# UML

## Class

Сверху всегда пишется имя класса

Под ним идут атрибуты класса (свойства)

Последние — операции (методы)

- закрытые

+ публичные

# защищенные (может использоваться самим классом, всеми суб-классами и всеми классами внутри package

Затем следуют операции (методы класса)

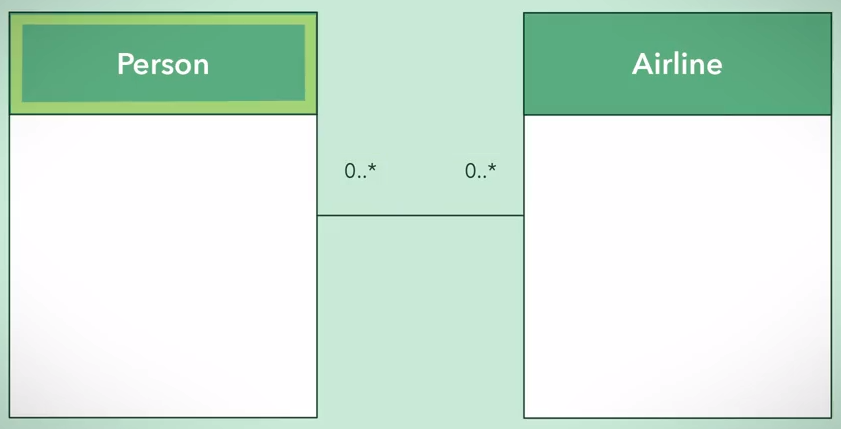
|  |
| --- |
| ClassName |
| Attributes:  -privateAttr  -publicAttr |
| Operations:  +setPublicAttr(String): void  -setPrivateAttr(int):void  +getPublicAttr():String |

# Decomposition

## Acquaintance

*использует, но не является*

Осведомленность — один обьект получает ссылку на другого, но не управляет временем его жизни. Loose partnership — свободное партнерство

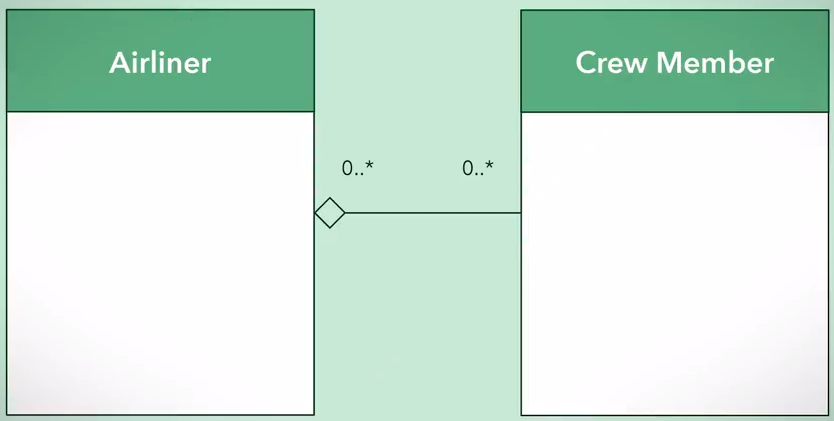


## Delegation

*не является*, *но содержит*.

Частью первого объекта является второй, но они могут существовать и раздельно. Обычно владелец поручает выполнение операции специалисту — включенному объекту.

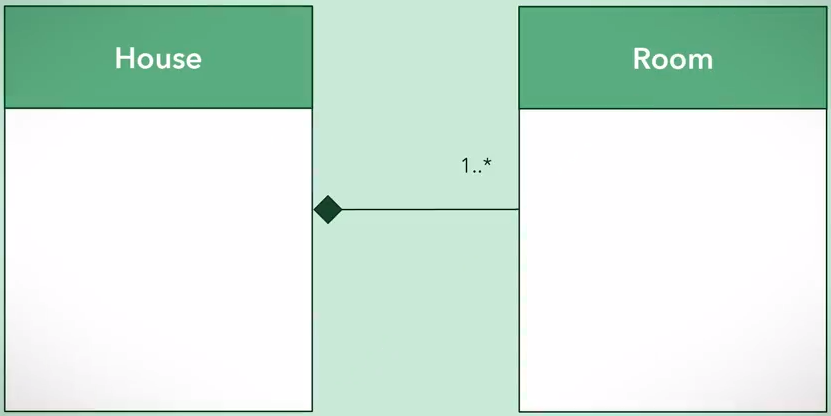
Пустой ромб указывает на объект-владелец. Другой конец указывает на объект, который включен во владельца



