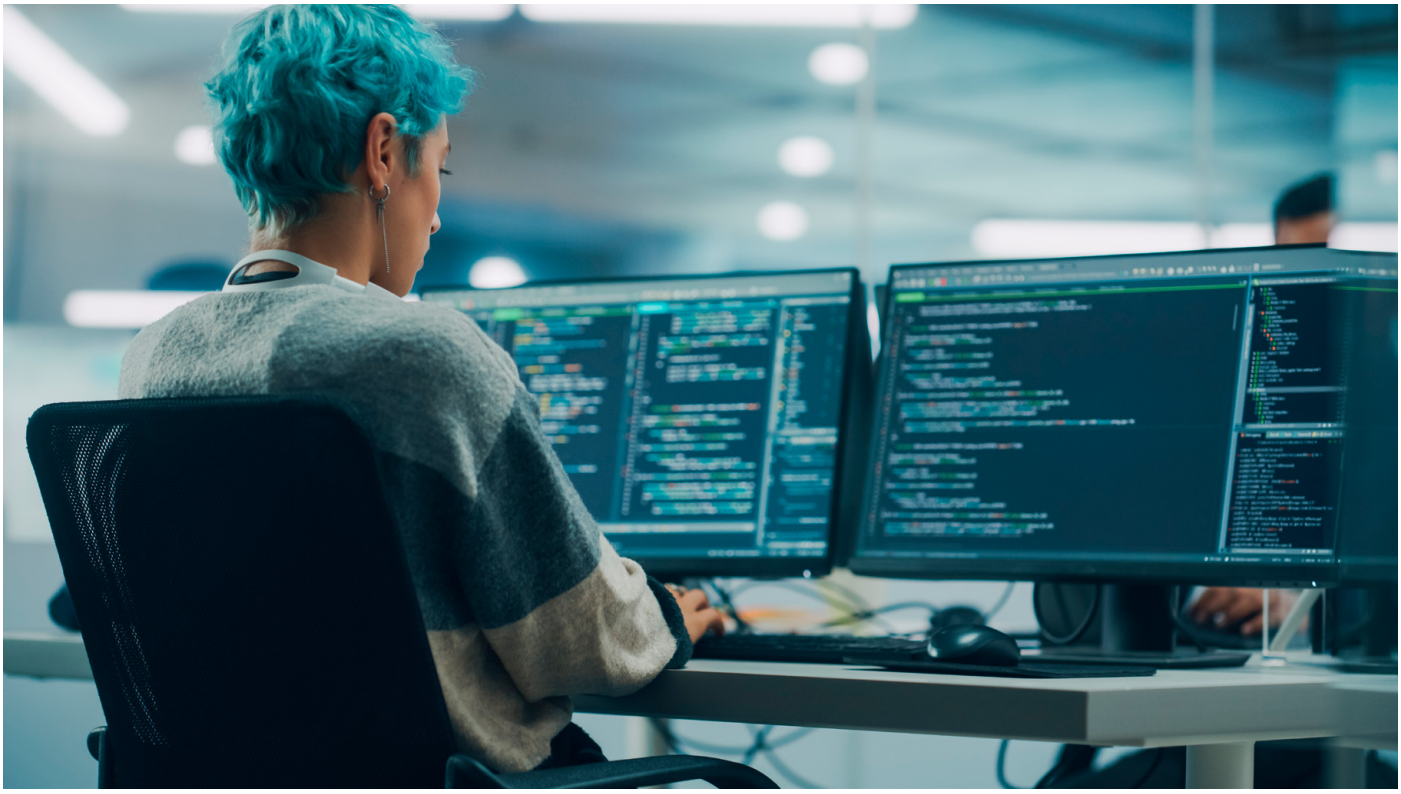
При работе со статическими событиями особенно тщательно управляйте подписками. Не забудьте отписаться в OnDisable:

```
InputReader.MoveP1Event -= OnMoveP1;

InputReader.MoveP2Event -= OnMoveP2;
```

Статические события всегда остаются доступными и при наличии активных подписчиков не освобождаются сборщиком мусора. Забытые подписчики будут препятствовать очистке до завершения работы приложения.

Однако статические события не сериализуются. Если нужна интерактивная работа в редакторе Unity, используйте нестатические события и укажите в окне Inspector ссылку на соответствующий ScriptableObject.

Паттерн «Команда»



Вместо прямого вызова метода паттерн «Команда» позволяет инкапсулировать один или несколько вызовов методов в «объекте команды».

Затем объекты команд сохраняются в коллекции, например в очереди или стеке, которая служит небольшим буфером. Это дает дополнительный контроль над выполнением каждой команды. Типичные сценарии включают воспроизведение последовательности действий с заданными интервалами и поддержку их отмены.

