

МОДУЛЬ КОМПЕТЕНЦИИ 1: «ПРОЕКТИРОВАНИЕ»	1
Инструментальные средства для анализа и проектирования программных решений.	1
Работа с MS SQL Server.....	72
МОДУЛЬ КОМПЕТЕНЦИИ 2: «РАЗРАБОТКА ОКОННЫХ ПРИЛОЖЕНИЙ»	82
СОЗДАНИЕ ПРОЕКТА	99
СОЗДАНИЕ РЕПОЗИТОРИЯ	100
СОХРАНЕНИЕ ИЗМЕНЕНИЙ В РЕПОЗИТОРИИ	103
СОЗДАНИЕ СЛОВАРЯ СТИЛЕЙ	106
РАЗМЕЩЕНИЕ КОНТЕНТА ПО ЦЕНТРУ ФОРМЫ	108
СОЗДАНИЕ МЕНЮ ПОЛЬЗОВАТЕЛЯ	113
СОЗДАНИЕ ПОДКЛЮЧЕНИЯ К БАЗЕ ДАННЫХ	114
Создание формы авторизации	119
ПЕРЕХОД МЕЖДУ СТРАНИЦАМИ	125
СОЗДАНИЕ ЮНИТ ТЕСТА	140

МОДУЛЬ КОМПЕТЕНЦИИ 1: «ПРОЕКТИРОВАНИЕ»

Инструментальные средства для анализа и проектирования программных решений.

Для того, чтобы разработать программную систему, приносящую реальные выгоды определенным пользователям, необходимо сначала выяснить, какие же задачи она должна решать для этих людей и какими свойствами обладать.

Требования к ПО определяют, какие свойства и характеристики оно должно иметь для удовлетворения потребностей пользователей и других заинтересованных лиц. Однако сформулировать требования к сложной системе не так легко. В большинстве случаев будущие пользователи могут перечислить набор свойств, который они хотели бы видеть, но никто не даст гарантий, что это – исчерпывающий список. Кроме того, часто сама формулировка этих свойств будет непонятна большинству программистов: могут прозвучать фразы типа «должно использоваться и частотное, и временное уплотнение каналов», «передача клиента должна быть мягкой», «для обычных швов отмечайте бригаду, а для доверительных – конкретных сварщиков», и это еще не самые тяжелые для понимания примеры.

Чтобы ПО было действительно полезным, важно, чтобы оно удовлетворяло реальные потребности людей и организаций, которые часто отличаются от непосредственно выражаемых пользователями желаний. Для выявления этих потребностей, а также для выяснения смысла высказанных требований приходится проводить достаточно большую дополнительную работу, которая называется анализом предметной области или бизнес-моделированием, если речь идет о потребностях коммерческой организации. В результате этой деятельности разработчики должны научиться понимать язык, на котором говорят пользователи и заказчики, выявить цели их деятельности, определить набор задач, решаемых ими. В дополнение стоит выяснить, какие вообще задачи нужно уметь решать для достижения этих целей, выяснить свойства результатов, которые хотелось бы получить, а также определить набор сущностей, с которыми приходится иметь дело при решении этих задач. Кроме того, анализ предметной области позволяет выявить места возможных улучшений и оценить последствия принимаемых решений о реализации тех или иных функций.

После этого можно определять область ответственности будущей программной системы – какие именно из выявленных задач будут ею решаться, при решении каких задач она может оказать существенную помощь и чем именно. Определив эти задачи в рамках общей системы задач и деятельности пользователей, можно уже более точно сформулировать требования к ПО.

Анализом предметной области занимаются системные аналитики или бизнес-аналитики, которые передают полученные ими знания другим членам проектной команды, сформулировав их на более понятном разработчикам языке. Для передачи этих знаний обычно служит некоторый набор моделей, в виде графических схем и текстовых документов.

Часто для описания поведения сложных систем и деятельности крупных организаций используются диаграммы потоков данных (data flow diagrams). Эти диаграммы содержат 4 вида графических элементов: процессы, представляющие собой любые трансформации данных в рамках описываемой системы, хранилища данных, внешние по отношению к системе сущности и потоки данных между элементами трех предыдущих видов.

Процессы на диаграммах потоков данных могут уточняться: если некоторый процесс устроен достаточно сложно, для него можно нарисовать отдельную диаграмму, описывающую потоки данных внутри этого процесса. На ней показываются те элементы, с которыми этот процесс связан потоками данных, и составляющие его более мелкие процессы и хранилища. Таким образом, возникает иерархическая структура процессов. Обычно на самом верхнем уровне находится один процесс, представляющий собой систему в целом, и набор внешних сущностей, с которыми она взаимодействует.

Диаграммы потоков данных появились как один из первых инструментов представления деятельности сложных систем при использовании структурного анализа. Для представления структуры данных в этом подходе используются диаграммы сущностей и связей (entity-relationship diagrams, ER diagrams), изображающие набор сущностей предметной области и связей между ними. И сущности, и связи на таких диаграммах могут иметь атрибуты.

Хотя методы структурного анализа могут значительно помочь при анализе систем и организаций, дальнейшая разработка системы, поддерживающей их деятельность, с использованием объектноориентированного подхода часто требует дополнительной работы по переводу полученной информации в объектно-ориентированные модели.

Методы объектно-ориентированного анализа предназначены для обеспечения более удобной передачи информации между моделями анализируемых систем и моделями разрабатываемого ПО. В качестве графических моделей в этих методах вместо диаграмм потоков данных используются рассматривавшиеся при обсуждении RUP диаграммы вариантов использования, а вместо диаграмм сущностей и связей – диаграммы классов.

Однако диаграммы вариантов использования несут несколько меньше информации по сравнению с соответствующими диаграммами потоков данных: на них процессы и хранилища в соответствии с принципом объединения данных и методов работы с ними объединяются в варианты использования, и остаются только связи между вариантами использования и действующими лицами (аналогом внешних сущностей). Для представления остальной информации каждый вариант использования может дополняться набором разнообразных диаграмм UML – диаграммами деятельности, диаграммами сценариев, и пр.

Выделение и анализ требований

После получения общего представления о деятельности и целях организаций, в которых будет работать будущая программная система, и о ее предметной области, можно определить более четко, какие именно задачи система будет решать. Кроме того, важно понимать, какие из задач стоят наиболее остро и обязательно должны быть поддержаны уже в первой версии, а какие могут быть отложены до

следующих версий или вообще вынесены за рамки области ответственности системы. Эта информация выявляется при анализе потребностей возможных пользователей и заказчиков.

Потребности определяются на основе наиболее актуальных проблем и задач, которые пользователи и заказчики видят перед собой. При этом требуется аккуратное выявление значимых проблем, определение того, насколько хорошо они решаются при текущем положении дел, и расстановка приоритетов при рассмотрении недостаточно хорошо решаемых, поскольку чаще всего решить сразу все проблемы невозможно.

Формулировка потребностей может быть разбита на следующие этапы.

- Выделить одну-две-три основных проблемы.
- Определить причины возникновения проблем, оценить степень их влияния и выделить наиболее существенные из проблем, влекущие появление остальных.
- Определить ограничения на возможные решения.

Формулировка потребностей не должна накладывать лишних ограничений на возможные решения, удовлетворяющие им. Нужно попытаться сформулировать, что именно является проблемой, а не предлагать сразу возможные решения.

При выявлении потребностей пользователей анализируются модели деятельности пользователей и организаций, в которых они работают, для выявления проблемных мест. Также используются такие приемы, как анкетирование, демонстрация возможных сеансов работы будущей системы, интерактивные опросы, где пользователям предоставляется возможность самим предложить варианты внешнего вида системы и ее работы или поменять предложенные кем-то другим, демонстрация прототипа системы и др.

После выделения основных потребностей нужно решить вопрос о разграничении области ответственности будущей системы, т.е. определить, какие из потребностей надо пытаться удовлетворить в ее рамках, а какие – нет.

При этом все заинтересованные лица делятся на пользователей, которые будут непосредственно использовать создаваемую систему для решения своих задач, и вторичных заинтересованных лиц, которые не решают своих задач с ее помощью, но чьи интересы так или иначе затрагиваются ею. Потребности пользователей нужно удовлетворить в первую очередь и на это нужно выделить больше усилий, а интересы вторичных заинтересованных лиц должны быть только адекватно учтены в итоговой системе.

На основе выделенных потребностей пользователей, отнесенных к области ответственности системы, формулируются возможные функции будущей системы, которые представляют собой услуги, предоставляемые системой и удовлетворяющие потребности одной или нескольких групп пользователей (или других заинтересованных лиц). Идеи для определения таких функций можно брать из имеющегося опыта разработчиков (наиболее часто используемый источник) или из результатов мозговых штурмов и других форм выработки идей.

При этом часто нужно учитывать, что ПО является частью программно-аппаратной системы, требования к которой надо

преобразовать в требования к программной и аппаратной ее составляющим. В последнее время, в связи со значительным падением цен на мощное аппаратное обеспечение общего назначения, фокус внимания переместился, в основном, на программное обеспечение. Во многих проектах аппаратная платформа определяется из общих соображений, а поддержку большинства нужных функций осуществляет ПО.

Каждое требование раскрывает детали поведения системы при выполнении ею некоторой функции в некоторых обстоятельствах. При этом часть требований исходит из потребностей и пожеланий заинтересованных лиц и решений, удовлетворяющих эти потребности и пожелания, а часть – из внешних ограничений, накладываемых на систему, например, основными законами той предметной области, в рамках которой системе придется работать, государственным законодательством, корпоративной политикой и пр.

Еще до перехода от функций к требованиям полезно расставить приоритеты и оценить трудоемкость их реализации и рискованность. Это позволит отказаться от реализации наименее важных и наиболее трудоемких, не соответствующих бюджету проекта функций еще до их детальной проработки, а также выявить возможные проблемные места проекта – наиболее трудоемкие и неясные из вошедших в него функций.

Для представления архитектуры, а точнее – различных входящих в нее структур, удобно использовать графические языки. На настоящий момент наиболее проработанным и наиболее широко используемым из них является унифицированный язык моделирования (Unified Modeling Language, UML), хотя достаточно часто архитектуру системы описывают просто набором именованных прямоугольников, соединенных линиями и стрелками, которые представляют возможные связи.

UML предлагает использовать для описания архитектуры 8 видов диаграмм. 9-й вид UML диаграмм, диаграммы вариантов использования, не относится к архитектурным представлениям. Кроме того, и другие виды диаграмм можно использовать для описания внутренней структуры компонентов или сценариев действий пользователей и прочих элементов, к архитектуре часто не относящихся. В этом курсе мы не будем разбирать диаграммы UML в деталях, а ограничимся обзором их основных элементов, необходимым для общего понимания смысла того, что изображено на таких диаграммах.

Диаграммы UML делятся на две группы – статические и динамические диаграммы.

Статические диаграммы представляют либо постоянно присутствующие в системе сущности и связи между ними, либо суммарную информацию о сущностях и связях, либо сущности и связи, существующие в какой-то определенный момент времени. Они не показывают способов поведения этих сущностей. К этому типу относятся диаграммы классов, объектов, компонентов и диаграммы развертывания.

Диаграммы классов (class diagrams) показывают классы или типы сущностей системы, характеристики классов (поля и операции) и возможные связи между ними. Пример диаграммы классов изображен на Рис. 31.

Классы представляются прямоугольниками, поделенными на три части. В верхней части показывают имя класса, в средней – набор

его полей, с именами, типами, модификаторами доступа (public '+', protected '#', private '-') и начальными значениями, в нижней – набор операций класса. Для каждой операции показывается ее модификатор доступа и сигнатура.

На Рис. 1 изображены классы Account, Person, Organization, Address, CreditAccount и абстрактный класс Client.

Класс CreditAccount имеет private поле maximumCredit типа double, а также public, метод getCredit() и protected метод setCredit().

Интерфейсы, т.е. типы, имеющие только набор операций и не определяющие способов их реализации, часто показываются в виде небольших кружков, хотя могут изображаться и как обычные классы. На Рис. 1 представлен интерфейс AccountInterface.

