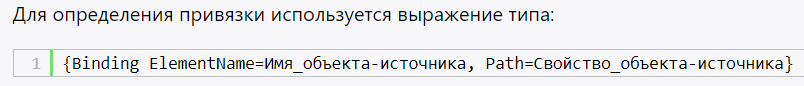
**ЛАБА 10**

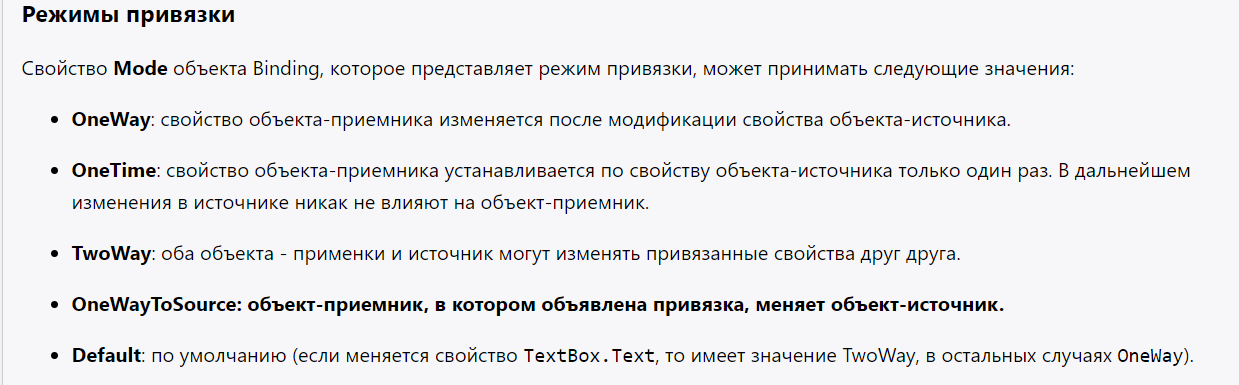
**1. Что такое Binding в WPF? Какие атрибуты мы можем указывать и для чего?**

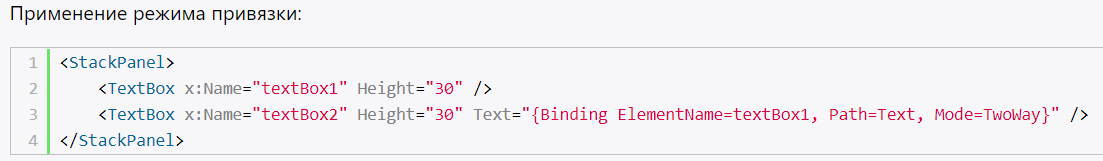
Привязка подразумевает взаимодействие двух объектов: источника и приемника. Объект-приемник создает привязку к определенному свойству объекта-источника. В случае модификации объекта-источника, объект-приемник также будет модифицирован.



То есть в данном случае у нас элемент TextBox является источником, а TextBlock - приемником привязки. Свойство Text элемента TextBlock привязывается к свойству Text элемента TextBox. В итоге при осуществлении ввода в текстовое поле синхронно будут происходить изменения в текстовом блоке.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_



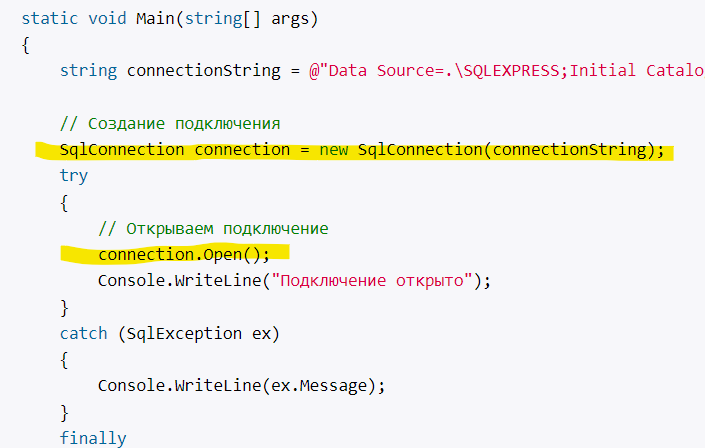


2. Что такое ADO.NET? Какие существуют режимы работы?

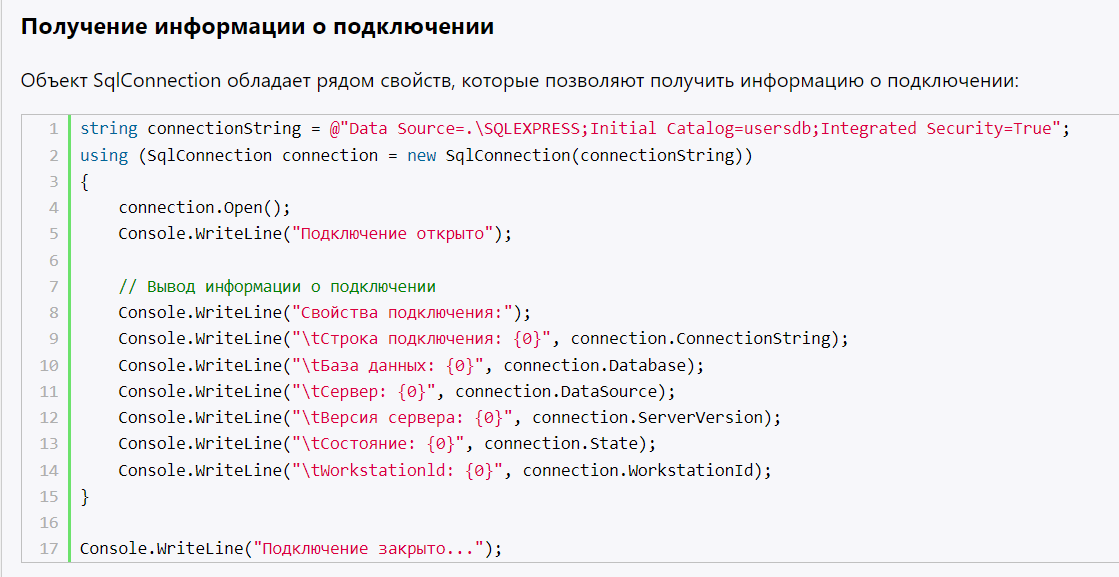
ADO.NET предоставляет собой технологию работы с данными, которая основана на платформе .NET Framework. Эта технология представляет нам набор классов, через которые мы можем отправлять запросы к базам данных, устанавливать подключения, получать ответ от базы данных и производить ряд других операций.

**3. Объясните назначение класса SqlConnection.**

Чтобы подключаться к базе данных, нам необходимо создать и использовать объект SqlConnection:



В конструктор объекту SqlConnection передается строка подключения, которая инициализирует объект. Чтобы использовать этот объект и подключаться к базе данных, мы должны выполнить его метод Open(), а после завершения работы с базой данных нам надо вызвать метод Close() для закрытия подключения.



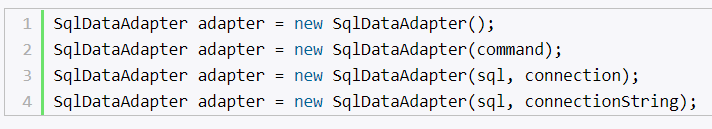
**4. Объясните назначение класса SqlDataAdapter**

Объект [SqlDataAdapter](https://docs.microsoft.com/ru-ru/dotnet/api/system.data.sqlclient.sqldataadapter?view=dotnet-plat-ext-5.0)служит мостом между [DataSet](https://docs.microsoft.com/ru-ru/dotnet/api/system.data.dataset?view=dotnet-plat-ext-5.0) SQL Server для извлечения и сохранения данных. Этот [SqlDataAdapter](https://docs.microsoft.com/ru-ru/dotnet/api/system.data.sqlclient.sqldataadapter?view=dotnet-plat-ext-5.0) мост предоставляется путем сопоставления[Fill](https://docs.microsoft.com/ru-ru/dotnet/api/system.data.common.dbdataadapter.fill?view=dotnet-plat-ext-5.0), который изменяет данные в [DataSet](https://docs.microsoft.com/ru-ru/dotnet/api/system.data.dataset?view=dotnet-plat-ext-5.0) соответствии с данными в источнике данных и [Update](https://docs.microsoft.com/ru-ru/dotnet/api/system.data.common.dbdataadapter.update?view=dotnet-plat-ext-5.0)который изменяет данные в источнике данных в соответствии с данными, [DataSet](https://docs.microsoft.com/ru-ru/dotnet/api/system.data.dataset?view=dotnet-plat-ext-5.0)используя соответствующие инструкции Transact-SQL к источнику данных.

Представляет набор команд данных и подключение к базе данных, которые используются для заполнения [DataSet](https://docs.microsoft.com/ru-ru/dotnet/api/system.data.dataset?view=dotnet-plat-ext-5.0) и обновления базы данных SQL Server. Этот класс не наследуется

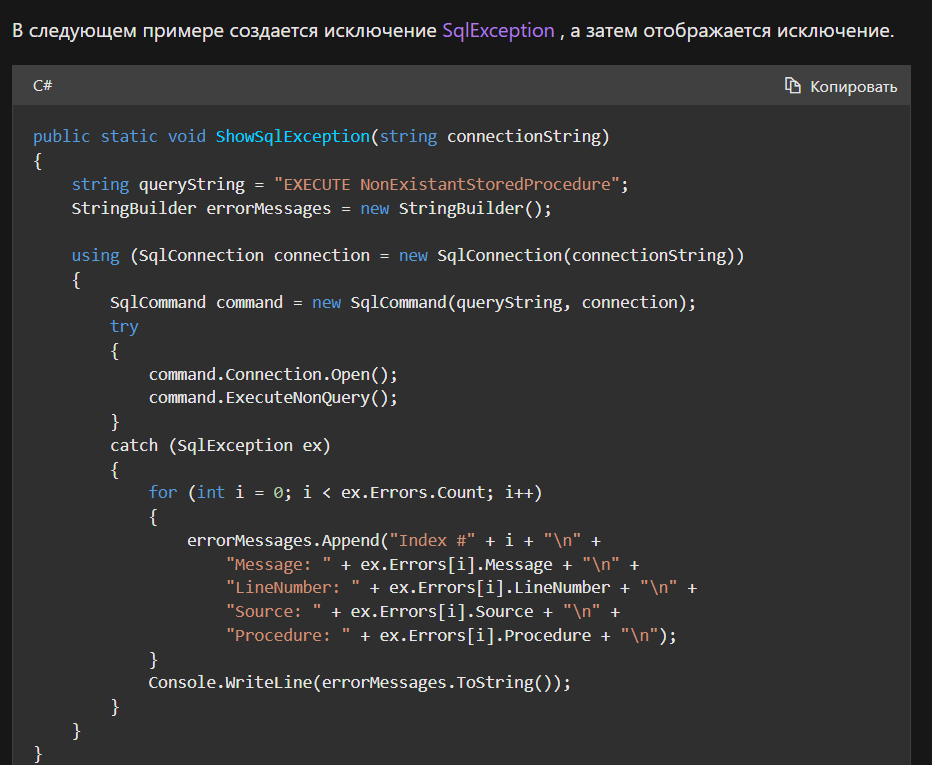
Для работы с БД SQL Server пакет Microsoft.Data.SqlClient как раз предоставляет реализацию DataAdapter в виде класса **SqlDataAdapter**.

Для получения данных через объект SqlDataAdapter необходимо организовать подключение к БД и выполнить команду SELECT. Есть несколько способов создания SqlDataAdapter:



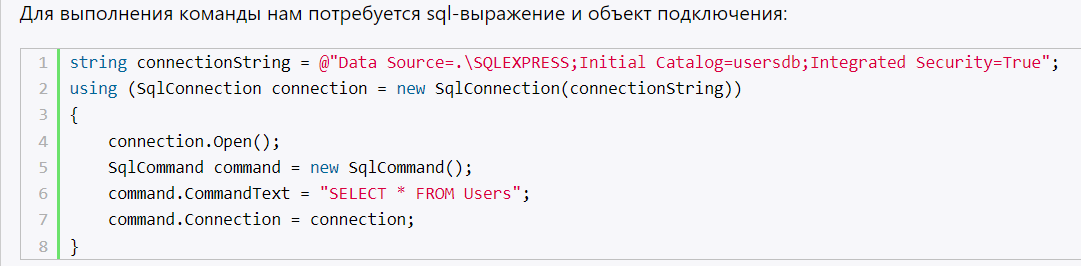
**5. Объясните назначение класса SqlException. Чем данный класс отключается от класса-родителя Exception?**

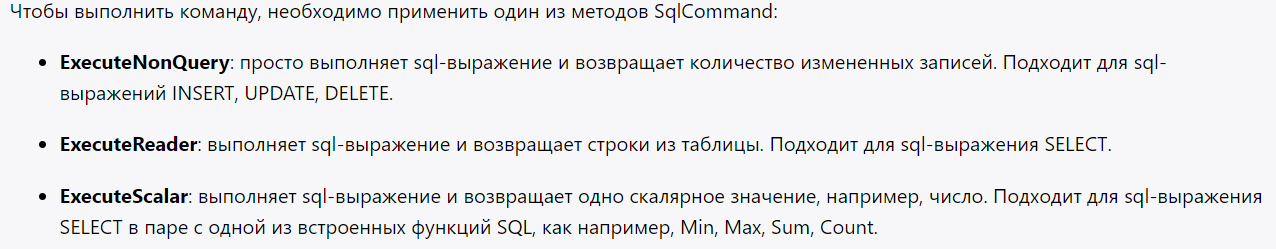
Класс предоставляет информацию о сгенерированных ошибках, их кодах и связанных сообщениях.