Вероятно, вы уже встречали этот прием в любимых игровых жанрах:

- В стратегии в реальном времени паттерн «Команда» можно использовать, чтобы помещать в очередь действия юнитов и зданий. Игра будет выполнять команды для зданий по мере появления ресурсов, а для юнитов - последовательности перемещений.

- В пошаговой стратегии игрок может выбрать юнита, а затем сохранить его перемещения или действия в очереди либо другой коллекции. В конце хода игра выполнит все команды из очереди игрока.

- В головоломке паттерн «Команда» позволяет игроку отменять и повторно выполнять действия.

- В файтинге определенная последовательность нажатий кнопок или движений стика геймпада может запускать комбо и специальные приемы.

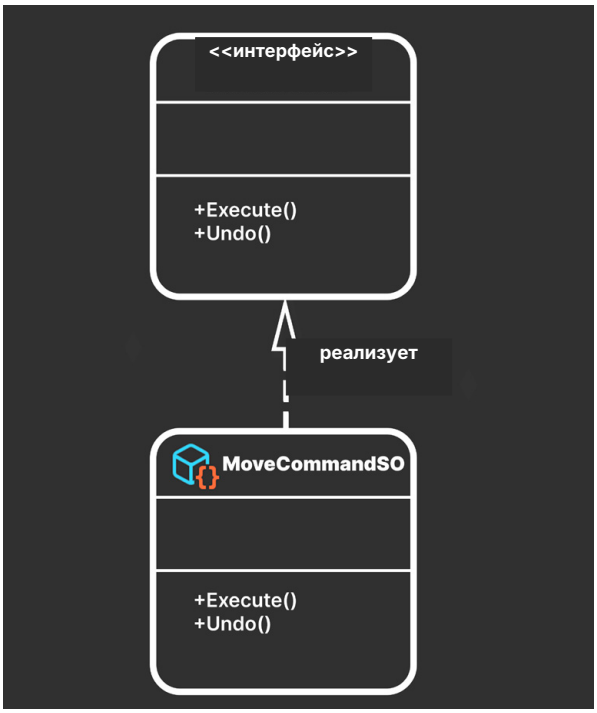
Для реализации паттерна «Команда» можно использовать ScriptableObject.

Например, можно создать команду, задающую перемещение Transform. Если каждое действие обернуть в отдельный объект, управлять им становится проще.

Сначала определите интерфейс ICommand с методами Execute и Undo. Вместо интерфейса также можно использовать абстрактный класс:

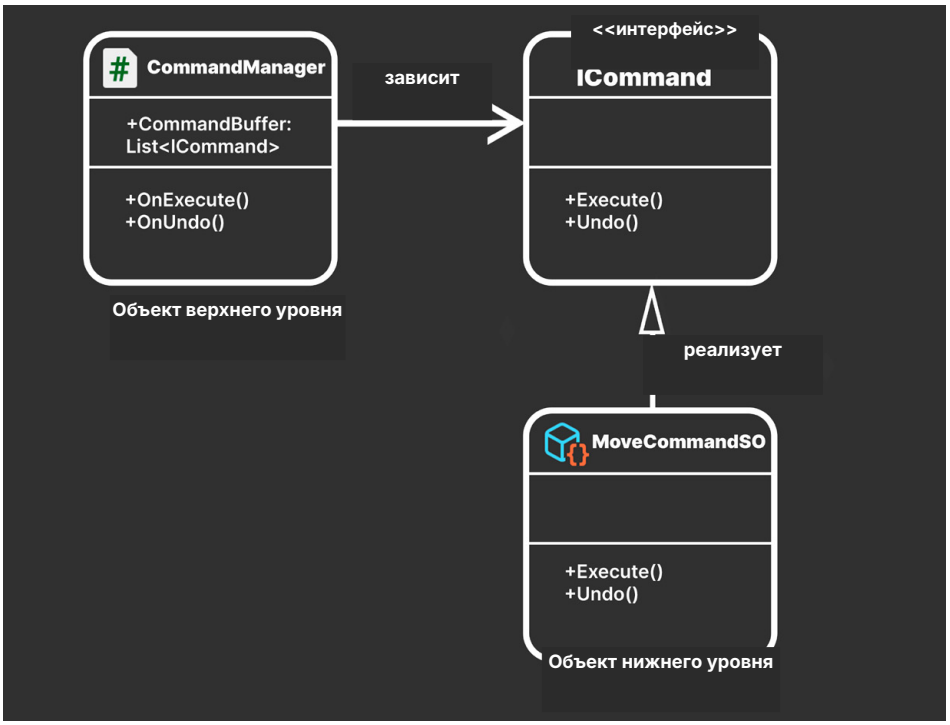
```
public interface ICommand
{
    public abstract void Execute();
    public abstract void Undo();
}
```

Затем реализуйте интерфейс ICommand в ScriptableObject. Каждый объект команды получает собственные реализации методов Execute и Undo.



