→ ЭЛЕКТРОННАЯ КНИГА

Руководство по стилю C# для чистого и масштабируемого игрового кода

Unity®

Содержание

Введение

Чистый код всегда выглядит так, будто его написал человек, которому не все равно.

— Майкл Физерс, автор книги «Эффективная работа с унаследованным кодом»

Большинство разработчиков игр согласны: код должен быть чистым — легко читаться, сопровождаться и использоваться повторно. Когда логика уже работает, наступает время рефакторинга и наведения порядка.

У такого единодушия есть веские причины. То, что очевидно вам как автору кода, другому разработчику может быть непонятно. Да и вы сами через три месяца можете уже не вспомнить, что делает написанный сегодня фрагмент.

Вероятно, у вас уже сложились собственные предпочтения в написании кода. Но у будущих коллег они наверняка окажутся другими. Единственно правильного подхода нет, поэтому в этом руководстве собраны советы о том, как с помощью руководства по стилю согласовать общие стандарты.

В этом руководстве собраны рекомендации отраслевых экспертов по использованию и адаптации общепринятых руководств по стилю кода. Единое руководство для всей команды позволит кодовой базе расти вместе с проектом вплоть до коммерческого масштаба.

Эти рекомендации принесут пользу процессу разработки в долгосрочной перспективе, даже если поначалу потребуют дополнительных усилий. Чистая и масштабируемая кодовая база также облегчает адаптацию новых разработчиков по мере роста команды.

Поддерживайте код в чистоте — так вы облегчите жизнь и себе, и всем участникам проекта.

Чистый код помогает масштабировать разработку и соблюдать производственные стандарты, в том числе следующие:

— Придерживайтесь единых соглашений об именовании

— Форматируйте код так, чтобы его было легко читать

— Делайте классы и методы небольшими и понятными

— Комментируйте код, смысл которого неочевиден

Создаете ли вы мобильную головоломку или масштабную MMORPG для консолей, чистая кодовая база снижает совокупную стоимость сопровождения программного обеспечения. В результате внедрять новые возможности и выпускать исправления для существующего продукта становится значительно проще.

За это вам будут благодарны и будущие коллеги, и вы сами в будущем.

Примечание. Это второе издание электронной книги, обновленное для Unity 6.

Авторы

Руководство написал независимый разработчик игр и преподаватель Уилмер Лин. Значительный вклад также внесли менеджер по маркетингу технического контента Томас Крог-Якобсен и старшие инженеры Unity Питер Андреасен, Скотт Билас и Михай Попеску.

Командная разработка

Любой дурак способен написать код, понятный компьютеру. Хорошие программисты пишут код, понятный людям.

— Мартин Фаулер, автор книги «Рефакторинг»

Разработчик не существует в изоляции. По мере роста технических требований игры вам понадобится помощь, и команда неизбежно пополнится специалистами с разными навыками. Чистый код задает стандарты для растущей команды и помогает всем придерживаться общего подхода. Благодаря этому каждый может работать над одним проектом по единым правилам.

Прежде чем переходить к конкретным руководствам по стилю кода, вспомним несколько общих, не зависящих от стиля правил, которые помогут масштабировать разработку на Unity.

KISS («не усложняй»)

Признаем: инженеры и разработчики склонны все усложнять, хотя вычисления и программирование и без того непросты. Пусть принцип KISS — «не усложняй» — помогает находить самое простое решение текущей задачи.

Не нужно изобретать велосипед, если задачу уже решает проверенный и простой прием. В Scripting API Unity есть множество готовых решений. Например, если для вашей стратегии подходит Hexagonal Tilemap, а UnityEngine.Pool удовлетворяет требованиям к пулу объектов, не пишите собственную реализацию. Лучший код — тот, который не пришлось писать.

Принцип KISS

Известный принцип KISS ставит во главу угла простоту проектирования. Эта идея была популярна в разные эпохи, что подтверждают следующие высказывания:

«Простота — высшая степень изощренности».

— Леонардо да Винчи

«Делайте простые задачи простыми!»

— Бьёрн Страуструп

«Простота — необходимое условие надежности».

— Эдсгер В. Дейкстра

«Все следует упрощать настолько, насколько возможно, но не более того».

— Альберт Эйнштейн

В программировании это означает: делайте код как можно проще и не вносите в него лишнюю сложность.

Принцип YAGNI

Связанный с ним принцип YAGNI («вам это не понадобится») предписывает реализовывать возможности только тогда, когда они действительно нужны. Не тратьте силы на функции, которые, возможно, пригодятся при идеальном стечении обстоятельств. Создайте самое простое решение, необходимое сейчас, и добейтесь его работоспособности.

Не маскируйте проблему кодом

Первый шаг разработки программного обеспечения — понять, какую именно проблему вы решаете. Это кажется очевидным, однако разработчики слишком часто погружаются в реализацию, не разобравшись в сути задачи, или меняют код, пока он не заработает, так и не поняв почему.

Допустим, вы устранили NullReferenceException, добавив в начало метода простую проверку на null. Уверены ли вы, что причина была именно в этом, а не в вызове другого метода где-то глубже в коде?

Не добавляйте код лишь для того, чтобы скрыть симптом. Исследуйте первопричину и спросите себя, почему возникает ошибка, вместо того чтобы ставить временную заплатку.

Улучшайте понемногу каждый день

Создание чистого кода — непрерывный итерационный процесс. Вся команда должна разделять этот подход и воспринимать наведение порядка в коде как часть повседневной работы разработчика. Обычно никто не намерен писать плохой код — он постепенно становится таким со временем. Кодовая база требует постоянного ухода, поэтому заранее выделяйте на это время и действительно выполняйте такую работу.

Делайте хорошо, а не идеально

В то же время не стремитесь к недостижимому совершенству. Когда код соответствует производственным стандартам, зафиксируйте изменения и двигайтесь дальше.

В конечном счете код должен решать задачу. Соблюдайте баланс между реализацией новых возможностей и наведением порядка. Не проводите рефакторинг ради самого рефакторинга — делайте это, когда он принесет пользу вам или другим разработчикам.

Планируйте, но адаптируйтесь

В книге «Программист-прагматик» Энди Хант и Дэйв Томас пишут: «Программирование больше похоже не на строительство, а на садоводство». Разработка программного обеспечения — живой процесс, поэтому будьте готовы адаптироваться, когда все идет не по плану.

Даже самый подробный план сада на бумаге не гарантирует желаемого результата: растения могут цвести не так, как вы ожидали. Чтобы ваш сад процветал, придется подрезать, пересаживать и заменять отдельные части кода.

Проектирование программного обеспечения не вполне похоже на архитектурные чертежи: оно гибче и менее механистично. По мере роста кодовой базы вам придется реагировать на изменения.

Будьте последовательны

Выбрав способ решения задачи, применяйте тот же подход в похожих ситуациях. Это несложно, но требует постоянной дисциплины. Следуйте этому принципу во всем: от именования классов и методов и выбора стиля регистра до организации папок и ресурсов проекта.

Главное — согласуйте руководство по стилю всей командой и затем соблюдайте его.

Качество кода — общее дело

Как говорит американский инженер-программист и автор Роберт К. Мартин, известный также как «Дядя Боб», чистый и простой код отвечает интересам всех, но «чисто и просто» не означает «легко». Такая работа требует усилий как от начинающих, так и от опытных разработчиков.

Если ничего не контролировать, проект неизбежно придет в беспорядок. Это естественное следствие того, что множество людей работают над разными его частями. Предотвращать захламление кода обязаны все, поэтому каждый участник команды должен прочитать руководство по стилю и следовать ему.

Следуйте руководству по стилю

В информатике есть только две по-настоящему сложные задачи: инвалидация кеша и именование сущностей.

— Фил Карлтон, инженер-программист

Руководство по стилю для вас и вашей команды

В этом руководстве рассматриваются наиболее распространенные соглашения, с которыми вы столкнетесь при разработке на Unity. В основном это подмножество рекомендаций Microsoft Framework Design Guidelines, содержащих гораздо больше правил, чем представлено здесь.

Это рекомендации, а не незыблемые правила. Адаптируйте их к предпочтениям своей команды. Выберите стиль, который устраивает всех, и следите за его соблюдением.

Главное — единообразие. Если в будущем руководство по стилю потребуется изменить, несколько операций поиска и замены позволят быстро обновить всю кодовую базу.

Если ваше руководство по стилю расходится с этим документом или Microsoft Framework Design Guidelines, приоритет следует отдать собственному руководству: так команда сохранит единый стиль во всем проекте.

Приложение — общий результат труда людей, которые могут мыслить по-разному. Руководство по стилю сглаживает эти различия и помогает создать цельный продукт. Сколько бы участников ни работало над проектом Unity, код должен выглядеть так, словно его написал один автор.

Руководство по стилю устраняет неопределенность в соглашениях о кодировании и форматировании. Поддерживать единообразие становится просто: достаточно следовать установленным правилам.

Если вы работаете в одиночку, поначалу это может казаться ограничением, однако в командной разработке руководство по стилю необходимо.

Считайте руководство по стилю первоначальным вложением, которое окупится позднее. Единый набор стандартов сокращает время на повторное освоение правил, когда разработчик переходит в другой проект.

Microsoft и Google предлагают подробные руководства-примеры, которые принято считать отраслевыми стандартами:

— Соглашения Microsoft по написанию кода C#

— Руководство Google по стилю C#

Они служат отличной отправной точкой для разработки на Unity и предлагают решения по именованию, форматированию и комментированию кода.

В Unity мы не навязываем единый стиль кода всем учебным материалам для начинающих, документации и примерам проектов. Строгое руководство может усложнить обучение новичкам, тогда как большим командам и сложным проектам нужны более жесткие правила. У человека, который только осваивает Unity и, возможно, свой первый язык программирования, совсем иные потребности, чем у инженерной организации из более чем 2000 сотрудников.

Поэтому здесь используется стиль, расширяющий Microsoft Framework Design Guidelines, где определен ряд правил, не вошедших в этот документ. Он близок к стилю наших инженерных команд и авторов документации и, на наш взгляд, подходит большинству проектов Unity: в нем прагматизм поставлен выше формальной строгости.

В конечном счете важно выбрать подход, который устраивает вас и вашу команду, и последовательно применять его во всем проекте.

Мы подготовили пример руководства по стилю C#, которым можно пользоваться при создании собственного. Свободно копируйте и изменяйте его под свои задачи.

Перейдем к делу.

Соглашения об именовании

В выборе имени заключено больше смысла, чем кажется. Имя показывает, какое место сущность занимает в системе: что это, к чему относится и какую роль выполняет.

Имена переменных, классов и методов — не просто метки: они несут смысл. Удачные соглашения об именовании напрямую влияют на то, насколько легко читатель программы поймет заложенную в нее идею.

При выборе имен учитывайте следующие рекомендации.

Имена идентификаторов

Идентификатор — это любое имя, присвоенное типу (классу, интерфейсу, структуре, делегату или перечислению), члену типа, переменной либо пространству имен.

Хотя C# допускает специальные знаки, символы Unicode и обратную косую черту в идентификаторах, избегайте их. Они могут мешать работе некоторых инструментов командной строки Unity. Необычные символы также снижают совместимость с разными платформами.

Стили регистра

В имени переменной нельзя использовать пробелы: C# разделяет ими идентификаторы. Для составных имен и фраз в исходном коде применяют разные стили регистра. Существует несколько общепринятых соглашений об именовании и регистре.

Стиль camelCase (camelCase)

В стиле camelCase фразы записывают без пробелов и знаков препинания, а каждое следующее слово начинают с прописной буквы. Первая буква остается строчной. Так оформляют локальные переменные и параметры методов. Например:

```
examplePlayerController
maxHealthPoints
endOfFile
```

Стиль PascalCase (PascalCase)

PascalCase — разновидность camelCase, в которой первая буква тоже прописная. При разработке на Unity этот стиль используют для имен классов, открытых полей и методов. Например:

```
ExamplePlayerController
MaxHealthPoints
EndOfFile
```

Стиль snake_case (snake_case)

В этом стиле пробелы между словами заменяют символами подчеркивания. Например:

```
example_player_controller
max_health_points
end_of_file
```

Стиль kebab-case (kebab-case)

Здесь пробелы между словами заменяют дефисами, словно нанизывая слова на «шампур». Например:

```
example-player-controller
Max-health-points
end-of-file
naming-conventions-methodology
```

Стиль kebab-case широко применяется в веб-технологиях, особенно в CSS. Мы также рекомендуем использовать его в USS для UI Toolkit, о чем подробнее расскажем далее.

Венгерская нотация

В таком имени переменной или функции обычно кодируется ее назначение либо тип. Например:

```
int iCounter
string strPlayerName
```

Венгерская нотация — устаревшее соглашение, которое редко применяется при разработке на Unity.

Поля и переменные

При именовании переменных и полей соблюдайте следующие правила:

— Используйте существительные для имен переменных. Имя должно быть содержательным, понятным и однозначным, поскольку переменная обозначает объект или состояние. Исключение составляют переменные типа bool, о которых сказано ниже.

— Начинайте имена логических переменных с глагола. Такие переменные содержат true или false и часто отвечают на вопрос: бежит ли игрок, окончена ли игра? Глагол делает смысл имени очевиднее. Обычно за ним следует описание или условие, например isDead, isWalking или hasDamageMultiplier.