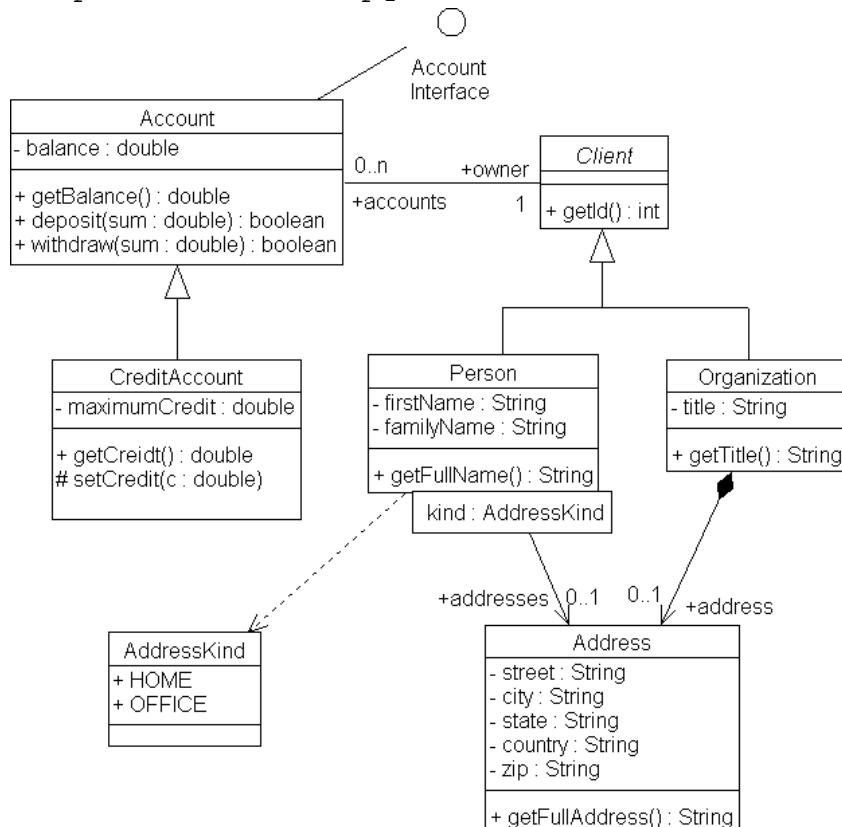


Рисунок 1. Диаграмма классов

Наиболее часто используется три вида связей между классами – связи по композиции, ссылки, связи по наследованию и реализации.

Композиция описывает ситуацию, в которой объекты класса А включают в себя объекты класса В, причем последние не могут разделяться (объект класса В, являющийся частью объекта класса А, не может являться частью другого объекта класса А) и существуют только в рамках объемлющих объектов (уничтожаются при уничтожении объемлющего объекта).

Композицией на Рис. 1 является связь между классами Organization и Address. Ссылочная связь (или слабая агрегация) обозначает, что объект некоторого класса А имеет в качестве поля ссылку на объект другого (или того же самого) класса В, причем ссылки на один и тот же объект класса В могут иметься в нескольких объектах класса А.

И композиция, и ссылочная связь изображаются стрелками, ведущими от класса А к классу В. Композиция дополнительно имеет закрашенный ромбик у начала этой стрелки. Двусторонние ссылочные связи, обозначающие, что объекты могут иметь ссылки друг на друга, показываются линиями без стрелок. Такая связь показана на Рис. 1 между классами Account и Client.

Эти связи могут иметь описание множественности, показывающее, сколько объектов класса В может быть связано с одним объектом класса А. Оно изображается в виде текстовой метки около конца стрелки, содержащей точное число или нижние и верхние границы, причем бесконечность изображается звездочкой или буквой n. Для двусторонних связей множественности могут показываться с обеих сторон. На Рис. 1 множественности, изображенные для связи между классами Account и Client, обозначают, что один клиент может иметь много счетов, а может и не иметь ни одного, и счет всегда привязан ровно к одному клиенту.

Наследование классов изображается стрелкой с пустым наконечником, ведущей от наследника к предку. На Рис. 1 класс CreditAccount наследует классу Account, а классы Person и Organization – классу Client.

Реализация интерфейсов показывается в виде пунктирной стрелки с пустым наконечником, ведущей от класса к реализуемому им интерфейсу, если тот показан в виде прямоугольника. Если же интерфейс изображен в виде кружка, то связь по реализации показывается обычной сплошной линией (в этом случае неоднозначности в ее толковании не возникает). Такая связь изображена на Рис. 1 между классом Account и интерфейсом AccountInterface.

Один класс использует другой, если этот другой класс является типом параметра или результата операции первого класса. Иногда связи по использованию показываются в виде пунктирных стрелок. Пример такой связи между классом Person и перечислимым типом AddressKind можно видеть на Рис. 1.

Ссылочные связи, реализованные в виде ассоциативных массивов или отображений (map) – такая связь в зависимости от некоторого набора ключей определяет набор ссылок-значений – показываются при помощи стрелок, имеющих прямоугольник с перечислением типов и имен ключей, примыкающий к изображению класса, от которого идет стрелка.

Множественность на конце стрелки при этом обозначает количество ссылок, соответствующее одному набору значений ключей.

На Рис. 1 такая связь ведет от класса Person к классу Address, показывая, что объект класса Person может иметь один адрес для каждого значения ключа kind, т.е. один домашний и один рабочий адреса.

Диаграммы классов используются чаще других видов диаграмм.

Диаграммы объектов (object diagrams) показывают часть объектов системы и связи между ними в некотором конкретном состоянии или суммарно, за некоторый интервал времени. Объекты изображаются прямоугольниками с идентификаторами ролей объектов (в контексте тех состояний, которые изображены на диаграмме) и типами. Однородные коллекции объектов могут изображаться накладывающимися друг на друга прямоугольниками.

Такие диаграммы используются довольно редко.

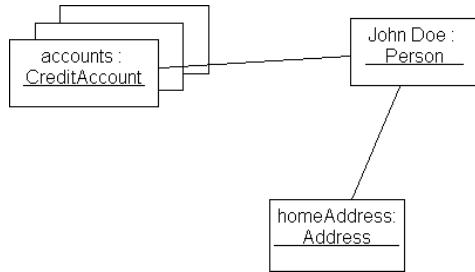


Рисунок 2. Диаграмма объектов.

Диаграммы компонентов (component diagrams) представляют компоненты в нескольких смыслах – атомарные составляющие системы с точки зрения ее сборки, конфигурационного управления и развертывания. Компоненты сборки и конфигурационного управления обычно представляют собой файлы с исходным кодом, динамически подгружаемые библиотеки, HTML-страницы и пр., компоненты развертывания – это компоненты JavaBeans, CORBA, COM и т.д.

Компонент изображается в виде прямоугольника с несколькими прямоугольными или другой формы «зубами» на левой стороне.

Связи, показывающие зависимости между компонентами, изображаются пунктирными стрелками. Один компонент зависит от другого, если он не может быть использован в отсутствии этого другого компонента в конфигурации системы. Компоненты могут также реализовывать интерфейсы.

Диаграммы этого вида используются редко.

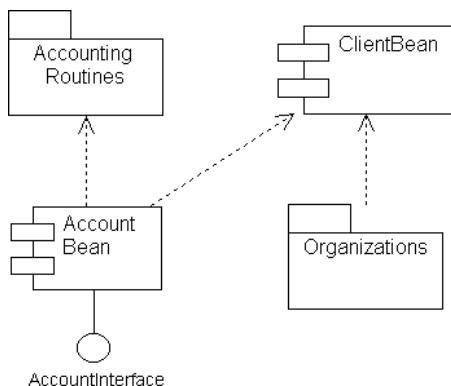


Рисунок 3. Диаграмма компонентов.

На диаграмме компонентов, изображенной на Рис. 3, можно также увидеть пакеты, изображаемые в виде «папок», точнее – прямоугольников с прямоугольными «наростами» над левым верхним углом. Пакеты являются пространствами имён и средством группировки диаграмм и других модельных элементов UML – классов, компонентов и пр. Они могут появляться на диаграммах классов и компонентов для указания зависимостей между ними и отдельными классами и компонентами. Иногда на такой диаграмме могут присутствовать только пакеты с зависимостями между ними.

Диаграммы развертывания (deployment diagrams) показывают декомпозицию системы на физические устройства различных видов – серверы, рабочие станции, терминалы, принтеры, маршрутизаторы и

пр. – и связи между ними, представленные различного рода сетевыми и индивидуальными соединениями.

Физические устройства, называемые узлами системы (nodes), изображаются в виде кубов или параллелепипедов, а физические соединения между ними – в виде линий.

На диаграммах развертывания может быть показана привязка (в некоторый момент времени или постоянная) компонентов развертывания системы к физическим устройствам

– например, для указания того, что компонент EJB AccountEJB исполняется на сервере приложений, а аплет AccountInfoEditor – на рабочей станции оператора банка.

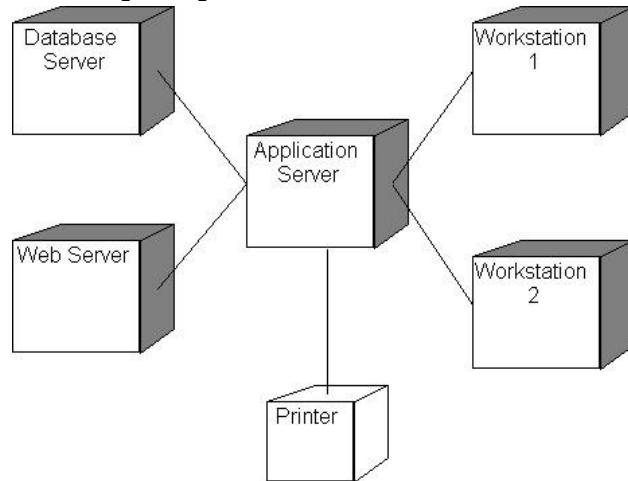


Рисунок 4. Диаграмма развертывания.

Эти диаграммы используются достаточно редко. Пример диаграммы развертывания изображен на Рис. 4.

Динамические диаграммы описывают происходящие в системе процессы. К ним относятся диаграммы деятельности, сценариев, диаграммы взаимодействия и диаграммы состояний.

Диаграммы деятельности (activity diagrams) иллюстрируют набор процессов – деятельности и потоки данных между ними, а также возможные их синхронизации друг с другом.

Деятельность изображается в виде прямоугольника с закругленными сторонами, слева и справа, помеченного именем деятельности.

Потоки данных показываются в виде стрелок. Синхронизации двух видов – разветвления (forks) и слияния (joins) – показываются жирными короткими линиями (кто-то может посчитать их и тонкими закрашенными прямоугольниками), к которым сходятся или от которых расходятся потоки данных. Кроме синхронизаций, на диаграммах деятельности могут быть показаны разветвления потоков данных, связанных с выбором того или иного направления в зависимости от некоторого условия. Такие разветвления показываются в виде небольших ромбов.

Диаграмма может быть поделена на несколько горизонтальных или вертикальных областей, называемых дорожками (swimlanes). Дорожки служат для группировки деятельности в соответствии с выполняющими их подразделением организации, ролью, приложением, подсистемой и пр.

Диаграммы деятельности могут заменять часто используемые диаграммы потоков данных, поэтому применяются достаточно широко. Пример такой диаграммы показан на Рис. 5.

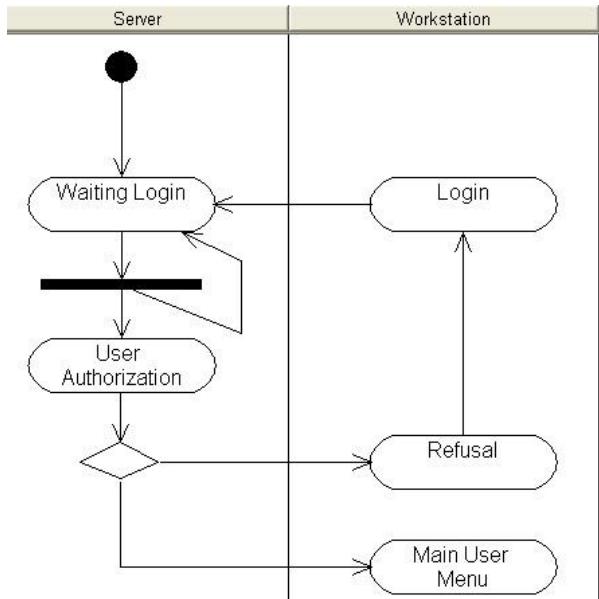


Рисунок 5. Диаграмма деятельности.