## Aggregation

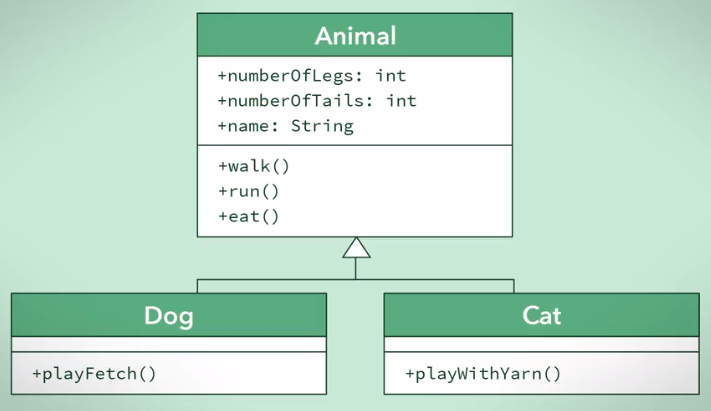
Один обьект является частью другого. Когда внешний обьект, умирает и внутренний.

Закрашенный ромб указывает на обьект-владалец. Другой конец указывает на обьект, который включен во владельца и который будет уничтожен после уничтожения владельца.



## **Generalization**

Вынос повторяющихся кусков отдельно. Вычленение родительского общего класса. На диаграмме указываетяся в форме треугльника, направленного на родительский класс со сплошной линией. Подклассы всегда находятся ниже родительского



**Интерфейс класса**

В Java можно определить класс как интерфейс, который описывает только сигнатуры методов (название метода, параметры на вход и выход), но не включает атрибуты (свойства) или тело методов

Если класс определен только как абстрактный интерфейс, перед его названием ставят «I». Классы, которые наследуют интерфейс, обязаны наследовать его полностью, при этом самостоятельно определить поведение каждого из методов. Один класс может наследовать несколько интрефейсов

На диаграмме указывается в форме треугльника, направленного на родительский класс с пунктирной линией

public interface IPublicSpeaking {

public void givePresentation();

public void speak();

}

public interface IPrivateConversation {

public void lowerVoiceVolume();

public void speak();

}

public class Person implements IPublicSpeaking, IPrivateConversation {

public void speak(){

System.out.println("hello");

}

}

**Sequence diagrams (схемы последовательности)**

Фигура человечка — актер

: Object — указание обьекта, пишется через двоеточие

Вертикальная пунктирная линия — время жизни обьекта

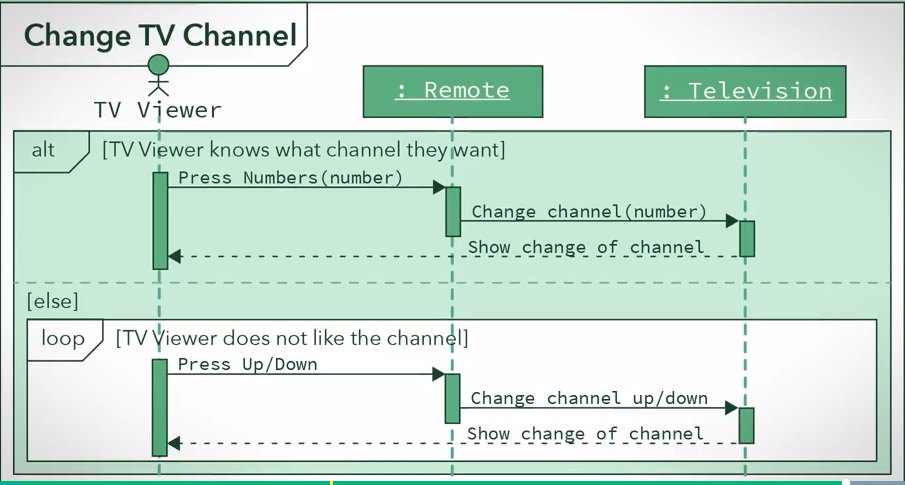
Вертикальный прямоугольник — обьект активен

Горизонтальная линия — взаимодействие обьекта с другими обьектами

На примере условие с циклом:

- юзер знает какой канал хочет смотреть. Он нажимает на пульт, пульт перенаправляет сигнал телевизору, телевизор отображает картинку для юзера

- юзер переключает каналы. Схема та же самая, но указан цикл



**State diagrams**

Может описывать одиночный обьект и иллюстрировать, как обьект реагирует на серию событий в системе

Точка состояния отображается кружком, описание состояния пишется рядом с прямоугольнике с закругленными краями. Прямоугольник поделен на 3 строки:

* название состояния — пишется сверху
* переменные состояния — пишутся посредние
* процессы проходящие в данный момент — в самом конце