— Выбирайте осмысленные имена и не сокращайте их без необходимости, кроме общепринятых математических обозначений. Имя должно ясно выражать назначение, легко произноситься и находиться поиском. Это важно не только для коллег: содержательные имена дают больше контекста инструментам ИИ и помогают им точнее генерировать код и рекомендации. Например, свойство HorizontalAlignment читается лучше, чем AlignmentHorizontal.

Однобуквенные переменные допустимы в циклах и математических выражениях, но в остальных случаях не сокращайте имена. Ясность важнее нескольких секунд, сэкономленных на пропущенных буквах.

При прототипировании возникает соблазн использовать короткие бессодержательные имена, однако это не сэкономит время, если позднее код придется рефакторить. Выбирайте осмысленные имена с самого начала.

Не рекомендуется	Рекомендуемый вариант	Примечания
`int d`	`int elapsedTimeInDays`	Избегайте однобуквенных сокращений, кроме счётчиков и выражений. Указывайте единицу измерения.

`int hp,`
`string tName,`
`int mvmtSpeed`
	`int healthPoints,`
	`string teamName,`
	`int movementSpeed`
		Имена переменных должны передавать смысл. Выбирайте легко произносимые имена, по которым удобно выполнять поиск.

`int getMovement-` `Speed`	`int movementSpeed`	Используйте существительные. Глаголы оставляйте для методов, кроме логических переменных (см. ниже).
`bool dead`	`bool isDead,` `bool isPlayerDead`	Имена логических переменных формулируйте как вопрос, на который можно ответить true или false.

— Используйте PascalCase (MyPropertyName) для открытых полей, а camelCase (myPrivateVariable) — для закрытых переменных. Вместо открытых полей можно применять свойства с открытым методом доступа get (см. раздел «Форматирование» ниже).

— Рассмотрите возможность использования префиксов или иной системы обозначений. Некоторые руководства советуют начинать имена закрытых полей с подчеркивания (_), чтобы отличать их от локальных переменных. В наших руководствах используются префиксы m_ для закрытых полей, k_ для констант и s_ для статических переменных: так назначение переменной видно сразу. Например, movementSpeed превращается в m_movementSpeed.

> Допустимо сочетать префикс с PascalCase, например m_MovementSpeed, однако в современном C# такой вариант встречается реже.
>
> Другой вариант — различать поля и локальные переменные по ключевому слову this и отказаться от префикса. У открытых полей и свойств префиксов обычно нет. Локальные переменные и параметры оформляют в camelCase без префикса.
>
> Многие разработчики отказываются от префиксов и полагаются на редактор: современные IDE поддерживают подсветку, цветовое оформление и подробную контекстную информацию.

— Поля автоматически получают значения по умолчанию. Для числовых типов, например int, это обычно 0; поля ссылочных типов, например объектов, инициализируются значением null, а поля bool — значением false. Поэтому явно присваивать полю его значение по умолчанию обычно не требуется.

— Именуйте константы в PascalCase и добавляйте префикс k_. Это позволяет отличать их от обычных переменных и свойств, а также упрощает чтение и сопровождение кода.

```
// ПРИМЕР: константы
public class MathConstants
{
    public const int k_MaxItems = 100;
}
```

— Последовательно указывайте или опускайте модификаторы доступа. Если модификатор не задан, компилятор считает уровень доступа private. Это допустимо, но выбранного подхода нужно придерживаться во всем коде.

> В рекомендациях Microsoft предлагается явно указывать private, чтобы уровень доступа был очевиден и не возникало неоднозначности. Другие руководства советуют опускать избыточные модификаторы (не писать private в области типа) и инициализаторы (например, = 0 для int и = null для ссылочных типов). Помните: если член позднее понадобится подклассу, потребуется protected. Мы рекомендуем ради простоты опускать неявные и потому избыточные элементы, такие как private, если это не ухудшает читаемость для вашей команды.

— Ставьте читаемость выше краткости. Как показывает пример из документации Microsoft, имя свойства CanScrollHorizontally лучше, чем ScrollableX, поскольку в имени ScrollableX неясно, что X обозначает горизонтальную ось.

Примеры фрагментов кода

Фрагменты кода в этом руководстве сокращены и не предназначены для выполнения. Они демонстрируют только стиль и форматирование.

Можно также обратиться к этому примеру руководства по стилю C# для разработчиков Unity — измененной версии Microsoft Framework Design Guidelines. Это лишь один из возможных вариантов организации командного руководства.

Просмотрите каждое правило примера и адаптируйте его к предпочтениям команды. Детали отдельного правила менее важны, чем общее согласие соблюдать его последовательно. При разногласиях в вопросах стиля опирайтесь на руководство своей команды.

```
// ПРИМЕР: открытые и закрытые переменные сгруппированы
public float DamageMultiplier = 1.5f;
public float MaxHealth;
public bool IsInvincible;
private bool m_isDead;
```

```
private float m_currentHealth;

public void InflictDamage(float damage, bool isSpecialDamage)
{
    // локальная переменная
    int totalDamage = damage;
    // локальная переменная и открытое поле
    if (isSpecialDamage)
    {
       totalDamage *= DamageMultiplier;
    }
    // локальная переменная и закрытое поле
    if (totalDamage > _currentHealth)
    {
        /// ...
    }
}
```

— Объявляйте по одной переменной в строке. Код получится менее компактным, но более читаемым.

— Избегайте избыточных имен. Если класс называется Player, его поля не нужно называть PlayerScore или PlayerTarget — достаточно Score и Target.

— Опускайте избыточные инициализаторы: не пишите = 0 для int, = null для ссылочных типов и тому подобное.

— Избегайте шуток и каламбуров. Имена вроде infiniteMonkeys или dudeWheresMyChar могут вызвать улыбку сейчас, но быстро утомят после нескольких десятков прочтений. Что еще важнее, они противоречат цели выбирать имена, раскрывающие контекст.

— Избегайте неоднозначности и повышайте читаемость, но используйте var, когда тип очевиден из контекста. При удачных именах переменных намерение и так понятно. Рефакторинг с var проще: конкретный тип абстрагирован, поэтому при его изменении нужно обновлять меньше участков кода. В циклах foreach var гарантирует соответствие переменной итерации типу элементов, выдаваемых перечислителем. Явно указанный несовместимый тип компилятор иногда допускает, что приводит к ошибкам во время выполнения.

```
// ПРИМЕР: уместное использование var
var powerUps = new List<PowerUps>();
var dictionary = new Dictionary<string, List<GameObject>>();

// НЕ РЕКОМЕНДУЕТСЯ: возможна неоднозначность
var powerUps = PowerUpManager.GetPowerUps();
```

Перечисления

Перечисления — это особые типы значений, определенные набором именованных констант. По умолчанию константы имеют тип int и нумеруются начиная с 0.

Оформляйте имена перечислений и их значений в PascalCase. Открытое перечисление можно объявить вне класса, сделав доступным глобально. Имя перечисления должно быть существительным в единственном числе, поскольку обозначает одно значение из набора. Не добавляйте к нему префиксы или суффиксы.

Примечание. Исключение составляют битовые перечисления с атрибутом System.FlagsAttribute. Их имена обычно ставят во множественное число, поскольку значение может представлять сразу несколько вариантов.

```
// ПРИМЕР: имя перечисления – существительное в единственном числе
public enum WeaponType
{
    Knife,
    Gun,
    RocketLauncher,
    BFG
}

public enum FireMode
{
    None = 0,
    Single = 5,
    Burst = 7,
    Auto = 8,
}

// ПРИМЕР: имя перечисления флагов – во множественном числе (допустимо 1 << bitnum)

[Flags]
public enum AttackModes
{
  // Десятичное                    // Двоичное
   None = 0,                        // 000000
   Melee = 1,                       // 000001
   Ranged = 2,                      // 000010
   Special = 4,                     // 000100

   MeleeAndSpecial = Melee | Special  // 000101
}
```

Классы и интерфейсы

При именовании классов и интерфейсов соблюдайте следующие стандартные правила:

— Именуйте классы существительными или именными словосочетаниями в PascalCase. Так имена типов отличаются от методов, которые называют глагольными выражениями.

— Если файл содержит MonoBehaviour, имя исходного файла должно совпадать с именем этого класса. В файле могут находиться и другие внутренние классы, но MonoBehaviour должен быть только один.

— Начинайте имя интерфейса с прописной I, а затем добавляйте прилагательное, описывающее его назначение.

```
// ПРИМЕР: оформление класса
public class ExampleClass : MonoBehaviour
{

    public int PublicField;
    public static int MyStaticField;
    private int m_packagePrivate;
    private int m_myPrivate;

    private static int m_myPrivate;

    protected int m_myProtected;

    public void DoSomething()
    {

    }
}

// ПРИМЕР: интерфейсы
public interface IKillable
{
    void Kill();
}

public interface IDamageable<T>
{
    void Damage(T damageTaken);
}
```

Методы

В C# каждая выполняемая инструкция находится в контексте метода.

Примечание. В разработке на Unity слова «функция» и «метод» нередко употребляют как синонимы. Однако в C# функцию нельзя написать вне класса, поэтому принят термин «метод».

Методы выполняют действия, поэтому именуйте их по следующим правилам:

— Начинайте имя с глагола или глагольного выражения и при необходимости уточняйте контекст, например GetDirection или FindTarget.

— Используйте camelCase для параметров. Оформляйте параметры метода так же, как локальные переменные.

— Методы, возвращающие bool, должны задавать вопрос. Как и имена логических переменных, начинайте такие методы с глагола, чтобы имя выражало условие true или false, например IsGameOver или HasStartedTurn.

```
// ПРИМЕР: имя метода начинается с глагола
public void SetInitialPosition(float x, float y, float z)
{
    transform.position = new Vector3(x, y, z);
}

// ПРИМЕР: метод, возвращающий bool, задает вопрос
public bool IsNewPosition(Vector3 currentPosition)
{
    return (transform.position == newPosition);
}
```

События и обработчики событий

События C# реализуют паттерн «Наблюдатель». Он определяет отношения, при которых один объект — субъект, или издатель, — уведомляет зависимые объекты, называемые наблюдателями, или подписчиками. Так субъект сообщает наблюдателям об изменении состояния без жесткой связанности между объектами. Подробнее о «Наблюдателе» и других паттернах для проектов Unity рассказывается в электронной книге «Совершенствуйте код с помощью шаблонов проектирования и SOLID».

Для событий и связанных с ними методов субъекта и наблюдателей существует несколько схем именования. Рекомендуем следующие приемы:

— Называйте событие глагольным выражением, точно передающим изменение состояния. Причастием настоящего или прошедшего времени обозначайте событие «до» или «после». Например, OpeningDoor — событие перед открытием двери, а DoorOpened — после него.

— Используйте для событий делегаты System.Action. В большинстве игровых сценариев достаточно Action либо обобщенных вариантов Action<T...>. Они позволяют передавать от 0 до 16 параметров разных типов и не возвращают значения (void). Готовые делегаты сокращают объем кода.

Примечание. Можно также использовать делегаты EventHandler и EventHandler<TEventArgs>. Команда должна заранее договориться о едином способе реализации событий.

```
// ПРИМЕР: события

// используется делегат System.Action

public event Action OpeningDoor;    // событие «до»
public event Action DoorOpened;     // событие «после»

public event Action<int> PointsScored;
public event Action<CustomEventArgs> ThingHappened;
```

— Начинайте имя метода, вызывающего событие в субъекте, с On. Субъект обычно вызывает событие из метода с таким префиксом, например OnOpeningDoor или OnDoorOpened.

```
// вызывает событие, если есть подписчики
public void OnDoorOpened()
{
    DoorOpened?.Invoke();
}

public void OnPointsScored(int points)
{
    PointsScored?.Invoke(points);
}
```

— Рассмотрите схему, в которой имя метода обработки события в наблюдателе начинается с имени субъекта и символа подчеркивания (_). Если субъект называется GameEvents, методы наблюдателей могут называться GameEvents_OpeningDoor и GameEvents_DoorOpened.

Такой метод называется методом обработки события; не путайте его с делегатом EventHandler.

— Создавайте собственный EventArgs только при необходимости. Если событию нужно передавать пользовательские данные, определите новый тип EventArgs, унаследованный от System.EventArgs, либо пользовательскую структуру.

```
// при необходимости определите EventArgs

// ПРИМЕР: неизменяемая пользовательская структура для передачи ID и Color
public struct CustomEventArgs
{
    public int ObjectID { get; }
    public Color Color { get; }

      public CustomEventArgs(int objectId, Color color)
      {
          this.ObjectID = objectId;
          this.Color = color;
      }
}
```

Пространства имен

Используйте пространства имен, чтобы классы, интерфейсы, перечисления и другие типы не конфликтовали с одноименными сущностями из других пространств имен или глобального пространства. Они также предотвращают конфликты со сторонними ресурсами из Asset Store.

При использовании пространств имен:

— Используйте PascalCase без специальных символов и подчеркиваний.

— Добавляйте директиву using в начало файла, чтобы не повторять префикс пространства имен.

— Создавайте вложенные пространства имен. Разделяйте уровни оператором точки (.), чтобы организовать скрипты по иерархическим категориям. Например, логические компоненты игры можно распределить между MyApplication.GameFlow, MyApplication.AI, MyApplication.UI и другими пространствами.