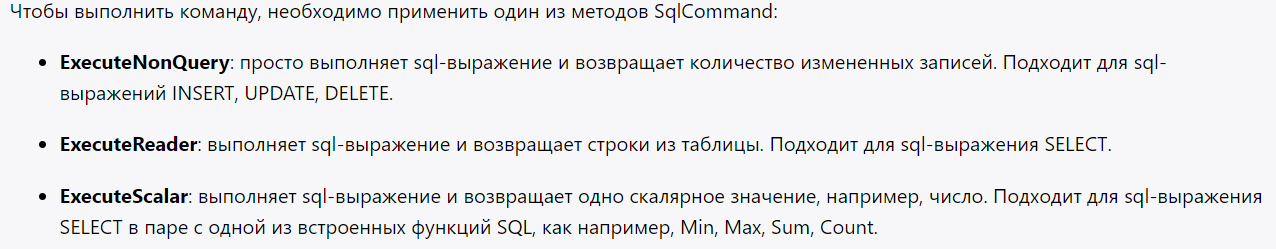
1. **Объясните назначение класса SqlCommand.**

После установки подключения мы можем выполнить к базе данных какие-либо команды, например, добавить в базу данных объект, удалить, изменить его или просто извлечь. Команды представлены объектом интерфейса System.Data.IDbCommand. Провайдер для MS SQL предоставляет его реализацию в виде класса **SqlCommand**. Этот класс инкапсулирует sql-выражение, которое должно быть выполнено.





1. **В чем разница между методами ExecuteNonQuery, ExecuteReader и ExecuteScalar?**



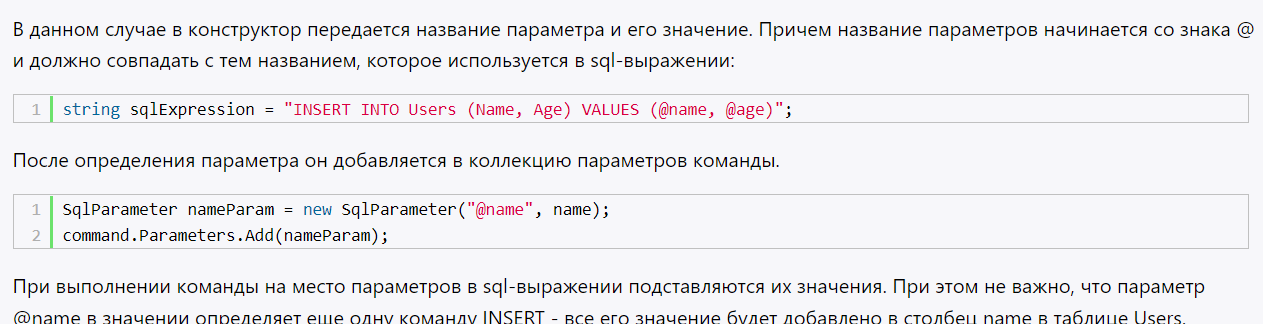
1. **Объясните назначение класса SqlDataReader.**

Предоставляет способ чтения потока строк последовательного доступа из базы данных SQL Server. Этот класс не наследуется.

Основная функция **класса** **SqlDataReader** заключается в чтении потока строк из базы данных SQL Server только прямым способом. Объект **класса** создается путем вызова метода ExecuteReader() **класса** SqlCommand. Объект SqlConnection, связанный с объектом **SqlDataReader**, не может выполнять никаких других операций до тех пор, пока он не будет использован объектом **SqlDataReader**.

1. **Объясните назначение класса SqlParameter.**

Представляет параметр для объекта [SqlCommand](https://docs.microsoft.com/ru-ru/dotnet/api/system.data.sqlclient.sqlcommand?view=netframework-4.8) и (необязательно) его сопоставление со столбцами [DataSet](https://docs.microsoft.com/ru-ru/dotnet/api/system.data.dataset?view=netframework-4.8). Этот класс не наследуется. Эти параметры используются для выбора данных из источника данных и их вставки в [DataSet](https://docs.microsoft.com/ru-ru/dotnet/api/system.data.dataset?view=netframework-4.8). После определения параметров они добавляются в коллекцию **Parameters** объекта SqlCommand.



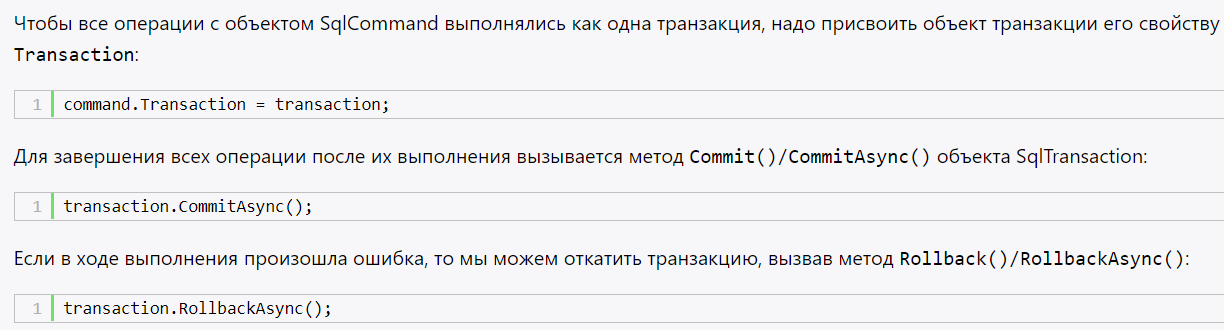
1. **Объясните назначение класса SqlTransaction.**

Предоставляет транзакцию Transact-SQL, которая должна быть выполнена над базой данных SQL Server. Транзакция откатывается при любой ошибке или удаляется без предварительной фиксации. Try/Catch Обработка ошибок используется для обработки ошибок при попытке фиксации или отката транзакции.

Транзакции позволяют выполнять набор операций в виде одного целостного пакета. И если хотя бы одна из операций из этого набора завершится неудачно, то произойдет откат выполнения остальных операций.

Для начала транзакции надо вызвать метод

**BeginTransaction() / BeginTransactionAsync()** объекта SqlConnection:



**11.Объясните назначение классов DataSet, DataTable, DataColumn, DataRow, DataRelation.**

После получения данных из базы данных через SqlDataAdapter в DataSet мы можем локально работать с этими данными вне зависимости от наличия подключения. Объект DataSet содержит таблицы, которые представлены типом **DataTable**. Таблица, в свою очередь, состоит из столбцов и строк. Каждый столбец представляет объект **DataColumn**, а строка - объект **DataRow**. Все данные строки хранятся в свойстве **ItemArray**, который представляет массив объектов - значений отдельных ячеек строки. **DataRelation** предназначен для создания родительских отношений между двумя или более объектами DataTable и DataSet.

**ЛАБА 11**

**1.Что такое ORM?**

**ORM** - (**Object** Relative **Mapping**) - это паттерн проектирования, который позволяет наладить взаимосвязь между классом и таблицей в **Базе** **Данных**. **ORM** системы присутствуют в любом большом фреймворке - Laravel, ASP .Net, **ORM** Django и у многих других.

object-relational mapping - отображения данных на реальные объекты.

Entity Framework представляет ORM-технологию (object-relational mapping - отображения данных на реальные объекты) от компании Microsoft для доступа к данным. Entity Framework Core позволяет абстрагироваться от самой базы данных и ее таблиц и работать с данными как с объектами классом независимо от типа хранилища.

**2. Зачем используются ORM-решения при работе с базами данных? Опишите их достоинства и недостатки.**

**ORM** - (**Object** Relative **Mapping**) - это паттерн проектирования, который позволяет наладить взаимосвязь между классом и таблицей в **Базе** **Данных. ORM** системы присутствуют в любом большом фреймворке - Laravel, ASP .Net, **ORM** Django и у многих других.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Отличительной чертой Entity Framework Core, как технологии ORM, является использование запросов LINQ для выборки данных из БД. С помощью LINQ мы можем создавать различные запросы на выборку объектов, в том числе связанных различными ассоциативными связями.

­­­­­­­­­­­­­­­­­­­­­­­­­­­­­­\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

## **Преимущества:**

* Ускоряет разработку - устраняет необходимость в повторяющемся коде SQL.
* Сокращает время разработки.
* Снижает затраты на разработку.
* Устраняет различия в SQL, зависящие от поставщика - ORM знает, как писать SQL, зависящий от поставщика, поэтому вам не нужно.

## **Недостатки:**

* Снижение производительности разработчиков, пока они учатся программировать с помощью ORM.
* Разработчики теряют понимание того, что на самом деле делает код - разработчик больше контролирует, используя SQL.
* ORM имеет тенденцию быть медленным.
* ORM не может конкурировать с SQL-запросами для сложных запросов.

**3. Что такое Entity Framework?**

Entity Framework представляет ORM-технологию (object-relational mapping - отображения данных на реальные объекты) от компании Microsoft для доступа к данным. Entity Framework Core позволяет абстрагироваться от самой базы данных и ее таблиц и работать с данными как с объектами классом независимо от типа хранилища.

**4. Что такое EDM в Entity Framework и из каких уровней она состоит?**

Entity Data Model состоит из трех уровней: концептуального, уровень хранилища и уровень сопоставления (маппинга).

На концептуальном уровне происходит определение классов сущностей, используемых в приложении.

Уровень хранилища определяет таблицы, столбцы, отношения между таблицами и типы данных, с которыми сопоставляется используемая база данных.

Уровень сопоставления (маппинга) служит посредником между предыдущими двумя, определяя сопоставление между свойствами класса сущности и столбцами таблиц.

Таким образом, мы можем через классы, определенные в приложении, взаимодействовать с таблицами из базы данных.

5. Перечислите способы создания моделей. Опишите их действия.

6. Назовите основные отличия EF от других технологий работы с базами данных.

**7. Для чего и как используется DbContext?**

DbContext — это сочетание шаблонов единиц работы и репозитория. Экземпляр DbContext представляет сеанс с базой данных и может использоваться для запроса и сохранения экземпляров сущностей(Сущность представляет набор данных, ассоциированных с определенным объектом. Поэтому данная технология предполагает работу не с таблицами, а с объектами и их наборами.).

!!!!!! Entity Framework Core не поддерживает несколько параллельных операций, выполняемых в одном экземпляре DbContext. включая параллельное выполнение асинхронных запросов и любое явное использование экземпляра из нескольких потоков одновременно

**8. Как и для чего используются аннотации при настройке конфигураций Code First?**

Entity Framework Code First позволяет использовать собственные классы предметной области для представления модели, которую EF использует для выполнения запросов, отслеживания изменений и обновления функций. Code First использует шаблон программирования, называемый "соглашением по конфигурации". Code First предполагает, что классы следуют соглашениям Entity Framework, и в этом случае автоматически выработает, как выполнять свою работу. Однако если классы не соответствуют этим соглашениям, вы можете добавлять конфигурации в классы для предоставления EF необходимых сведений.

Code First предоставляет два способа добавления этих конфигураций в классы.

**Один** из них использует **простые атрибуты, называемые DataAnnotations**,

а **второй** — **API Fluent Code First**, который позволяет описать конфигурации в коде.

