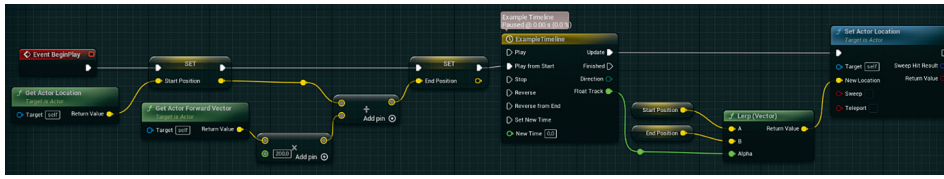

Рисунок 122. Полная схема, реализующая движение кубика

Объектно ориентированное программирование в Unreal Engine5

В этом разделе мы познакомимся с классами в Unreal Engine 5. Для начинающих мы подготовили небольшое введение по теме «Основы объектно ориентированного программирования», в котором очень сжато изложены базовые понятия этой концепции разработки. Изложенный во введении материал, конечно, не сделает вас асом объектно ориентированного подхода, но все же позволит получить базовое понимание этой философии и даст вам путеводную нить в эту занимательную область программирования. Если у вас есть опыт в ООП-разработке, вы можете пропустить вводный текст и сразу перейти к основному материалу раздела.

БАЗОВЫЕ ПОНЯТИЯ ООП

Специалисты в области проектирования программного обеспечения ежедневно имеют дело с очень сложными системами[12] — программными системами. Компьютерные игры — это тоже сложные системы, и чем дальше вы будете пробираться сквозь чащу игровой разработки, тем больше «системного» будет возникать вокруг вас: программные подсистемы, игровые подсистемы, системный гейм-дизайн и прочее. Уже в процессе прототипирования игры приходится думать о том, как будет организована работа большой и сложной системы. Но поскольку история программной разработки насчитывает не один десяток лет, изобретать велосипед вам не понадобится. Например, давно

[12] В этой книге мы не будем вводить понятие «система», хотя с точки зрения строгости изложения это было бы правильно. Вы можете самостоятельно ознакомиться с упомянутым термином и современным системно-инженерным подходом к нему, воспользовавшись работами А. И. Левенчука (начать можно с его личного блога «Лабораторный журнал», доступного по адресу: https://ailev.livejournal.com).

уже стало очевидно, что разбиение большой системы на более мелкие подсистемы, с каждой из которых можно работать независимо, сильно упрощает работу со сложным «организмом». Такой подход в общем случае называется **декомпозицией**.

Если вам уже приходилось писать программный код, вы наверняка сталкивались с алгоритмической декомпозицией (algorithmic decomposition) — процессом разделения системы на части, каждая из которых отражает некоторый этап общего процесса. Гейм-дизайнеры используют декомпозицию, чтобы свести комплексные механики к атомарным[13]. Декомпозиции окружают нас повсюду и, говоря об объектно ориентированном подходе, мы будем подразумевать объектовую декомпозицию.

> **Объектно ориентированный подход** (object oriented approach) — подход к решению проблем, предполагающий объектовую декомпозицию проблемы.
>
> **Объектно ориентированное программирование (ООП)** (object oriented programming) — это парадигма программирования, основанная на представлении программы в виде совокупности взаимодействующих объектов, каждый из которых является экземпляром некоторого класса, входящего в иерархию наследования.

Из всего вышесказанного становится понятно, что в ООП все вертится вокруг понятия «**объект»** [object]. Объект — это произвольная сущность, для которой достаточно легко определить *границы*, *состояние* и *поведение*. Каждая из этих характеристик имеет формальное определение, но об этом мы поговорим позже, а пока что давайте попробуем понять их на интуитивном уровне, рассмотрев следующий пример. Итак, что же такое объекты? На самом деле мы каждый день взаимодействуем с ними: ложками, чашками, ручками, ноутбуками, мылом, стиральной машиной, трамваем, в конце концов, друзьями и коллегами (да простят авторов последние). Материальные границы всех этих сущностей позволяют нам отличать друг от друга объекты разных типов: ложки от чашек, чашки от плошек, утюги от кушаков и т. д. Отличать друг от друга объекты одного типа (ложки от ложек, чашки от чашек, игрока от игрока) нам помогает следующая характеристика — состояние. Например, ложка может быть алюминиевая или серебряная, то есть ее состояние определяется ее материалом, кроме того, ложка может быть алюминиевая теплая или серебряная холодная, в этом случае ее состояние выражено уже двумя характеристиками: материалом и температурой. И конечно, объект может позволять с собой что-то делать. Да-да, именно так! Поведение — это не только делать что-то самому, это еще и страдательный залог: позволять

---

[13] *Гимельрейх С.* «Игровая механика, динамика и машина состояний: Часть I» (https://gdcuffs.com/game_mechanics_deconstruct_1/), «Игровые механики: Часть II» (https://gdcuffs.com/game_mechanics_deconstruct_2/).