— Некоторые разработчики строят пространства имен по структуре папок проекта. Логическая группировка связанных классов и компонентов упрощает поиск и понимание организации кодовой базы.

```
namespace Enemy
{
    public class Controller1 : MonoBehaviour
    {
        ...
    }

    public class Controller2 : MonoBehaviour
    {
        ...
    }
}
```

В коде эти классы обозначаются соответственно как Enemy.Controller1 и Enemy.Controller2. Чтобы не вводить префикс каждый раз, добавьте директиву using:

```
using Enemy;
```

Встретив имена классов Controller1 и Controller2, компилятор поймет, что речь идет об Enemy.Controller1 и Enemy.Controller2.

Если скрипт должен обращаться к одноименным классам из разных пространств имен, различайте их по префиксу. Например, если классы Controller1 и Controller2 находятся также в пространстве имен Player, указывайте полные имена Player.Controller1 и Player.Controller2, чтобы избежать конфликта. Иначе компилятор сообщит об ошибке.

Форматирование

Если хотите, чтобы код было легко писать, сделайте его удобным для чтения.

— Роберт К. Мартин, автор книг «Чистый код» и «Гибкая разработка программного обеспечения»

Чем меньше вы думаете о форматировании, тем больше времени остается на другие задачи.

Наряду с именованием форматирование уменьшает количество догадок и делает код понятнее. При едином руководстве по стилю во время проверки кода обсуждают прежде всего то, как он работает, а не как выглядит.

Исключайте, дополняйте или изменяйте предлагаемые правила с учетом потребностей команды.

Продумайте, как команда будет применять каждое правило форматирования, и добейтесь единообразия. При разногласиях сверяйтесь с принятым в команде стилем.

Учитывайте следующие рекомендации по форматированию кода при подготовке руководства по стилю разработки на Unity.

Свойства

Свойство — это гибкий механизм чтения, записи и вычисления значений класса. Свойства выглядят и используются как открытые поля, но фактически представляют собой специальные методы доступа. Свойство может иметь методы доступа get и set, работающие с закрытым полем — резервным полем (backing field).

Так свойство инкапсулирует данные и защищает их от нежелательных изменений пользователем или внешними объектами. У методов get и set могут быть разные модификаторы доступа, поэтому свойство может поддерживать чтение и запись, только чтение либо только запись.

Методы доступа также позволяют проверять или преобразовывать данные: например, контролировать формат значения или переводить его в нужные единицы.

Синтаксис свойств может различаться, поэтому руководство по стилю должно определять правила их оформления. Следующие рекомендации помогут сохранить единообразие:

— Для однострочных свойств только для чтения используйте тело выражения (=>): оно возвращает закрытое резервное поле.

```
// ПРИМЕР: свойства с телом выражения
public class PlayerHealth
{
    // резервное поле
    private int m_maxHealth;

    // только чтение: возвращает резервное поле
    public int MaxHealth => m_maxHealth;

    // эквивалентно:
    // public int MaxHealth { get; private set; }
}
```

— Во всех остальных случаях используйте обычный синтаксис { get; set; }. Если нужно открыть свойство без отдельного резервного поля, применяйте автоматически реализуемое свойство.

> Применяйте синтаксис с телом выражения и к методам доступа get и set.
>
> Если внешняя запись запрещена, сделайте set закрытым. В многострочных блоках закрывающая фигурная скобка должна быть выровнена с открывающей.

```
// ПРИМЕР: свойства с телом выражения
public class PlayerHealth
{
    // резервное поле
    private int m_maxHealth;

    // явная реализация методов get и set
    public int MaxHealth
    {
        get => m_maxHealth;
        set => m_maxHealth = value;
    }

    // только запись (без резервного поля)
    public int Health { private get; set; }

    // только запись, без явного set
    public SetMaxHealth(int newMaxValue) => _maxHealth = newMaxValue;
}
```

— Хотя закрытые данные можно открывать и через методы, как в примере ниже, для простых операций get/set обычно рекомендуются свойства. Для сложной логики или вычислений лучше использовать методы.

```
// ПРИМЕР: свойства с телом выражения
public class PlayerHealth
{
    // резервное поле
    private int m_maxHealth;

    public int GetMaxHealth
    {
        return m_maxHealth;
    }
}
```

Сериализация

Сериализация скриптов — это автоматическое преобразование структур данных или состояния объектов в формат, который Unity может сохранить и позднее восстановить. Из соображений производительности Unity выполняет сериализацию иначе, чем другие среды программирования.

Сериализованные поля отображаются в Inspector, однако статические, константные и доступные только для чтения поля сериализовать нельзя. Поле должно быть открытым либо помеченным атрибутом [SerializeField]. Unity поддерживает сериализацию лишь определенных типов полей; полный набор правил приведен в документации.

При работе с сериализованными полями соблюдайте несколько основных рекомендаций:

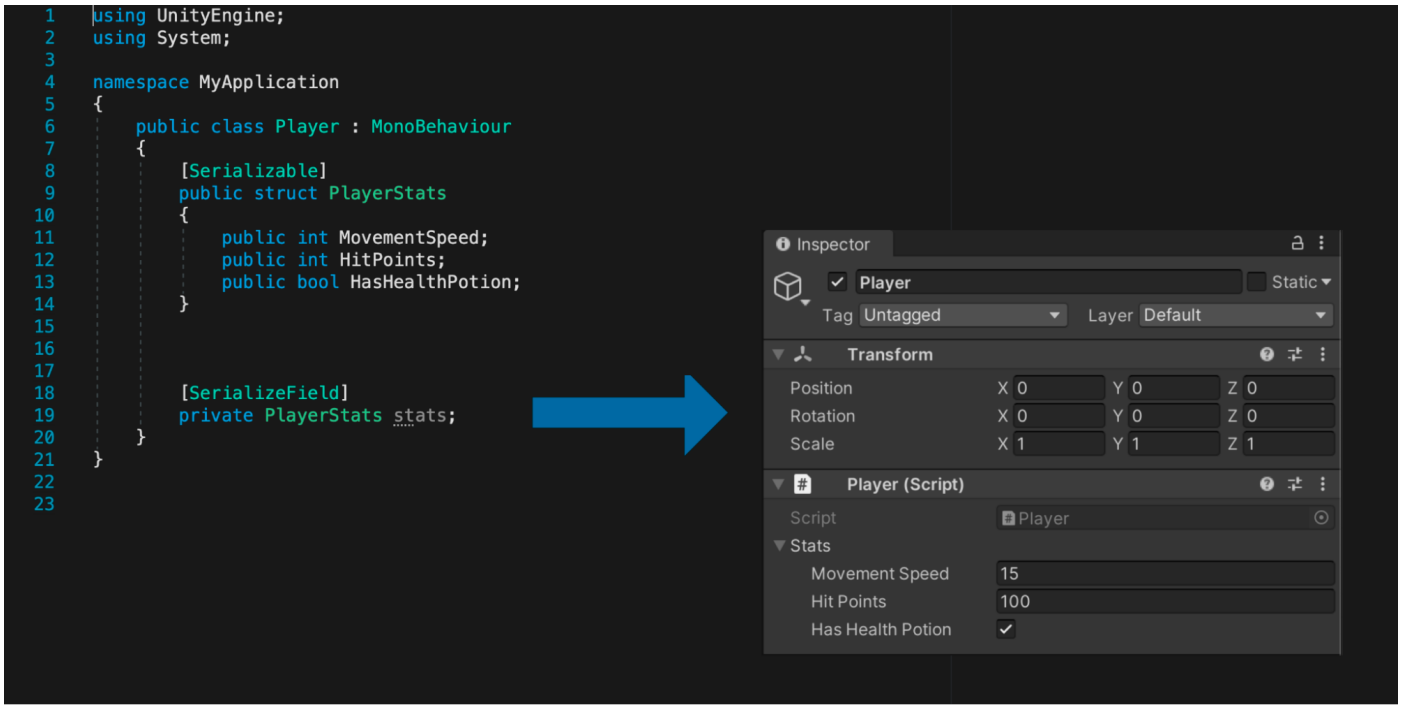
— Используйте атрибут [SerializeField]. Он позволяет отображать закрытые и защищенные переменные в Inspector. Это лучше инкапсулирует данные, чем объявление переменной открытой, и не дает внешнему объекту перезаписывать ее значения.

— Задавайте минимальное и максимальное значения атрибутом Range. Атрибут [Range(min, max)] ограничивает допустимое значение числового поля и удобно отображает его в Inspector в виде ползунка.

— Группируйте данные в сериализуемые классы или структуры, чтобы упорядочить Inspector. Объявите открытый класс или структуру, пометьте атрибутом [Serializable] и создайте открытые переменные всех типов, которые должны отображаться в Inspector.

```
// ПРИМЕР: сериализуемый класс PlayerStats

using System;
using UnityEngine;

public class Player : MonoBehaviour
{
    [Serializable]
    public struct PlayerStats
    {
        public int MovementSpeed;
        public int HitPoints;
        public bool HasHealthPotion;
    }

// ПРИМЕР: закрытое поле отображается в Inspector

    [SerializeField]
    private PlayerStats m_stats;
}
```

Добавьте в другой класс поле типа этого сериализуемого класса. Его переменные появятся в Inspector в сворачиваемых группах.



Сериализуемый класс или структура помогают упорядочить Inspector.

Стиль скобок и отступов

В C# распространены два стиля отступов:

— В стиле Олмана открывающая фигурная скобка размещается с новой строки. Этот стиль также называют BSD-стилем, по имени BSD Unix.

— В стиле K&R, также известном как «единственно верный стиль скобок», открывающая скобка остается в той же строке, что и предшествующая конструкция.

```
// ПРИМЕР: в стиле Олмана, или BSD, открывающая скобка находится на новой строке.

void DisplayMouseCursor(bool showMouse)
{
     if (!showMouse)
     {
          Cursor.lockState = CursorLockMode.Locked;
          Cursor.visible = false;
     }
     else
     {
```

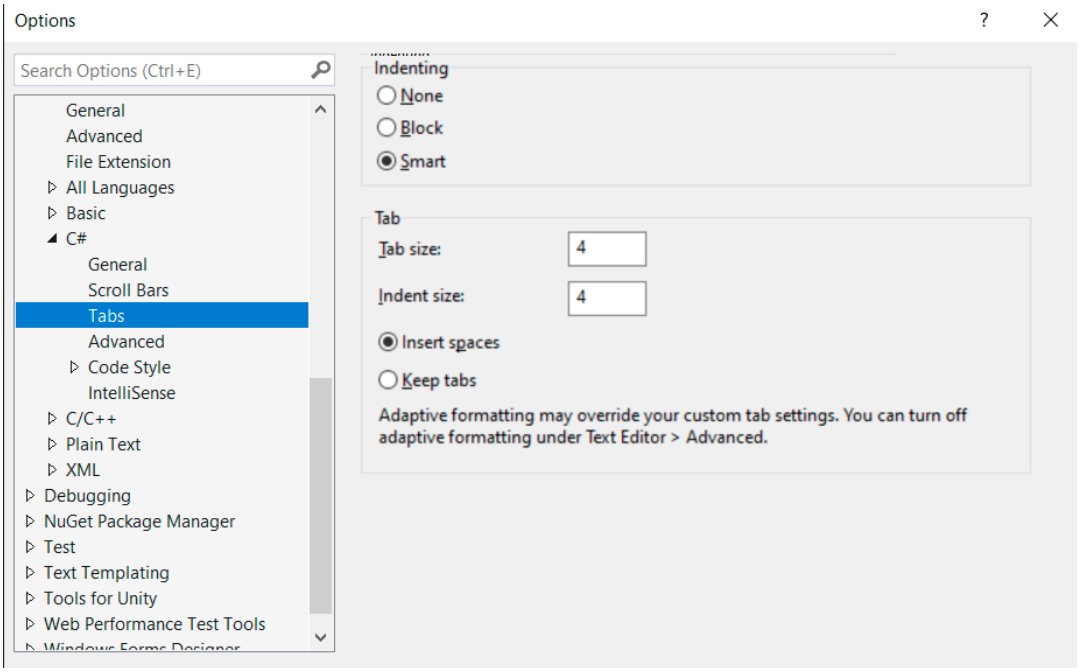
```
          Cursor.lockState = CursorLockMode.None;
          Cursor.visible = true;
     }
}

// ПРИМЕР: в стиле K&R открывающая скобка остается в строке заголовка.

void DisplayMouseCursor(bool showMouse){
     if (!showMouse) {
          Cursor.lockState = CursorLockMode.Locked;
          Cursor.visible = false;
     }
     else {
          Cursor.lockState = CursorLockMode.None;
          Cursor.visible = true;
     }
}
```

Существуют и другие варианты этих стилей. В примерах руководства применяется стиль Олмана из Microsoft Framework Design Guidelines. Какой бы вариант ни выбрала команда, все должны придерживаться одинаковых правил отступов и расстановки скобок.