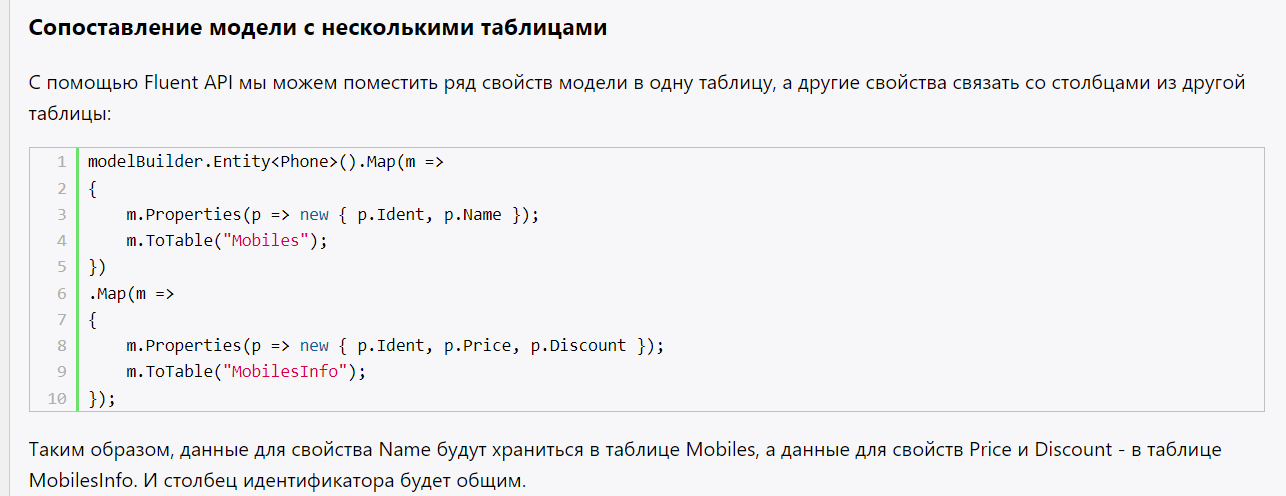
**9. Как и для чего используются Fluent API при настройке конфигураций Code First?**

ли мы используем подход Code First, то классы моделей сопоставляются с таблицами с помощью ряда правил в Entity Framework. Но иногда необходимо изменить и переопределить логику этих правил. Для этого используется Fluent API и аннотации данных.

Fluent API по большому счету представляет набор методов, которые определяются сопоставление между классами и их свойствами и таблицами и их столбцами. Как правило, функционал Fluent API задействуется при переопределении метода **OnModelCreating**.

### *Сопоставление класса с таблицей*

По умолчанию EF сопоставляет модель с одноименной таблицей, но мы можем переопределить это поведение с помощью метода **ToTable()**

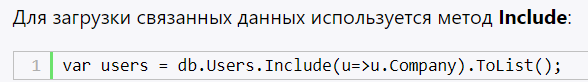


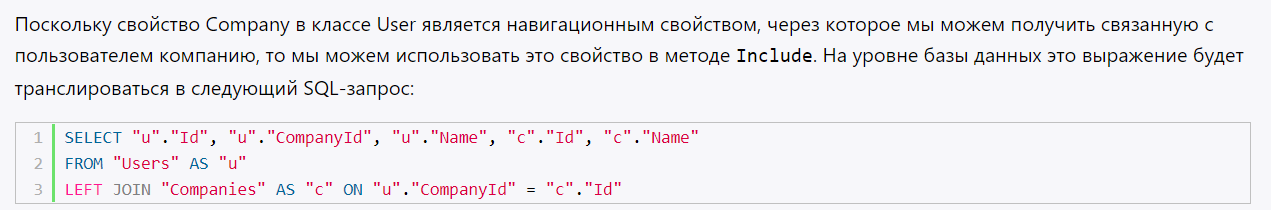
**10.Какие существуют способы загрузки связанных данных?**

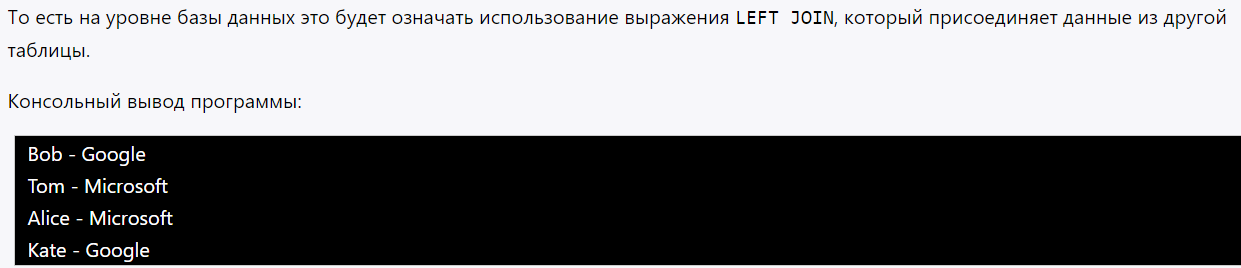
Через навигационные свойства мы можем загружать связанные данные. И здесь у нас три стратегии загрузки:

* **Eager loading** (прямая загрузка)
* **Explicit loading** (явная загрузка)
* **Lazy loading** (отложенная загрузка)

В начале рассмотрим, что предствляет собой **eager loading** или прямая загрузка. Она позволяет загружать связанные данные с помощью метода **Include()**, в который передается навигационное свойство.





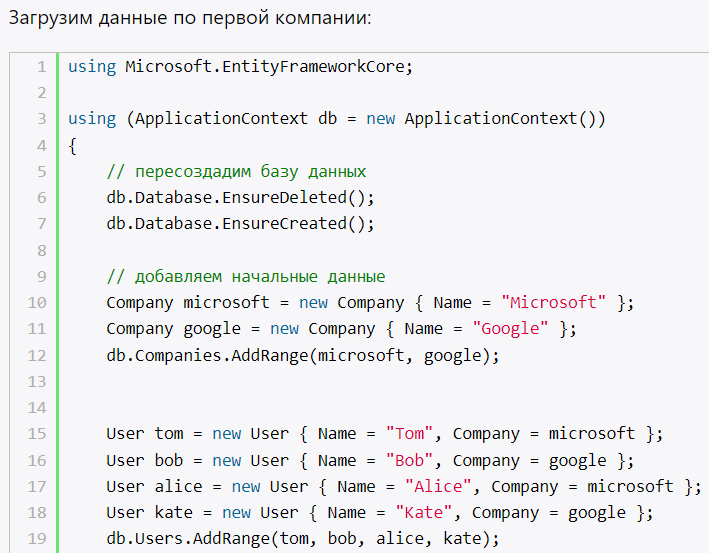


\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Стратегия **Explicit loading** предполагает явную загрузку данных с помощью метода **Load()**.

Важно, что здесь подгружаются только те данные, которые непосредственно связаны с запросом. Если нам надо загрузить в контекст вообще все объекты (например из таблицы Users), то можно было бы использовать следующее выражение db.Users.Load()





Выражение db.Users.Where(p=>p.CompanyId==company.Id).Load() загружает пользователей в контекст. Подвыражение Where(p=>p.CompanyId==company.Id) означает, что загружаются только те пользователи, у которых свойство CompanyId соответствует свойству Id ранее полученной компании. После этого нам не надо подгружать связанные данные, так как они уже есть в контексте.

#### Eager Loading

Суть Eager Loading заключается в том, чтобы использовать для подгрузки связанных по внешнему ключу данных метод **Include**.

#### Explicit Loading

Явная загрузка предусмативает применение метода **Load()** для загрузки данных в контекст.

Чтобы подгрузить данные, здесь идет обращение к методу db.Entry(), в который передается нужный объект. Для подгрузки связанного объекта, который не представляет коллекцию, используется метод **Reference()**. В этот метод переается навигационное свойство, по которому надо подгрузить данные.

Если связанные объект представляет коллекцию, то применяется метод Collection(), в который также передается навигационное свойство в виде строки.

#### Lazy Loading

Еще один способ представляет так называемая "ленивая загрузка" или **lazy loading**. При таком способе подгрузки при первом обращении к объекту, если связанные данные не нужны, то они не подгружаются. Однако при первом же обращении к навигационному свойству эти данные автоматически подгружаются из бд.

При использовании ленивой загрузки надо иметь в виду некоторые моменты при объявлении классов. Так, классы, использующие ленивую загрузку должны быть публичными, а их свойства должны иметь модификаторы **public** и **virtual**.

**11.В чем разница между концепциями Code First и Database First?**

Code-First: нет бд, сначала кодом пишем бд, все свойства для бд(таблицы) и по коду создается бд и потом само приложение (МОДЕЛЬКИ)

Database-First: создаем бд, потом только разрабатываем приложение

12.Зачем EF использует миграции? Как создать новую миграцию?

Миграции позволяют вносить изменения в базу данных при изменениях моделей и контекста данных.

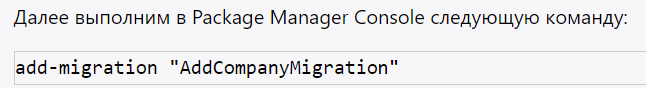
мы можем применить миграции.

Для этого в Visual Studio перейдем к окну **Package Manager Console**, которое можно найти внизу VS. Если такого окна нет, то его можно открыть, перейдя к меню **View**->**Other Window**->**Package Manager Console**

Для добавления функционала миграций введем в это окно следующую команду:



После ввода команды нажмем на Enter. И в результате выполнения данной команды в проект будет добавлена папка *Migrations*, в которой будут два файла: *Configration.cs* (содержит базовую конфигурацию миграций) и файл начальной миграции, название которого может отличаться. Файл начальной миграции устанавливает, как база данных определяется на данный момент.





В миграции определяются два метода: Up() и Down(). В методе Up с помощью вызова метода AddColumn добавляется новый столбец Company в уже имеющуюся таблицу dbo.Phones. Метод Down удаляет столбец на случай, если они существуют. Фактически эти методы равнозначны выражению ALTER в языке SQL, которое меняет структуру базы данных и ее таблиц.

И в завершении чтобы выполнить миграцию, применим этот класс, набрав в той же консоли команду:

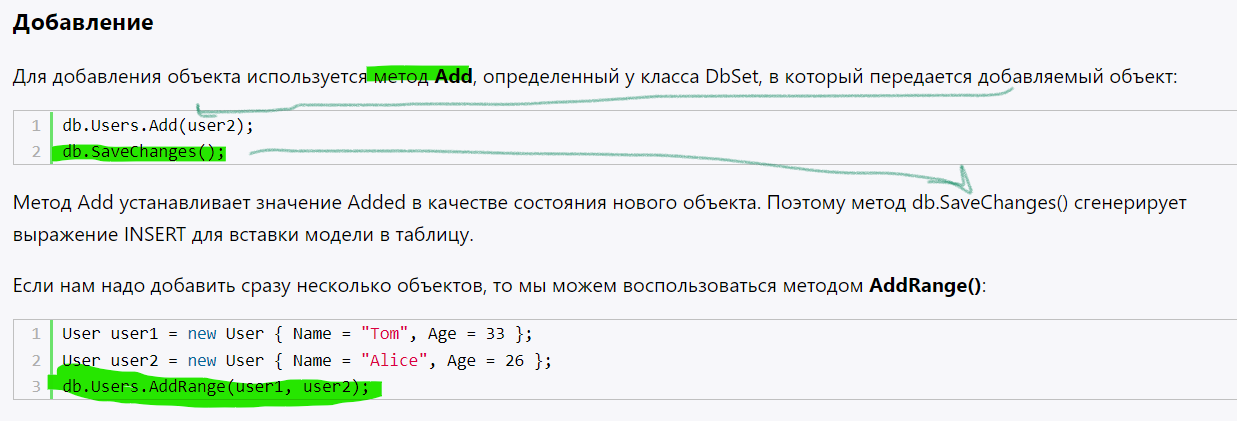


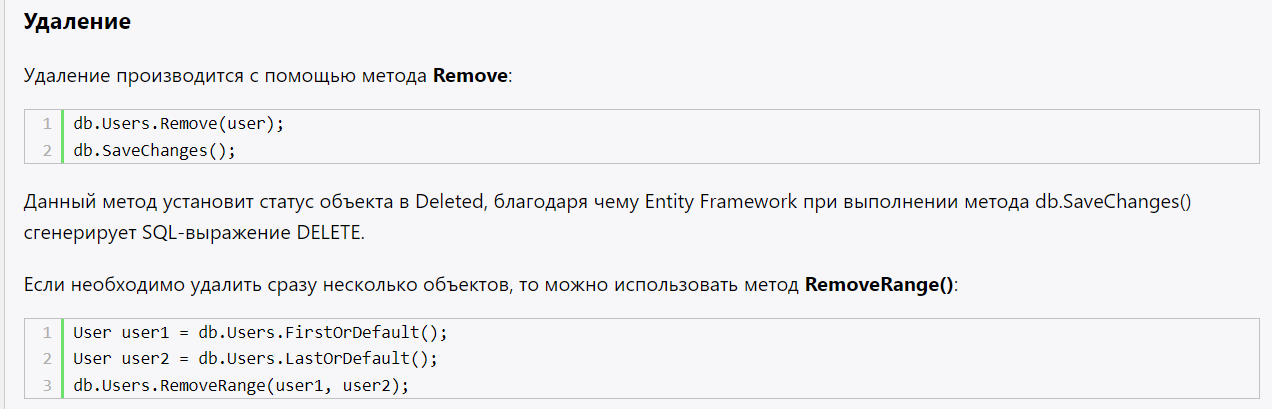
та команда обновит базу данных, добавив в нее новый столбец. Причем данные, которые уже были в таблицы, сохранятся.