что-то делать с собой. Например, ложку можно помыть (и возможно, от мытья в горячей воде она изменит состояние с холодной на горячую), стиральную машину можно включить, студента можно отчислить, игровой объект активировать и т. п.

> **Состояние объекта** определяется совокупностью значений атрибутов.
>
> **Атрибут** (attribute) — свойство объекта.
>
> **Поведение объекта** определяется совокупностью методов.
>
> **Метод** (method) — вариант поведения объекта; отметим, что хоть в Blueprints и применяется название «функция», суть явления остается той же.

Самое интересное начинается дальше. Не все действия, которые может выполнять объект, доступны для вызова другими объектами. Одна часть поведения объекта доступна только ему, другая позволяет объекту взаимодействовать с внешним миром — она называется интерфейсом объекта. У интерфейса тоже есть формальное определение, которое мы приведем ниже, но пока что давайте еще немного задержимся на этапе объяснений и примеров. Особенности интерфейса можно отразить одной очень точной, но при этом, вероятно, и очень загадочной фразой: *объект позволяет делать с собой только то, что позволяет*. Это значит, что никакой другой объект не сможет сделать с объектом ничего, кроме тех действий, что доступны в интерфейсе объекта. Допустим, ваша стиральная машина имеет 10 режимов стирки. Если не влезать в программный код командного аппарата машины, вы никоим образом не сможете создать какую-либо комбинацию из доступных программ стирки. В данном случае вы обращаетесь именно к интерфейсу машины, который вам предоставляет фирма-производитель, — нажимаете на уже существующие кнопки, но создавать новые вы не можете.

> **Интерфейс** (interface) **объекта** — внешнее представление объекта, то есть всего, его атрибуты и методы. Описание интерфейса задает границы объекта путем включения или невключения тех или иных атрибутов или методов.

Итак, теперь мы знаем достаточно, чтобы понять суть классов. Помните однотипные объекты: ложки, чашки, плошки? Именно их описание — обозначение типа объекта — и приводит нас к понятию класса: класс ложек, класс чашек, класс плошек. Очевидно, что все ложки примерно одинаковы и отличаются совокупностями значений атрибутов (состоянием), все чашки тоже примерно одинаковы и тоже отличаются друг от друга состоянием, и т. д. С любым объектом одного типа можно взаимодействовать

примерно одинаково — ложками черпать, в чашки наливать или насыпать (в крайнем случае разбивать о пол).

> **Класс** (class) — описание однотипных объектов, то есть их интерфейса.

Иначе говоря, *класс можно представить в виде шаблона, описывающего возможные состояния и поведение объектов данного класса.* Объект, соответствующий этому шаблону, называется **экземпляром** (instance) **класса**.

Если вы вспомните наши рассуждения о типах данных, то заметите некоторое сходство с разговором о классах. И эта общность совсем не случайна, фактически классы задают новые типы данных. Когда мы в программе описываем экземпляры класса (объекты), происходит фиксация их принадлежности к типу. Например, переменная A связывается с объектом типа «Чашка», а переменная B связывается с объектом типа «Ложка». И эта принадлежность к типу позволяет нам пользоваться описанными выше особенностями классов и объектов.

Программа считается объектной, если[14]:

1) она использует в качестве основных конструктивных элементов объекты, а не алгоритмы;
2) каждый объект программы является экземпляром определенного класса;
3) классы образуют иерархии.

Раз уж мы заговорили об иерархиях, настало время познакомиться с перечнем базовых принципов ООП:

— объектовая инкапсуляция (object encapsulation);
— наследование (inheritance);
— полиморфизм (polymorphism).

Каждому из этих принципов можно посвятить отдельную главу, и даже не одну, но подробный разбор этих тем является предметом специализированных учебников программирования, а в рамках данной книги мы попытаемся дать лишь общее представление о принципах ООП, тем самым помогая читателю справиться с материалом

[14] *Буч Г., Максимчук Р. А., Энгл М. У., Янг Б. Д., Коналлен Д., Хьюстон К.А.*. Объектно ориентированный анализ и проектирование с примерами приложений. М. : Вильямс, 2017.