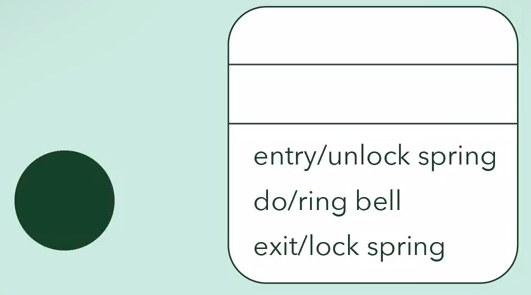
в свою очередь, процессы делятся на 3 вида:

- начальное состояние — процессы которые начинаются, когда текущее состояние стартует

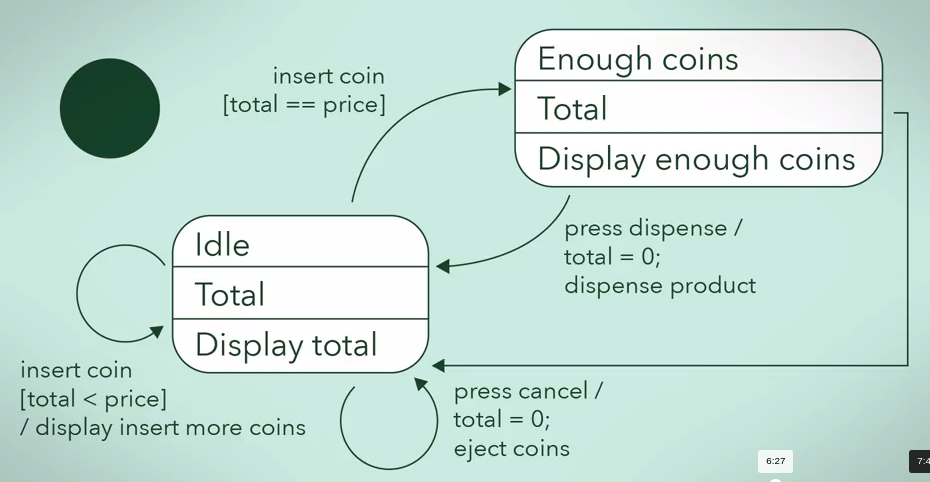
- состояние во время процесса — процессы которые происходят единожды или множество раз, когда обьект в этом состоянии

- состояние на выходе — процессы которые происходят, когда текущее состояние переходит в следующее состояние

*Пример будильника:* когда будильник готовится звенеть, он разблокирует колокольчик, затем звенит, затем блокирует колокольчик



Пример 2: автомат по выдаче шоколадок. Юзер вставляет монетку и получает шоколадку. Так же он может нажать «отмена»



**KRUTCHEN`S 4+1 VIEW MODEL**

Для того, что бы полностью смоделировать поведение и разработку программной системы, нужны разные точки зрения. Эта модель предполагает:

**Logical view**

Фокусируется на функциональности и необходимых объектах.

Разбирается, что система должна делать для удовлетворения потребностей клиентов с точки зрения функциональности, какие для этог нужны объекты.

Контекст — это сервисы, которые должны быть предоставлены конечным юзерам. На этом этапе включается UML class and state diagrams

* Создание словаря проблемы в рамках системы
* обозначение всех классов, их атрибутов и поведения

**Process view**

Фокусируется на процессах, полученных как результат работы обьектов в *logical view*. Разбирается с точки зрения эффективности системы или взаимодействия подпроцессов во время исполнения программы. Система рассматривается в разрезе производительности, конкурирующих запросов и тд. На этом этапе включают UML sequence and activity diagram. Разбирается через призму атрибутов качества, таких как

* производительность системы
* доступность системы

**Development view**

Фокусируется на внедрении стандартов и соглашений, таких как иерархическая структура ПО. Выбранные программный язык имеет сильное значение на конечную структуру и следовательно, привносит свои ограничения. Это распространяется на данные прожект менеджмента, такие как планирование, бюджет и задания

* Языки программирования
* библиотеки
* тулзы

**Physical view**

Фокусируется на физических компонентах системы и их взаимодействию — количество серверов, что на них будет находится и тд.

Тут строится deployment UML diagram

**Scenarios**

Сценарии описывают варианты использования, которые требовались конечными пользователями. Сценарии предоставляют контекст, чтобы помочь детализировать четыре *views*.