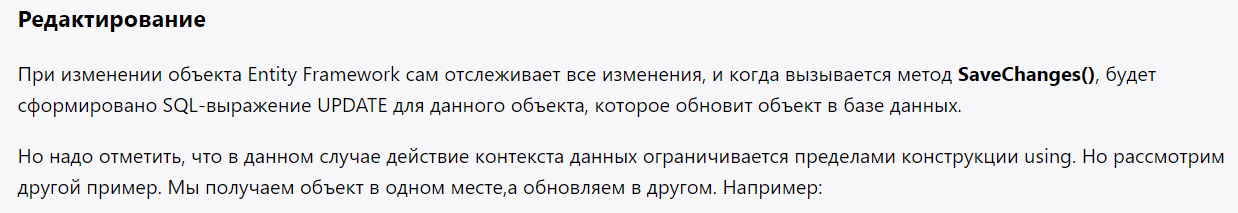
**13.Расскажите алгоритм выполнения CRUD-запросов при работе с БД через библиотеку Entity Framework**

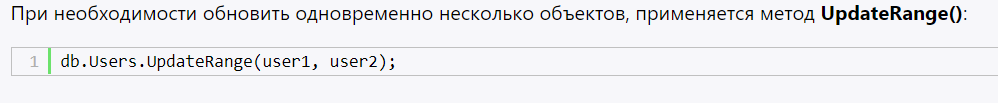
Большинство операций с данными так или иначе представляют собой CRUD операции (Create, Read, Update, Delete), то есть создание, получение, обновление и удаление. Entity Framework Core позволяет легко выполнять все эти действия.

Для примера создадим проект по типу **Console App (.NET Core)**. И после создания проекта сразу добавим в него функциональность EF Core. Для этого в проект через NuGet пакет **Microsoft.EntityFrameworkCore.SqlServer**.









\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

### **Способы взаимодействия с БД**

Entity Framework предполагает три возможных способа взаимодействия с базой данных:

* **Database first**: Entity Framework создает набор классов, которые отражают модель конкретной базы данных
* **Model first**: сначала разработчик создает модель базы данных, по которой затем Entity Framework создает реальную базу данных на сервере.
* **Code first**: разработчик создает класс модели данных, которые будут храниться в бд, а затем Entity Framework по этой модели генерирует базу данных и ее таблицы

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**ЛАБА 12**

1. **Для чего используют многоуровневые архитектуры?**

Этот архитектурный подход разделяет комплекс ПО на уровни по принципу взаимодействия “клиент-сервер”. Архитектура может иметь один, два и больше уровней, разделяющих ответственности между поставщиком данных и потребителем.

Если мы говорим о трехуровневой архитектуре, то n-tier приложение могло быть разделено на такие уровни:

- сервер базы данных,

-веб-приложение на веб-сервере

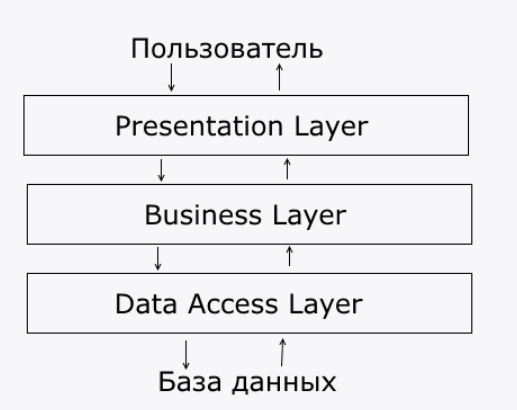
-браузер пользователя.

То есть каждый уровень представлял бы особый отдельный физический процесс, даже если бы и сервер баз данных, и веб-сервер, и браузер пользователя находились бы на одном компьютере. Если бы в качестве клиента альтернативно использовалось мобильное приложение, то это был бы еще один физический уровень.

Подходит для создания новых приложений, которые нужно развернуть по-быстрому. Это своеобразный «шаблон общего назначения».

*Плюсы:*  
-Каждый уровень этой архитектуры выполняет строго ограниченный набор функций (которые не повторяются от слоя к слою) и не знает о том, как устроены остальные уровни. Поэтому «содержимое» уровней можно изменять без риска глобальных конфликтов между слоями.  
*Недостатки:*  
-низкая скорость работы. Очень много информации начинает бесполезно проходить от слоя к слою, когда количество бесполезных операций начинает преобладать над полезными.  
-Поиск багов в многоуровневых системах также [может быть затруднен](https://medium.com/code-smells/layered-architecture-f11bc04c5d6c). Прежде чем попасть в базу данных, информация проходит через все уровни (так как БД является конечным компонентом). Если по какой-то причине эта информация повреждается (или теряется по пути), то для поиска ошибки приходится анализировать каждый уровень по отдельности.

1. **Опишите назначение слоев многоуровневой архитектуре. Business layer (уровень бизнес-логики), Data Access layer (уровень доступа к данным).**



**Business layer** (уровень бизнес-логики): содержит набор компонентов, которые отвечают за обработку полученных от уровня представлений данных, реализует всю необходимую логику приложения, все вычисления, взаимодействует с базой данных и передает уровню представления результат обработки.

**Data Access layer** (уровень доступа к данным): хранит модели, описывающие используемые сущности, также здесь размещаются специфичные классы для работы с разными технологиями доступа к данным, например, класс контекста данных Entity Framework. Здесь также хранятся репозитории, через которые уровень бизнес-логики взаимодействует с базой данных.

**Presentation layer** (уровень представления): это тот уровень, с которым непосредственно взаимодействует пользователь. Этот уровень включает компоненты пользовательского интерфейса, механизм получения ввода от пользователя. Применительно к asp.net mvc на данном уровне расположены представления и все те компоненты, который составляют пользовательский интерфейс (стили, статичные страницы html, javascript), а также модели представлений, контроллеры, объекты контекста запроса.

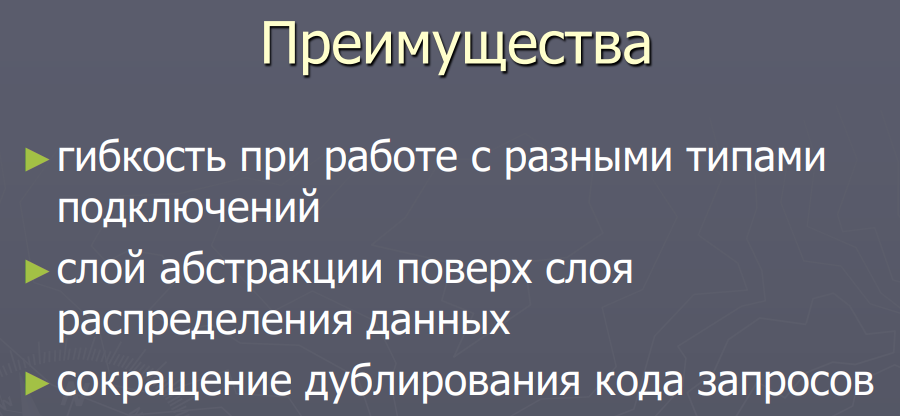
При этом надо отметить, что крайние уровни не могут взаимодействовать между собой, то есть уровень представления не могут напрямую обращаться к базе данных и даже к уровню доступа к данным, а только через уровень бизнес-логики.

Уровень доступа к данным не зависит от других уровней, уровень бизнес-логики зависит от уровня доступа к данным, а уровень представления - от уровня бизнес-логики.

1. **Поясните назначение и принцип использования паттерна Repository?**

<https://metanit.com/sharp/articles/mvc/11.php>





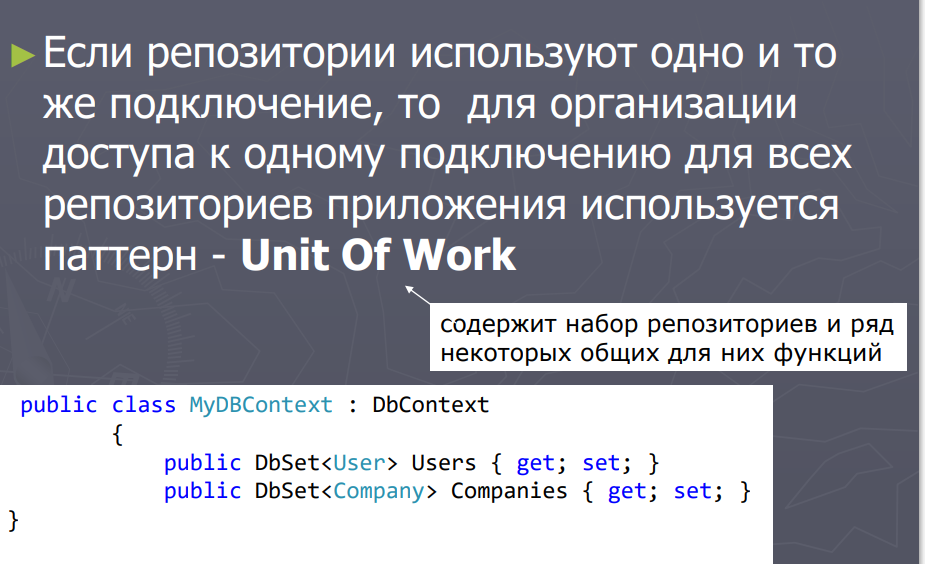
Репозитории представляют коллекции. Коллекция, которая содержит сущности и может фильтровать и возвращать результат обратно в зависимости от требований вашего приложения.

Репозиторий позволяет абстрагироваться от конкретных подключений к источникам данных, с которыми работает программа, и является промежуточным звеном между классами, непосредственно взаимодействующими с данными, и остальной программой.

Допустим, у нас есть одно подключение к базе данных MS SQL Server. Однако, если мы захотим сменить подключение с MS SQL на другое .При стандартном подходе даже в небольшом приложении, осуществляющем выборку, добавление, изменение и удаление данных, нам бы пришлось сделать большое количество изменений. Либо в процессе работы программы в зависимости от разных условий мы хотим использовать два разных подключения. **Таким образом, репозиторий добавляет программе гибкость при работе с разными типами подключений.**

1. **Поясните назначение и принцип использования паттерна Unit of Work?**

https://metanit.com/sharp/mvc5/23.3.php



Паттерн Unit of Work позволяет упростить работу с различными репозиториями и дает уверенность, что все репозитории будут использовать один и тот же контекст данных.

Класс UnitOfWork предоставляет доступ к репозиториям через отдельные свойства и определяет общий контекст для обоих репозиториев.

Кроме того, данный класс содержит дополнительные методы Save() и Dispose(), которые в иной ситуации мы могли б определить в репозиториях. Но так как этот функционал будет общим для обоих репозиториев, то его лучше вынести в класс UnitOfWork.

1. **Опишите основные особенности библиотеки Entity Framework?**

<https://metanit.com/sharp/entityframework/1.1.php>

**Entity Framework** представляет специальную объектно-ориентированную технологию на базе фреймворка .NET для работы с данными.

Entity Framework представляет собой более высокий уровень абстракции, который позволяет абстрагироваться от самой базы данных и работать с данными независимо от типа хранилища. Если на физическом уровне мы оперируем таблицами, индексами, первичными и внешними ключами, но на концептуальном уровне, который нам предлагает Entity Framework, мы уже работает с объектами.