Следуйте этим рекомендациям:

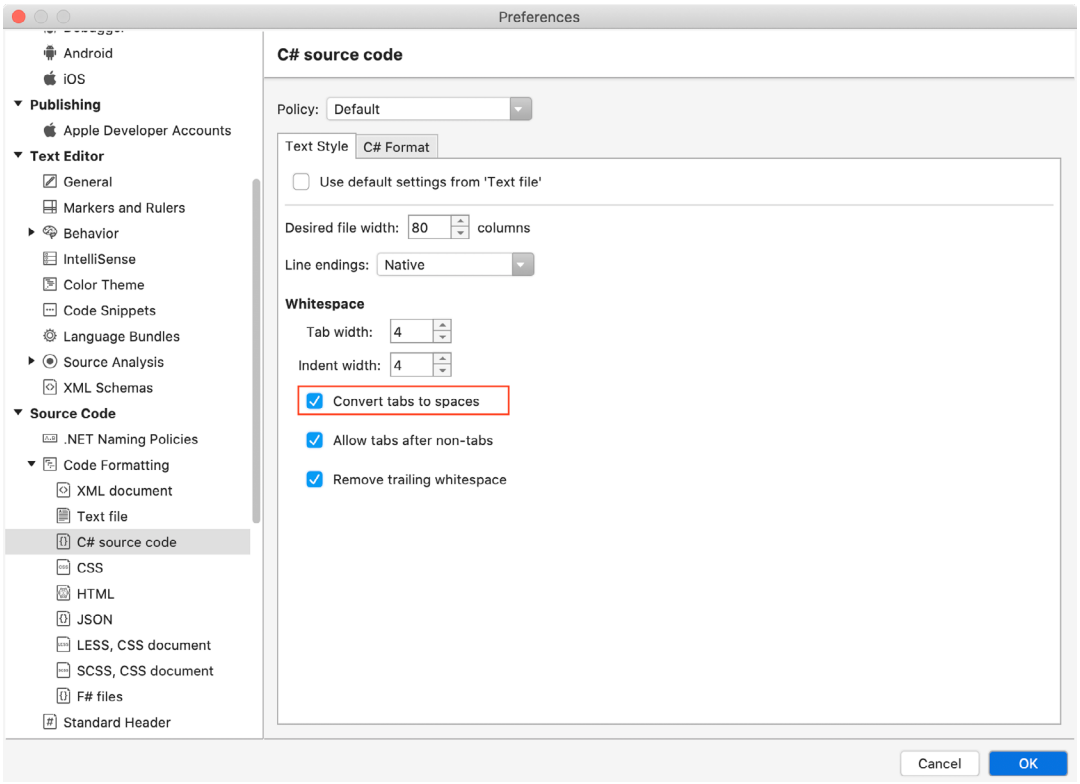
— Установите единый отступ. Обычно используют два или четыре пробела. Согласуйте настройку редактора со всей командой, не разжигая спор о табуляции и пробелах. Visual Studio умеет преобразовывать знаки табуляции в пробелы.

В Visual Studio (Windows) выберите Tools > Options > Text Editor > C# > Tabs.



Настройки табуляции в Visual Studio

В Visual Studio for Mac выберите Visual Studio > Preferences > Source Code > C# Source Code, а затем настройте параметры на вкладке Text Style.



Преобразуйте знаки табуляции в пробелы, чтобы сделать отступы единообразными.

— По возможности не опускайте фигурные скобки даже в однострочных конструкциях. Это повышает единообразие и упрощает чтение и сопровождение кода. В примере скобки четко отделяют вызов DoSomething от цикла.

> Если позднее понадобится добавить строку отладочного вывода или вызвать DoSomethingElse, скобки уже будут на месте. Некоторые программисты также считают, что отдельная строка для выражения позволяет проще поставить точку останова.

```
// ПРИМЕР: сохраняйте скобки для ясности...

for (int i = 0; i < 100; i++) { DoSomething(i); }

// …и/или вынесите выражение в отдельную строку.
for (int i = 0; i < 100; i++)
{
    DoSomething(i);
}

// ИЗБЕГАЙТЕ: опускать скобки
for (int i = 0; i < 100; i++) DoSomething(i);
```

— Не удаляйте скобки из вложенных многострочных конструкций. Ошибки компиляции не возникнет, но код станет запутаннее. Используйте скобки для ясности, даже когда они необязательны. Кроме того, с ними можно безопасно добавлять новую логику, не перестраивая окружающий код.

```
// ПРИМЕР: сохраняйте скобки для ясности
for (int i = 0; i < 10; i++)
{
      for (int j = 0; j < 10; j++)
     {
            ExampleAction();
     }
}
// ИЗБЕГАЙТЕ: удалять скобки из вложенных многострочных конструкций
for (int i = 0; i < 10; i++)
      for (int j = 0; j < 10; j++)
            ExampleAction();
```

— Унифицируйте оформление операторов switch. Для удобства чтения длинные цепочки if-else обычно лучше заменять оператором switch. В этом примере метки case оформлены с отступом. Как правило, стоит добавлять и ветвь default. Даже если все варианты уже охвачены, ветвь default позволит обработать неожиданное значение.

```
// ПРИМЕР: метки case имеют отступ относительно switch
switch (someExpression)
{
   case 0:
      DoSomething();
      break;
   case 1:
      DoSomethingElse();
      break;
   case 2:
      int n = 1;
      DoAnotherThing(n);
      break;
    default:
      // Обработка неожиданного значения или ветви default
      break;
}
```

Что такое EditorConfig?

Если над одним проектом работают несколько разработчиков, использующих разные редакторы и IDE, стоит задействовать файл EditorConfig.

EditorConfig помогает определить единый стиль кода для всей команды. Многие IDE, включая Visual Studio и Rider, поддерживают его изначально и не требуют отдельного плагина.

Файлы EditorConfig легко читать и удобно хранить в системе контроля версий. Пример такого файла приведен здесь. Настройки стиля из EditorConfig хранятся вместе с кодом и могут применяться даже вне Visual Studio.

Параметры EditorConfig имеют приоритет над глобальными настройками текстового редактора Visual Studio. Личные настройки редактора продолжают действовать в проектах без файла .editorconfig и для параметров, которые этот файл не переопределяет.

Практические примеры можно найти в репозитории GitHub.

Горизонтальные пробелы

Даже обычные пробелы могут улучшить внешний вид кода на экране. Предпочтения в форматировании различаются, но следующие рекомендации помогут повысить читаемость:

— Добавляйте пробелы, чтобы снизить плотность кода. Дополнительное пустое пространство визуально разделяет части строки и облегчает чтение.

```
// ПРИМЕР: пробелы делают строку удобнее для чтения
for (int i = 0; i < 100; i++) { DoSomething(i); }

// НЕВЕРНО: без пробелов
for(inti=0;i<100;i++){DoSomething(i);}
```

— Ставьте один пробел после запятой между аргументами метода.

```
// ПРИМЕР: один пробел после запятой между аргументами
CollectItem(myObject, 0, 1);

// ИЗБЕГАЙТЕ: пропускать пробел
CollectItem(myObject,0,1);
```

— Не ставьте пробелы внутри круглых скобок: ни после открывающей, ни перед закрывающей.

```
// ПРИМЕР: без пробелов внутри круглых скобок
DropPowerUp(myPrefab, 0, 1);

// ИЗБЕГАЙТЕ:
DropPowerUp( myPrefab, 0, 1 );
```

— Не ставьте пробел между именем метода и открывающей скобкой.

```
// ПРИМЕР: без пробела между именем метода и открывающей скобкой.
DoSomething()

// ИЗБЕГАЙТЕ
DoSomething ()
```

— Не добавляйте пробелы внутри квадратных скобок.

```
// ПРИМЕР: без пробелов внутри квадратных скобок
x = dataArray[index];

// ИЗБЕГАЙТЕ
x = dataArray[ index ];
```

— Ставьте один пробел перед условием управляющей конструкции: отделяйте условие в круглых скобках от ключевого слова.

```
// ПРИМЕР: пробел перед условием; круглые скобки отделены пробелом.
while (x == y)

// ИЗБЕГАЙТЕ
while(x==y)
```

— Ставьте по одному пробелу до и после операторов сравнения.

```
// ПРИМЕР: пробелы до и после оператора сравнения.
if (x == y)

// ИЗБЕГАЙТЕ
if (x==y)
```

— Делайте строки короткими и учитывайте горизонтальные пробелы. Установите стандартную ширину строки в 80–120 символов. Длинную строку лучше разбить на несколько выражений, чем допустить ее переполнение.

— Соблюдайте отступы и иерархию. Отступы повышают читаемость кода.

— Не выравнивайте код по столбцам без необходимости. Такое размещение выравнивает переменные, но мешает сопоставлять тип с именем.

> При этом выравнивание по столбцам может быть полезно для побитовых выражений или структур с большим объемом данных. Помните, что при добавлении элементов его придется поддерживать вручную, а некоторые автоформаттеры могут изменить выравниваемую часть столбца.

```
// ПРИМЕР: один пробел между типом и именем

    public float Speed = 12f;
    public float Gravity = -10f;
    public float JumpHeight = 2f;

    public Transform GroundCheck;
    public float GroundDistance = 0.4f;
    public LayerMask GroundMask;

// ИЗБЕГАЙТЕ: выравнивания по столбцам

    public float          Speed = 12f;
    public float          Gravity = -10f;
    public float          JumpHeight = 2f;
    public Transform      GroundCheck;
    public float          GroundDistance = 0.4f;
    public LayerMask      GroundMask;
```

Вертикальные отступы

Вертикальные отступы тоже помогают организовать код. Располагайте связанные части скрипта рядом и осмысленно используйте пустые строки. Следующие рекомендации упорядочат код сверху вниз:

— Группируйте зависимые и похожие методы. Код должен быть логичным и связным. Размещайте рядом методы, выполняющие сходные задачи, чтобы читателю не приходилось перемещаться по всему файлу.

— Отделяйте разные части класса пустыми строками. Например, можно оставить две пустые строки между:

— объявлениями переменных и методами

— классами и интерфейсами

— блоками if-then-else, если это улучшает читаемость

Не злоупотребляйте этим приемом и зафиксируйте в руководстве по стилю, где он уместен.

Области #region

Директива #region позволяет сворачивать и скрывать участки кода в файлах C#, упрощая работу с большими файлами и их чтение.

Однако при соблюдении рекомендаций этого руководства по размеру классов директива #region становится лишней. Вместо того чтобы скрывать блоки за областями, разбивайте код на небольшие классы. В коротком исходном файле потребность в #region возникает реже.

Примечание. Многие разработчики считают области признаком проблемного кода или антипаттерном. Команде следует договориться о едином подходе.

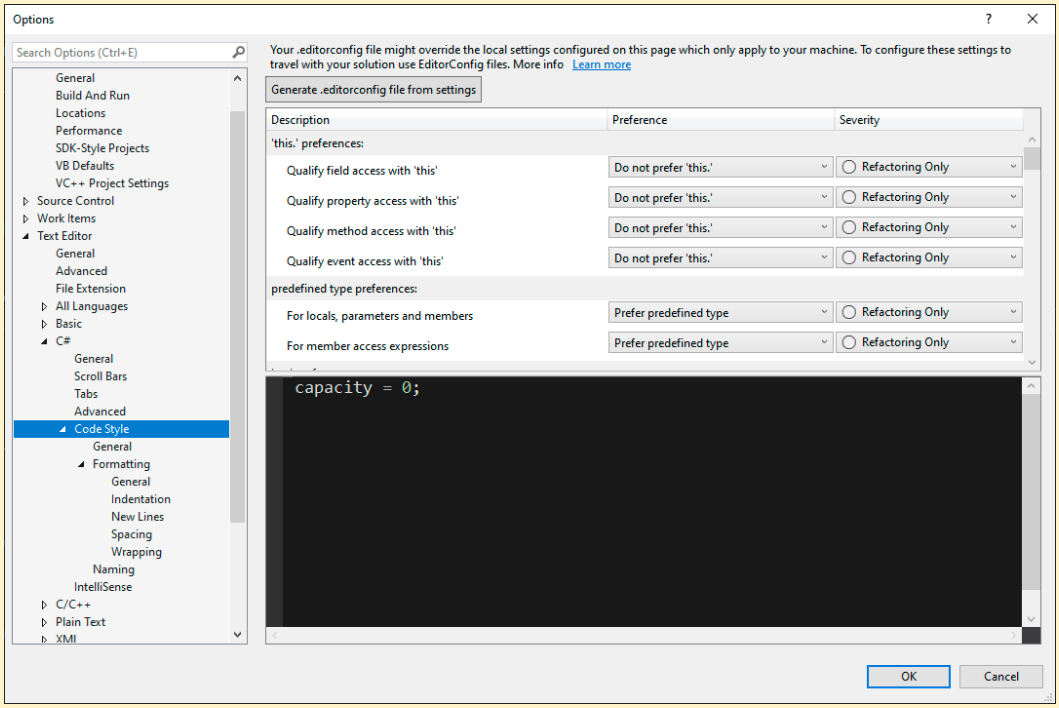
Форматирование кода в Visual Studio

Не пугайтесь, если правил форматирования кажется слишком много. Современные IDE позволяют быстро настраивать и применять их. Можно создать шаблон правил форматирования и сразу применить его ко всем файлам проекта.

Чтобы настроить правила форматирования в редакторе скриптов:

— В Visual Studio (Windows) выберите Tools > Options, затем Text Editor > C# > Code Style > Formatting.