книги и не пугая его обилием теории. Кроме того, полученная информация позволит вам впоследствии легче подойти к изучению «взрослого» ООП.

Будет нелишним отметить, что среди прочего базовые принципы ООП нацелены на поддержку повторного использования кода. Если какой-то фрагмент программы можно вызвать повторно или же сэкономить время на написании нового кода за счет «отсылки» к существующему, то ленивый (в данном случае это синоним рациональности) программист обязан так и сделать. Дублирование и создание плохо продуманных, малоразличимых между собой фрагментов кода является антипаттерном программирования (шаблоном «так делать нельзя») под названием WET (We Enjoy Typing). Эти же явления породили и шаблон (паттерн) DRY (Don't Repeat Yourself).

Как правило, классы образуют иерархии, происходит это благодаря механизму наследования. С точки зрения пишущего код человека наследование — это способ создать один тип данных на основе другого типа данных, о котором мы уже все знаем (вспомните определение классов и объектов) и не желаем перепечатывать один и тот же код.

Формально наследование можно охарактеризовать так:

— Происходит не только наследование интерфейса как обязательство предоставить «не худший интерфейс», но и наследование реализации.

— Существуют правила совместимости типов класса-потомка и класса-предка.

— Переопределение виртуальных методов позволяет единообразно использовать полиморфизм.

Как и в случае алгоритмов, для визуализации классов тоже существует специальная нотация — язык UML (**Unified Modeling Language**), которым мы будем пользоваться в дальнейших пояснениях, при этом стараясь не переусложнять схемы. При помощи UML классы можно изображать в виде ясных диаграмм, которые так и называются — **диаграммы классов**. В UML классы изображаются в виде прямоугольников, в верхней секции которых размещается название класса (имя типа), в средней — атрибуты, а в нижней — методы, составляющие интерфейс класса (рисунок 123). На рисунке видно, что все атрибуты и методы помечены знаком минус «–», его следует воспринимать не как маркер списка, но как специальный символ, указывающий, что доступ к методу и атрибуту возможен только внутри кода класса, любой другой программный код не сможет этим воспользоваться. Открытый доступ помечается

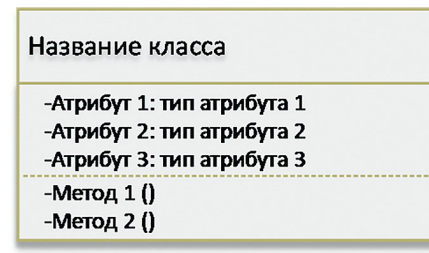

Рисунок 123. Общий вид класса в нотации UML

знаком «+», а защищенный — знаком «#» (диез). Мы еще не говорили о видах доступа к данным, пока просто примите к сведению эту информацию.

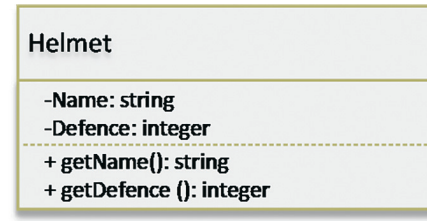

Рисунок 124. Диаграмма класса Helmet, представляющего в программе шлем

Давайте сконструируем класс, представляющий в игре предмет инвентаря типа «шлем». Основные атрибуты шлема: название и количество единиц защиты; интерфейс: получение названия шлема, получение значения количества единиц защиты. На рисунке 124 изображена UML-диаграмма класса Helmet, в нашей программе новый тип данных будет иметь такое же название. Мы должны указать типы данных для атрибутов класса, поэтому для имени Name установим тип string, а защите Defence — целочисленный тип integer. Поскольку мы хотим иметь возможность во внешнем коде обращаться к данным шлема и получать информацию о его названии и количестве очков защиты, методы getName() и getDefence() помечены плюсиком, а после двоеточия указан тип возвращаемого методом значения. Спроектированный нами класс выглядит неплохо, но чуть позже вы узнаете, как его можно сделать еще лучше.

Наследование на диаграммах классов изображается сплошной стрелочкой, направленной от класса-наследника к классу-родителю, например на рисунке 125 показан класс-родитель Class 2 и два его наследника: Inherited Class 1 и Inherited Class 2.