Реализация паттерна «Команда» с помощью ScriptableObject

После этого MonoBehaviour или ScriptableObject может определить буфер команд, содержащий объекты команд. В качестве буфера подойдет коллекция: список, стек, массив или очередь.



CommandManager хранит коллекцию объектов ICommand.

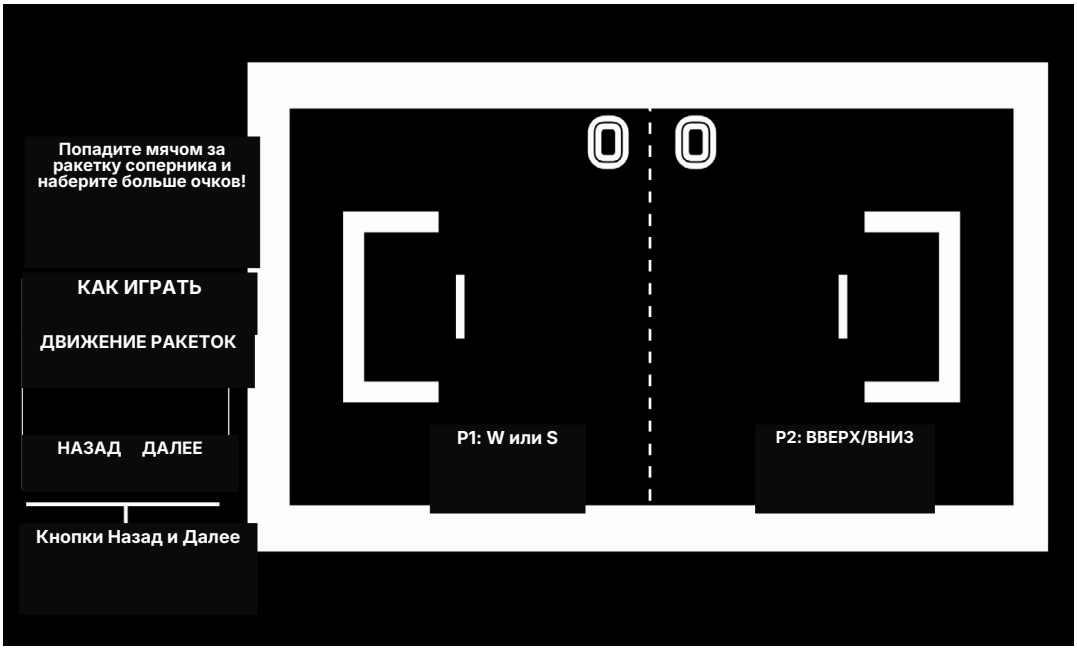
Такая простая структура позволяет выполнять команды по очереди. Представьте обучающий эпизод или кат-сцену, в которой GameObject выполняет заданную последовательность действий или анимаций. Паттерн «Команда» отлично подходит для этой задачи.

Каждая команда является отдельным объектом, поэтому их легко менять местами. Достаточно выбрать способ организации CommandBuffer:

- Если использовать стек, при выполнении помещайте команды в него. При отмене извлекайте команду из основного стека и помещайте в отдельный стек повторного выполнения.
- Если используется список или массив, отслеживайте индекс текущей команды. При отмене уменьшайте индекс, а при повторном выполнении увеличивайте.

Один из вариантов реализации отменяемого перемещения показан в классах MoveCommandSO и CommandManager из примера проекта. В простой обучающей сцене подписи к элементам игрового поля объясняют правила игры.

Нажмите кнопку Next («Далее»), чтобы перейти к следующему пояснению. Кнопка Back («Назад») позволяет просматривать инструкции в обратном порядке.



Простая реализация паттерна «Команда».

Подробнее о паттерне «Команда» можно узнать из электронной книги «Повышаем уровень кода с паттернами проектирования и SOLID». Также ознакомьтесь с публикацией сообщества «Паттерн "Команда" с ScriptableObject», где показана реализация этого паттерна с помощью ScriptableObject.

ScriptableObject или классы C#?

Выбирая архитектуру кода для проекта, важно учитывать навыки и предпочтения команды, а также требования игры к производительности. Одним дизайнерам удобнее работать интерактивно в редакторе Unity, а другим полностью в коде C#.

Учитывайте эти факторы, чтобы с кодовой базой было удобно работать всем участникам команды. Ни один паттерн не подходит абсолютно для всех задач, поэтому перед внедрением тщательно оцените его преимущества и недостатки.

Помните: «правильная» архитектура кода просто та, которая лучше всего подходит вашей команде и проекту.