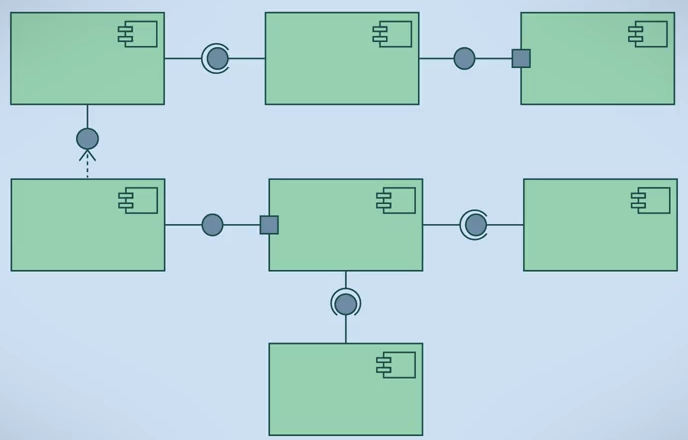
Каждый сценарий является скриптом, в котором описана последовательность взаимодействий между объектами и процессами. Это включает ключевые объекты, определенные в logical view, процессы, описанные в process view, иерархию и разные ветки, описанные в physical view. Сценарии их объединяют для цельной картины

Далеко не все системы нужно описывать через призму данной модели, часто некоторые view можно исключить. Например, если logical и development view изумительно похожи, они могут быть описаны вместе

**UML DIAGRAMS**

**Component diagram**

Компоненты — независимые инкапсулированные элементы системы. У каждого компонента есть интерфейс для взаимодействия с другими компонентами



Диаграммы компонентов — высокоуровневые структуры, на которых опущены детали

При разработке диаграмм компонентов сначала определяются Главные Объекты системы, затем библиотеки для системы, затем составляются отношения между компонентами

Коннекторы отношений

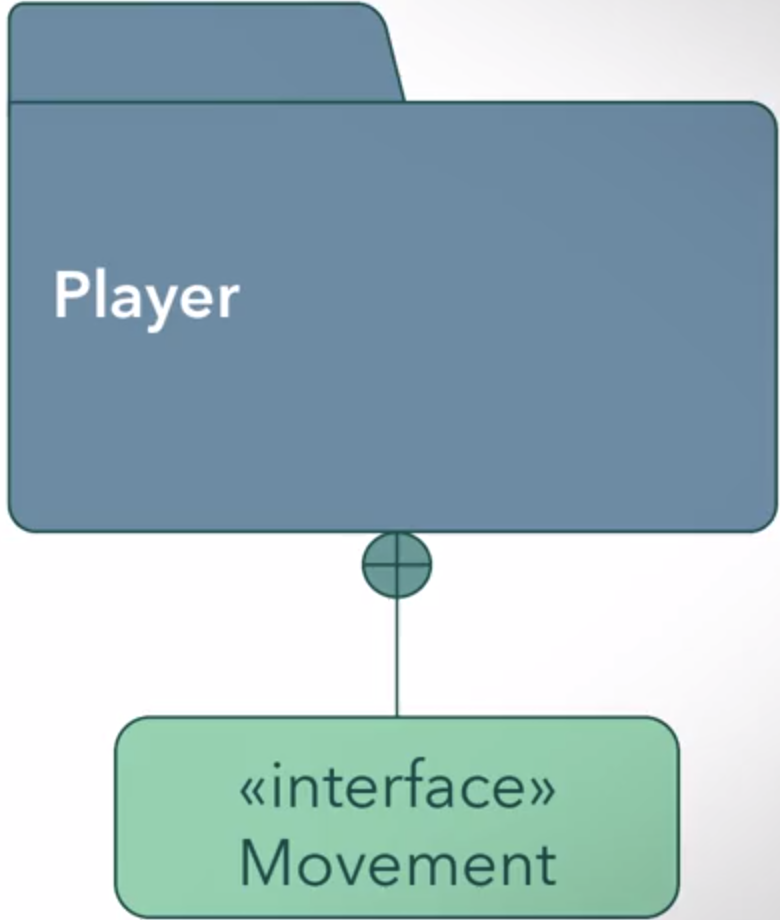
* полумесяц — socket connector — необходимый интерфейс — компонент принимает запрос, какие данные он может принять
* закрашенный круг — ball connector — предоставляемый интерфейс — компонент делает запрос и передает данные

**Package diagram**

Пакет группирует родственные элементы ПО, например данные, классы или по задачам юзерам или даже другие пакеты

Package так же определяет namespace для всех внутренних элементов

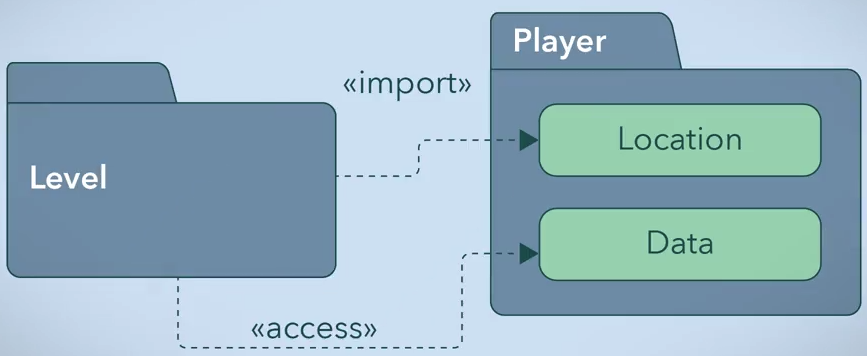
*Qualified name* — имя пакета + имя класса, которые делают его уникальным в системе