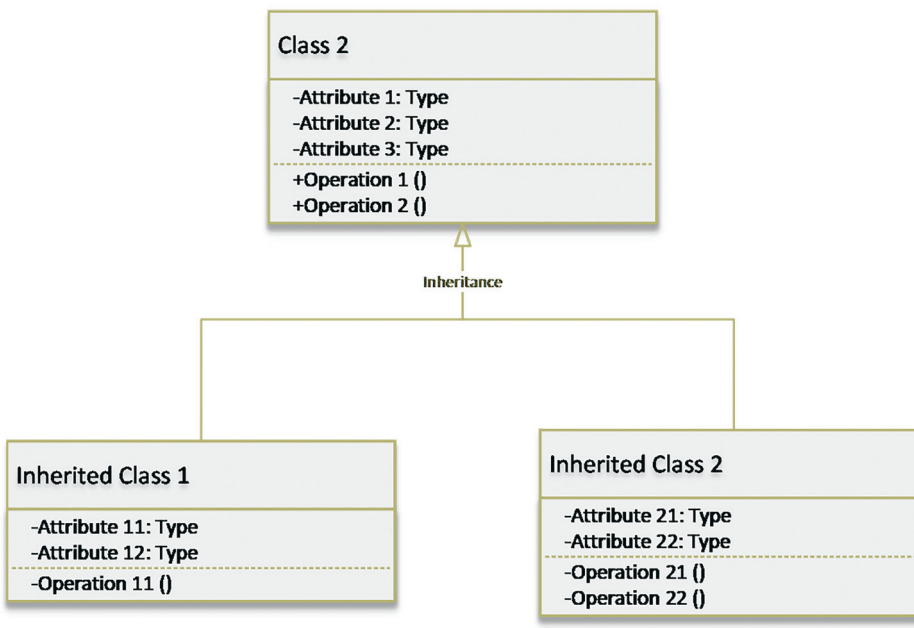

Рисунок 125. Пример стандартной заготовки родителя и двух наследников (изображение из программы моделирования Microsoft Visio)

Возвращаясь к примеру со шлемом (тип Helmet), рассмотрим ситуацию расширения иерархии типов. Предположим, что у нас есть еще один вид шлемов: магический шлем, добавляющий параметры магической защиты и позволяющий владельцу быть невидимым. Очевидно, что в этом случае просто объектом типа Helmet со значением атрибута Name = "Magic Helmet" обойтись не удастся, потому что часть новых данных попросту негде разместить. Для решения этой проблемы можно создать новый тип данных MagicHelmet, который будет дополнять структуру и интерфейс класса Helmet. Воспользуемся концепцией наследования, чтобы не дублировать код, и таким образом расширим (иногда говорят — специализируем) тип Helmet до типа MagicHelmet.

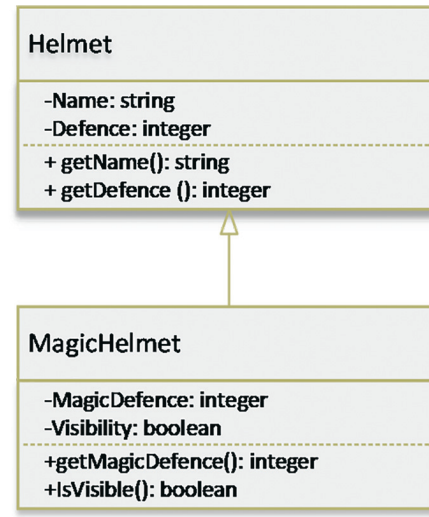

Рисунок 126. Иерархия наследования классов Helmet и MagicHelmet

На рисунке 126 представлена диаграмма возникшей иерархии наследования. Как видите, в интерфейс класса MagicHelmet добавлены методы getMagicDefence() и IsVisible(), но при этом надо не забывать, что на самом деле в интерфейсе этого типа содержатся четыре метода, так как еще два были унаследованы из типа Helmet.

Итак, мы немного разобрались с наследованием, и теперь можем перейти к следующему базовому принципу ООП и поговорим об объектовой инкапсуляции. Если вы сталкивались с декомпозицией программ по частям процессов, вы знаете, что такое функция (или процедура, метод, подпрограмма). Вызов функции — это по большому счету обращение к размещенной в ней (скрытой или инкапсулированной) реализации алгоритма. Начинающие разработчики, пользуясь библиотечными функциями, не всегда знают или просто не задумываются о том, как это реализовано. Все что требуется — передать в функцию при ее вызове верные параметры и корректно обойтись с возвращаемым значением. Так снаружи выглядит механизм процедурной инкапсуляции, который скрывает от нас детали реализации отдельных подпрограмм. Часто говорят, что инкапсуляция скрывает детали реализации. Разберемся, а что же скрывает объектовая инкапсуляция в ООП? Короткий ответ: состояние объекта.