Диаграммы сценариев (или диаграммы последовательности, sequence diagrams) показывают возможные сценарии обмена сообщениями или вызовами во времени между различными компонентами системы (здесь имеются в виду архитектурные компоненты, компоненты в широком смысле – это могут быть компоненты развертывания, обычные объекты, подсистемы и пр.). Эти диаграммы являются подмножеством специального графического языка – языка диаграмм последовательностей сообщений (Message Sequence Charts, MSC), который был придуман раньше UML и достаточно долго развивается параллельно ему.

Компоненты, участвующие во взаимодействии, изображаются прямоугольниками вверху диаграммы. От каждого компонента вниз идет вертикальная линия, называемая его линией жизни. Считается, что ось времени направлена вертикально вниз. Интервалы времени, в которые компонент активен, т.е. управление находится в одной из его операций, представлены тонким прямоугольником, для которого линия жизни компонента является осью симметрии.

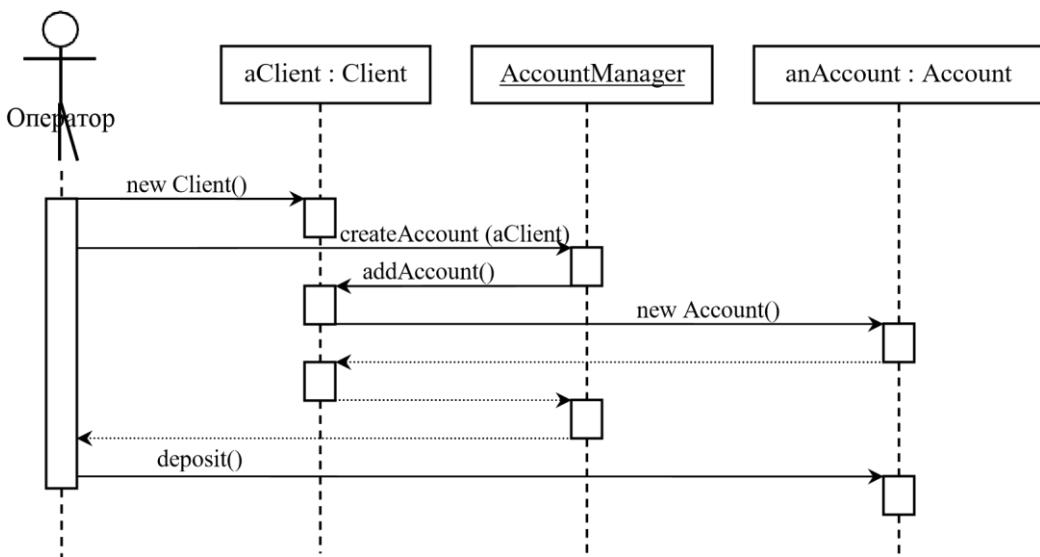


Рисунок 6. Пример диаграммы сценария открытия счета.

Передача сообщения или вызов изображаются стрелкой от компонента-источника к компоненту-приемнику. Возврат управления показан пунктирной стрелкой, обратной к соответствующему вызову.

Эти диаграммы используются достаточно часто, например, при детализации сценариев, входящих в варианты использования. Пример такой диаграммы изображен на Рис. 6.

Диаграммы взаимодействия (collaboration diagrams) показывают ту же информацию, что и диаграммы сценариев, но привязывают обмен сообщениями/вызовами не к времени, а к связями между компонентами.

На диаграмме изображаются компоненты в виде прямоугольников и связи между ними. Вдоль связей могут передаваться сообщения, показываемые в виде небольших стрелок, параллельных связи. Стрелки нумеруются в соответствии с порядком происходящих событий. Нумерация может быть иерархической, чтобы показать вложенность действий друг в друга (т.е. если вызов некоторой операции имеет номер 1, то вызовы, осуществляемые при выполнении этой операции, будут нумероваться как 1.1, 1.2, и т.д.). Диаграммы взаимодействия используются довольно редко.

Диаграммы состояний (statechart diagrams) показывают возможные состояния отдельных компонентов или системы в целом, переходы между ними в ответ на какие-либо события и выполняемые при этом действия.

Состояния показываются в виде прямоугольников с закругленными углами, переходы — в виде стрелок. Начальное состояние представляется как небольшой темный кружок, конечное — как пустой кружок с концентрически вложенным темным кружком. Вы могли обратить внимание на темный кружок на диаграмме деятельности на Рис. 5 — он тоже изображает начальное состояние: дело в том, что диаграммы деятельности являются диаграммами состояний специального рода, а деятельности — частный случай состояний. Пример диаграммы состояний приведен на Рис. 7.

Состояния могут быть устроены иерархически: они могут включать в себя другие состояния, даже целые отдельные диаграммы вложенных состояний и переходов между ними. Пребывая в таком состоянии, система находится ровно в одном из его подсостояний. На

Рис. 7 почти все изображенные состояния являются подсостояниями состояния Site.

Кроме того, в нижней части диаграммы три состояния объединены, чтобы показать, что переход по действию cancel возможен в каждом из них и приводит в одно и то же состояние Basket.

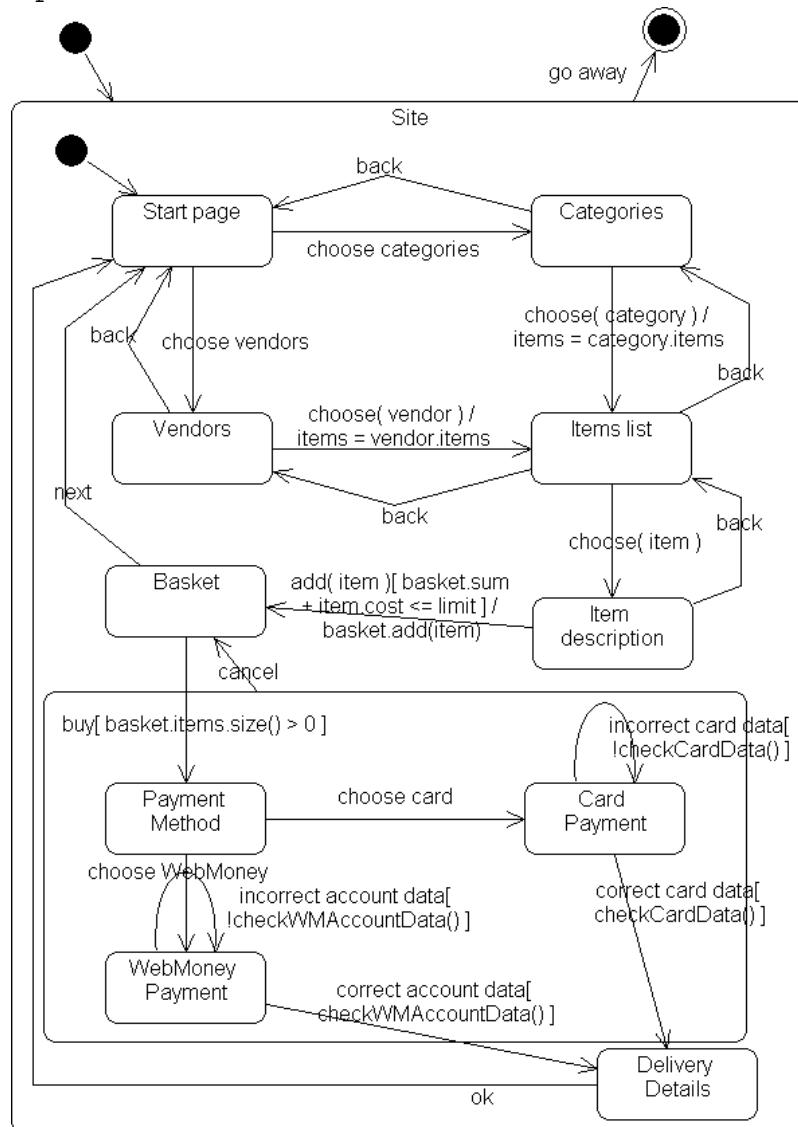


Рисунок 7. Пример диаграммы состояний, моделирующей сайт Интернет-магазина.

Состояние может декомпозироваться и на параллельные подсостояния. Они изображаются как области внутри объемлющего состояния, разделенные пунктирными линиями, их аналогом на диаграммах деятельности являются дорожки. Пребывая в объемлющем состоянии, система должна находиться одновременно в каждом из его параллельных подсостояний.

Помимо показанных на диаграмме состояний изображаемая подсистема может иметь глобальные (в ее рамках) переменные, хранящие какие-то данные. Значения этих переменных являются общими частями всех изображаемых состояний.

На Рис.7 примерами переменных являются список видимых пользователем товаров, items, и набор уже отобранных товаров с количеством для каждого, корзина, basket. Переходы могут происходить между состояниями одного уровня, но могут также вести из некоторого состояния в подсостояние соседнего или, наоборот, из

подсостояния в некоторое состояние, находящее на том же уровне, что и объемлющее состояние.

На переходе между состояниями указываются следующие данные:

Событие, приводящее к выполнению этого перехода. Обычно событие – это вызов некоторой операции в одном из объектов или приход некоторого сообщения, хотя могут указываться и абстрактные события.

Например, из состояния Categories на Рис. 7 можно выйти, выполнив команду браузера «Назад». Она соответствует событию back, инициирующему переход в состояние Start page. Другой переход из состояния Categories происходит при выборе категории товаров пользователем. Соответствующее событие имеет параметр – выбранную категорию. Оно изображено как choose(category).

Условие выполнения (охранное условие, guardian). Это условие, зависящее от параметров события и текущих значений глобальных переменных, выполнение которого необходимо для выполнения перехода. При наступлении нужного события переход выполняется, только если его условие тоже выполнено.

Условие перехода показывается в его метке в квадратных скобках.

На Рис. 7 примером условного перехода является переход из состояния Basket в состояние Payment Method. Он выполняется, только если пользователь выполняет команду «Оплатить» (событие buy) и при этом в его корзине есть хотя бы один товар.

Действие, выполняемое в дополнение к переходу между состояниями. Обычно это вызовы каких-то операций и изменения значения глобальных переменных. Действие показывается в метке перехода после слеша (символа '/') . При этом изменения значений переменных перечисляются в начале, затем, после знака '^', указывается вызов операции.

Например, на Рис. 7 при выборе пользователем категории товаров происходит переход из состояния Categories в Items list. При этом список товаров, видимый пользователю, инициализируется списком товаров выбранной категории.

Диаграммы состояний используются часто, хотя требуется довольно много усилий, чтобы разработать их с достаточной степенью детальности.

Рассмотрим проектирование диаграмм в среде PlantUML.

Диаграмма последовательности

Основные примеры

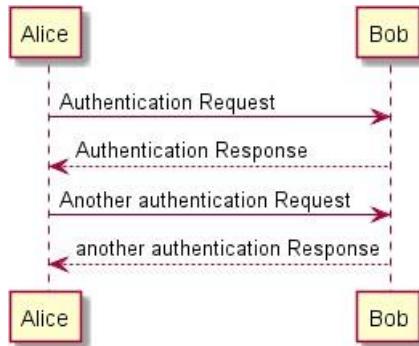
Последовательность `->` используется, чтобы отобразить сообщение между двумя участниками (participants). Не обязательно явно объявлять участников.

Для получения пунктирной стрелки (dotted arrow), используйте `-->`.