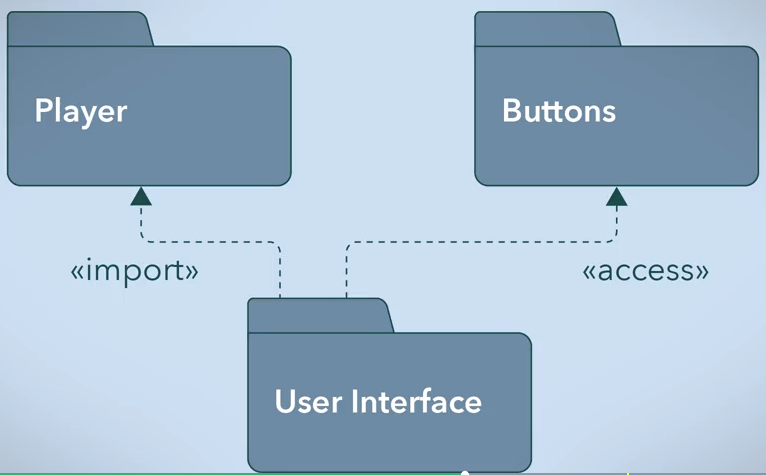


На диаграмме пакет можно детализировать, показав, что он включает в себя. На данной картинке пакет включает в себя интерфейс движения плеера (композиция)

Так же, можно указать внутрилежащие элементы, указав их доступность. На картинке 2, элемент *Audio* — публичный через интерфес, где необходимо указать полное *Qualified name* элемента

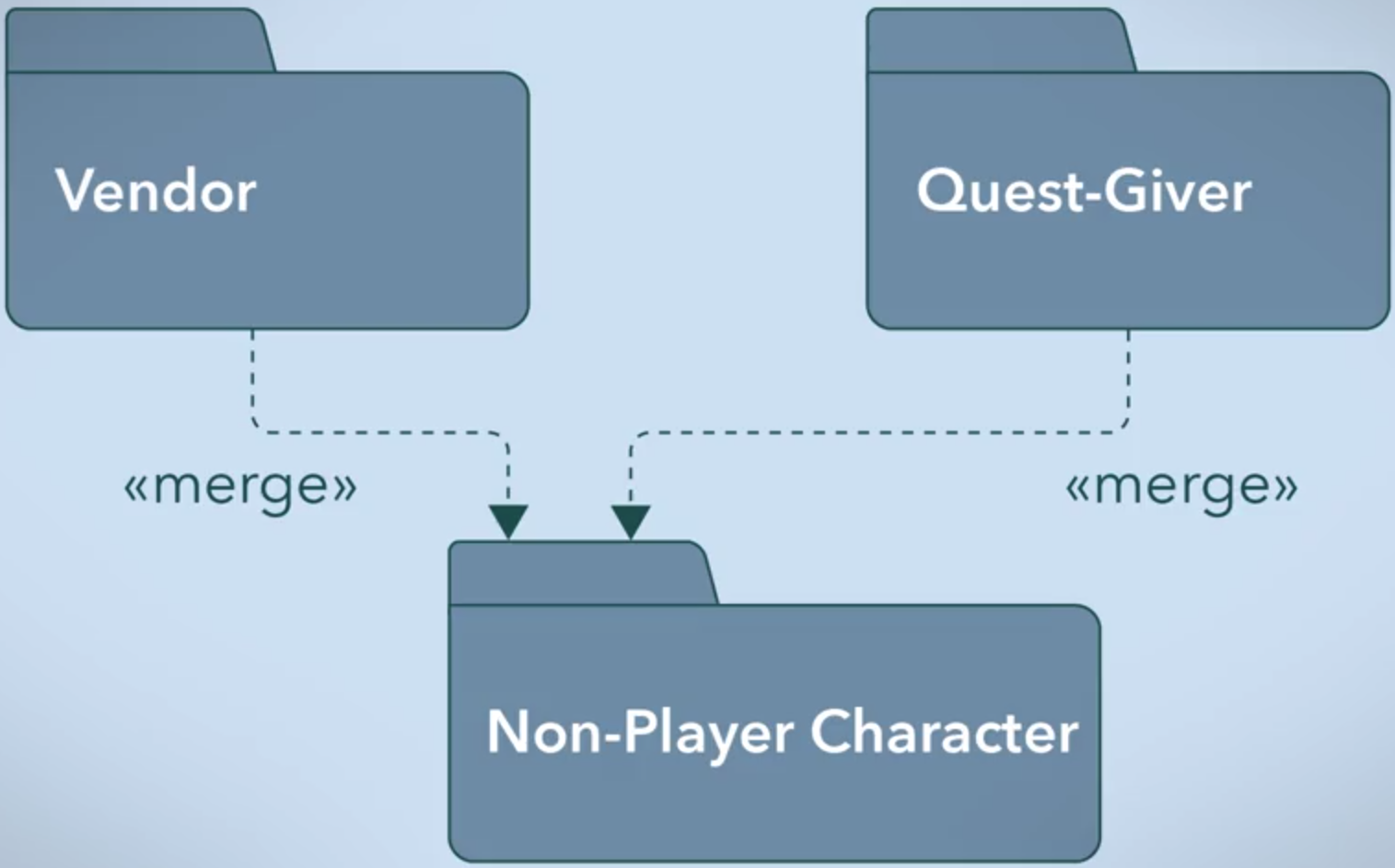
Пакет может импортировать элемент из другого пакета. Он даже может импортировать весь контент из другого пакета. Так же пакеты могут быть объединены. В данной картинке, *Level* импортирует *Location* из другого пакета, и это делает доступным элемент *Location* из пространства имен пакета *Level.*