Говоря формально, объектовая инкапсуляция:

— объединяет инкапсуляцию данных и инкапсуляцию процессов;

— основана на описании состояния и поведения объекта на уровне интерфейса общего шаблона объектов с идентичным поведением — класса;

— позволяет управлять видимостью элементов интерфейса;

— позволяет единообразно сформулировать понятие наследования.

Помните наши диаграммы классов? Знаком «-» мы обозначали закрытые члены классов, они скрыты внутри объектов, и только сами эти объекты знают о существовании этих атрибутов и методов; знаком «+» отмечались открытые члены классов, доступные любому вызывающему коду через объект этого типа; а вот про отметку «#» мы лишь упомянули, что она делает члены класса защищенными. Это означает, что снаружи объекта такой член класса не виден, но в отличие от закрытого атрибута или метода он доступен для наследования. Если в родительском классе есть защищенные члены, то они будут доступны всем классам-наследникам. По этой причине иногда говорят, что наследование нарушает инкапсуляцию или что при наследовании детали реализации родительского класса становятся доступными наследнику. Мы не будем углубляться в разбор этого вопроса, в полном объеме он обсуждается в специальной литературе по ООП и его философии.

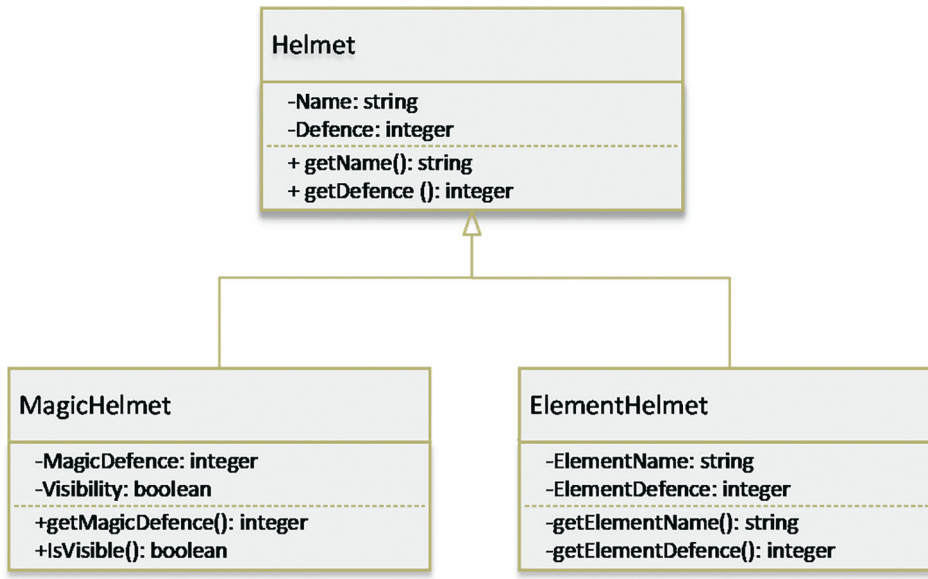

Рисунок 127. Иерархия наследования классов Helmet, MagicHelmet и ElementHelmet

Итак, нам осталось разобраться только с полиморфизмом. Само название этой концепции (поли-) говорит о том, что она связана со множественностью форм, давайте попробуем разобраться, о чем здесь идет речь. Благодаря знакомству с принципом наследования понять суть полиморфизма будет легче. Итак, предположим, что в нашей иерархии шлемов появился еще один — шлем стихий (ElementHelmet). Помимо свойств простого шлема шлем стихий дополнительно характеризуется названием

стихии, от которой он защищает игрока (ElementName), и степенью этой защиты (ElementDefence), поэтому мы добавили в интерфейс класса соответствующие методы для доступа к этим свойствам.