Также возможно использовать `<-` и `<--`. Это не изменит отображение, но может улучшить читабельность. Заметьте, что это верно только для диаграмм последовательности, для других диаграмм правила другие. @startuml

```
Alice -> Bob: Authentication Request
Bob --> Alice:
Authentication Response
```

```
Alice -> Bob: Authentication Request
Alice <- Bob:
another authentication Response
@enduml
```



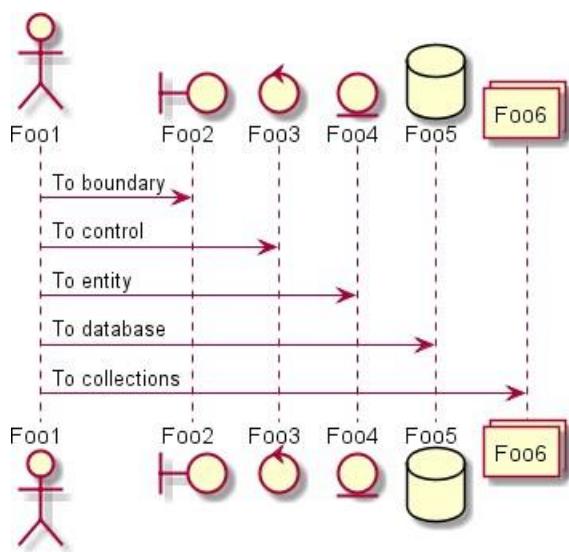
Объявление участников

Возможно изменить порядок участников, используя ключевое слово `participant`. Так же возможно использование других ключевых слов, для объявления `participant'a`:

```

actor
boundary
control
entity
database
@startuml actor Foo1 boundary Foo2 control Foo3 entity Foo4
database Foo5 collections Foo6
    Foo1 -> Foo2 : To boundary
    Foo1 -> Foo3 : To control
    Foo1 -> Foo4 : To entity
    Foo1 -> Foo5 : To database
    Foo1 -> Foo6 : To collections
    
```

```
@enduml
```

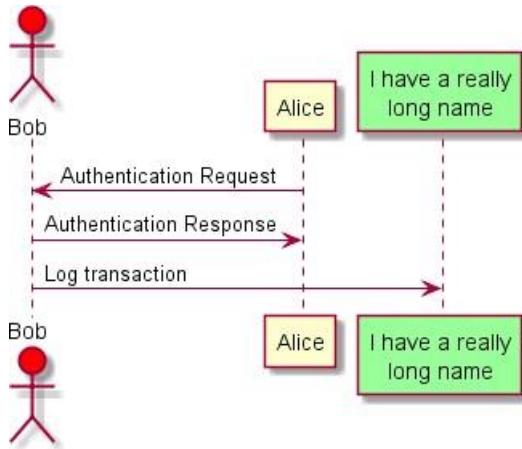


Можно переименовать участника используя ключевое слово `as`
Также возможно изменить цвет фона `actor-a` или участника,
используя имя цвета или его html-код

```
@startuml actor Bob #red
```

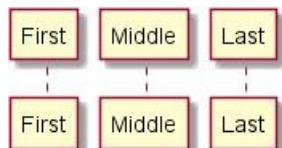
```
' The only difference between actor 'and participant is the
drawing participant Alice participant "I have a really\nlong name"
as L #99FF99 /' You can also declare: participant L as "I have a
really\nlong name" #99FF99 '/
```

```
Alice->Bob: Authentication Request Bob->Alice: Authentication
Response Bob->L: Log transaction
@enduml
```



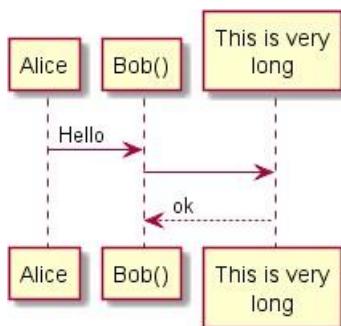
Использовав ключевое поле `order` (порядок) можно настроить порядок печати `participant` (участника).

```
@startuml
participant Last order 30 participant Middle order 20
participant First order 10 @enduml
```



Использование небуквенных символов в названиях участников
Вы можете использовать кавычки для задания участников. Также
Вы можете использовать ключевое слово `as` для
присвоения псевдонимов к этим участникам.

```
@startuml
Alice -> "Bob()": Hello
"Bob()" -> "This is very\nlong" as Long
' You can also
declare:
' "Bob()" -> Long as "This is very\nlong"
Long --> "Bob()": ok @enduml
```

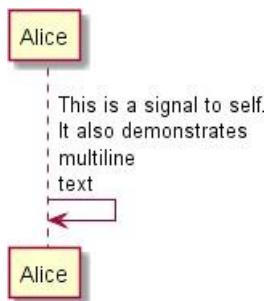


Сообщения к самому себе

Участник может посыпать сообщения сам себе.

Также возможно создание многострочных используя \n. @startuml

```
Alice->Alice: This is a signal to self.\nIt also  
demonstrates\nmultiline \n\ntext @enduml
```



Изменить стиль стрелок

Вы можете изменить стиль стрелок следующими способами:

закончить стрелку с помощью x для обозначения потерянного сообщения используя \ или / вместо < или > для создания только верхней

или нижней части стрелки.

повторите окончание стрелки (например, >> or //) для тонкой отрисовки.

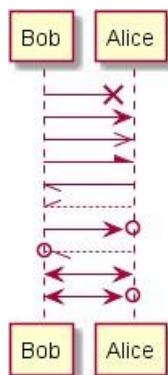
используйте -- вместо - для создания пунктирной стрелки

заканчивать символом "o" в острие стрелки использовать двунаправленные стрелки <-> @startuml

```
Bob ->x Alice Bob -> Alice Bob ->> Alice Bob -\ Alice Bob \\-  
Alice Bob //-- Alice
```

```
Bob ->o Alice Bob o\\-- Alice
```

```
Bob <-> Alice Bob <->o Alice @enduml
```

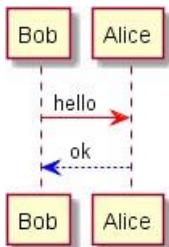


Изменить цвет стрелок

Вы можете изменить цвет отдельных стрелок, используя следующие правила:

```
@startuml
```

```
Bob -[#red]> Alice : hello Alice -[#0000FF]->Bob : ok @enduml
```



Нумерация сообщений в последовательностях

Ключевое слово autonumber используется для автоматической нумерации сообщений.

```

@startuml autonumber
Bob -> Alice : Authentication Request Bob <- Alice :
Authentication Response @enduml
  
```



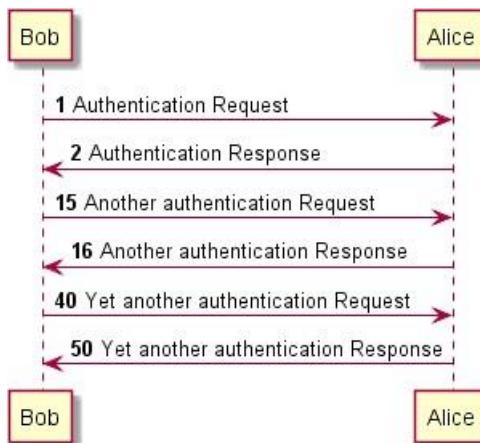
Вы можете обозначить число с которого начнется отсчет autonumber start , и число которое будет использоваться в качестве инкремента autonumber start increment. @startuml autonumber

```

Bob -> Alice : Authentication Request Bob <- Alice :
Authentication Response
    autonumber
    15
    Bob -> Alice : Another authentication Request Bob <- Alice :
    Another authentication Response

    autonumber 40 10
    Bob -> Alice : Yet another authentication Request Bob <- Alice
    : Yet another authentication Response
  
```

@enduml



Можно задавать формат чисел, указав его в двойных кавычках.

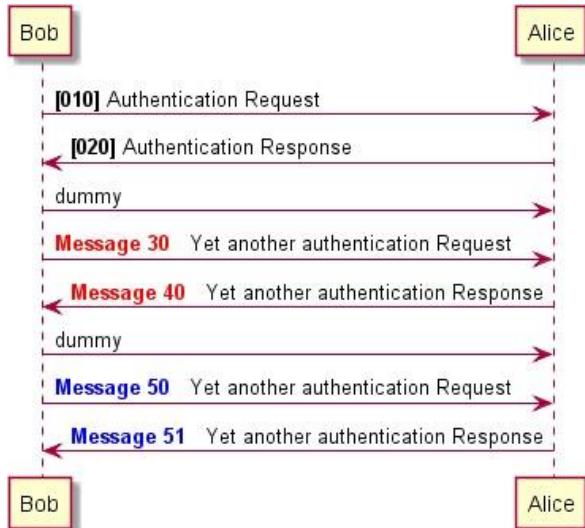
Форматирование выполнено с использованием класса Java DecimalFormat (0 означает цифру, # означает цифру или ноль если отсутствует).

```
При форматировании также можно использовать теги html.  
@startuml autonumber  
"<b>[000]"  
Bob -> Alice : Authentication Request Bob <- Alice :  
Authentication Response  
  
autonumber 15 "<b>(<u>##</u>)"  
Bob -> Alice : Another authentication Request Bob <- Alice :  
Another authentication Response  
  
autonumber 40 10 "<font color=red><b>Message 0 " Bob -> Alice  
: Yet another authentication Request Bob <- Alice : Yet another  
authentication Response  
@enduml
```

```
sequenceDiagram
    participant Bob
    participant Alice
    Bob->>Alice: [001] Authentication Request
    Alice-->Bob: [002] Authentication Response
    Bob->>Alice: (15) Another authentication Request
    Alice-->Bob: (16) Another authentication Response
    Bob->>Alice: Message 40 Yet another authentication Request
    Bob->>Alice: Message 50 Yet another authentication Response
```

Вы так же можете использовать autonumber stop и autonumber resume increment format чтобы соответственно остановить и продолжить автоматическое нумерование.

```
@startuml  
autonumber 10 10 "<b>[000]"  
Bob -> Alice : Authentication Request Bob <- Alice :  
Authentication Response  
  
autonumber stop Bob  
-> Alice : dummy  
  
autonumber resume "<font color=red><b>Message 0 " Bob -> Alice  
: Yet another authentication Request Bob <- Alice : Yet another  
authentication Response  
  
autonumber stop Bob  
-> Alice : dummy  
  
autonumber resume 1 "<font color=blue><b>Message 0 " Bob ->  
Alice : Yet another authentication Request Bob <- Alice : Yet  
another authentication Response @enduml
```



Page Title, Header and Footer

1.8 Page Title, Header and Footer

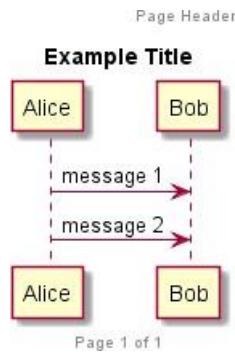
1 ДИАГРАММА ПОСЛЕДОВАТЕЛЬНОСТИ

The `title` keyword is used to add a title to the page.

Pages can display headers and footers using header and footer.

```
@startuml header Page Header footer Page
    page of lastpage title Example Title
    Alice %-> Bob : message 1 Alice -> Bob : message 2
```

@enduml



Разбиение диаграмм

Ключевое слово newpage используется для разбиения диаграмм на несколько изображений. Вы можете указать название страницы сразу после ключевого слова newpage.