*Level* так же импортирует *Data* из Player, но «*access*» флаг указывает на то, что *Data* является приватным свойством, потому этот элемент видим только внутри *Level*, и не является публичным



Тут происходит импорт целого пакета *Player* пакетом *User Interface,* делая публичными все видимые элементы пакета *Player*

Так же, *User Interface* импортирует *Buttons*, но делает все его видимые свойства приватными



Тут происходит объединение двух пакетов в один

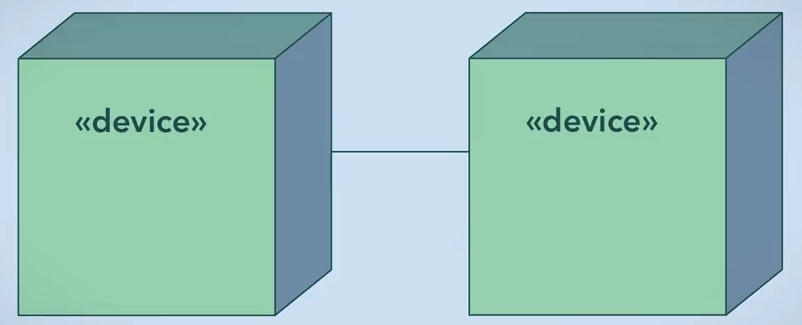
**Deployment diagram**

Это диаграма использует высокоуровневый обзор:

* Артефактов — физическая сущность программы. Например экзешник, конфигурационный файл и тд.
* Библиотек — любые подключенные third-party модули
* Главных компонентов
* Машин
* Девайсов

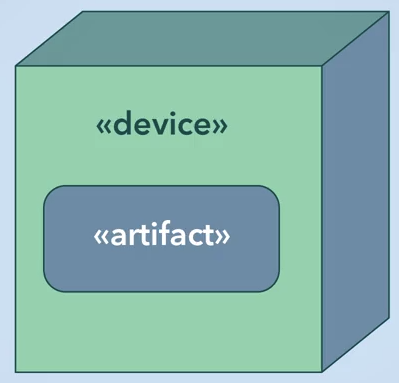
Существует два вида Deplyment diagrams:

* *Specification level diagram* — дает обзор артефактов и целей развертки, без указания специфических деталей, таких как конкретная OS. Он фокусируется на общем обзоре развертки, не включая детали
* *Instance level diagram —* более конкретный подход, который отображает специфичные артифекты по отношению к специфичным целям развертки. Например — exe файл для Windows и .sh для Linux. В частности, эта диаграма может указывать на специализированные машины и девайсы. This approach is used to highlight the differences in deployments along development, staging and release builds



Nodes — конкретные девайсы, обычно изображаются в виде 3d коробок. Например MacBook Pro будет отдельной нодой

Отношения между нодами обозначаются линией, обычно это обозначает протокол для коммуникации