О чем нам говорит рисунок 127? Он сообщает, что классы MagicHelmet и ElementHelmet расположены на одном уровне иерархии наследования, то есть в некотором смысле являются «братьями»; при этом они оба являются описанием шлема, то есть могут быть соотнесены с типом Helmet. На уровне кода это означает, что мы можем создать переменную типа Helmet и связать ее как с объектом типа MagicHelmet, так и с объектом типа ElementHelmet. Более того, нам даже не требуется выяснять тип объекта, ссылка с типом родительского класса позволит вызвать любой метод, описанный в его интерфейсе. Это и есть работа полиморфизма, который позволяет избежать проверки типа данных с помощью конструкции выбора. Вызывая методы getName() и getDefence(), мы можем быть уверены в их существовании, потому что любой из классов-наследников принадлежит к типу Helmet, в котором имеются эти два метода.

Но если мы попробуем через созданную нами переменную родительского типа обратиться к тем элементам интерфейса, которые определены в классах-наследниках, нас ожидает фиаско. Выяснится, что класс Helmet понятия не имеет о запрашиваемых членах классов (например, методе IsVisible() из типа MagicHelmet), и поэтому через переменные своего типа обращаться к ним не может. Возможно ли решить эту проблему? Конечно! Полиморфизм позволяет задействовать позднее связывание и выбирать поведение в зависимости от фактического типа объекта, но это опять-таки тема для книги, посвященной ООП.

Отношения между классами

Упомянутое выше наследование — хотя и является основой основ ООП — относится к уровню отношений между классами. Очевидно, что отношения между классами определяют специфику отношений и между экземплярами (объектами) этих классов.

Самый простой вариант отношений между классами — это их **независимость** (independency) друг от друга. Понятно, что и объекты, порождаемые такими классами, тоже будут независимы. Несмотря на то что на первый взгляд концепция независимости выглядит прозрачной, она не так проста. Поэтому мы на некоторое время задержимся на ней, чтобы разобраться в этой теме более детально. Если вы только недавно познакомились с ООП, попробуйте ответить на вопрос — что значит «независимость объектов»?

> **Время жизни объекта** (жизненный цикл) — временной промежуток между созданием объекта и его уничтожением.

Говорят, что между объектами существует зависимость, если один объект может каким-либо образом воздействовать на время жизни другого объекта, например

создавать его, удалять или даже просто «знать» о существовании этой другой сущности. Конечно, выполнения только этого условия недостаточно, существуют и другие характеристики. Возвращаясь к концепции независимости, отметим, что время жизни независимых объектов друг с другом никак не связано.

Отношение **зависимости** (dependency) в противовес независимости описывает более общий тип отношений между классами, говоря о том, что один класс зависим от особенностей реализации другого класса, при этом не указывая на особенности этой зависимости.

Между классами могут существовать следующие виды отношений: **ассоциация**, **использование**, **агрегация** и **композиция**. Последние два являются наиболее жесткими. Давайте рассмотрим их поближе.

Отношение **ассоциации** (association) появляется в случае взаимного включения классов, в таких случаях говорят, что между ними возникает двунаправленная связь. Это означает, что объекты классов «знают» о существовании друг друга, в коде же это знание выражается наличием ссылок соответствующих типов в полях классов. С ассоциациями связано понятие **множественности** или **кратности** ассоциации, то есть числа объектов, участвующих в ассоциации с каждой стороны:

— 1:1 (один к одному) «игрок — активная (текущая) локация»;

— 1:n (один ко многим) «клан — игроки»;

— M:N (многие ко многим) «открытые локации — игроки».