Это очень полезно для печати длинных диаграмм на нескольких страницах.

```
@startuml
```

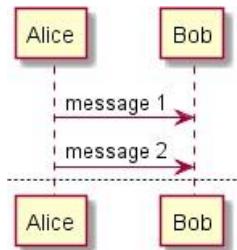
Alice → Bob : message 1 Alice → Bob : message 2

newpage

```

Alice -> Bob : message 3 Alice -> Bob : message 4
newpage A title for the\nlast page Alice -> Bob : message 5
Alice -> Bob : message 6
@enduml

```



Группировка сообщений

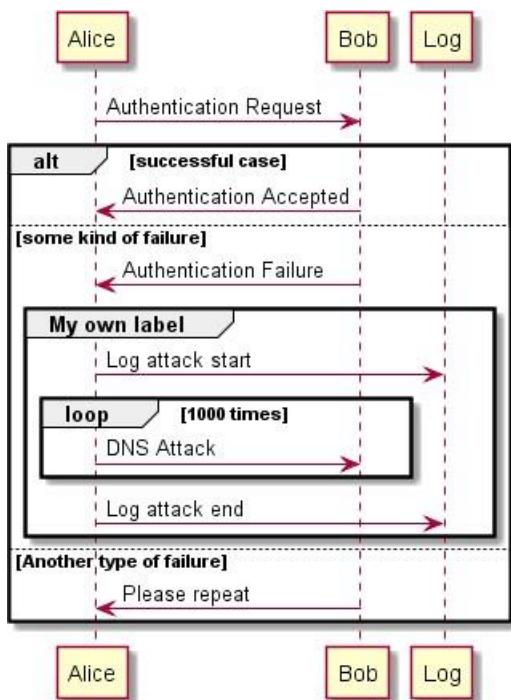
Группировать сообщения возможно используя следующие ключевые слова:

```

alt/els
e opt
loop
par
break
critica
l
group, соответствует тексту который должен быть отображен
Имеется возможность добавить текст который должен быть
отображен в заголовке. Ключевое слово end используется для
завершения группы. Имейте ввиду что допускаются вложенные
группы. Ключевое слово end закрывает группу.
Допустимо вложение группы в группу.
@startuml
Alice -> Bob: Authentication Request alt successful case
Bob -> Alice: Authentication Accepted else some kind of
failure
Bob -> Alice: Authentication Failure group My own label
Alice -> Log : Log attack start loop 1000 times
Alice -> Bob: DNS Attack
end
Alice -> Log : Log attack end end

else Another type of failure Bob -> Alice: Please repeat end
@enduml

```



Примечания в сообщениях

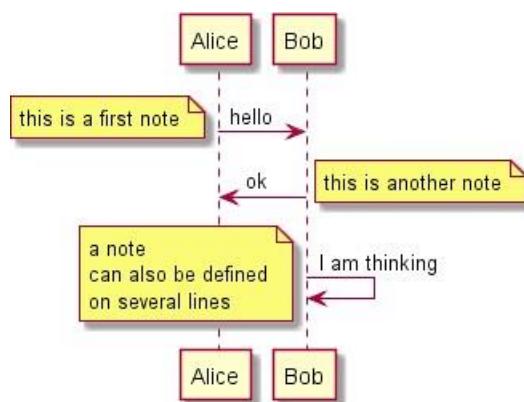
Можно помещать заметки к сообщениям, используя ключевые слова `note left` или `note right` сразу после сообщения.

Можно делать многострочные заметки используя ключевое слово `end note` для завершения.

```
@startuml Alice->Bob
: hello
note left: this is a first note
```

```
Bob->Alice : ok
note right: this is another note
```

```
Bob->Bob : I am thinking
note left a
note
can also be defined on several lines
end note
@enduml
```



Другие примечания

Так же возможно размещение примечаний относительно участников с использованием ключевых слов

<code>note left of</code> , note right of или note over. Возможно выделить примечание изменив цвет фона.

Так же возможно многостройное примечани, для этого существует ключевое слово end note.

```
@startuml participant Alice participant Bob  
note left of Alice #aqua This is displayed  
left of Alice. end note
```

note right of Alice: This is displayed right of Alice. note
over Alice: This is displayed over Alice.

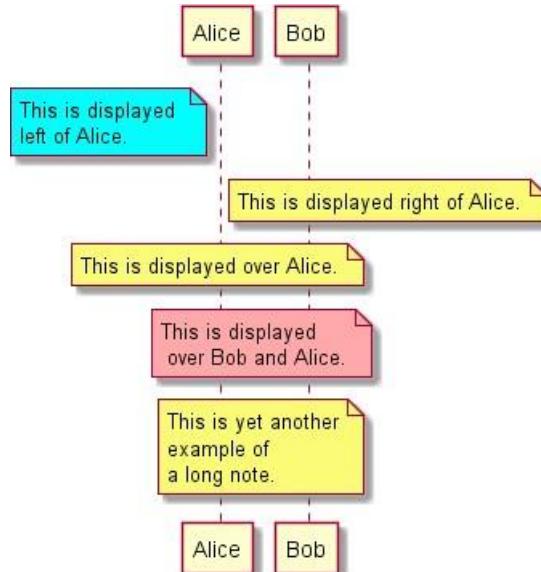
note over Alice, Bob #FFAAAA: This is displayed\n over Bob
and Alice.

note over Bob, Alice This is yet another example of

1.12 Другие примечания

I ДИАГРАММА ПОСЛЕДОВАТЕЛЬНОСТИ

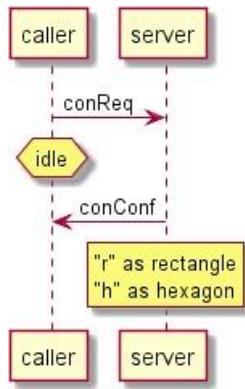
a long note. end note @enduml



Изменение формы примечаний

Вы можете использовать hnote и rnote для изменения формы примечаний. @startuml

```
caller -> server : conReq hnote over caller : idle caller <-  
server : conConf rnote over server  
"r" as rectangle "h" as hexagon  
endrnote @enduml
```



Creole и HTML

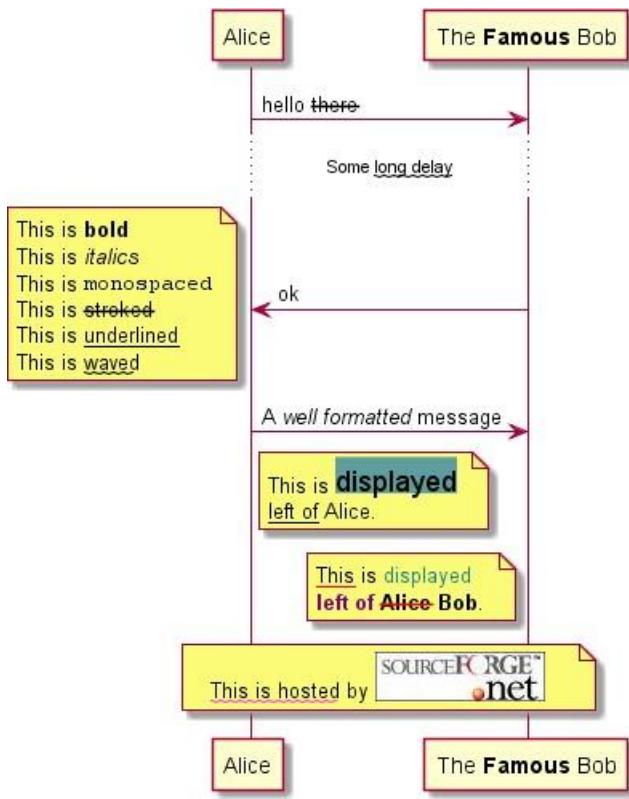
Так же можно использовать форматирование на Creole:

```
@startuml
participant Alice
participant "The **Famous** Bob" as Bob
```

```
Alice -> Bob : hello --there--
... Some ~~long delay~~ ...
Bob -> Alice : ok note left
This is **bold** This is //italics//  

This is ""monospaced"" This is --stroked-- This is underlined
    This is ~~waved~~ end
note
```

```
Alice -> Bob : A //well formatted// message note right of
Alice
    This is
    <back:cadetblue><size:18>displayed</size></back>
    left of Alice. end note note left of Bob
    <u:red>This</u> is <color #118888>displayed</color>
    **<color purple>left of</color> <s:red>Alice</strike> Bob**.
end note note over Alice,
    Bob
    <w:#FF33FF>This is hosted</w> by <img sourceforge.jpg> end
note
@enduml
```



Разделитель

Вы можете использовать разделитель "==" , чтобы разбить диаграмму на несколько этапов.

@startuml

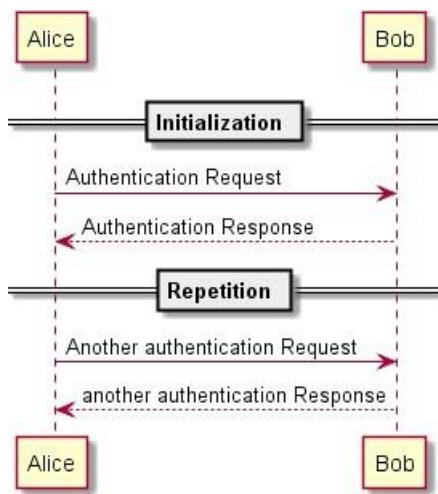
== Initialization ==

Alice -> Bob: Authentication Request Bob --> Alice:
Authentication Response

== Repetition ==

Alice -> Bob: Another authentication Request Alice <-- Bob:
another authentication Response

@enduml



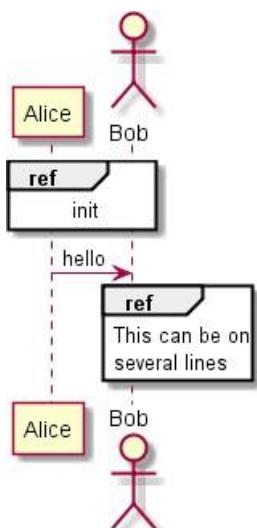
Ссылки

Вы можете использовать ссылки в диаграммах с помощью ключевого слова ref over. @startuml participant Alice actor Bob

```

ref over Alice, Bob : init Alice -> Bob :
hello ref over Bob This can be on several
lines end ref @enduml

```



Задержка на диаграммах

Вы можете использовать конструкцию ... для представления временной задержки в процессе на диаграмме. При необходимости можно снабдить задержку комментарием.

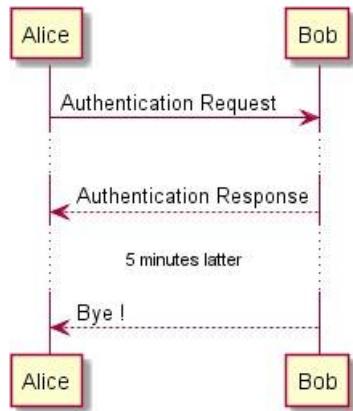
```
@startuml
```

```

Alice -> Bob: Authentication Request
...
Bob --> Alice: Authentication Response
...5 minutes later...
Bob --> Alice: Bye !

```

```
@enduml
```



Промежутки

Вы можете использовать ||| чтобы показать промежутки в диаграммах.. Так же возможно указать промежуток в пикселях.

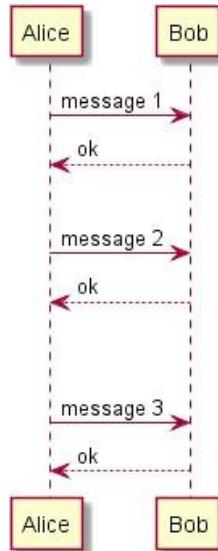
@startuml

```

Alice -> Bob: message 1 Bob --> Alice: ok |||
Alice -> Bob: message 2 Bob --> Alice: ok
||45||
Alice -> Bob: message 3 Bob --> Alice: ok

```

@enduml



Активация и деактивация линии существования

activate и deactivate используются чтобы обозначить активацию участника. Линия существования появляется в момент активации участника.

activate и deactivate применяются к предыдущему сообщению. destroy обозначает конец линии существования участника.