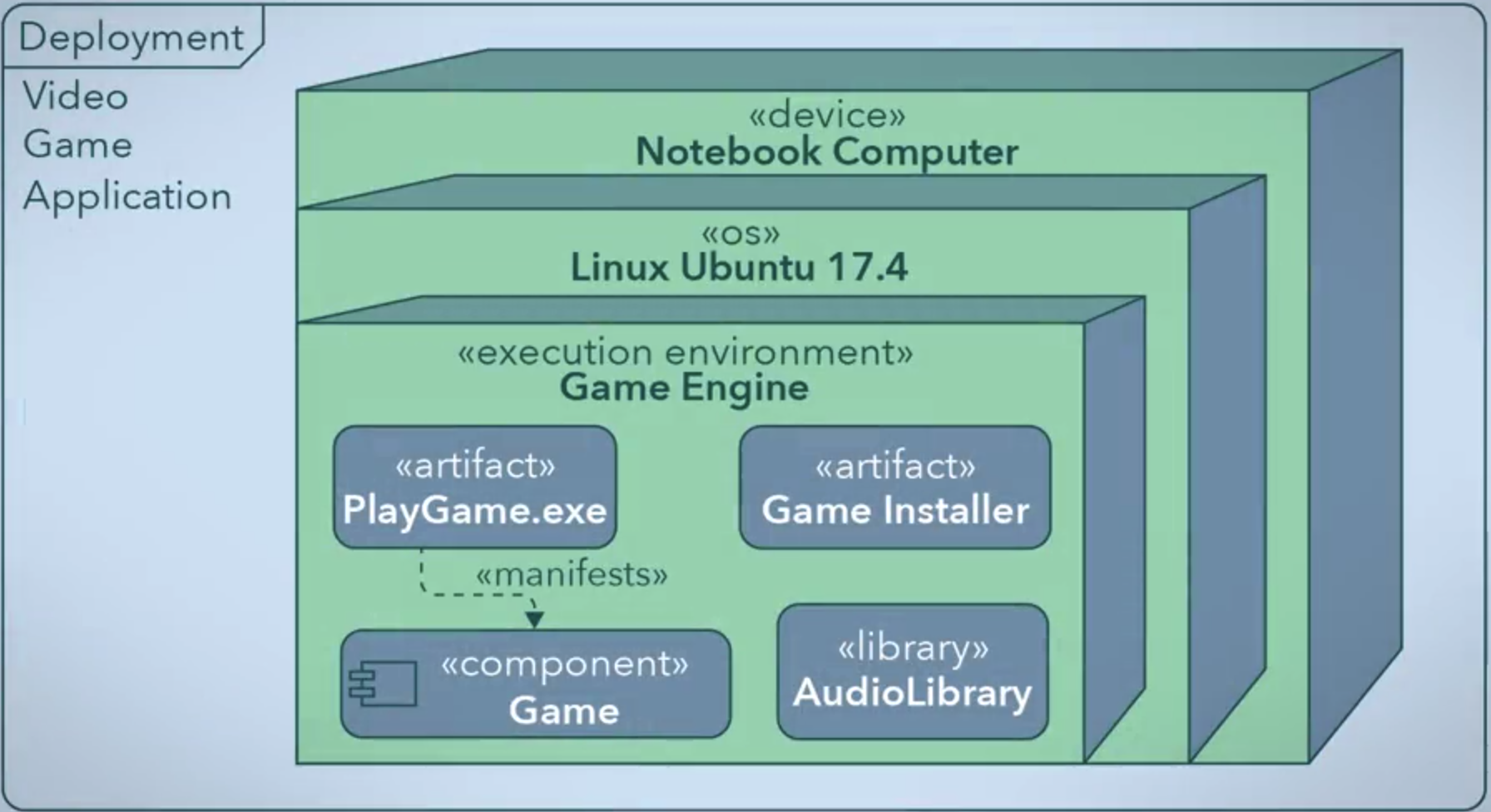
Если артефакт был развернут в ноде, это изображается «внутри» коробки. Это так же означает, что артефакт не может работать вне этого Node.



Манифестация — артефакт является физической реализаций софтверного компонента.

Class Player содержит всю функциональность, которую содержит компонент Player. Class Player появляется в результате компилирования компонента Player

Пример простой диаграммы. Тут exe включает целый компонент Game, который в свою очередь включает всю логику программы. В целях сокрытия деталей, логика программы опущена