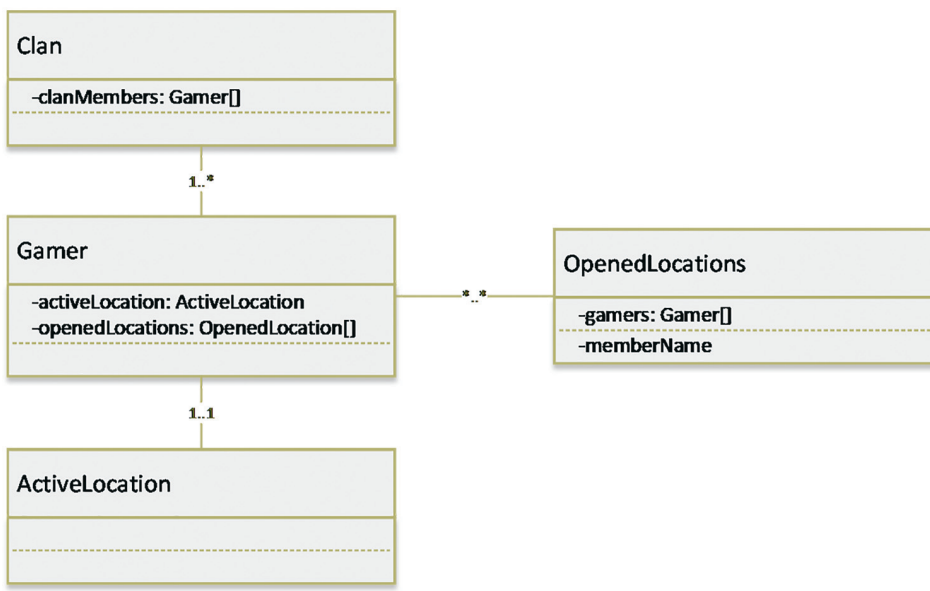

Рисунок 128. Пример обозначения отношения «ассоциация» (разной кратности) между классами

Чтобы было легче понять сказанное, изобразим на UML-диаграмме примеры ассоциаций разных кратностей (рисунок 128). Обычная ассоциация изображена сплошной линией, над которой указывается кратность. Для ассоциации с кратностью 1:1 обозначение показателя может быть опущено. Для иллюстрации отношений «один ко многим» и «многие ко многим» мы будем использовать массивы, но на практике вместо списков могут быть и другие типы контейнеров.

Некоторым промежуточным вариантом между независимостью и зависимостью является отношение **использования** (use), когда один класс при реализации своих методов обращается к интерфейсу (то есть к методам) другого класса. Пожалуй, отношение использования является одним из самых распространенных, но, как и независимость, не означает наличия каких-либо обязательств по управлению временем жизни объектов, классы которых связаны таким образом. При этом отношения **ассоциации** и **использования** между классами всегда отражаются в объектах-потомках, как это показано на диаграмме выше. Если объект *x* вызывает метод объекта *y*, то говорят о **явной ассоциации** между классами *X* и *Y*. А если объект *x* вызывает метод объекта *z*, который, в свою очередь, вызывает метод объекта *y*, то говорят, что между классами *X* и *Y* существует **косвенная ассоциация**.

Рассказ об ассоциациях будет неполным, если мы не упомянем такой ее специфический вид, как агрегация (включение) (aggregation). Агрегация — это отношение вида «являться частью» или «часть/целое» (также известна как HAS A). Другими словами, когда что-то целое можно представить в виде составных частей, то между целым и составными частями возникает агрегация. Примером могут являться детальки Lego, которые существуют каждая по отдельности, но при этом из них можно собрать какой-либо объект. С точки зрения программирования отношение агрегации позволяет «включать» один класс в состав другого. Выглядит это так: в классе объявляется поле, тип которого является другим классом. Слово «включать» забрано в кавычки, потому что в коде не будет фактического описания одного класса внутри другого. Такая ситуация описывается отношением **вложения**, и в этом случае говорят о внутреннем и внешнем классе. Кстати, многие языки программирования позволяют описывать внутренние классы.

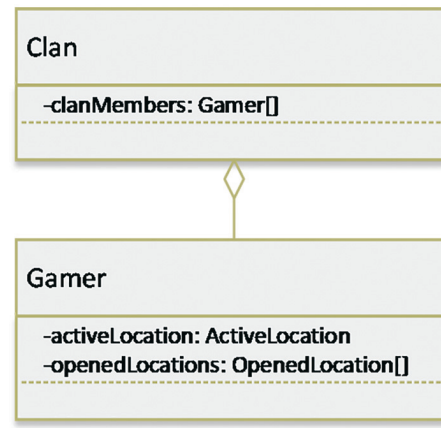

Рисунок 129. Представление отношения агрегации на UML-диаграмме классов

Уже рассмотренный пример ассоциации, в котором игроки объединялись в клан, может быть описан более строго: клан — целое, игроки — части (на UML-диаграммах агрегация изображается стрелочкой с не закрашенным ромбом, ведущей от части к целому, см. рисунок 129). Очевидно, что, если мы уничтожим клан,

удаление не должно коснуться игроков, так как они являются самостоятельными сущностями и должны продолжать «жить». Этот момент является главной особенностью агрегации — она не управляет временем жизни своих составных частей, то есть целое не берет на себя обязательства по уничтожению частей при завершении своего жизненного цикла.

Вероятно, вы уже обратили внимание на то, что в рассказе об агрегации используется то контекст классов, то объектов. Безусловно, это всего лишь легкая терминологическая вольность, так как мы все же говорим о классах, но ведь **отношение агрегации между объектами** и является следствием агрегации между классами.