@startuml participant User

User -> A: DoWork activate A

-> B: << createRequest >> activate B

```

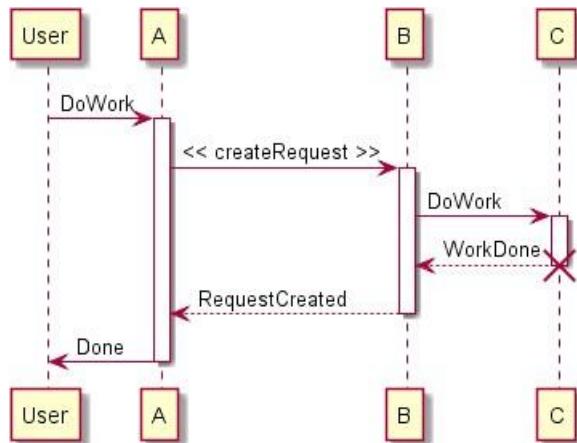
-> C: DoWork activate C
--> B: WorkDone destroy C

B --> A: RequestCreated deactivate B

A -> User: Done deactivate A

```

@enduml



Можно использовать вложенные линии существования, и возможно добавлять цвет линии существования

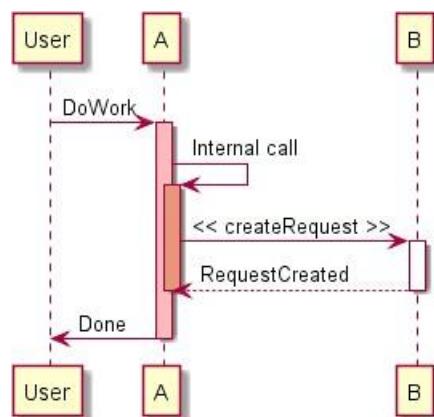
@startuml participant User

```

User -> A: DoWork activate A #FFBBBB
A -> A: Internal call activate A #DarkSalmon
-> B: << createRequest >> activate B
--> A: RequestCreated deactivate
B deactivate A
A -> User: Done deactivate A

```

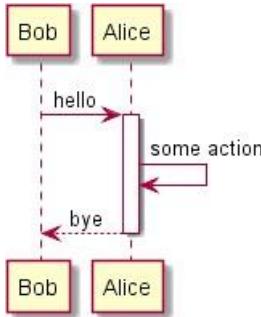
@enduml



Return

A new command return for generating a return message with optional text label. The point returned to is the point that cause the most recently activated life-line. The syntax is simply return label where label, if provided, can be any string acceptable on conventional messages.

```
@startuml
Bob -> Alice : hello activate Alice
Alice -> Alice : some action return bye
@enduml
```



Отображение создания участника процессом

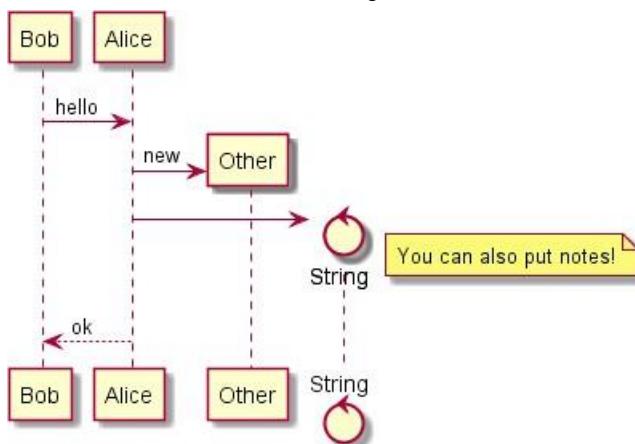
Вы можете использовать ключевое слово create перед декларацией сообщения для акцентирования факта, что принимающий участник создается данным сообщением.

```
@startuml
Bob -> Alice : hello

create Other
Alice -> Other : new

create control String Alice -> String note
right : You can also put notes!

Alice --> Bob : ok @enduml
```



Входящие и исходящие сообщения

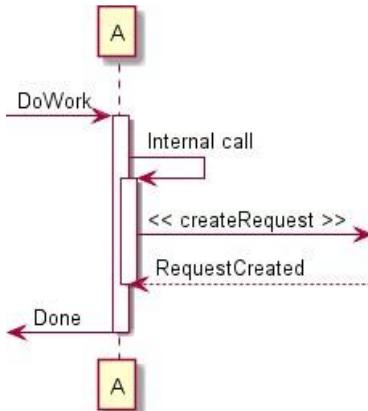
Вы можете использовать входящие или исходящие стрелки если вы хотите сфокусироваться на части диаграммы.

Используйте квадратные скобки для указания левой "[" или правой "]" стороны диаграммы

```
@startuml  
[-> A: DoWork activate A  
A -> A: Internal call activate A  
  
A ->] : << createRequest >>
```

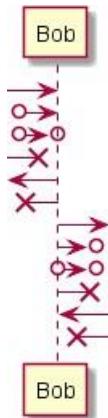
```
A<--] : RequestCreated deactivate A  
[<- A: Done deactivate A @enduml
```

Вы также можете использовать следующий синтаксис:



```
@startuml [-> Bob  
[o-> Bob [o->o Bob [x-> Bob  
  
[<- Bob [x<- Bob  
  
Bob ->] Bob ->o] Bob o->o] Bob ->x]
```

```
Bob <-] Bob x<-] @enduml
```

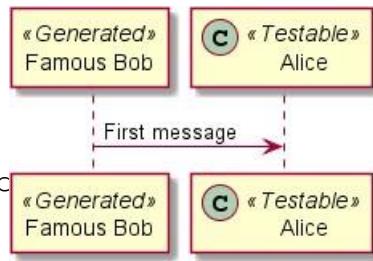


Шаблоны и отметки

Можно добавить шаблоны к участникам используя << и >>.

В шаблоне вы можете добавить отмеченного участника в цветном круге используя синтаксис (X,color). @startuml participant "Famous Bob" as Bob << Generated >> participant Alice << (C,#ADD1B2) Testable >>

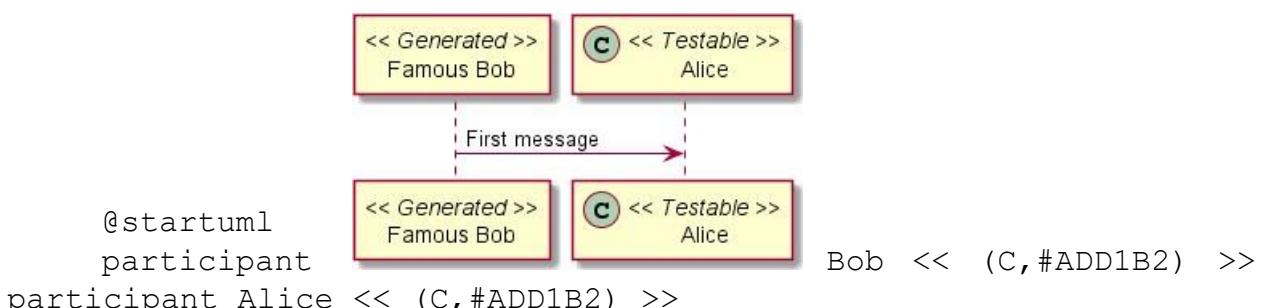
```
Bob->Alice: First message @enduml
```



По умолчанию, скрипты @startuml и @enduml пользуются для отображения шаблона. Вы можете изменить это поведение, используя skinparam guillemet:

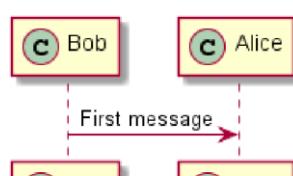
```
@startuml  
skinparam guillemet false  
participant "Famous Bob" as Bob << Generated >> participant Alice << (C,#ADD1B2) Testable >>
```

```
Bob->Alice: First message @enduml
```



```
Bob->Alice: First message @enduml
```

Страница 90 из

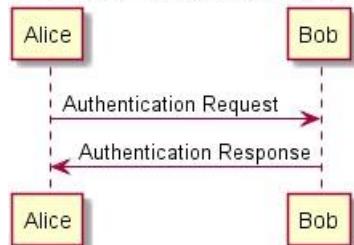


Больше информации в заголовках

Вы можете использовать форматирование на Creole для заголовков. @startuml title Simple **communication** example Alice -> Bob:
Authentication Request

Bob -> Alice: Authentication Response @enduml

Simple communication example



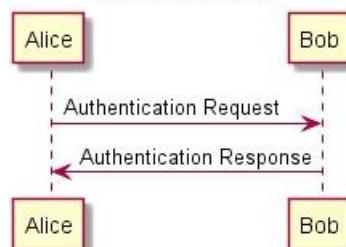
С помощью последовательности символов \n вы можете добавить перевод строки в заголовок.

```

@startuml
title Simple communication example\non several lines
Alice -> Bob: Authentication Request Bob -> Alice:
Authentication Response
  
```

@enduml

Simple communication example on several lines



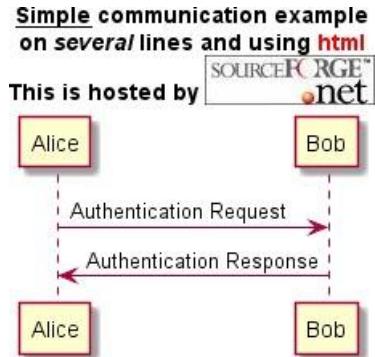
Вы также можете задать заголовок на нескольких строках, используя ключевые слова title и end title .

```

@startuml
1 title
<u>Simple</u> communication example
on <i>several</i> lines and using <font color=red>html</font>
This is hosted by <img:sourceforge.jpg> end
title
  
```

Alice -> Bob: Authentication Request Bob -> Alice:
Authentication Response

```
@enduml
```

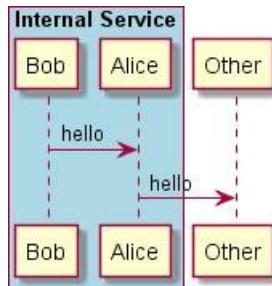


Группировка участников

Можно создать прямоугольник вокруг участников, используя команды box и end box. Вы можете задать optionalный заголовок и цвет фона, после команды the box. @startuml box "Internal Service" #LightBlue participant Bob participant Alice end box participant Other

```
Bob -> Alice : hello
Alice -> Other : hello @enduml
```

Удаление футера
Вы можете
hide footbox для



использовать ключевое слово
удаления футера из

```
@startuml hide
footbox title
Footer removed
```

```
Alice -> Bob: Authentication Request Bob --> Alice:
Authentication Response
```

```
@enduml
```



Skinparam

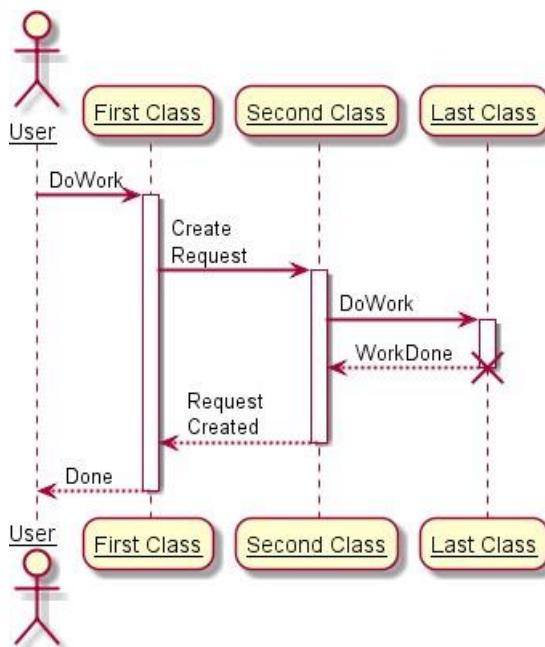
Вы можете использовать команду `skinparam` для изменения шрифтов и цветов диаграммы. Вы можете использовать данную команду:

- В определении диаграммы, как любую другую команду,
- В подключенном файле,
- В конфигурационном файле, указанном в командной строке в задании ANT.