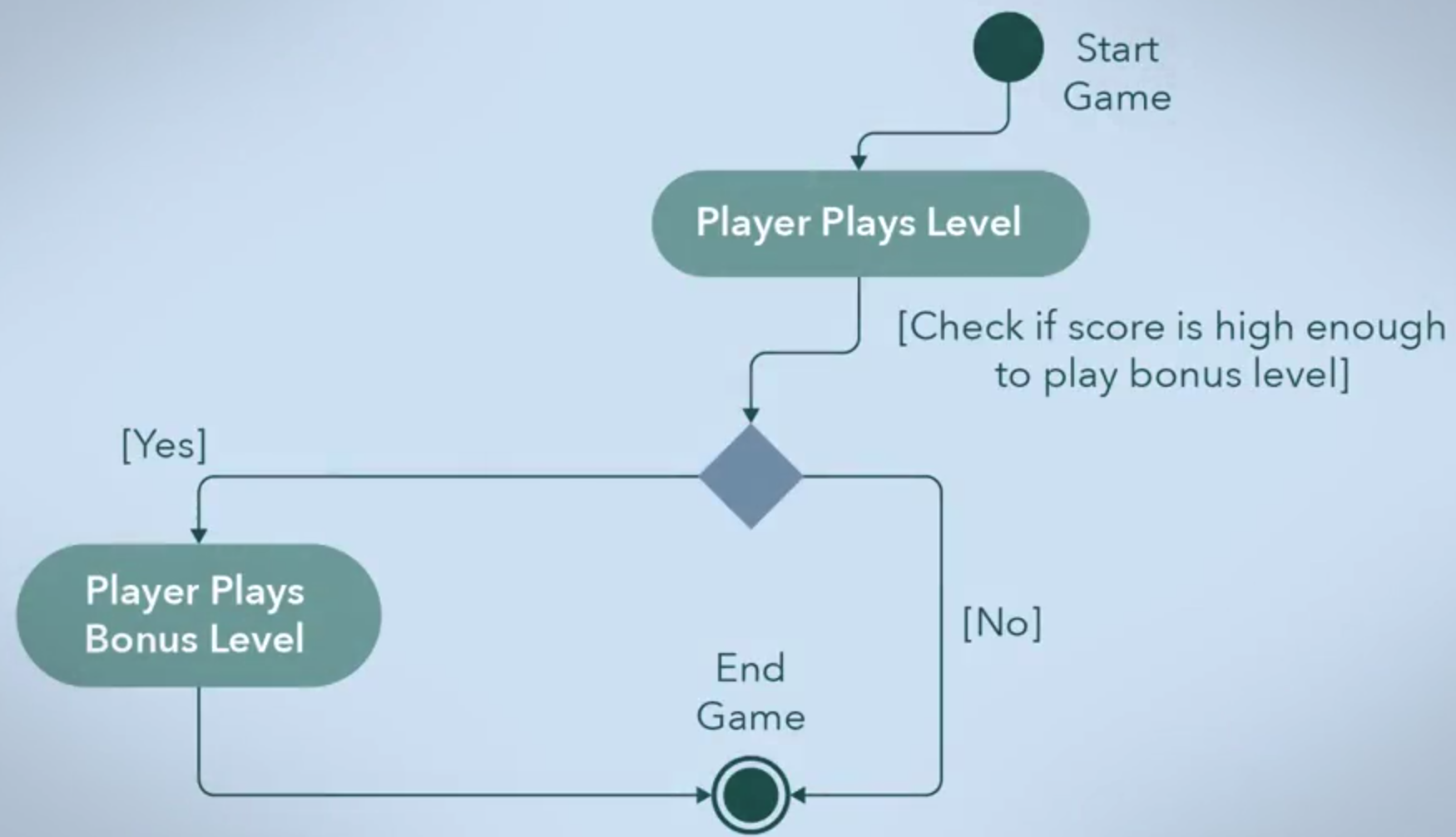


**Activity diagram**

Представляет контроль потока управления (control flow) в разных окружениях

Действия (actions) вызывают другие действия, например объекты создают новые объекты и т.д.

Цель это диаграммы — отобразить изменяющееся поведение системы, как поток управления переходит из одного действия в другое.

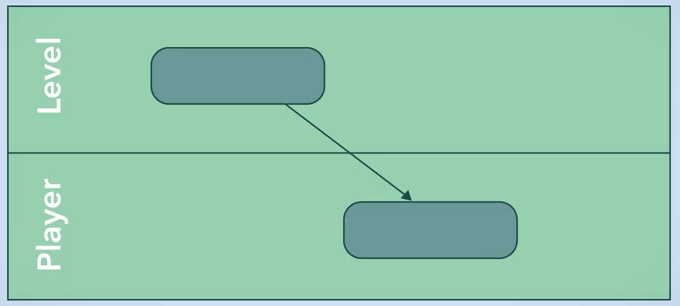
Круг — начало,

Двойной круг — конец,

Активность — в овалах,

Выбор — в ромбе

Так же могут быть параллельные активности, например игрок начал игру и параллельно начала играть музыка. Так же параллельные потоки могут быть объединены в один поток

*Partitions*

Используют, что бы отобразить конкурирующие активити. Partitions разделяют активити на разные категории, такие, в которых активити начинается или когда вовлекается юзер. Например, все активити видеоигры относительно уровней могут быть сгруппированы в одну группу, а все активити связанные с игроком — в другую