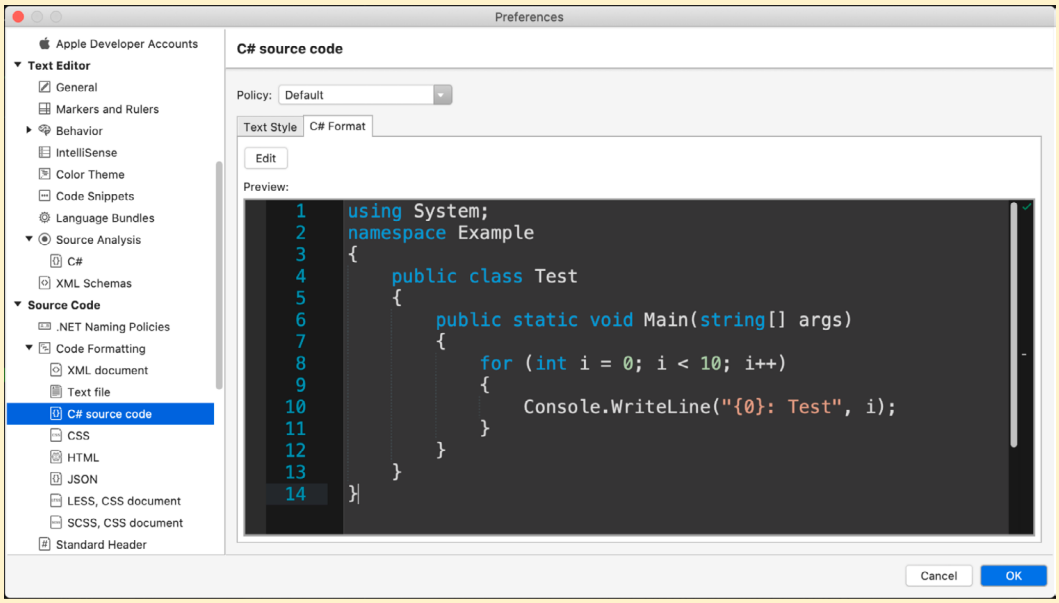
Настройте параметры General, Indentation, New Lines, Spacing и Wrapping.



Параметры форматирования кода

— В Visual Studio for Mac выберите Visual Studio > Preferences, затем Source Code > Code Formatting > C# Source Code.

В верхней части окна выберите Policy. Настройте пробелы и отступы на вкладке Text Style, а параметры Indentation, New Lines, Spacing и Wrapping — на вкладке C# Format.



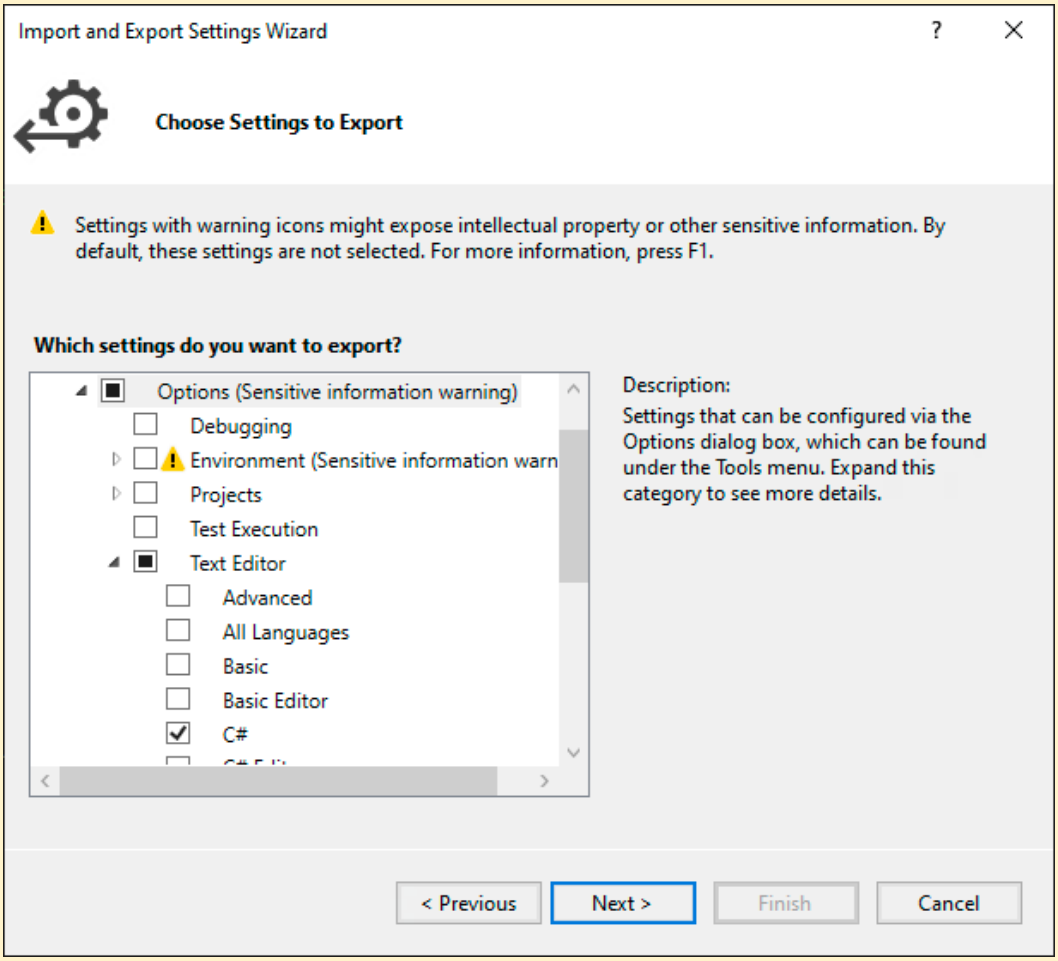
В окне Preview отображается результат выбранных настроек стиля.

Чтобы в любой момент привести файл скрипта в соответствие с руководством по стилю:

— В Visual Studio (Windows) выберите Edit > Advanced > Format Document (Ctrl+K, Ctrl+D). Если нужно отформатировать только пробелы и выровнять табуляцию, в нижней части редактора можно запустить Run Code Cleanup (Ctrl+K, Ctrl+E).

— В Visual Studio for Mac выберите Edit > Format Document (Ctrl+I).

В Windows настройки редактора можно передать через Tools > Import and Export Settings. Экспортируйте файл с правилами форматирования C# из руководства по стилю и попросите каждого участника команды импортировать его.



Экспорт настроек форматирования C# для совместного использования.

Visual Studio упрощает соблюдение руководства по стилю: для форматирования достаточно сочетания клавиш.

Примечание. Вместо импорта и экспорта настроек Visual Studio можно настроить файл EditorConfig, описанный выше. Так правила форматирования проще использовать в разных IDE и хранить в системе контроля версий. Дополнительные сведения см. в параметрах правил стиля кода .NET.

Хотя это не относится напрямую к чистому коду, рекомендуем посмотреть доклад GDC «Советы и приемы Visual Studio для повышения продуктивности». Эти советы упрощают форматирование и рефакторинг чистого кода.

Чтобы настроить файл .editorconfig в Visual Studio Code:

1. Создайте в корневом каталоге проекта файл с именем .editorconfig.

2. Откройте файл .editorconfig и добавьте нужные параметры. Ниже приведен пример конфигурации для C#:

```
# корневой файл EditorConfig
root = true

# переводы строк в стиле Unix; каждый файл заканчивается новой строкой
[*]
end_of_line = lf
insert_final_newline = true

# отступ в 4 пробела
[*.cs]
indent_style = space
indent_size = 4
charset = utf-8
trim_trailing_whitespace = true

# отступы табуляцией для Makefile
[Makefile]
indent_style = tab

# отдельные параметры для файлов JSON
[*.json]
indent_style = space
indent_size = 2
```

Классы

За всю недолгую историю вычислительной техники никто не написал идеального программного обеспечения. Маловероятно, что первым станете вы.

— Энди Хант, автор книги «Программист-прагматик»

Согласно книге Роберта К. Мартина «Чистый код», первое правило классов гласит: классы должны быть небольшими. Второе правило: они должны быть еще меньше.

Ограничение размера класса делает его более целенаправленным и связным. Существующий класс легко обрастает функциями, пока не становится перегруженным. Осознанно делайте классы небольшими: крупные раздутые классы трудно читать и отлаживать.

Метафора газетной статьи

Представьте исходный код класса как газетную статью. Чтение начинается сверху: сначала внимание привлекают заголовок и имя автора. Вводный абзац дает общее представление, а по мере продвижения вниз раскрываются подробности.

Журналисты называют такую структуру перевернутой пирамидой. Самое важное излагается в начале, а тонкости раскрываются ближе к концу статьи.

Класс должен следовать тому же принципу. Располагайте код сверху вниз и выстраивайте методы в иерархию. Методы верхнего уровня задают общую картину — размещайте их первыми. Ниже располагайте низкоуровневые методы с деталями реализации.

Например, метод ThrowBall может вызывать SetInitialVelocity и CalculateTrajectory. Сначала разместите ThrowBall, поскольку он описывает основное действие, а затем добавьте вспомогательные методы.

Каждая газетная статья коротка, но множество статей вместе образуют единое целое — газету или новостной сайт. Так же устроен проект Unity: многочисленные классы должны совместно образовывать крупное, но связное приложение.

Организация класса

Каждому классу нужна единая структура. Для порядка группируйте члены класса по разделам:

— Поля

— Свойства

— События и делегаты

— Методы MonoBehaviour (Awake, Start, OnEnable, OnDisable, OnDestroy и т. д.)

— Открытые методы

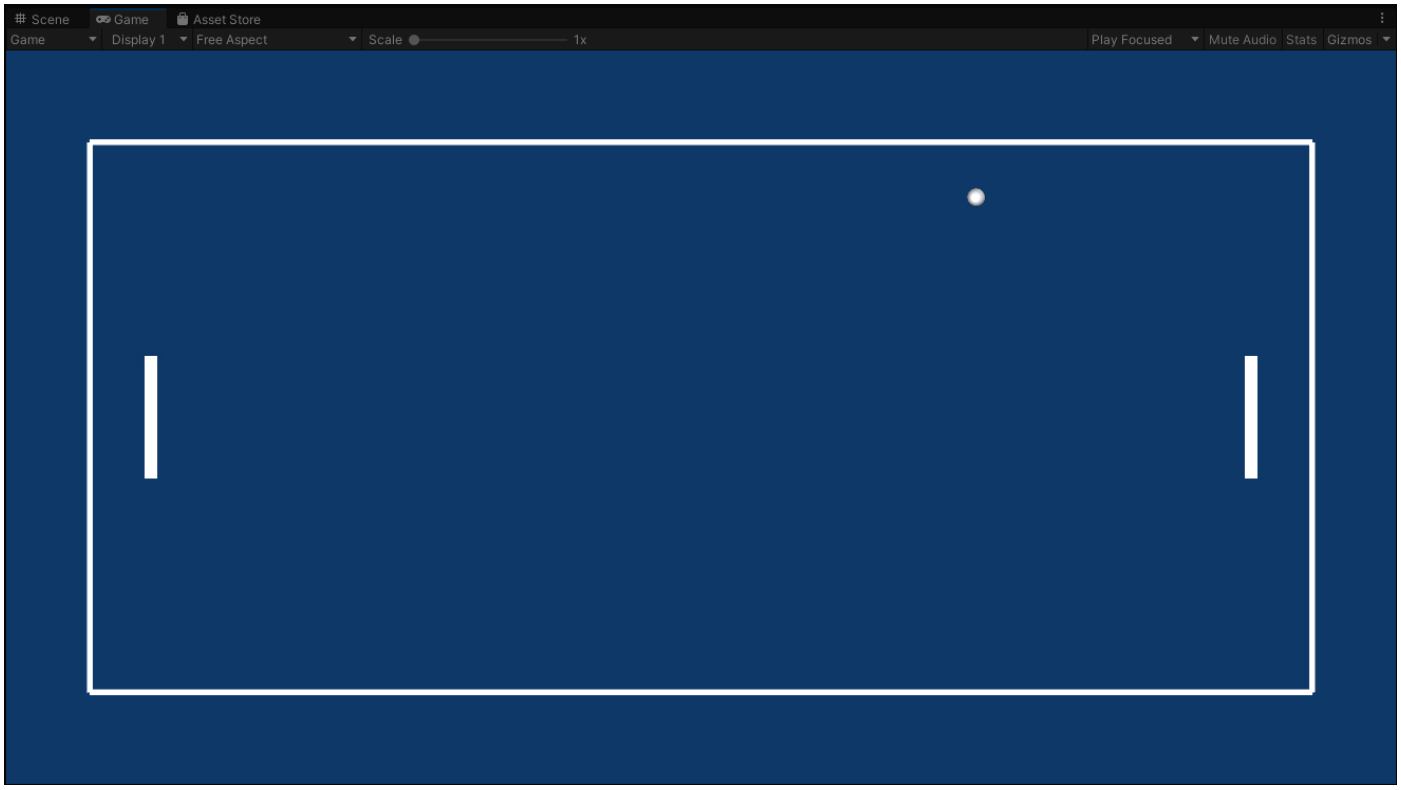
— Закрытые методы

Помните правило именования классов в Unity: имя исходного файла должно совпадать с именем содержащегося в нем MonoBehaviour. В файле могут находиться и другие внутренние классы, однако MonoBehaviour в каждом файле должен быть только один, если только несколько MonoBehaviour не разделены пространствами имен.

Принцип единственной ответственности

Помните: каждый класс должен оставаться небольшим. В проектировании ПО принцип единственной ответственности помогает сохранять простоту.