Паттерн Runtime Set

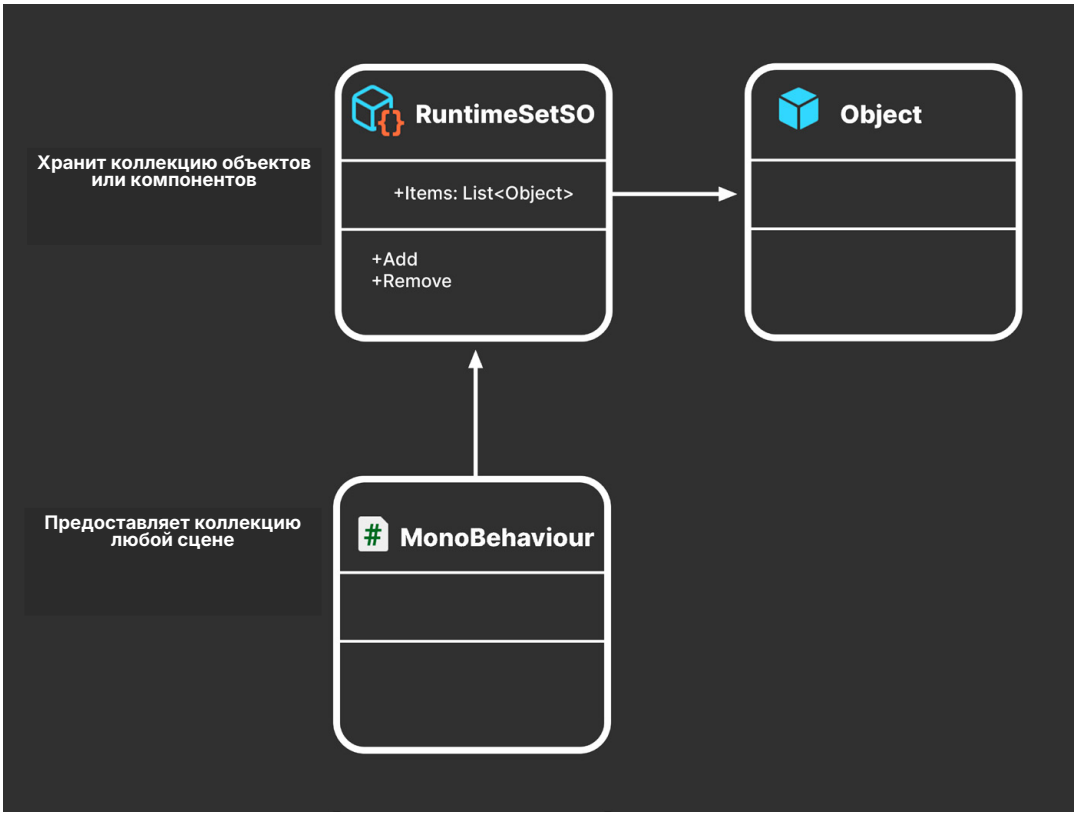
Во время выполнения часто требуется отслеживать список объектов GameObject или компонентов в сцене. Например, может понадобиться список противников или NPC.

Экземпляр ScriptableObject существует на уровне проекта, поэтому может хранить данные, доступные любому объекту из любой сцены. Это обеспечивает почти такой же удобный глобальный доступ, как синглтон, но без известных недостатков этого паттерна.

Чтение данных напрямую из ScriptableObject также эффективнее, чем поиск в иерархии сцены через Object.FindObjectOfType или GameObject.FindWithTag. В зависимости от задачи и размера иерархии эти методы сравнительно затратны и не подходят для вызова в каждом кадре.

Базовый Runtime Set

Вместо этого можно хранить данные в ScriptableObject как Runtime Set. Это специализированный контейнер данных с общедоступной коллекцией элементов и базовыми методами добавления в коллекцию и удаления из нее.