**Способы взаимодействия с БД**

Entity Framework предполагает три возможных способа взаимодействия с базой данных:

* **Database first**: Entity Framework создает набор классов, которые отражают модель конкретной базы данных
* **Model first**: сначала разработчик создает модель базы данных, по которой затем Entity Framework создает реальную базу данных на сервере.
* **Code first**: разработчик создает класс модели данных, которые будут храниться в бд, а затем Entity Framework по этой модели генерирует базу данных и ее таблицы

**6. Какие преимущества обеспечивает использование слабосвязанного кода?**

https://metanit.com/sharp/mvc5/18.4.php

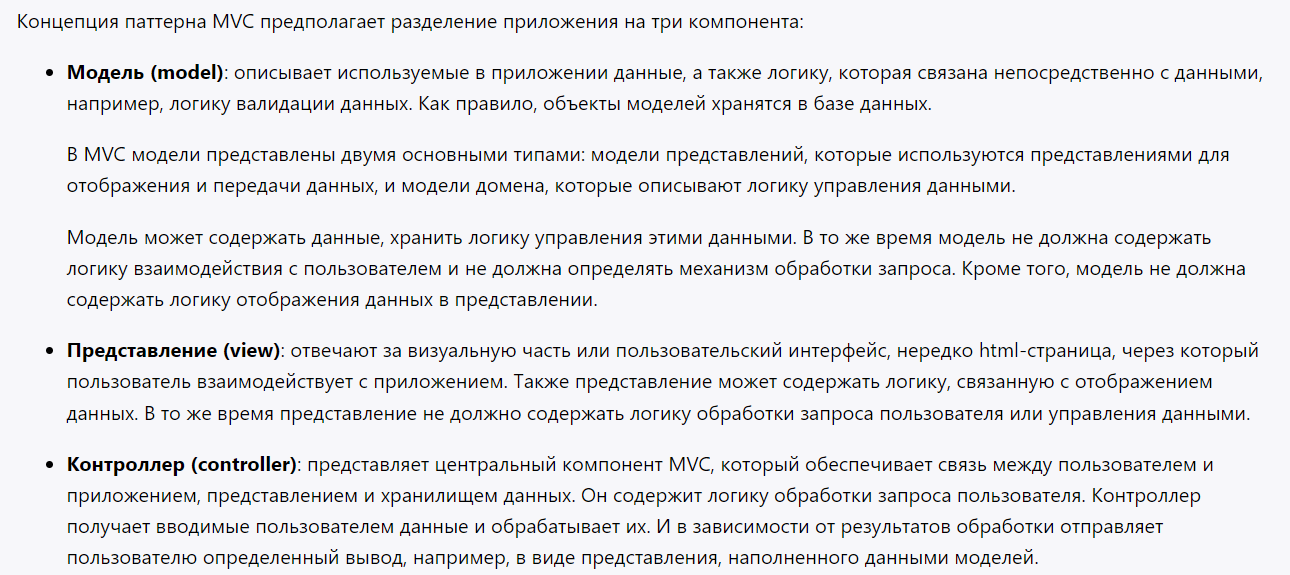
 вместо жесткой связи объектов используют **слабосвязанные объекты** (loosely coupled objects).

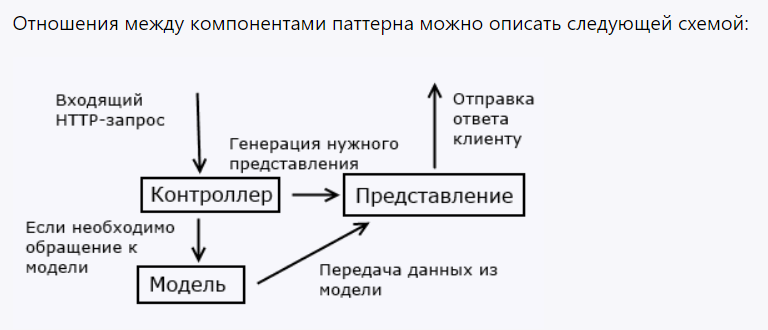
Слабая связь позволяет вносить изменения в одной области приложения, не затрагивая другие. Теоретически он позволяет вам делать такие вещи, как изменение уровня доступа к данным, не перестраивая уровень бизнеса или пользовательского интерфейса.

Это определенно делает ваши приложения более гибкими, более приспособленными к изменениям и более простыми в обслуживании (поскольку вам не нужно беспокоиться о том, что изменение в одной области приложения нарушит работу другой).

**ЛАБА 13**

1. **Что такое MVC, MVP, MVVM?**

****

****

**MVP** — это паттерн программирования графических интерфейсов. В нём приложение делится на три компонента:

Model (Модель) работает с данными, проводит вычисления и руководит всеми бизнес-процессами

View (Вид или представление) показывает пользователю интерфейс и данные из модели

Presenter (Представитель) служит прослойкой между моделью и видом

**MVVM** — это паттерн разработки, позволяющий разделить приложение на три функциональные части:

Model — основная логика программы (работа с данными, вычисления, запросы и так далее)

View — вид или представление (пользовательский интерфейс)

ViewModel — модель представления, которая служит прослойкой между View и Model

**Mvc** - взаимодействие между моделью и вьюшкой происходит посредством контроллеров, которые определяют работу приложения, работающие с данными и формируют нужную вьюшку. В MVP получается третий компонент – Presenter

**Mvp** - преимущественно для пользовательского интерфейса, например какие-то манипуляции с мышкой и так же нужнн иногда для представления(вью)

получается, что MVC там контроллеры определяют работу приложения, Mvp это манипуляции "для пользователя", а Mvvm это разделение функционала приложения

**Но!!!!!!!!!!!**

**Mvc** тоже разделяет функционал на model view controller. Просто тут контроллеры, для каждой вьюшки определен контролер свой. В мввм там просто логика прописана в вм

1. **Объясните из каких компонентов состоит паттерн MVVM?**

Паттерн **MVVM (Model-View-ViewModel)** позволяет отделить логику приложения от визуальной части (представления). Данный паттерн является архитектурным, то есть он задает общую архитектуру приложения.

****

### **Model**

Модель описывает используемые в приложении данные. Модели могут содержать логику, непосредственно связанную этими данными, например, логику валидации свойств модели. В то же время модель не должна содержать никакой логики, связанной с отображением данных и взаимодействием с визуальными элементами управления.

### **View**

View или представление определяет визуальный интерфейс, через который пользователь взаимодействует с приложением. Применительно к WPF представление - это код в xaml, который определяет интерфейс в виде кнопок, текстовых полей и прочих визуальных элементов.

### **ViewModel**

ViewModel или модель представления связывает модель и представление через механизм привязки данных. Если в модели изменяются значения свойств, при реализации моделью интерфейса INotifyPropertyChanged автоматически идет изменение отображаемых данных в представлении, хотя напрямую модель и представление не связаны.

ViewModel также содержит логику по получению данных из модели, которые потом передаются в представление. И также VewModel определяет логику по обновлению данных в модели.

1. **Каким образом паттерн можно реализовать в WPF?**

Для работы с паттерном MVVM создадим новый проект. По умолчанию в проект добавляется стартовое окно MainWindow - это и будет представление.

Также добавим в проект новый класс ApplicationViewModel, который будет представлять модель представления

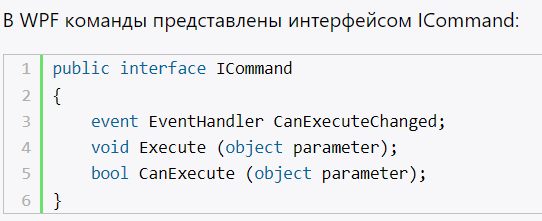
Это класс модели представления, через который будут связаны модель Phone и представление MainWindow.xaml. В этом классе определен список объектов Phone и свойство, которое указывает на выделенный элемент в этом списке.

1. **Объясните назначение интерфейса INotifyPropertyChanged?**

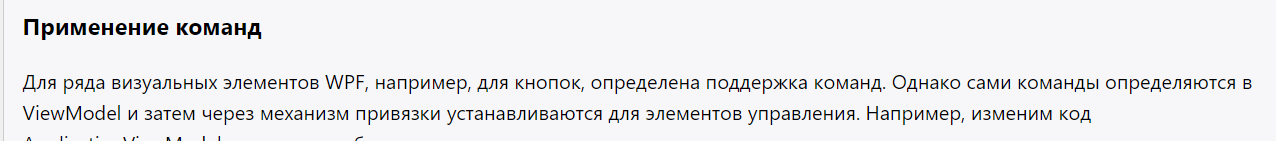
Интерфейс **INotifyPropertyChanged** используется для уведомления клиентов, как правило, привязки клиентов о том, что значение свойства изменилось

1. **Зачем нужен интерфейс ICommand и как паттерн MVVM использует команды в WPF-приложениях.**

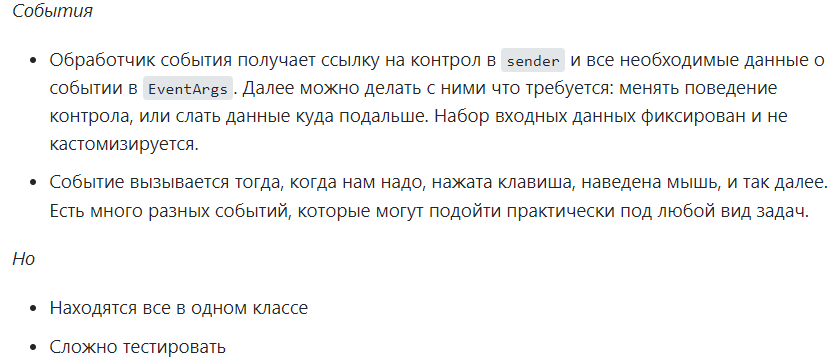
Для взаимодействия пользователя и приложения в MVVM используются команды.

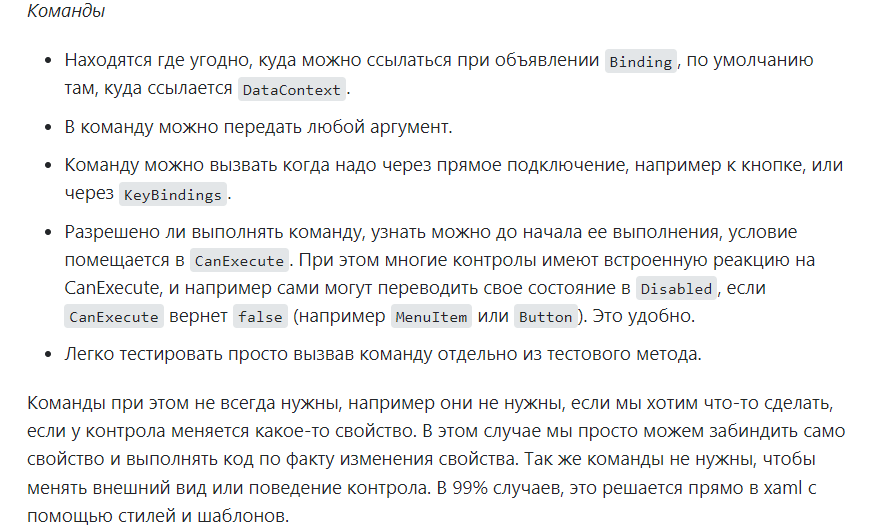
****

Однако WPF имеет в качестве реализации этого интерфейса имеет класс System.Windows.Input.RoutedCommand, который ограничен по функциональности. Поэтому, как правило, придется реализовывать свои собственные команды с помощью реализации ICommand.

****

1. **Чем отличаются команды от событий?**

****

****

1. **Расскажите о назначении и об основных возможностях следующих библиотек и фреймворков: Light Toolkit, Catel и Prism.**

**Light Toolkit –** автоматически создает структуру проекта, и нам не надо это делать

**Catel –** Catel содержит множество функций: ведение журнала, диагностика, отражение, MVVM, пользовательские элементы управления и формы. Все эти функции дополняют другие функции. Самым важным является то, что Catel можно использовать с другими фреймворками

**Prism –** Prism разработано вокруг основных принципов архитектурного дизайна, таких как разделение ответственности и слабая связанность. Это позволяет получить множество выгод, таких как:

* **Повторное использование кода.**
* **Расширяемость**
* **Гибкость.**
* **Командная разработка.**