Вы можете изменить другие параметры отображения, как видно из следующих примеров: @startuml

```
skinparam sequenceArrowThickness  
2 skinparam roundcorner 20  
skinparam maxmessagesize 60  
skinparam sequenceParticipant underline  
actor  
User  
participant "First Class" as A participant "Second Class" as B  
participant "Last Class" as C  
User -> A: DoWork activate A  
  
-> B: Create Request activate B  
  
-> C: DoWork activate C  
--> B: WorkDone destroy C  
  
B --> A: Request Created deactivate B  
  
A --> User: Done deactivate A
```

@enduml



@startuml

```
skinparam backgroundColor #EEEBCD skinparam handwritten true
```

```

skinparam sequence { ArrowColor DeepSkyBlue ActorBorderColor
DeepSkyBlue LifeLineBorderColor blue LifeLineBackgroundColor
#A9DCDF
    ParticipantBorderColor DeepSkyBlue
    ParticipantBackgroundColor DodgerBlue ParticipantFontName Impact
    ParticipantFontSize 17 ParticipantFontColor #A9DCDF
        ActorBackgroundColor aqua ActorFontColor DeepSkyBlue
        ActorFontSize 17 ActorFontName Aapex
    }
    actor
User
participant "First Class" as A participant "Second Class" as
B participant "Last Class" as C
User -> A: DoWork activate A

-> B: Create Request activate B

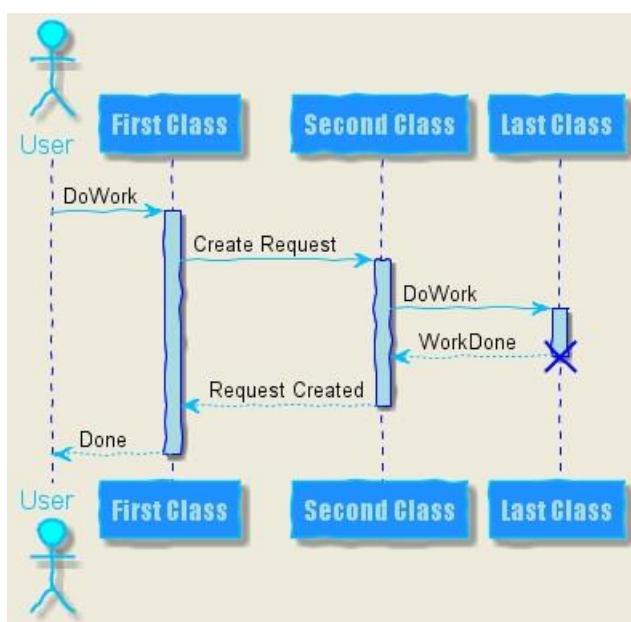
-> C: DoWork activate C
--> B: WorkDone destroy C

B --> A: Request Created deactivate B

A --> User: Done deactivate A

```

@enduml



Изменение отступов

Вы можете изменить некоторые настройки отступов

@startuml

```

skinparam ParticipantPadding 20 skinparam
BoxPadding 10

```

```

box "Fool" participant Alice1 participant
Alice2 end box

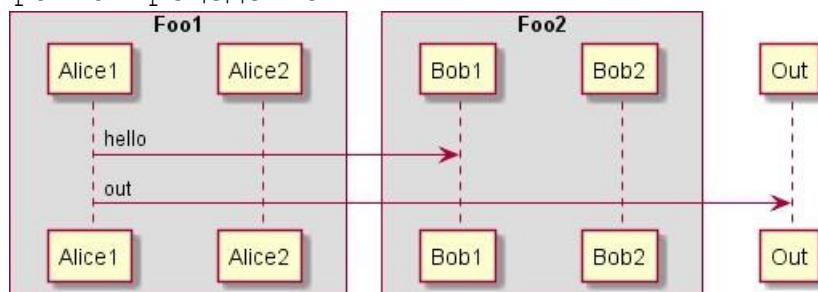
```

```

box "Foo2" participant Bob1 participant Bob2 end box
Alice1 -> Bob1 : hello Alice1 -> Out : out @enduml

```

Диаграмма прецедентов



Рассмотрим несколько примеров:

Заметьте, что Вы можете отключить тени, используя команду `skinparam shadowing false`.

Прецеденты

Прецеденты заключаются в две скобки (потому что две скобки выглядят как овал).

Вы можете использовать `usecase` для создания прецедента. также вы можете создать псевдоним, используя

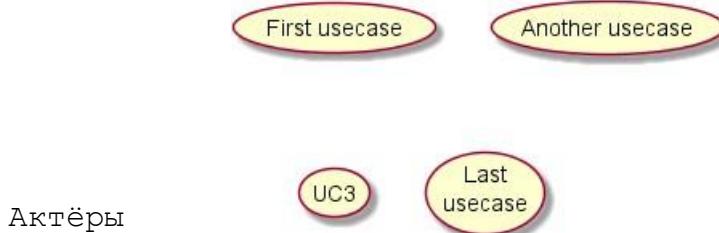
`as keyword`. Этот псевдоним будет использоваться позже во время определения связей

```
@startuml
```

```

(First usecase)
(Another usecase) as (UC2) usecase UC3 usecase
(Last\nusecase) as UC4 @enduml

```



Актеры обозначаются заключёнными между двумя точками.

Также Вы можете использовать ключевое слово `actor` для определения актёра. И вы можете создать псевдоним, используя ключевое слово `as`. Этот псевдоним будет использован позднее, при определении отношений.

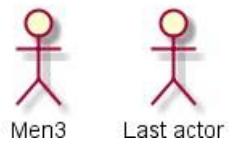
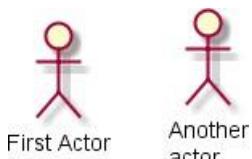
Мы увидим, что определения актеров не обязательны.

```
@startuml
```

```

:First Actor:
:Another\nactor: as Men2 actor Men3 actor
:Last actor: as Men4 @enduml

```



Описание прецедентов

Если вы хотите описание на несколько строк, можете использовать кавычки.

Вы также можете использовать следующие разделители: -- .. ==

. И вы можете вставлять заголовки внутри разделителей.

```
@startuml
usecase UC1 as "You can use
several lines to define your usecase. You can also use
separators."
```

--
Several separators are possible.

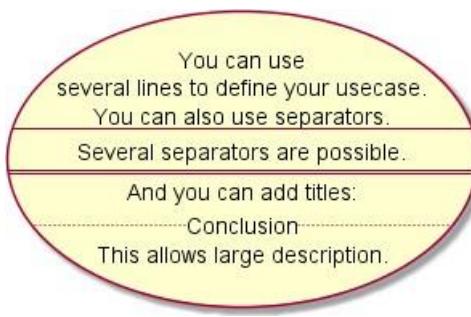
==
And you can add titles:

..Conclusion..

This allows large description." @enduml

Простой

пример



Для соединения актеров и прецедентов, используется стрелка - >.

Чем больше тире - в стрелке, тем она длиннее. Вы можете добавить метку на стрелку, добавив символ : при определении стрелки.

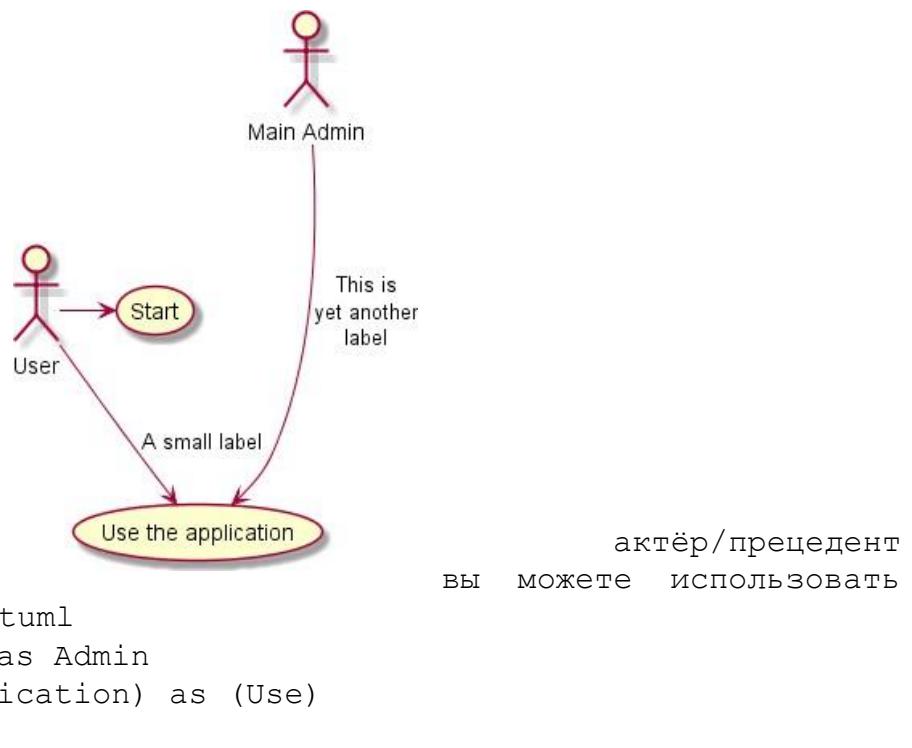
В этом примере, вы можете видеть, что User не определён ранее и используется как актёр.

```
@startuml
```

User -> (Start)

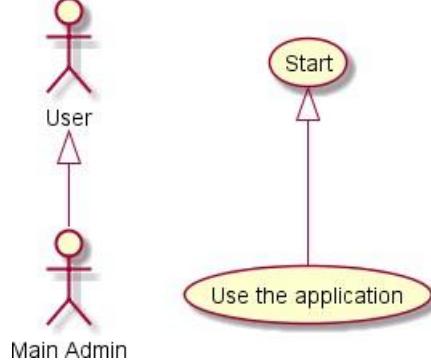
User --> (Use the application) : A small label

```
:Main Admin: ---> (Use the application) : This is\nyet another\nlabel @enduml
```



```
User <|-- Admin (Start) <|-- (Use)
```

```
@enduml
```



Использование заметок

Вы можете использовать ключевые слова note left of , note right of , note top of , note bottom of чтобы создать заметку относящуюся к одному объекту.

Заметка так же может быть создана с помощью ключевого слова note , а затем прикреплена к другому объекту используя символ ... @startuml

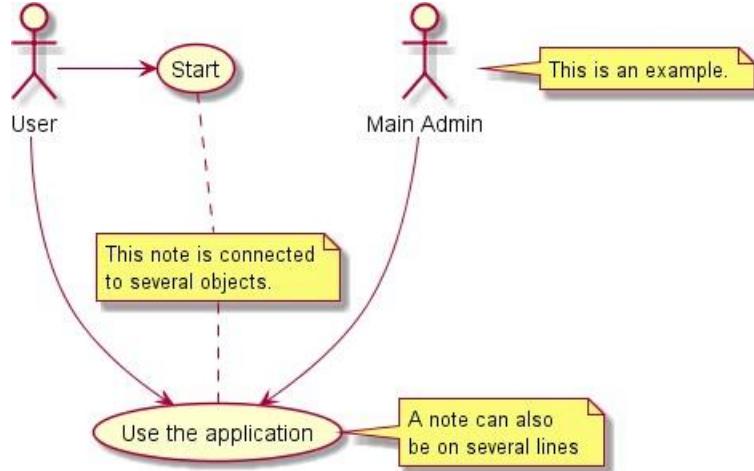
```
:Main Admin: as Admin  
(Use the application) as (Use) User -> (Start)  
User --> (Use)
```

```

Admin ---> (Use)
note right of Admin : This is an example. note right of (Use)
A note can also
be on several lines end note

note "This note is connected\nto several objects." as N2
(Start) .. N2
N2 .. (Use)
@enduml

```



2.7 Шаблоны

2 ДИАГРАММА ПРЕЦЕДЕНТОВ

Шаблоны

Вы можете добавить шаблоны когда определяете актёров и прецеденты, используя << и >>. @startuml

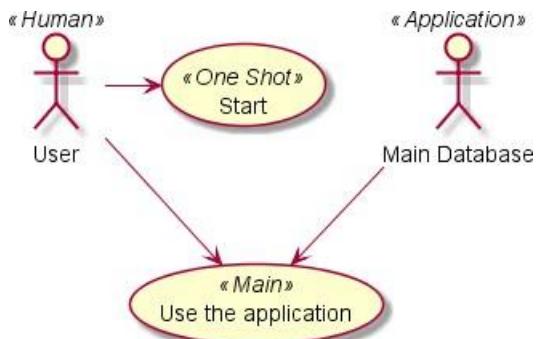
```

User << Human >>
:Main Database: as MySql << Application >> (Start) << One
Shot >>
(Use the application) as (Use) << Main >>

```

User -> (Start) User --> (Use)