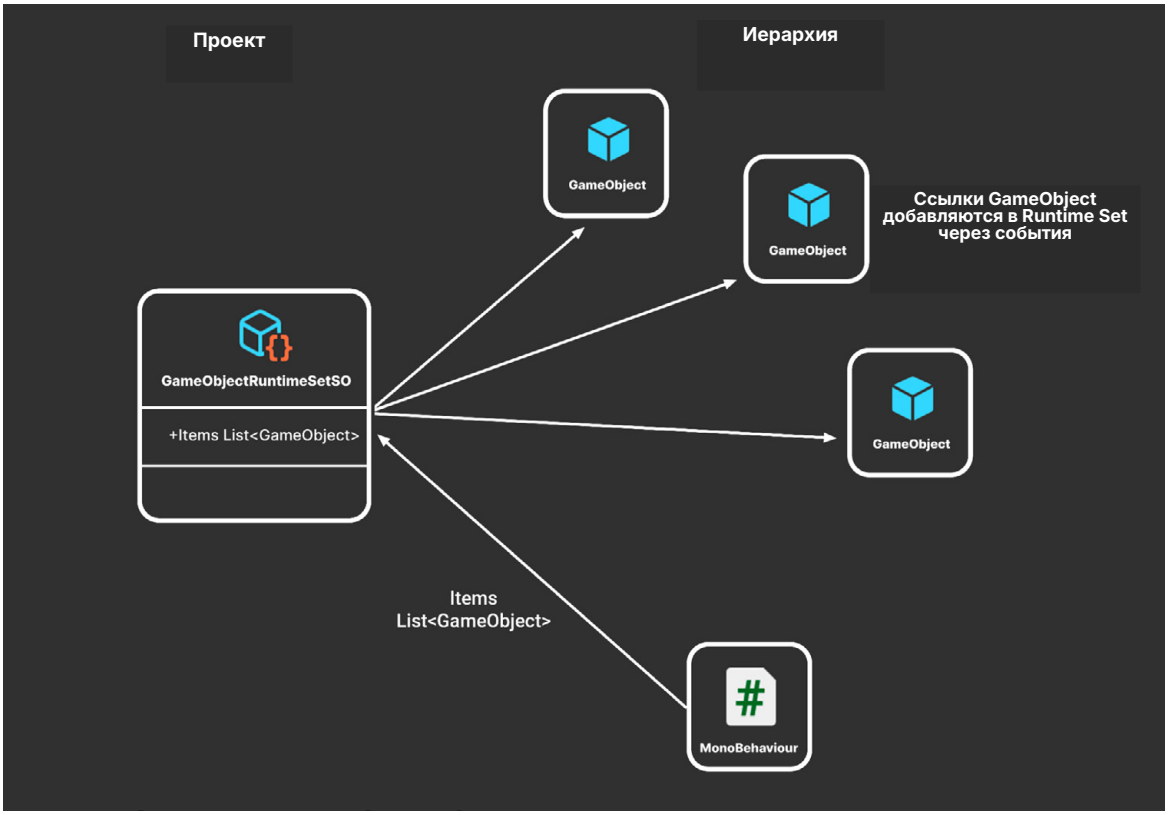
Runtime Set предоставляет глобальный доступ к коллекции данных.

Ниже приведен базовый Runtime Set, который отслеживает список объектов GameObject:

```
using System.Collections.Generic;
using UnityEngine;

[CreateAssetMenu(menuName = "GameObject Runtime Set",
    fileName = "GORuntimeSet")]
public class GameObjectRuntimeSetSO : ScriptableObject
{
    private List<GameObject> items = new List<GameObject>();
    public List<GameObject> Items => items;

    public void Add(GameObject thingToAdd)
    {
        if (!items.Contains(thingToAdd))
            items.Add(thingToAdd);
    }
```

```
    public void Remove(GameObject thingToRemove)
    {
        if (items.Contains(thingToRemove))
            items.Remove(thingToRemove);
    }
}
```

Во время выполнения любой MonoBehaviour может обратиться к общедоступному свойству Items и получить полный список. За добавление объектов GameObject в этот список и удаление из него должен отвечать другой скрипт или компонент.



Runtime Set для GameObject

Добавьте ссылку на Runtime Set в MonoBehaviour. Затем в функциях событий OnEnable и OnDisable добавляйте объект в список Items этого Runtime Set и удаляйте его оттуда. В качестве альтернативы используйте канал событий, передающий GameObject в качестве полезной нагрузки, например GameObjectEventChannel.

Обобщенная версия

Иногда Runtime Set требуется для конкретного типа MonoBehaviour. Например, так можно хранить доступный всем объектам сцены список противников или подбираемых предметов. В этом случае для каждого типа можно создать отдельный Runtime Set, например EnemyRuntimeSet или PickupRuntimeSet.

Упростить создание дополнительных Runtime Set можно с помощью обобщенного абстрактного класса:

```
public abstract class RuntimeSetSO<T> : ScriptableObject
 {
     [HideInInspector]
     public List<T> Items = new List<T>();


     public void Add(T thing)
     {
         if (!Items.Contains(thing))
             Items.Add(thing);
     }


     public void Remove(T thing)
     {
         if (Items.Contains(thing))
             Items.Remove(thing);
     }
 }
```

Он работает аналогично исходному GameObjectRuntimeSet, но дает больше гибкости. Чтобы создать Runtime Set для пользовательского компонента Foo, определите конкретный FooRuntimeSetSO следующим образом:

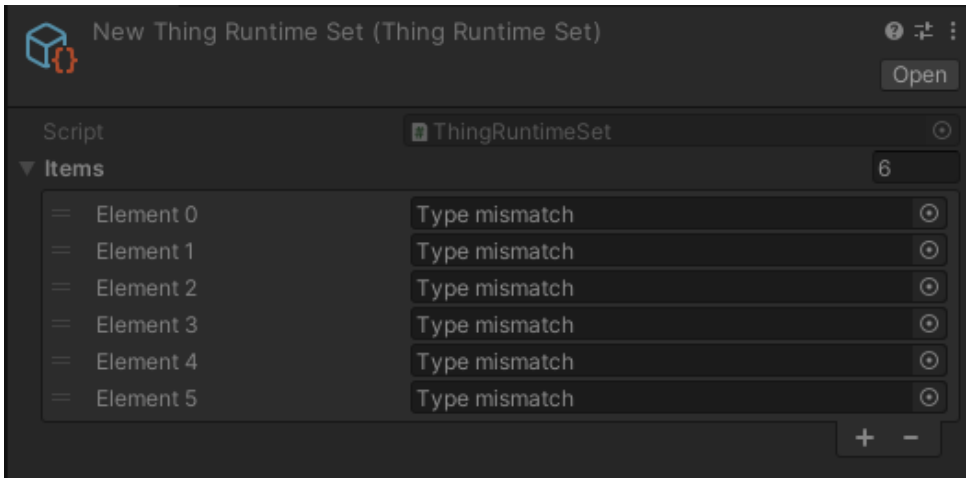
```
[CreateAssetMenu(menuName = "Foo Runtime Set",
    fileName = "FooRuntimeSet")]
public class FooRuntimeSetSO : RuntimeSetSO<Foo>
{
}
```

Создайте столько конкретных классов, сколько требуется игровой логике. Противники, NPC, предметы инвентаря, задания и другие сущности могут иметь собственные Runtime Set. Для этого достаточно объявить новый пустой класс с нужным типом.

Вместо событий каждый компонент Foo может самостоятельно добавлять и удалять себя в методах OnEnable и OnDisable. Если назначить ссылку в поле FooRuntimeSet окна Inspector, компонент Foo будет автоматически появляться в Runtime Set. Это особенно удобно при использовании компонента Foo с префабами.

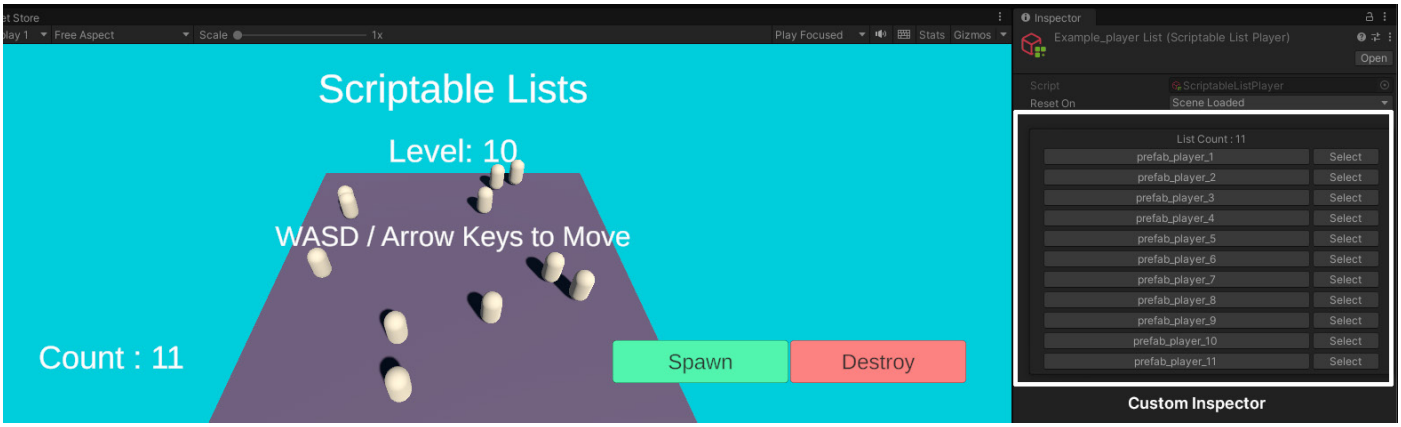
```
public class Foo : MonoBehaviour
 {
       public FooRuntimeSetSO RuntimeSet;

       private void OnEnable()
       {
           RuntimeSet.Add(this);
       }

       private void OnDisable()
       {
           RuntimeSet.Remove(this);
       }
 }
```

Примечание. У этого приема есть ограничение: если просматривать ScriptableObject во время выполнения, содержимое списка Runtime Set не будет видно в окне Inspector. Если сделать список общедоступным и вывести его в окно Inspector, результат будет таким:



Runtime Set не отображает объекты и компоненты сцены в окне Inspector.