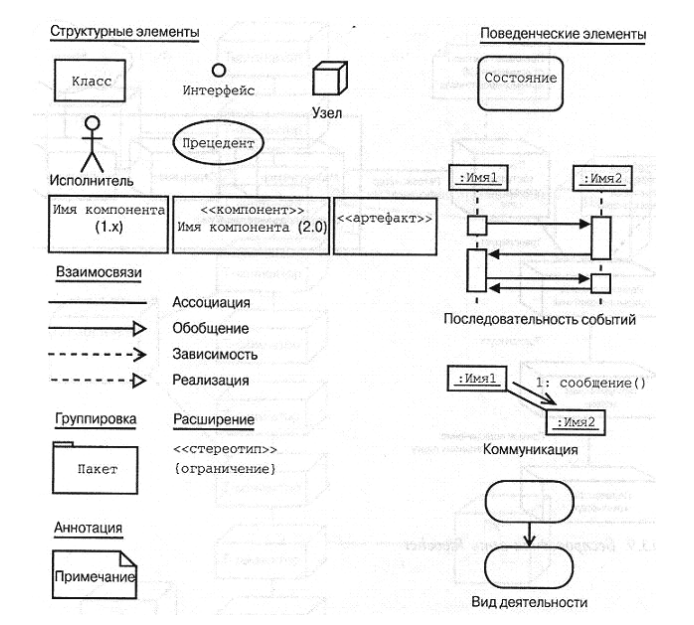
**ЛАБА 14**

**1. Что такое UML?**

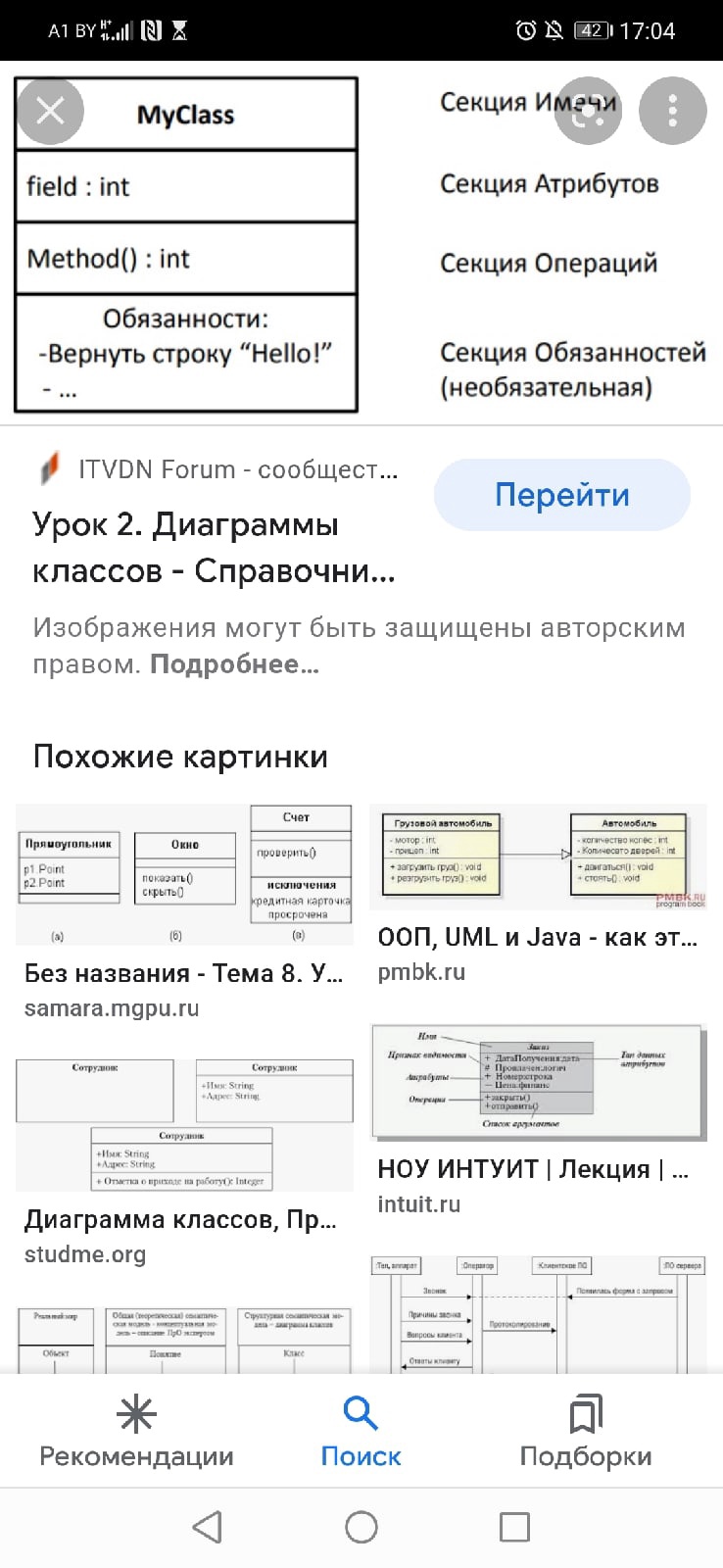
**UML** – унифицированный язык моделирования (Unified Modeling Language) – это система обозначений, которую можно применять для объектно-ориентированного анализа и проектирования. Его можно использовать для визуализации, спецификации, конструирования и документирования программных систем.

Проще говоря, если посмотреть картинки в поисковых системах, то станет понятно, что UML – это что-то про схемы, стрелочки и квадратики.

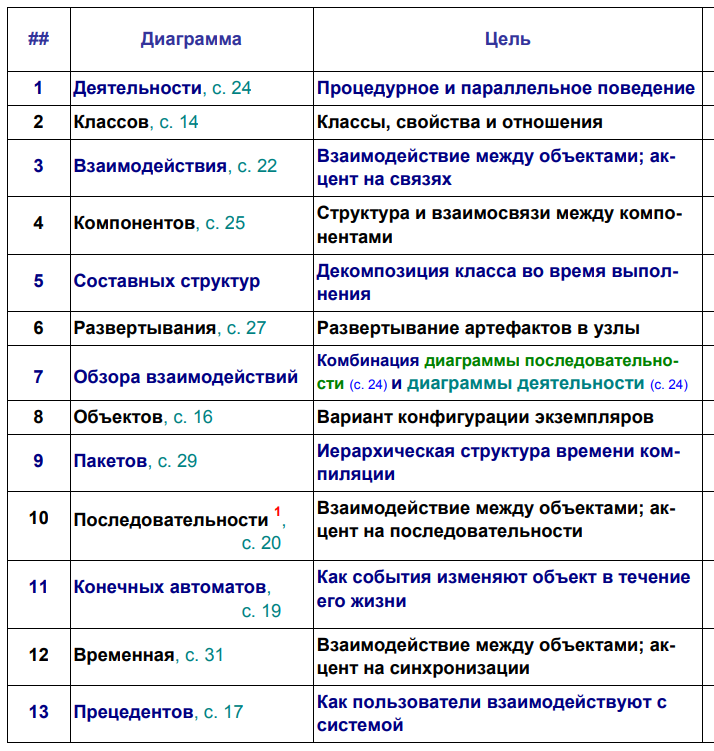
**UML — унифицированный язык моделирования, который позволяет наглядно описать архитектуру, проектирование и реализацию комплексных программных систем.**



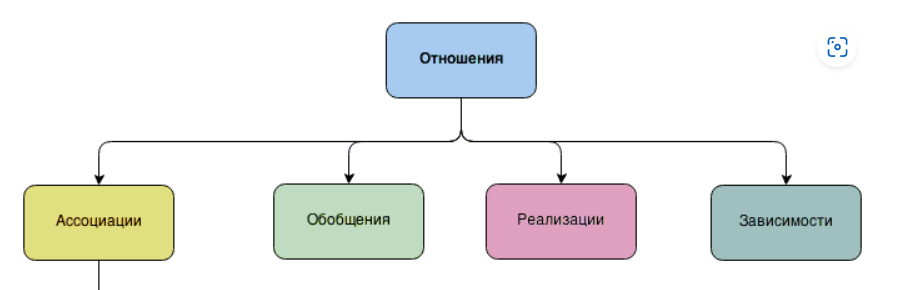




**2. Перечислите типы диаграмм UML.**



**3. Какие отношения между классами могут быть на диаграмме классов?**

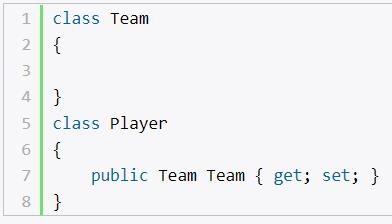


Эти связи представляют собой базовые строительные блоки для описания отношений в UML, используемые для разработки хорошо согласованных моделей.

* **Ассоциация** 

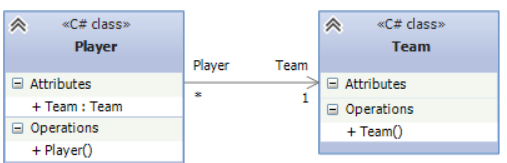
Ассоциация - это отношение, при котором объекты одного типа неким образом связаны с объектами другого типа. Например, объект одного типа содержит или использует объект другого типа. Ассоциация, связывающая два класса, называется бинарной

Например, игрок играет в определенной команде:

****

Класс Player связан отношением ассоциации с класом Team.

*На схемах UML* ассоциация обозначается в виде обычной стрелки:

****

Нередко при отношении ассоциации указывается кратность связей. В данном случае единица у Team и звездочка у Player на диаграмме отражает связь 1 ко многим. То есть одна команда будет соответствовать многим игрокам.

*Агрегация и композиция являются частными случаями ассоциации.*

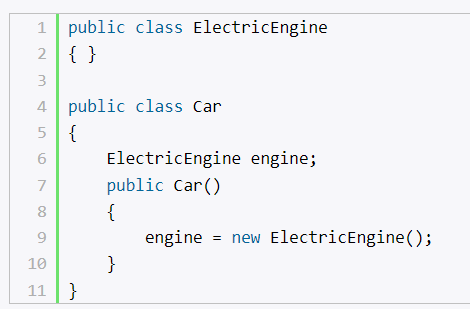
ассоциация между двумя классами отражает структурное отношение между равноправными сущностями, когда оба класса находятся на одном концептуальном уровне и ни один не является более важным, чем другой.

Включает:

**а) Композицию** 

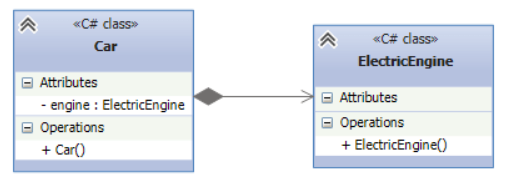
Композиция определяет отношение **HAS A**, то есть отношение "имеет".

Например, в класс автомобиля содержит объект класса электрического двигателя:

****

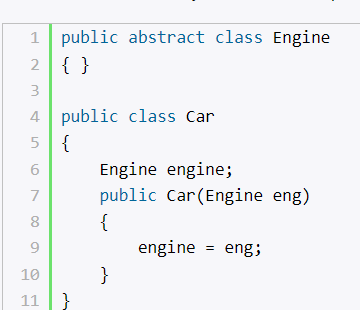
При этом класс автомобиля полностью управляет жизненным циклом объекта двигателя. При уничтожении объекта автомобиля в области памяти вместе с ним будет уничтожен и объект двигателя. И в этом плане объект автомобиля является главным, а объект двигателя - зависимой.

*На диаграммах UML* отношение композиции проявляется в обычной стрелке от главной сущности к зависимой, при этом со стороны главной сущности, которая содержит, объект второй сущности, располагается закрашенный ромбик:



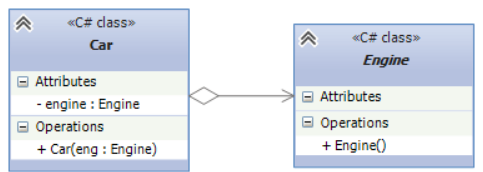
**б) Агрегацию** 

От композиции следует отличать агрегацию. Она также предполагает отношение **HAS A**, но реализуется она иначе:

****

При агрегации реализуется слабая связь, то есть в данном случае объекты Car и Engine будут равноправны. В конструктор Car передается ссылка на уже имеющийся объект Engine. И, как правило, определяется ссылка не на конкретный класс, а на абстрактный класс или интерфейс,  что увеличивает гибкость программы.

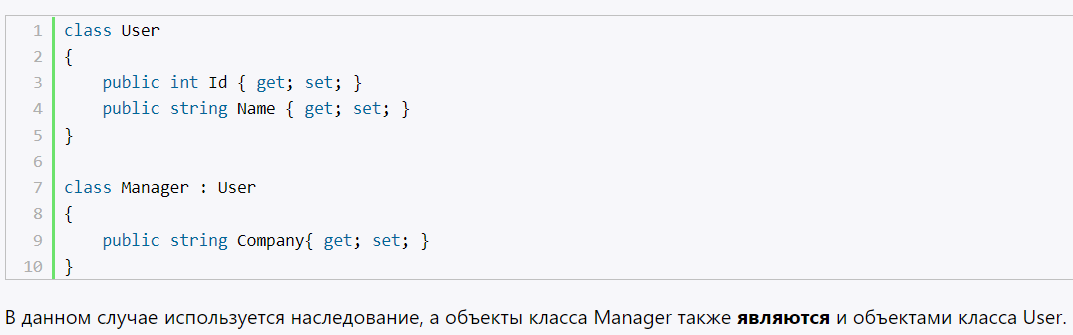
Отношение агрегации на диаграммах UML отображается также, как и отношение композиции, только теперь ромбик будет незакрашенным:

****

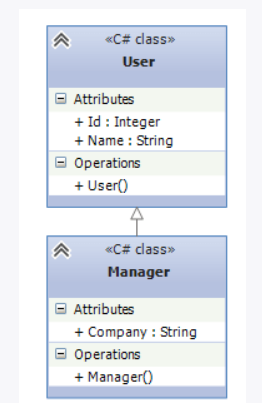
* **Обобщение (Наследование)** 

позволяет одному классу (наследнику) унаследовать функционал другого класса (родительского)

Наследование определяет отношение **IS A**, то есть "является"



С помощью диаграмм UML отношение между классами выражается в незакрашенной стрелочке от класса-наследника к классу-родителю:

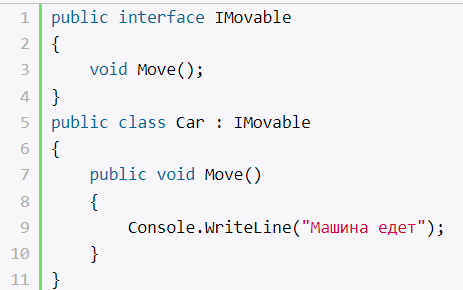


* **Реализация** 

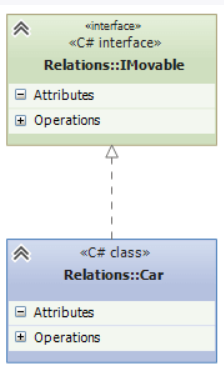
Реализация предполагает определение интерфейса и его реализация в классах.