Суть принципа в том, что каждый модуль, класс или функция отвечает только за одну задачу. Допустим, вы создаете игру Pong. Начать можно с классов ракетки, мяча и стены.



Сыграем в Pong?

Например, классу Paddle может потребоваться:

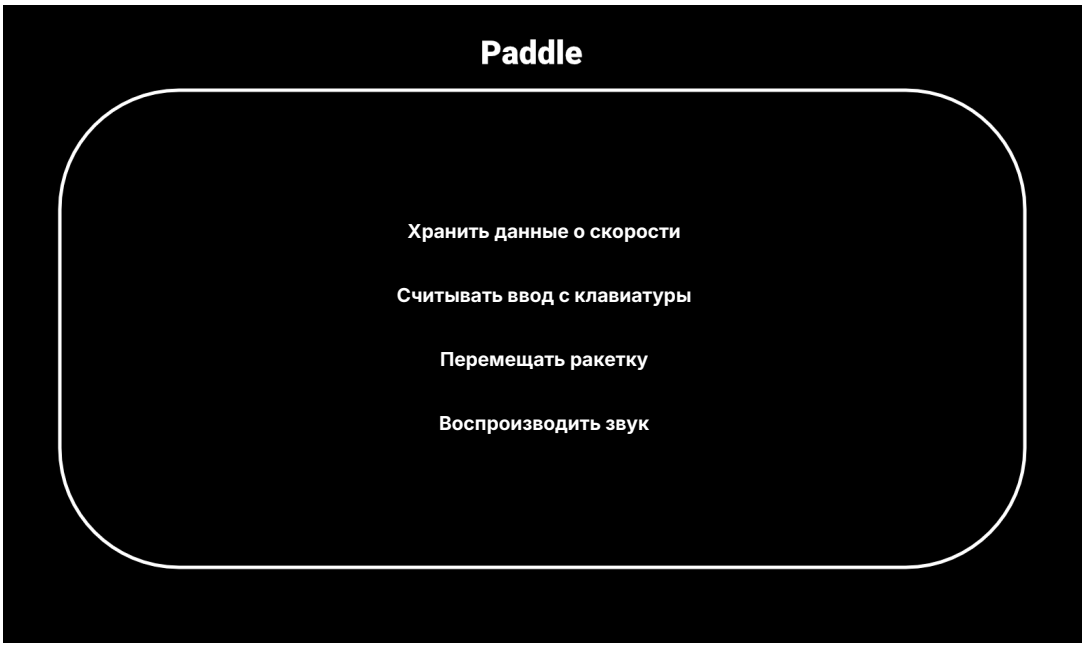
— хранить основные данные о скорости движения

— считывать ввод с клавиатуры

— перемещать ракетку в ответ на ввод

— воспроизводить звук при столкновении с мячом

Поскольку игра устроена просто, все перечисленные задачи можно объединить в базовом классе Paddle. Технически один MonoBehaviour способен выполнить все необходимое.

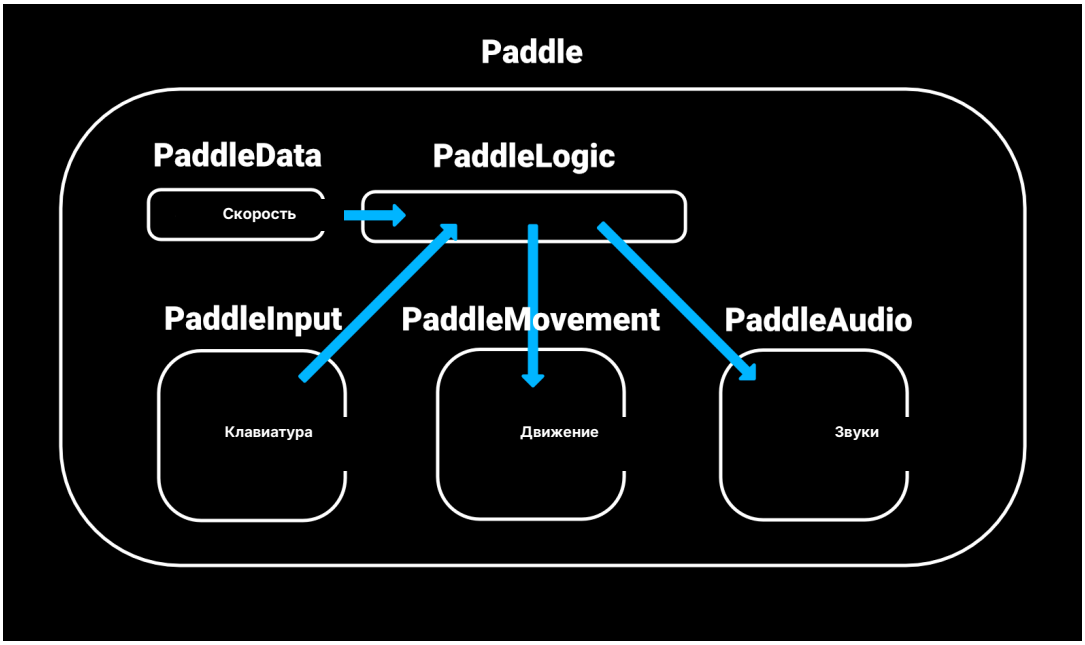


Один MonoBehaviour выполняет все задачи

Однако объединение всего в одном классе, даже небольшом, усложняет архитектуру из-за смешения обязанностей. Данные переплетаются с вводом, а класс должен обрабатывать и то и другое. Вопреки принципу KISS несколько простых задач оказываются тесно связанными.

Вместо этого разбейте Paddle на небольшие классы с одной ответственностью у каждого. Вынесите данные в PaddleData или ScriptableObject, а остальную логику распределите между PaddleInput, PaddleMovement и PaddleAudio.

Класс PaddleLogic может обрабатывать ввод из PaddleInput. Используя данные о скорости из PaddleData, он перемещает ракетку через PaddleMovement. Наконец, при столкновении мяча с ракеткой PaddleLogic сообщает PaddleAudio, что нужно воспроизвести звук.



Разделение обязанностей при рефакторинге класса Paddle

В новой архитектуре каждый класс выполняет одну задачу и остается небольшим и понятным. Чтобы проследить логику, больше не нужно прокручивать несколько экранов кода.

Скрипт Paddle по-прежнему нужен, но теперь он только связывает остальные классы. Основная функциональность распределена между ними.

Чистый код не всегда самый компактный. Даже при использовании небольших классов после рефакторинга общее число строк может вырасти, зато каждый класс читается проще. При отладке и добавлении функций такая структура помогает сохранять порядок.

Пример рефакторинга

Подробный разбор рефакторинга небольшого проекта приведен в статье «Как проектировать код по мере роста проекта». В ней показано, как разделить крупные MonoBehaviour на небольшие компоненты по принципу единственной ответственности.

Также можно посмотреть оригинальный доклад Микаэля Калмса «От Pong до проекта с командой из 15 человек», представленный на Unite Berlin.

Методы

Вы понимаете, что работаете с чистым кодом, когда каждая прочитанная процедура оказывается почти в точности такой, какой вы ее ожидали увидеть.

— Уорд Каннингем, создатель вики и один из основателей экстремального программирования

Как и классы, методы должны быть небольшими и иметь единственную ответственность. Каждый метод либо описывает одно действие, либо отвечает на один вопрос, но не делает и то и другое.

Хорошее имя метода отражает его назначение. Например, GetDistanceToTarget ясно описывает цель метода. Если в проекте используются разные единицы измерения, можно выбрать еще более точное имя GetDistanceToTargetInMeters, показывающее, что расстояние возвращается в метрах.

При создании методов пользовательских классов следуйте этим рекомендациям:

— Используйте меньше аргументов. Аргументы повышают сложность метода; сократите их число, чтобы упростить чтение и тестирование.

— Избегайте избыточной перегрузки. Можно создать бесконечное множество вариантов перегруженного метода. Оставьте только те, которые соответствуют реальным сценариям вызова.

> Если вы все же перегружаете метод, во избежание путаницы сделайте так, чтобы каждая перегрузка имела разное число параметров.

— Избегайте побочных эффектов. Метод должен делать только то, что заявлено в его имени. По возможности не изменяйте данные вне его области действия и передавайте аргументы по значению, а не по ссылке. Если результат возвращается через out или ref, это должно быть единственной целью метода.

> Для некоторых задач побочные эффекты полезны, но они могут привести к непредвиденным последствиям. Методы без побочных эффектов уменьшают вероятность неожиданного поведения.

— Вместо флага создайте отдельный метод. Не заставляйте один метод работать в двух режимах в зависимости от флага; создайте два метода с разными именами. Например, не используйте GetAngle, который по флагу возвращает градусы или радианы. Создайте GetAngleInDegrees и GetAngleInRadians.

.

> Булев флаг в списке аргументов выглядит безобидно, но может запутать реализацию или нарушить принцип единственной ответственности.

Методы или функции?

В Unity и C# принято говорить о методах, а не о функциях, поскольку метод — это функция, определенная в контексте класса или объекта. C# является объектно-ориентированным языком: его основа — классы и объекты, а методы описывают действия объектов. Функция — более общее название блока кода, который выполняет определенную задачу. В процедурных языках функции могут существовать независимо, но в объектно-ориентированных языках, таких как C#, они инкапсулированы в классах и называются методами.

Методы расширения

Методы расширения позволяют добавлять функциональность в классы, в том числе закрытые для наследования, и дают аккуратный способ расширять API UnityEngine.

Чтобы создать метод расширения, объявите статический метод и поставьте ключевое слово this перед первым параметром — типом, который требуется расширить.

Предположим, нужно создать метод ResetTransformation, который удаляет масштабирование, поворот и смещение объекта GameObject.

Можно создать статический метод, указав первым параметром Transform и поставив перед ним ключевое слово this:

```
// ПРИМЕР: определение метода расширения
public static class TransformExtensions
{
    public static void ResetTransformation(this Transform transform)
    {
        transform.position = Vector3.zero;
        transform.localRotation = Quaternion.identity;
        transform.localScale = Vector3.one;
    }
}
```

Затем, когда понадобится применить этот метод, вызовите ResetTransformation. Класс ResetOnStart вызывает его для текущего Transform в методе Start.

```
// ПРИМЕР: вызов метода расширения
public class ResetOnStart : MonoBehaviour
{
    void Start()
    {
        transform.ResetTransformation();
    }
}
```

Чтобы упорядочить код, определяйте методы расширения в статических классах. Например, методы, расширяющие Transform, можно поместить в класс TransformExtensions, методы для Vector3 — в Vector3Extensions и так далее.

Методы расширения позволяют создавать множество полезных утилит без дополнительных компонентов MonoBehaviour. Подробнее см. материал Unity Learn «Методы расширения»: он поможет пополнить арсенал приемов разработчика игр.

Принцип DRY: не повторяйтесь

В книге «Программист-прагматик» Энди Хант и Дейв Томас сформулировали принцип DRY — «не повторяйтесь». Этот известный принцип разработки программного обеспечения рекомендует избегать дублирования и повторения логики.

Так проще исправлять ошибки и снижать стоимость сопровождения. Если вы следуете принципу единственной ответственности, при изменении класса или метода не придется править несвязанный код. Устранив логическую ошибку в коде, соответствующем принципу DRY, вы исправите ее сразу во всех местах.

Противоположность DRY — WET («нам нравится печатать» или «пишите все дважды»). Код считается WET, если в нем есть ненужные повторения.

Представьте две системы ParticleSystem (explosionA и explosionB) и два объекта AudioClip (soundA и soundB). Каждая система ParticleSystem должна воспроизводиться вместе со своим звуком. Для этого можно написать простые методы, подобные следующим.

```
// ПРИМЕР: ПИШЕМ ВСЕ ДВАЖДЫ


    private void PlayExplosionA(Vector3 hitPosition)
    {
        explosionA.transform.position = hitPosition;
        explosionA.Stop();
        explosionA.Play();

        AudioSource.PlayClipAtPoint(soundA, hitPosition);
    }

    private void PlayExplosionB(Vector3 hitPosition)
    {
        explosionB.transform.position = hitPosition;
        explosionB.Stop();
        explosionB.Play();

        AudioSource.PlayClipAtPoint(soundB, hitPosition);
    }
```

Каждый метод принимает позицию типа Vector3, перемещает туда ParticleSystem и запускает эффект. Сначала система частиц останавливается на случай, если она уже воспроизводится, а затем симуляция запускается заново. После этого статический метод AudioSource.PlayClipAtPoint создает звуковой эффект в той же точке.

Один метод фактически является копией другого с небольшими заменами. Такой код работает, но для каждого нового взрыва придется создавать еще один метод с той же логикой.

Вместо этого выполните рефакторинг и сведите логику к одному методу PlayFXWithSound:

```
// ПРИМЕР: версия после рефакторинга по принципу DRY


    private void PlayFXWithSound(ParticleSystem particle, AudioClip
clip, Vector3 hitPosition)
    {
        particle.transform.position = hitPosition;
        particle.Stop();
        particle.Play();
```

```
        AudioSource.PlayClipAtPoint(clip, hitPosition);
    }
```

Добавляя новые объекты ParticleSystem и AudioClip, вы сможете по-прежнему воспроизводить их совместно тем же методом.

Обратите внимание: дублировать код можно и без нарушения принципа DRY. Главное — не дублировать логику.