Если смотреть с точки зрения управления временем жизни, то совсем иначе выглядит такой вариант агрегации как **композиция** (композитная агрегация, composition). В этом случае и целое (объект-агрегат) не существует без своих частей, и его части тоже не существуют без своего «агрегатора». Например, если в игре вы имеете дело с домом, то при его строительстве можно выбирать разные варианты крыши, балкона, фасада и прочих элементов, но если вы захотите разрушить постройку, то снесете ее целиком вместе с крышей, балконом и фасадом (на диаграмме классов UML композиция, как и агрегация, изображается стрелкой с ромбом, но в этом случае ромб будет закрашен, отражая тем самым неразрывную связь частей и целого, см. рисунок 130).

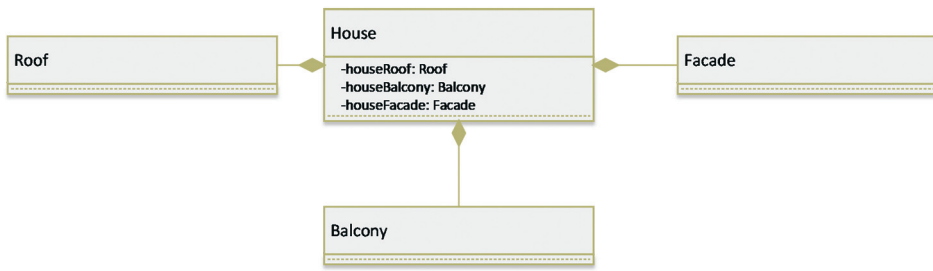

Рисунок 130. Представление композиции на UML-диаграмме классов

Мы уделили достаточно много времени концепции наследования. Пожалуй, ко всему сказанному в предыдущем разделе можно добавить еще одну важную деталь: **наследование** (inheritance) — это отражение отношения «являться экземпляром» или «частное/общее» («это есть»/IS A). Именно поэтому про объекты-наследники иногда говорят, что они специализируют родительский тип. С формальной точки зрения, наследование отражает иерархию классов как иерархию **обобщения** или как иерархию **специализации**. И так оно и есть, потому что каждый наследник, добавляя свои черты к родителю, уточняет последнего, но при этом каждый наследник продолжает быть сущностью родительского типа. Возвращаясь к недавнему примеру со строительством — дом с мезонином, конечно, уточнит понятие дома, но при этом все равно останется домом.

С предметом наследования в философии ООП связано очень много интересного, сложного и запутанного. В дальнейшем мы не будем рассматривать эту тему более детально, чем это уже сделано, но если вы всерьез увлечетесь программированием, то обязательно встретитесь с такими понятиями, как множественное наследование, конфликты имен, концепция виртуализации, абстрактные классы и т.д.

Завершим этот раздел кратким рассказом об отношении **инстанцирования** [instancing]. При этом типе отношений существует не только связь между классом и его экземплярами, но и, как это ни странно, связь одного класса с другим. Последнее возможно в том случае, если в выбранном вами языке программирования присутствуют обобщенные [generic] типы, являющиеся реализацией парадигмы обобщенного программирования, которая, к слову, имплементирована и базовом языке движка UE — C++.

Классы в Unreal Engine

Классы в UE бывают двух видов: те, что можно создавать на уровне (наследники типа Actor), и те, которые на уровне создать нельзя (наследники типа Object, в родителях нет типа Actor).

Ранее мы всегда использовали самый базовый класс, который можно создать на уровне: Actor. Для просмотра цепочки наследования откройте в Blueprint раздел Class Settings интересующего класса. Здесь же будет присутствовать и некоторая дополнительная информация о классе-родителе.

В UE5 содержится огромное количество классов, любой из которых можно посмотреть в окне Pick Parent Class, выбрав нужный элемент из раскрывающегося списка. Давайте попробуем поработать с новым для нас классом — Pawn. Родителем этого класса является тип Actor, представляющий в игре героя, которым можно управлять.

Чтобы создать класс Pawn, в окне Pick Parent Class выберите пункт Pawn.

Назовите его BP_Player и откройте. Персонажу с видом от третьего лица нужны две вещи:

1) тело;
2) камера.

Добавьте в блюпринт компонент Cube и сделайте его корнем. Далее добавьте компонент Camera и установите ее немного позади и над кубиком, чтобы восприятие героя было таким, каким оно бывает в играх от третьего лица.