По умолчанию в поле каждого элемента отображается Type mismatch («Несоответствие типов»), поскольку ассет ScriptableObject не может сериализовать ссылку на объект сцены. Список при этом работает нормально, но его содержимое отображается некорректно. Чтобы не вводить пользователя в заблуждение и скрыть список из окна Inspector, предоставьте к нему доступ через общедоступное свойство или примените атрибут HideInInspector. Проблему отображения также можно решить с помощью скрипта пользовательского Editor, который настраивает отображение в окне Inspector. Хороший пример приведен в SOAP (ScriptableObject Architectural Pattern) в Asset Store.



Пользовательский Editor в SOAP отображает содержимое Runtime Set.

Интересные факты о foo и bar

В программировании foo и bar часто используют как условные имена. Вероятно, их выбрали потому, что они короткие, легко запоминаются и необычно звучат.

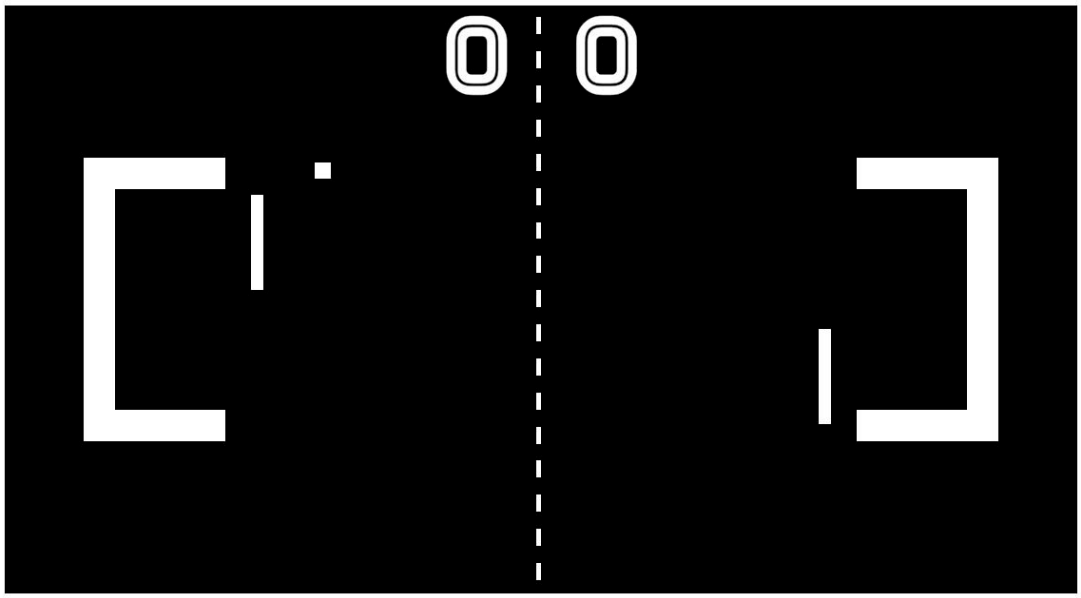
Точное происхождение этих терминов неизвестно. По одной из версий, их начали использовать операторы радиолокационных станций во время Второй мировой войны. Бессмысленное слово foo также стало крылатой фразой в комиксе 1930-х годов.

В программировании распространение foo и bar обычно связывают с клубом железнодорожного моделизма Tech Model Railroad Club при MIT примерно в 1960-х годах. В помещении клуба у двери находились две универсальные кнопки с надписями foo и bar. Хакеры MIT часто заимствовали эти названия для своих идей, и со временем foo и bar стали общими именами переменных. Сегодня их использование в качестве условных имен широко распространено среди программистов.

Знакомство с примером проекта

К этому руководству прилагается пример проекта, на котором можно изучить ScriptableObject. Проект основан на классической механике игры с мячом и ракетками и наглядно показывает, как ScriptableObject может улучшить архитектуру проекта Unity.

Применяя принципы SOLID и подходы, характерные для крупных проектов, вы увидите, как ScriptableObject помогает перестраивать код.



Пример проекта

Здесь вы увидите применение ScriptableObject на практике и лучше поймете, как с его помощью повысить эффективность разработки и улучшить организацию проектов.

В примере реализованы многие паттерны, рассмотренные в этом руководстве:

- Контейнеры данных: общие настройки вынесены в ассеты ScriptableObject. Параметры игрового процесса и общие состояния хранятся в проекте как ассеты. Исходную компоновку уровня также можно изменить, заменив соответствующие ассеты ScriptableObject.

- Расширяемые перечисления: пустые ассеты ScriptableObject выполняют роль перечислений для классификации игроков.

- Объекты-делегаты: простой аудиоделегат показывает, как рандомизировать звуки столкновения мяча одной лишь заменой ScriptableObject. Цели также представлены ассетами ScriptableObject, которые подключаются к системе управления игрой для определения условий победы и поражения.