Здесь основная функциональность вынесена в метод PlayFXWithSound. Если логику потребуется изменить, достаточно исправить один метод, а не оба — PlayExplosionA и PlayExplosionB.

Комментарии

Код похож на юмор: если его приходится объяснять, значит, он плох.

— Кори Хаус, архитектор программного обеспечения и автор

Уместные комментарии повышают читаемость кода, а чрезмерные или ненужные могут дать обратный эффект. Как и во всем, здесь важен баланс.

Большая часть кода не потребует комментариев, если следовать принципу KISS и разбивать логику на небольшие, понятные части. Хорошо названные переменные и функции говорят сами за себя.

Полезный комментарий отвечает не столько на вопрос «что?», сколько на вопрос «почему?». Принимали ли вы решения, причины которых неочевидны? Есть ли сложная логика, требующая пояснения? Хороший комментарий сообщает то, чего нельзя понять из самого кода.

Ниже приведены основные рекомендации по работе с комментариями:

— Не заменяйте плохой код комментариями. Если для объяснения запутанной логики нужен комментарий, переработайте код так, чтобы он стал понятнее. Тогда комментарий не понадобится.

— Правильно названный класс, переменная или метод заменяет комментарий. Если код понятен сам по себе, не создавайте лишний шум и обойдитесь без комментария.

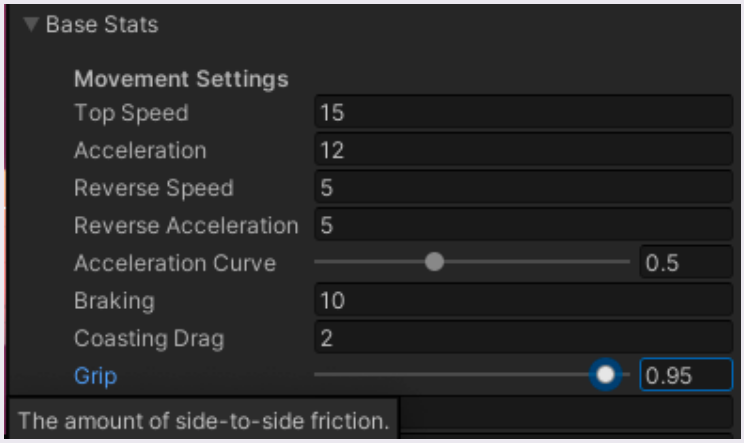
```
// ИЗБЕГАЙТЕ: лишних комментариев, повторяющих очевидное

// цель, по которой нужно выстрелить

Transform targetToShoot;
```

— По возможности размещайте комментарий на отдельной строке, а не в конце строки кода. Обычно отдельная строка делает его понятнее.

— В большинстве случаев используйте однострочные комментарии с двойной косой чертой (//). Размещайте комментарий рядом с кодом, который он поясняет, вместо большого многострочного блока в начале. Так читателю проще связать пояснение с логикой.

— Для сериализуемых полей используйте всплывающую подсказку вместо комментария. Если поле в Inspector требует пояснения, добавьте атрибут Tooltip и не создавайте отдельный комментарий: подсказка выполнит обе задачи.

```
// ПРИМЕР: Tooltip заменяет комментарий

[Tooltip("Величина бокового трения.")]
public float Grip;
```



Всплывающая подсказка в окне Inspector

— Перед открытыми методами и функциями также можно использовать XML-тег <summary>. Visual Studio отображает многие стандартные XML-комментарии через IntelliSense.

```
// ПРИМЕРЫ:
// Обычный комментарий.
// Используйте комментарии, чтобы показать намерение, ход логики и подход.

// Также можно использовать XML-тег <summary>.
//
/// <summary>
/// Управляет системой оружия: стрельбой, перезарядкой и нанесением урона.

/// </summary>
public void Fire()
{
    ...
}
```

— Между маркером комментария (//) и его текстом ставьте один пробел.

— Добавляйте юридические уведомления. Комментарий подходит для сведений о лицензии или авторских правах, но не вставляйте в код полный юридический документ. Вместо этого дайте ссылку на страницу с полной информацией.

— Соблюдайте единый стиль комментариев. Например, начинайте каждый комментарий с прописной буквы и завершайте точкой. Какой бы формат ни выбрала команда, закрепите его в руководстве по стилю и придерживайтесь его.

— Не окружайте комментарии декоративными блоками из звездочек или специальных символов. Это ухудшает читаемость и дополнительно загромождает код.

— Удаляйте закомментированный код. Временно отключать строки во время разработки и тестирования нормально, но не оставляйте такой код надолго. Предыдущие версии хранит система контроля версий — смело удаляйте ненужные строки.

— Поддерживайте комментарии TODO в актуальном состоянии. Завершив задачу, удалите оставленный для нее TODO: устаревшие напоминания только отвлекают.

> Для контекста и персональной ответственности в TODO можно указать имя и дату.
>
> Будьте реалистичны. До TODO, оставленного пять лет назад, вы, скорее всего, уже не доберетесь. Следуйте YAGNI: удалите этот комментарий и вернитесь к задаче лишь тогда, когда она действительно понадобится.

— Не превращайте комментарии в журнал разработки. Приступая к новому классу, не нужно записывать в комментариях каждое действие. При правильном использовании системы контроля версий это избыточно.

— Не указывайте авторство правок. Подписи вроде // добавлено devA или devB не нужны — эту информацию хранит система контроля версий.

Соглашения об именах в UI Toolkit

Если отладка — это процесс удаления ошибок из программного обеспечения, то программирование должно быть процессом их добавления.

— Эдсгер В. Дейкстра, один из пионеров информатики

До сих пор руководство было посвящено стилю кода C#, но стоит также рассмотреть соглашения об именах для UI Toolkit, UXML и CSS. В UI Toolkit визуальные элементы и Unity Style Sheets (USS) запрашиваются по строковым идентификаторам, поэтому единый набор правил сокращает число ошибок и делает код понятнее.

Для визуальных элементов в UXML и классах таблиц стилей обычно рекомендуется методология «Блок — Элемент — Модификатор» (Block Element Modifier, BEM). BEM широко применяется в CSS и современной веб-разработке, на идеях которых основан UI Toolkit.

По имени элемента в стиле BEM можно сразу понять, что он делает, где находится и как связан с окружающими элементами. Соглашение BEM включает три основных компонента:

```
block-name__element-name--modifier-name
```

Рассмотрим пример:

```
navbar-menu__shop-button--small
```

Каждая часть имени может содержать латинские буквы, цифры и дефисы. Части соединяются двойным подчеркиванием __ или двойным дефисом --.

Имя блока (block-name) обозначает высокоуровневый компонент, например меню навигации или панель характеристик персонажа, то есть самостоятельную и значимую часть интерфейса. Для универсальной кнопки, не относящейся к конкретному блоку, имя блока можно опустить: например, button--small.

Элемент element-name является дочерней частью блока и семантически с ним связан. Иными словами, контекст элемента задается блоком, и отдельно от него элемент не существует. В примере shop-button стиль показывает, что эта кнопка отличается от других кнопок блока navbar-menu: navbar-menu__shop-button.

Если новый элемент создает дочерние элементы в конструкторе, назначайте этим дочерним элементам соответствующие классы, например my-block__first-child и my-block__other-child.

Модификатор обозначает вариант или состояние блока либо элемента: нажатую кнопку, выбранный и подсвеченный элемент текстового поля или, как в нашем примере, уменьшенный вариант кнопки магазина. Это позволяет поддерживать разные состояния без дублирования кода.

Еще несколько примеров именования по BEM:

```
– menu__home-button – menu__shop-button – navbar-menu__shop-button--small
```

Имена классов BEM описывают сами себя: разработчикам проще понимать структуру и назначение компонентов, а четкая иерархия облегчает сопровождение и обновление стилей по мере роста проекта.

В этих примерах части имени разделены дефисами — это стиль kebab-case, распространенный в CSS. Как и в других вопросах стиля, команда может выбрать подходящую схему, но лучше сделать это в начале проекта и затем соблюдать ее последовательно.

Подробнее о соглашениях об именах в CSS читайте в этой статье и в документации UI Toolkit.

Рекомендации по именованию в UI Toolkit

Ниже приведены рекомендации по эффективному именованию:

— Делайте имена короткими, понятными и однозначными. Они должны быть лаконичными, но достаточно информативными, чтобы передавать назначение и роль элемента в интерфейсе.

— Не используйте в BEM-селекторах имена типов (Button, Label) или имена элементов (#my-button). Это избавляет от избыточности и возможной путаницы. Имена BEM должны описывать роль и состояние, а не тип элемента.

— Избегайте имен и модификаторов, смысл которых может измениться. Например, пока цветовая схема не утверждена, используйте button--quit, а не button--red. Семантические имена остаются актуальными даже после изменения оформления.

— Распространяйте эти соглашения на графические ресурсы интерфейса UI Toolkit, например спрайты и текстуры. Единое именование в коде и ресурсах сохраняет понятные связи и улучшает организацию проекта.

— Если элемент будет использоваться в других проектах, добавьте префикс к классам, чтобы избежать конфликтов с существующими пользовательскими именами. Пространства имен и префиксы предотвращают коллизии при интеграции с другими проектами и библиотеками.

— В конструкторе вызывайте AddToClassList(), чтобы назначить экземплярам элемента нужные классы USS. Классы добавляются при создании элемента, поэтому соответствующие стили применяются сразу, а код интерфейса остается последовательным и понятным.

Типичные ошибки

Чистый код не возникает случайно. Это результат осознанной работы людей, которые стараются мыслить и писать код как единая команда.

Разумеется, не все идет по плану. Как бы вы ни старались, нечистый код неизбежно появляется, поэтому его нужно уметь находить.

«Запах кода» — характерный признак того, что в проекте может скрываться проблемный участок. Перечисленные ниже симптомы не всегда означают серьезную проблему, но при их появлении код стоит проверить:

— Загадочные имена. Хорошая загадка уместна где угодно, но не в правилах написания кода. Классам, методам и переменным нужны понятные, однозначные имена. Подробнее см. раздел об именовании.

— Избыточная сложность. Чрезмерное проектирование возникает, когда класс пытаются заранее подготовить ко всем возможным потребностям. Так появляется «божественный объект» (God object) с длинными методами или огромный класс, который делает слишком много. Разделите монолитный класс на небольшие специализированные части, каждая из которых отвечает за одну задачу.

— Негибкость. Небольшое изменение не должно требовать множества исправлений в других местах. Если это происходит, проверьте, не нарушен ли принцип единственной ответственности.

> Объект с несколькими обязанностями ломается чаще, потому что предусмотреть все взаимодействия сложнее. Если метод выполняет одну задачу и после изменения по-прежнему работает правильно, остальная система также должна продолжить работу.

— Хрупкость. Если после небольшого изменения перестает работать вся система, это часто указывает на проблему в структуре кода.

— Неподвижность. Код нередко можно повторно использовать в другом контексте. Если перенести его мешает большое число зависимостей, отделите основную логику от окружения.

— Дублирование кода. Если видно, что код скопирован и вставлен, пора выполнить рефакторинг. Вынесите общую логику в отдельный метод и вызывайте его из остальных методов. Копии сложно сопровождать: при каждом изменении приходится исправлять несколько мест. Подробнее см. раздел о принципе DRY.

— Избыточные комментарии. Они помогают объяснить неочевидный код, но ими легко злоупотребить. Пошаговые пояснения к каждой переменной или инструкции не нужны. Лучше дать методу или классу имя, ясно выражающее намерение. Чем меньше и понятнее фрагменты логики, тем меньше пояснений им требуется.

Заключение

Программирование — не игра с нулевой суммой. Делясь знаниями с другим программистом, вы ничего не теряете.

— Джон Кармак, сооснователь id Software

Надеемся, вам понравилось это знакомство с принципами чистого кода и руководствами по стилю.

Представленные приемы — не столько строгие правила, сколько привычки. Как и любые привычки, их необходимо выработать самостоятельно ежедневной практикой.

Как уже упоминалось, вы можете взять эту памятку по стилю C# для разработчиков Unity за основу собственного руководства и изменить ее с учетом предпочтений команды. Главное — вместе договориться о стандарте и последовательно его соблюдать.

Но одних соглашений об оформлении недостаточно. Чтобы код масштабировался, разбивайте его на небольшие модульные части. В ходе долгой разработки будьте готовы неоднократно переписывать код: требования к продукту неизбежно меняются.

Если вам нужны дополнительные материалы о создании чистой, организованной и читаемой кодовой базы, ознакомьтесь с нашей электронной книгой «Повышайте уровень своего кода с помощью шаблонов проектирования и SOLID» и сопутствующим демонстрационным проектом в Unity Asset Store.

Источники

Это руководство представляет собой краткий перечень общепринятых практик разработки программного обеспечения. Дополнительные сведения см. в документе «Рекомендации по проектированию библиотек .NET Framework» (Microsoft Framework Design Guidelines), на котором основаны приведенные здесь правила стиля.

Подробнее о чистом коде можно узнать из посвященных ему фундаментальных книг. Ниже перечислены несколько изданий, которые мы рекомендуем:

«Чистый код: руководство по гибкой разработке программного обеспечения». Роберт К. Мартин, 2008. Prentice Hall. ISBN 978-0132350884.

«Программист-прагматик», юбилейное издание к 20-летию. Дэвид Томас и Эндрю Хант, 2019. Addison-Wesley. ISBN 978-0135957059.