Реализация — отношение между двумя элементами модели, в котором один элемент (клиент) реализует поведение, заданное другим (поставщиком). Реализация — отношение целое-часть. Явное выражение реализации в ABAP это реализация классом заданного интерфейса.

Например, имеется интерфейс IMovable с методом Move, который реализуется в классе Car:



С помощью диаграмм UML отношение реализации также выражается в незакрашенной стрелочке от класса к интерфейсу, только линия теперь пунктирная:



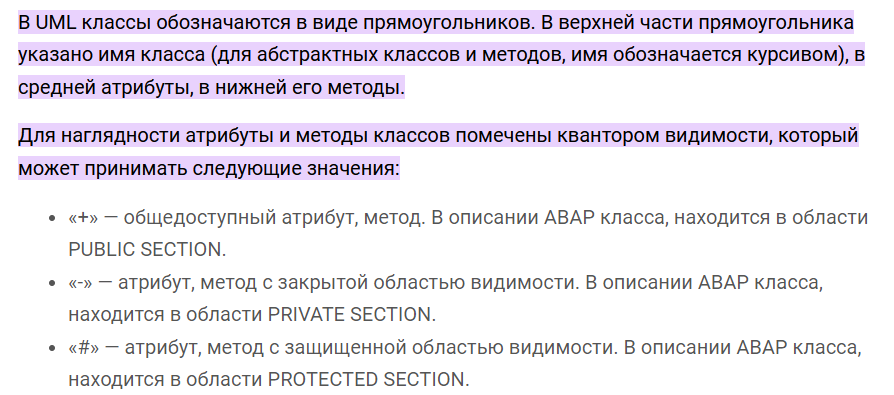
* **Зависимость** 

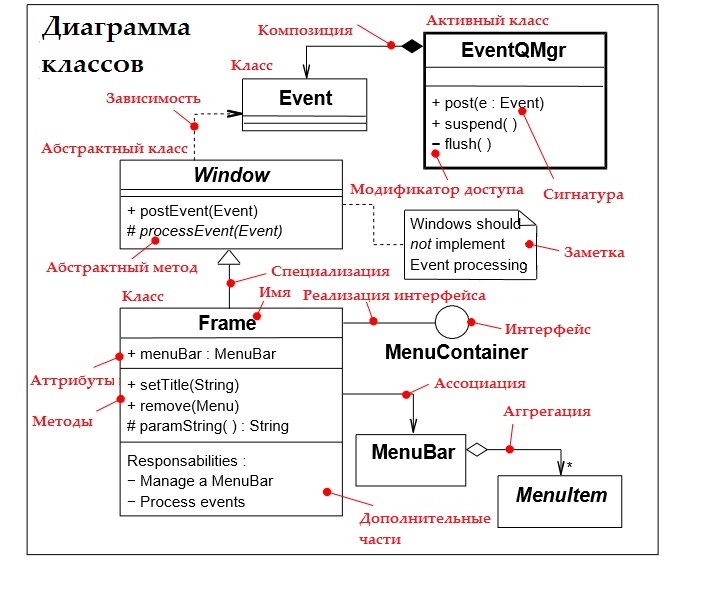
связь между двумя элементами модели, в которой изменение одного элемента (независимого) может привести к изменению семантики другого элемента (зависимого). Графически представлена пунктирной линией, иногда со стрелкой, направленной к той сущности, от которой зависит еще одна; может быть снабжена меткой.

Зависимость – это связь использования, указывающая, что **изменение** спецификаций **одной сущности может повлиять на другие сущности, которые используют ее**.

**4. Как обозначаются абстрактные классы на диаграмме классов?**

Имя абстрактного класса или абстрактной операции пишется **курсивом**. Кроме того, вы можете обозначить абстрактность элемента, поместив ключевое слово abstract в список [свойств](https://openu.ru/Books/UML/Property.asp)

****

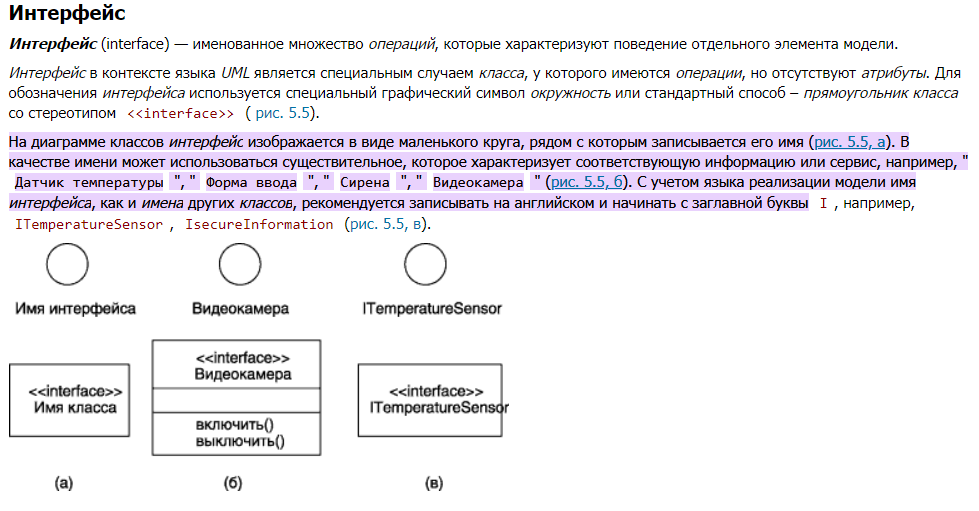


**5. Как обозначаются интерфейсы на диаграмме классов?**

***Интерфейс*** (interface) — именованное множество *операций*, которые характеризуют поведение отдельного элемента модели.

*Интерфейс* в контексте языка *UML* является специальным случаем *класса*, у которого **имеются*****операции*, но отсутствуют*****атрибуты*.** Для обозначения *интерфейса* используется специальный графический символ *окружность* или стандартный способ – *прямоугольник* *класса* со стереотипом <<interface>>

На диаграмме классов *интерфейс* изображается в виде маленького круга, рядом с которым записывается его имя, рекомендуется записывать на английском и начинать с заглавной буквы I, например, ITemperatureSensor, IsecureInformation

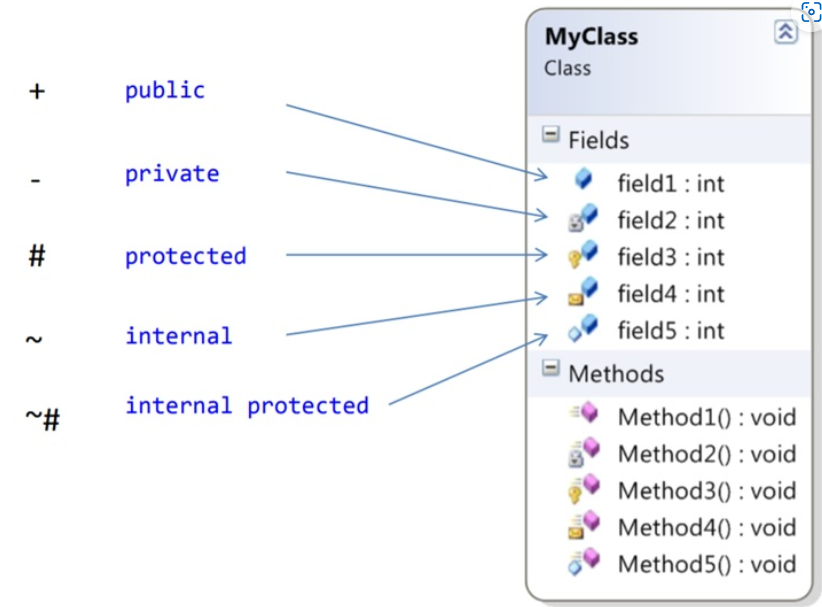


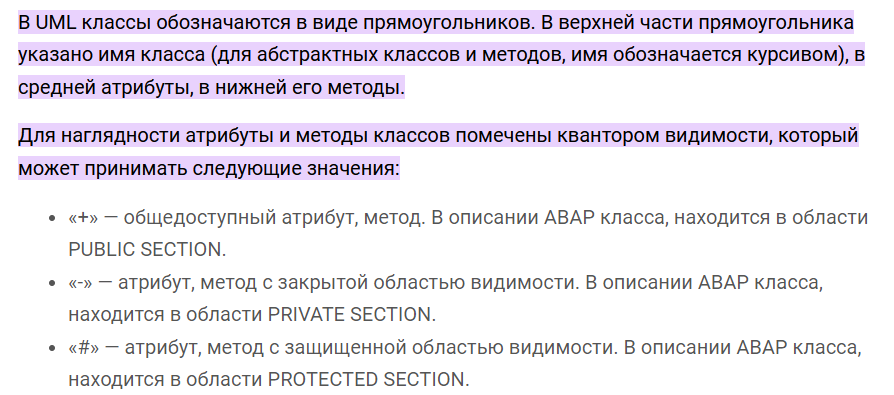
*Интерфейсы* на диаграмме служат для спецификации таких элементов модели, которые видимы извне, но их внутренняя структура остается скрытой от клиентов. Интерфейсы не могут содержать ни *атрибутов*, ни состояний, ни направленных ассоциаций. Они содержат только *операции* без указания особенностей их реализации.

**6. Как отображается доступность членов класса на диаграмме классов?**

Модификаторы доступа – это ключевые слова, задающие объявленную доступность члена или типа. При помощи модификаторов доступа можно задать следующие пять уровней доступности:

1. **public** – доступ к типу или члену возможен из любого другого кода в той же сборке или другой сборке, ссылающейся на него.
2. **protected** – доступ к типу или элементу можно получить только из кода в том же классе, либо в производном классе.
3. **internal** – доступ к типу или члену возможен из любого кода в той же сборке, но не из другой сборки.
4. **protected internal** – доступ ограничен текущей сборкой или типами, которые являются производными от содержащего класса.
5. **private** – доступ к типу или члену можно получить только из кода в том же классе или структуре.



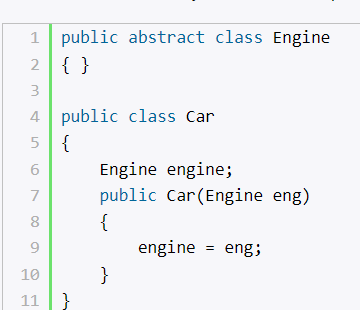
****

**7. Что такое агрегация? Как обозначается?**

**Агрегация** 

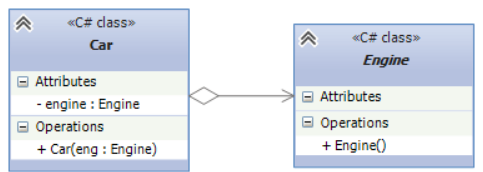
Агрегация (агрегирование по ссылке) **—** **отношение** **«часть-целое»** **между** **двумя** **равноправными** **объектами, когда** **один** **объект** **(контейнер)** **имеет** **ссылку** **на** **другой** **объект**. Оба объекта могут существовать независимо: если контейнер будет уничтожен, то его содержимое — нет

От композиции следует отличать агрегацию. Она также предполагает отношение **HAS A**, но реализуется она иначе:

****

При агрегации реализуется слабая связь, то есть в данном случае объекты Car и Engine будут равноправны. В конструктор Car передается ссылка на уже имеющийся объект Engine. И, как правило, определяется ссылка не на конкретный класс, а на абстрактный класс или интерфейс,  что увеличивает гибкость программы.