- Каналы событий: паттерн «Наблюдатель» помогает настроить игровые события для UI, звука и подсчета очков. Разные объекты GameObject могут подписываться на разные каналы событий, словно настраиваясь на определенную радиостанцию.

- Двойная сериализация: некоторые игровые данные, например компоновка уровня, хранятся в ScriptableObject для удобства работы в редакторе Unity, но при необходимости сохраняются в файлы JSON. Модифицированные вне Unity данные JSON затем можно использовать для воссоздания ScriptableObject, совместимого с исходным скриптом настройки.

Разумеется, сама игра не является главной частью примера. Аркадную игру с мячом и ракетками можно написать гораздо меньшим объемом кода. Цель примера проекта в другом: показать, как ScriptableObject помогает создавать тестируемые и масштабируемые компоненты, с которыми удобно работать дизайнерам.

Заключение

ScriptableObject может стать универсальным дополнением к вашему набору инструментов. Рассматривайте эти паттерны проектирования как дополнительные способы организовать разработку в Unity.

Многие приемы из этого руководства можно реализовать с помощью классов C#, не используя ScriptableObject, и в некоторых ситуациях разработчики предпочтут именно такой подход. Однако редактор Unity позволяет удобнее просматривать и редактировать ScriptableObject. Благодаря этому художники и дизайнеры могут взаимодействовать с проектом, а не полагаться во всем на код.

Хотят ли ваши дизайнеры самостоятельно настраивать данные игрового процесса, не обращаясь постоянно к разработчикам? Если да, ScriptableObject может найти место в вашем проекте.

Иногда отказ от паттерна не менее ценен, чем его применение. Решение, которое естественно подходит одному приложению, может не подойти другому. Перед внедрением паттерна оцените его преимущества и недостатки.

Строгих правил организации проекта Unity не существует. Согласуйте архитектуру кода с навыками и личными предпочтениями команды.

Тогда вы сможете сосредоточиться на главном: сделать игру по-настоящему увлекательной для игроков.

Дополнительные материалы

Документация

- API скриптинга ScriptableObject

- Страница руководства по ScriptableObject

Технические электронные книги Unity

О паттернах проектирования и программировании в Unity

- «Повышаем уровень кода с паттернами проектирования и SOLID»: введение в SOLID-разработку и паттерны проектирования в Unity.

- «Создание руководства по стилю C# (издание для Unity 6)»: обзор ряда передовых практик организации проекта.

Другие руководства по передовым практикам

- Расширенные руководства по передовым практикам в документации Unity

- Центр передовых практик Unity

Материалы Unite

- «Свержение тирании MonoBehaviour в славной революции ScriptableObject»

- «Архитектура игры с помощью ScriptableObject»

Для общего знакомства с архитектурой кода также рекомендуем:

- «От Pong до проекта команды из 15 человек»

Другие примеры проектов

В архитектуре этих проектов сообщества широко применяется подход на основе ScriptableObject:

- Презентация демонстрационного проекта Tanks с Unite, подготовленная Ричардом Файном, использует ScriptableObject для настройки звука, подключаемого ИИ и разрушаемых зданий.

- У Райана Хиппла, старшего разработчика игр полного цикла в Meta, есть проект на GitHub, иллюстрирующий значительную часть его выступления на Unite Austin. Также можно прочитать его публикацию в блоге на ту же тему.

- SOAP (ScriptableObject Architectural Pattern) в Asset Store реализует многие паттерны, описанные в этой электронной книге, и добавляет множество функций, упрощающих работу с ScriptableObject.

- Unity Atoms: библиотека с открытым исходным кодом, активно использующая ScriptableObject. Начать работу с ней поможет соответствующая страница.

- Reactive Menu System использует ScriptableObject для построения системы UGUI, которая обрабатывает изменения состояния UI и управление меню.

- Unity Open Project (Chop Chop) активно опирается на ScriptableObject и управляет игровым процессом с помощью каналов событий.

Для гейм-дизайнеров

Кристо Ноббс, старший технический гейм-дизайнер, специализирующийся на проектировании игровых систем и Unity (C#), участвовал в создании «Справочника гейм-дизайнера Unity» и был основным автором небольшой серии публикаций о проектировании игровых систем в Unity:

- «Системы, создающие экосистемы: эмерджентный гейм-дизайн»

- «Непредсказуемо и увлекательно: ценность рандомизации в гейм-дизайне»

- «Кривые анимации как главный рычаг дизайна»

Профессиональное обучение

Unity Professional Training проводит обучение онлайн и очно, помогая вам и вашей команде приобрести дополнительные навыки и знания для более продуктивной работы и эффективного сотрудничества в Unity. Подробнее о Unity Professional Training можно узнать здесь.

unity.com