«Рефакторинг: улучшение проекта существующего кода», второе издание. Кент Бек и Мартин Фаулер, 2018. Addison-Wesley. ISBN 978-0-134-75759-9.

«Разработка через тестирование», первое издание. Кент Бек, 2002. Addison-Wesley. ISBN 978-0321146533.

Приложение: шаблоны скриптов

Разговоры ничего не стоят. Покажите мне код.

— Линус Торвальдс, создатель Linux и Git

Шаблоны скриптов — это готовые фрагменты кода, которые Unity использует при создании новых файлов C#, например скриптов MonoBehaviour или ScriptableObject (Assets > Create > C# Script).

Определив правила форматирования в руководстве по стилю, настройте шаблоны скриптов в соответствии с ними. Это поможет поддерживать единообразие кодовой базы и сократит объем повторяющейся работы.

Эти файлы находятся по следующим путям:

Windows: C:\Program Files\Unity\Hub\Editor\[UnityVersion]\Editor\Data\Resources\ScriptTemplates

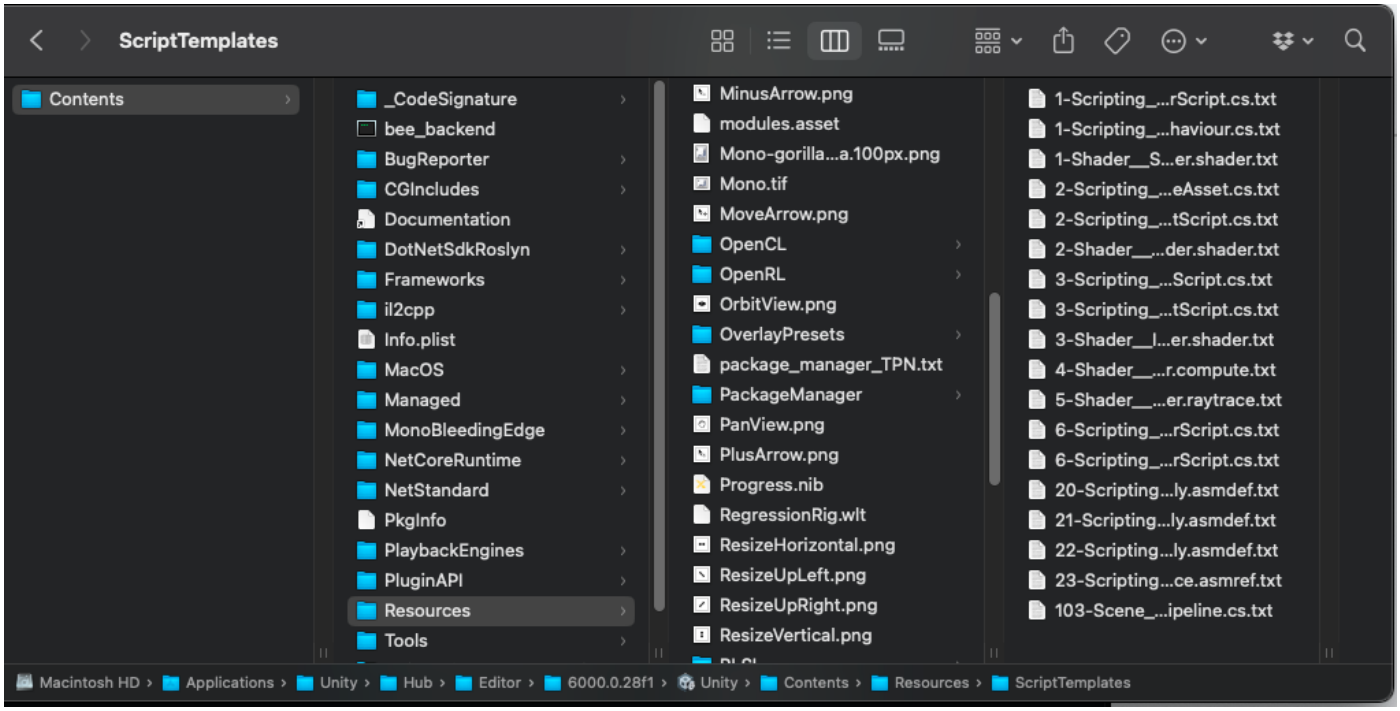
Mac: /Applications/Unity/Hub/Editor/[UnityVersion]/Unity.app/Contents/Resources/ScriptTemplates

В macOS откройте содержимое пакета Unity.app, чтобы увидеть подкаталог Resources.

В этом каталоге находятся стандартные шаблоны, например:

```
1-Scripting__MonoBehaviour Script-NewMonoBehaviourScript.cs.txt
```

```
2-Scripting__ScriptableObject Script-NewScriptableObjectScript.cs.txt
```



Когда вы создаете в окне Project новый скриптовый ассет через меню Create, Unity использует один из этих шаблонов.

Если открыть файл 1-Scripting__MonoBehaviour Script-NewMonoBehaviourScript.cs.txt в текстовом редакторе, вы увидите следующее:

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

public class #SCRIPTNAME# : MonoBehaviour
{
    // Start вызывается перед обновлением первого кадра
    void Start()
    {
        #NOTRIM#
    }
    // Update вызывается один раз за кадр
    void Update()
    {
        #NOTRIM#
    }
}
```

Обратите внимание на ключевые слова:

— #SCRIPTNAME#: указанное вами имя скрипта. Если его не изменить, будет использовано стандартное имя, например NewBehaviourScript.

— #NOTRIM# сохраняет пробельные символы и гарантирует наличие пустой строки между фигурными скобками.

Шаблоны скриптов можно настраивать: например, добавить пространство имен или удалить стандартный метод Update. Изменение шаблона экономит несколько действий при каждом создании скриптового ассета.

Имя файла шаблона скрипта строится по следующей схеме:

PriorityNumber-MenuPath-DefaultName.FileExtension.txt

Разные части имени разделяет символ дефиса (-):

— PriorityNumber определяет положение скрипта в меню Create. Чем меньше число, тем выше приоритет.

— MenuPath определяет расположение файла в меню Create. Категории можно создавать с помощью двойного подчеркивания (__).

> Например, CustomScript__Misc__ScriptableObject создает пункт ScriptableObject в меню Create > CustomScript > Misc.

— DefaultName — стандартное имя ассета, используемое, если вы не указали другое.

— FileExtension — расширение файла, добавляемое к имени ассета.

Обратите внимание: после FileExtension в имени каждого шаблона скрипта также добавляется суффикс .txt.

Чтобы применить шаблон скрипта только к определенному проекту Unity, скопируйте всю папку ScriptTemplates непосредственно в каталог Assets проекта: /Assets/ScriptTemplates. При необходимости можно скопировать лишь те разделы, которые вы собираетесь редактировать.

Затем создайте новые шаблоны скриптов или измените исходные по своему усмотрению. Удалите из проекта шаблоны, которые не планируете изменять. Можно также скопировать только нужные файлы.

Исходные шаблоны скриптов можно изменить и в ресурсах приложения, но будьте осторожны: изменения затронут все проекты, использующие эту версию Unity.

Подробнее о настройке шаблонов скриптов см. в этой статье службы поддержки. Дополнительные примеры также находятся в приложенном проекте.

Приложение: тестирование и отладка

Отладка похожа на работу детектива в криминальном фильме, где убийца — вы сами.

— Филипе Фортес

Автоматизированное тестирование помогает повышать качество кода и сокращать время на исправление ошибок. Разработка через тестирование (TDD) — это методология, при которой модульные тесты создаются одновременно с программным обеспечением. Как правило, каждый тест пишется до реализации соответствующей функции.

По мере разработки вы многократно запускаете весь набор автоматизированных тестов для проверки программы. Это принципиально отличается от подхода, при котором сначала пишут программу, а тесты добавляют позднее. В TDD написание кода, тестирование и рефакторинг тесно связаны.

Основная идея описана в книге Кента Бека «Разработка через тестирование на примере»:

1. Добавьте один модульный тест. Он описывает новую функцию, которую вы хотите добавить в приложение. Сформулируйте требования к ней на основе запросов команды или пользователей.

2. Запустите тест. Он должен завершиться неудачей, поскольку новая функция еще не реализована. Одновременно это подтверждает, что сам тест действительно способен обнаружить отсутствие функции и не проходит автоматически.

3. Напишите простейший код, который проходит новый тест. Реализуйте ровно столько логики, сколько необходимо. На этом этапе код не обязан быть чистым: допустимы неидеальная структура, жестко заданные магические числа и другие временные решения, если тест проходит.

4. Убедитесь, что проходят все тесты. Запустите полный набор автоматизированных тестов. Все прежние модульные тесты должны завершиться успешно, а новый код должен соответствовать как новым, так и существующим требованиям.

    Если это не так, изменяйте новый код — и только его — пока не пройдут все тесты.

5. Выполните рефакторинг. Вернитесь к новому коду, приведите его в порядок и убедитесь, что он соответствует руководству по стилю.

    Организуйте код логично: держите связанные классы и методы рядом, удаляйте дублирование и переименовывайте идентификаторы так, чтобы комментарии требовались как можно реже. Разделяйте чрезмерно длинные методы и классы.

    После каждого этапа рефакторинга запускайте набор автоматизированных тестов.

6. Повторяйте. Проходите этот цикл при добавлении каждой функции. Каждый шаг должен быть небольшим постепенным изменением. Часто создавайте коммиты в системе контроля версий. Тогда при отладке каждого модульного теста придется анализировать лишь небольшой объем нового кода. Если ничего не помогает, вернитесь к предыдущему коммиту и начните заново.

В этом суть подхода. Разработка по TDD естественным образом подталкивает к принципу KISS: добавляйте по одной функции, сразу тестируйте ее и непрерывно выполняйте рефакторинг, чтобы очистка кода стала постоянной практикой.

Как и большинство принципов чистого кода, TDD требует дополнительных усилий в краткосрочной перспективе, но обычно улучшает долгосрочное сопровождение и читаемость.

Unity Test Framework

Unity Test Framework (UTF), ранее называвшийся Unity Test Runner, предоставляет разработчикам Unity стандартный фреймворк тестирования. UTF использует NUnit — библиотеку тестирования с открытым исходным кодом для языков .NET.

Unity Test Framework выполняет модульные тесты в Editor — в Edit Mode или Play Mode — и на целевых платформах, включая Standalone, Android и iOS. Установите UTF через Package Manager. Начать работу поможет документация.

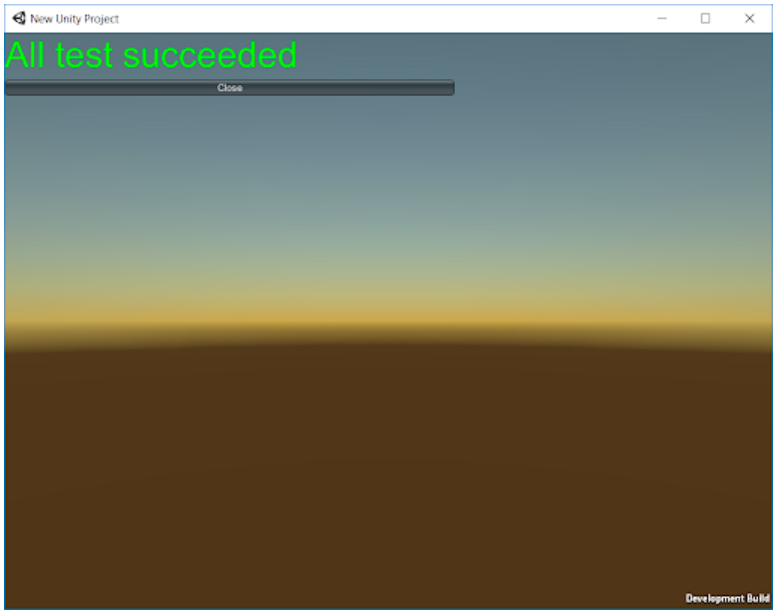
Общий рабочий процесс Unity Test Framework выглядит следующим образом:

— Создайте новый набор тестов — тестовую сборку (Test Assembly). Интерфейс Test Runner упрощает этот процесс и создает соответствующую папку в проекте.

— Создайте тест. Интерфейс Test Runner помогает управлять скриптами C#, используемыми как модульные тесты. Выберите папку Test Assembly и откройте Assets > Create > Testing > C# Test Script. Отредактируйте созданный скрипт и добавьте логику теста.

— Запустите тест. Через интерфейс Test Runner можно выполнить все модульные тесты или только выбранный. В JetBrains Rider UTF также запускается непосредственно из редактора скриптов.

— Добавьте тесты Play Mode для Editor или автономного запуска. Стандартная Test Assembly работает в Edit Mode. Чтобы тесты выполнялись во время работы приложения, создайте отдельную сборку Play Mode. Настройте их и для автономной сборки; результаты будут отображаться в Editor.



Unity Test Framework отображает в Editor результаты тестирования standalone-сборки.

Дополнительные сведения о UTF, тестировании и отладке проектов Unity доступны в следующих материалах:

— Микросайт Unity Test Framework

— Автоматизированное тестирование игр с помощью Unity Test Framework

— Отладка игрового кода с помощью Roslyn Analyzers

— Рекомендации по тестированию и обеспечению качества проектов Unity

— Ускорение отладки с помощью Microsoft Visual Studio Code

— Отладка кода с помощью Microsoft Visual Studio


Unity®