Отношение агрегации на диаграммах UML отображается также, как и отношение композиции, только теперь ромбик будет незакрашенным:

****

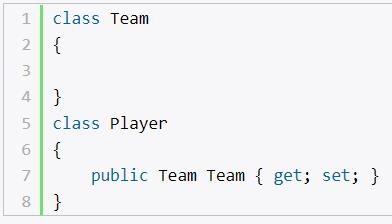
**8. Что такое ассоциация?**

* **Ассоциация** 

**Ассоциация** - это отношение, при котором объекты одного типа неким образом связаны с объектами другого типа. Например, объект одного типа содержит или использует объект другого типа.

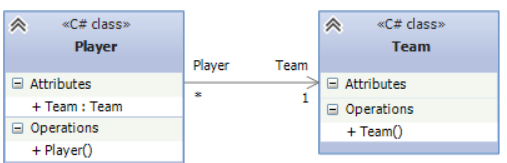
Ассоциация - это отношение, при котором объекты одного типа неким образом связаны с объектами другого типа. Например, объект одного типа содержит или использует объект другого типа. Ассоциация, связывающая два класса, называется бинарной

Например, игрок играет в определенной команде:

****

Класс Player связан отношением ассоциации с класом Team.

*На схемах UML* ассоциация обозначается в виде обычной стрелки:

****

Нередко при отношении ассоциации указывается кратность связей. В данном случае единица у Team и звездочка у Player на диаграмме отражает связь 1 ко многим. То есть одна команда будет соответствовать многим игрокам.

*Агрегация и композиция являются частными случаями ассоциации.*

ассоциация между двумя классами отражает структурное отношение между равноправными сущностями, когда оба класса находятся на одном концептуальном уровне и ни один не является более важным, чем другой.

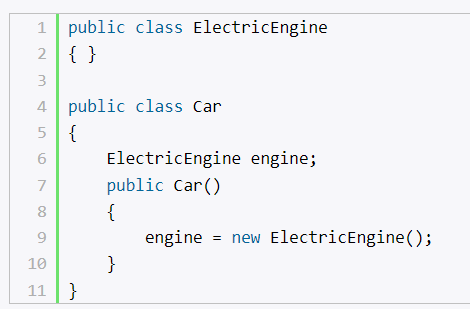
Включает:

**а) Композицию** 

**Композиция** – **это** когда двигатель не существует отдельно от автомобиля. Он создается при создании автомобиля и полностью управляется автомобилем. В типичном примере, экземпляр двигателя будет создаваться в конструкторе автомобиля.

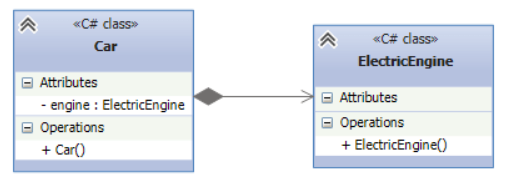
Композиция определяет отношение **HAS A**, то есть отношение "имеет".

Например, в класс автомобиля содержит объект класса электрического двигателя:

****

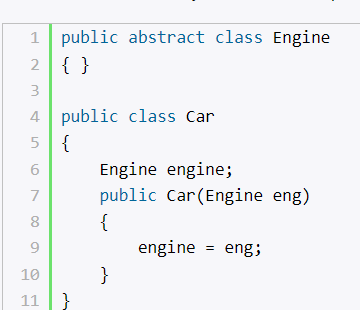
При этом класс автомобиля полностью управляет жизненным циклом объекта двигателя. При уничтожении объекта автомобиля в области памяти вместе с ним будет уничтожен и объект двигателя. И в этом плане объект автомобиля является главным, а объект двигателя - зависимой.

*На диаграммах UML* отношение композиции проявляется в обычной стрелке от главной сущности к зависимой, при этом со стороны главной сущности, которая содержит, объект второй сущности, располагается закрашенный ромбик:



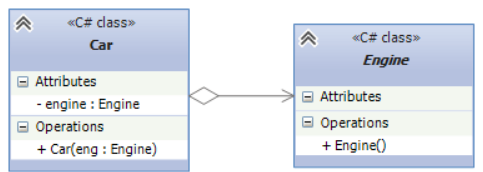
**б) Агрегацию** 

От композиции следует отличать агрегацию. Она также предполагает отношение **HAS A**, но реализуется она иначе:

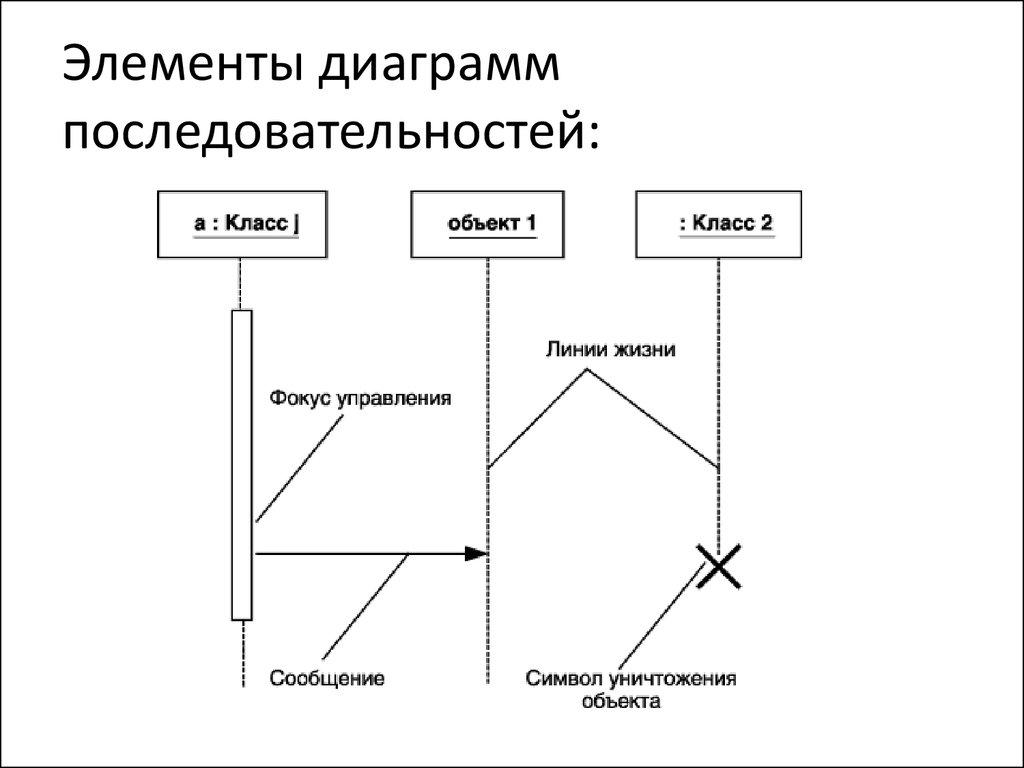
****

При агрегации реализуется слабая связь, то есть в данном случае объекты Car и Engine будут равноправны. В конструктор Car передается ссылка на уже имеющийся объект Engine. И, как правило, определяется ссылка не на конкретный класс, а на абстрактный класс или интерфейс,  что увеличивает гибкость программы.

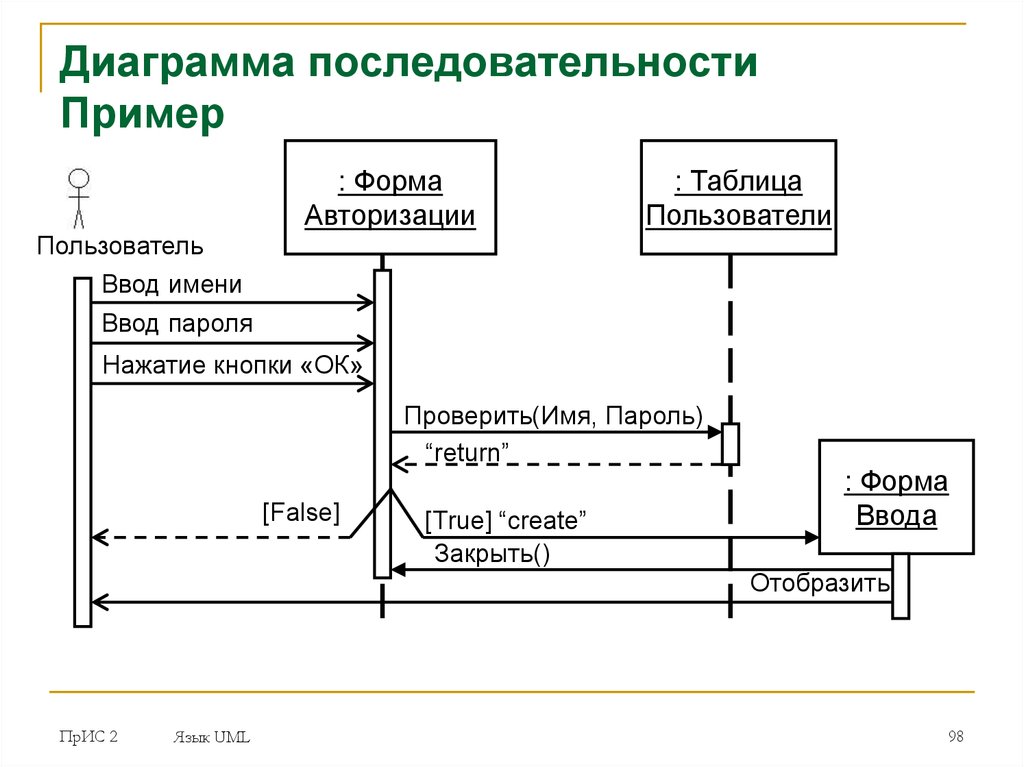
Отношение агрегации на диаграммах UML отображается также, как и отношение композиции, только теперь ромбик будет незакрашенным:

****

**9. Какие обозначения используют на диаграмме последовательности?**



Пример!!!!!!!!!!!!!!!!!!!!



[Диаграммы последовательностей](https://www.lucidchart.com/pages/uml-sequence-diagram), которые иногда также называют диаграммами (или сценариями) событий, показывают, в каком порядке происходит взаимодействие между объектами. С помощью таких диаграмм довольно легко составлять незамысловатые сценарии рабочих циклов.

**10. Для чего нужна диаграмма последовательности?**

[Диаграммы последовательностей](https://www.lucidchart.com/pages/uml-sequence-diagram), которые иногда также называют диаграммами (или сценариями) событий, показывают, **в каком порядке происходит взаимодействие между объектами**. С помощью таких диаграмм довольно легко составлять незамысловатые сценарии рабочих циклов.

**11.Каково назначение диаграммы использования, пакетов и активности**

Диаграмма вариантов использования (англ. use-case diagram) – диаграмма, описывающая, какой функционал разрабатываемой программной системы доступен каждой группе пользователей.

**Пакет** ([англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) *package*) в языке моделирования [UML](https://ru.wikipedia.org/wiki/UML) — основная группирующая сущность, с помощью которой организуются конкретные проектные решения в рамках используемой UML-модели. UML-пакет предназначен для группировки большого количества структурных, поведенческих и других сущностей в единое целое; изображается в виде стилизованной папки с закладкой, которая может иметь своё собственное имя[[1]](https://ru.wikipedia.org/wiki/%D0%9F%D0%B0%D0%BA%D0%B5%D1%82_(UML)#cite_note-_3cfd9cd84fcdcb60-1)[[2]](https://ru.wikipedia.org/wiki/%D0%9F%D0%B0%D0%BA%D0%B5%D1%82_(UML)#cite_note-_a0c0cfb6fa323833-2)

Для обозначения пакета UML используется изображение в виде большого [прямоугольника](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D1%8F%D0%BC%D0%BE%D1%83%D0%B3%D0%BE%D0%BB%D1%8C%D0%BD%D0%B8%D0%BA) с закладкой в левом верхнем углу; внутри прямоугольника можно разместить содержимое данного пакета[[3]](https://ru.wikipedia.org/wiki/%D0%9F%D0%B0%D0%BA%D0%B5%D1%82_(UML)#cite_note-_277621e6d1ec92b3-3). Зависимость между пакетами выражается с помощью [диаграммы пакетов](https://ru.wikipedia.org/wiki/%D0%94%D0%B8%D0%B0%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B0_%D0%BF%D0%B0%D0%BA%D0%B5%D1%82%D0%BE%D0%B2), которая несёт в себе информацию об управлении моделью и фактически является разновидностью [диаграммы классов](https://ru.wikipedia.org/wiki/%D0%94%D0%B8%D0%B0%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B0_%D0%BA%D0%BB%D0%B0%D1%81%D1%81%D0%BE%D0%B2)