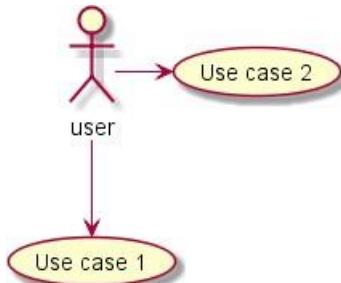
MySql --> (Use) @enduml



Смена направления стрелок

По умолчанию, связи между классами имеют два тире -- и вертикально ориентированы. Можно использовать горизонтальные связи, с помощью написание одного тире (или точки), вот так:

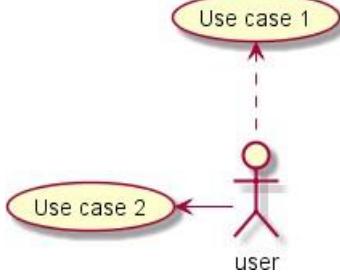
```
@startuml  
:user: --> (Use case 1)  
:user: -> (Use case 2) @enduml
```



Вы так же можете изменить направление с помощью переворачивания связи: @startuml

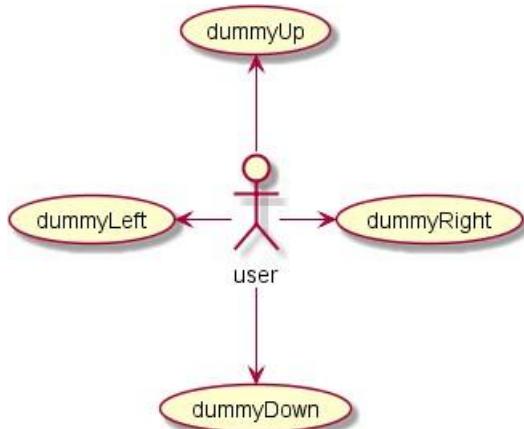
```
(Use case 1) <.. :user: (Use case 2) <- :user: @enduml
```

Так же возможно сменить направление добавляя ключевые слова



left, right, up или down внутри стрелки:

```
@startuml  
:user: -left-> (dummyLeft)  
:user: -right-> (dummyRight)  
:user: -up-> (dummyUp)  
:user: -down-> (dummyDown) @enduml
```



Вы можете записать короче, используя только первый символ названия направления (например, -d- вместо -down-) или первые два символа (-do-).

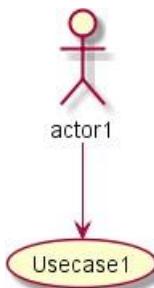
Пожалуйста, помните, что Вы не должны использовать эту функциональность без реальной необходимости:

GraphViz обычно даёт хороший результат без дополнительных настроек.

Разделение диаграмм

Ключевое слово newpage используется для разделения диаграмм на несколько страниц или изображений.

```
@startuml  
:actor1: --> (Usecase1) newpage  
:actor2: --> (Usecase2) @enduml
```



Направление слева направо

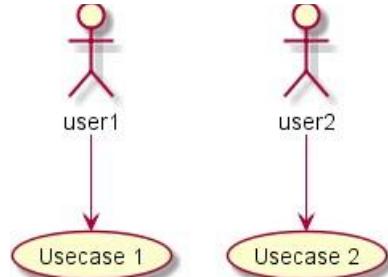
Общее поведение по умолчанию – построение диаграмм сверху вниз.

```

@startuml 'default
top to bottom direction user1 --> (Usecase 1) user2 -->
(Usecase 2)

@enduml

```



Вы можете изменить направление на слева направо используя команду left to right direction. Часто результат с таким направлением выглядит лучше.

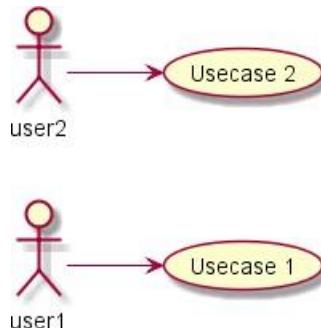
```

@startuml
left to right direction user1 --> (Usecase 1) user2 -->
(Usecase 2)

@enduml

```

`@enduml`



`Skinparam`

Вы можете использовать команду `skinparam` для изменения шрифтов и цветов диаграммы Вы можете использовать данную команду :

В определении диаграммы, как любую другую команду,
В подключенном файле,

В конфигурационном файле, указанном в командной строке в задании ANT. Вы можете задать цвет или шрифт для актёров или прецедентов с шаблонами. `@startuml skinparam handwritten true`

```
skinparam usecase { BackgroundColor DarkSeaGreen BorderColor  
DarkSlateGray
```

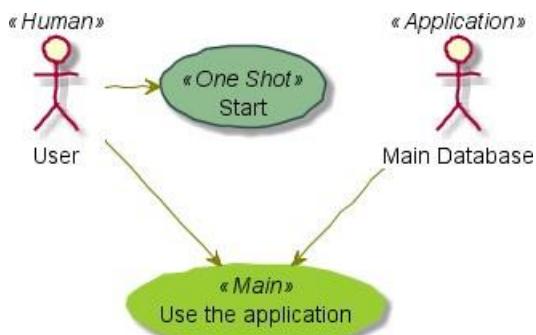
```
BackgroundColor<< Main >> YellowGreen BorderColor<< Main >>  
YellowGreen
```

```
ArrowColor Olive ActorBorderColor black ActorFontName Courier  
ActorBackgroundColor<< Human >> Gold  
}
```

```
User << Human >>  
:Main Database: as MySql << Application >> (Start) << One  
Shot >>  
(Use the application) as (Use) << Main >>
```

```
User -> (Start) User --> (Use)
```

```
MySql --> (Use) @enduml
```



Полноценный пример

```
@startuml  
left to right direction skinparam packageStyle rectangle  
actor customer actor clerk rectangle  
checkout {  
    customer -- (checkout) (checkout) .> (payment) : include  
(help) .> (checkout) : extends (checkout) -- clerk  
}  
@enduml
```

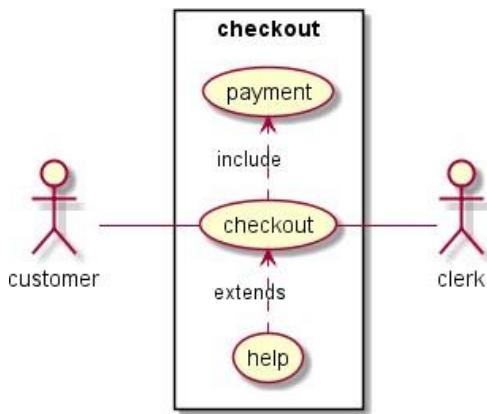


Диаграмма классов

Отношения между классами

Отношения между классами определяются с помощью следующих

Тип	Symbol	Drawing
Extension	< --	
Composition	--	
Agg	○	

символов: regation --

Можно заменить - на .., чтобы создать пунктирную линию. Зная эти правила можно нарисовать следующие изображения:

```

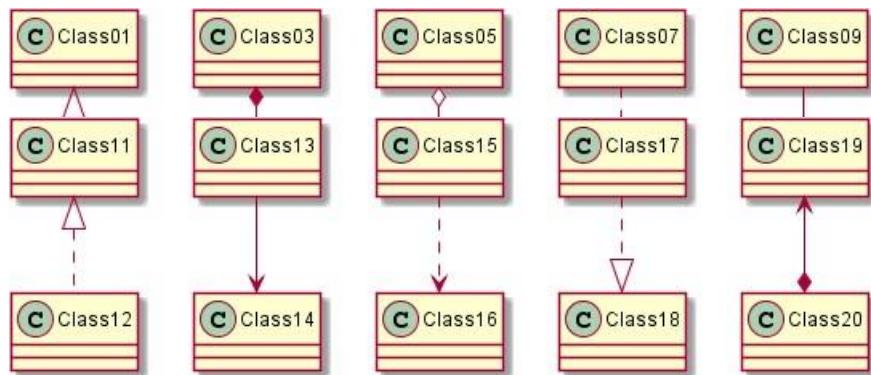
@startuml
Class01 <|.. Class02 Class03 *-- Class04
Class05 o-- Class06 Class07 ..
Class08 Class09 -- Class10 @enduml

```

```

@startuml
Class11 <|.. Class12 Class13 --> Class14 Class15 ..> Class16
Class17 ..|> Class18 Class19 <--* Class20 @enduml

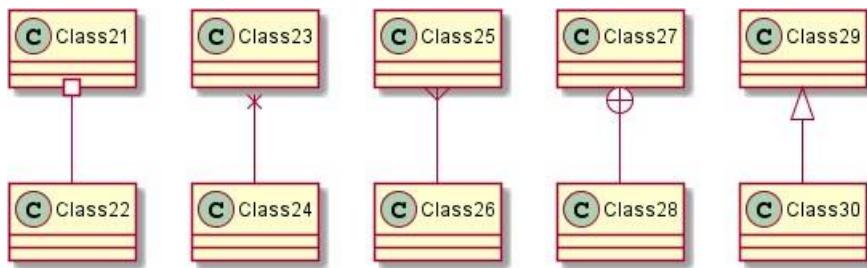
```



```

@startuml
Class21 #-- Class22 Class23 x-- Class24 Class25 }-- Class26
Class27 +-- Class28 Class29 ^-- Class30 @enduml

```



Метки на отношениях

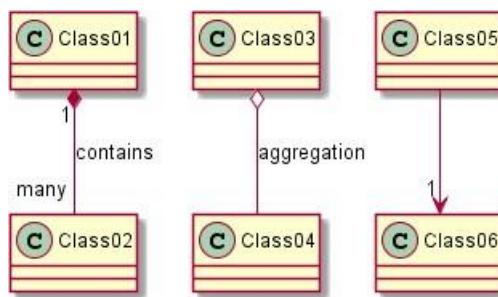
Для отношения можно добавить метку. Делается это с помощью указания символа :, после которого указывается текст метки.

Для указания количества элементов на каждой стороне отношения можно использовать двойные кавычки

"".

```
@startuml
```

```
Class01 "1" *-- "many" Class02 : contains Class03 o-- Class04
: aggregation Class05 --> "1" Class06
@enduml
```

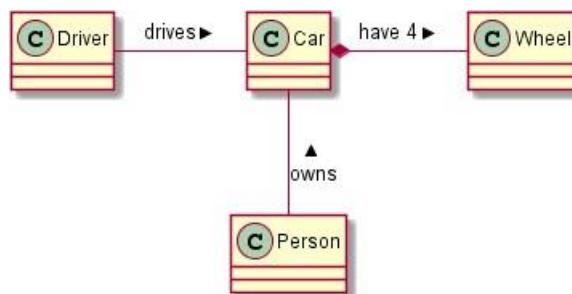


Вы можете добавить дополнительные стрелки < или > в начале или в конце метки, указывающие на использование одного из объектов другим объектом.

```
@startuml class Car
```

```
Driver -> Car : drives > Car *--> Wheel : have 4 > Car --> Person
: < owns
```

```
@enduml
```



Добавление методов

Для объявления полей и методов вы можете использовать символ `:`, после которого указывается имя поля или метода.

Для определения того, что вы указали метод или поле, система ищет скобки.

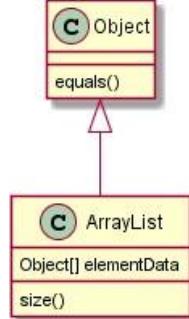
```
@startuml
```

```
Object <|-- ArrayList
```

```
Object : equals()
```

```
ArrayList : Object[] elementData ArrayList : size()
```

```
@enduml
```



Также можно группировать все поля и методы между фигурными скобками {}. Синтаксис порядка описания типа/имени довольно гибок.

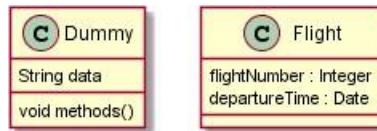
```
@startuml class Dummy {
```

```
String data void methods()
```

```
}
```

```
class Flight { flightNumber : Integer departureTime : Date
}
```

```
@enduml
```



You can use {field} and {method} modifiers to override default behaviour of the parser about fields and methods.

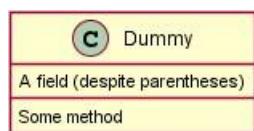
```
@startuml class Dummy {
```

```
{field} A field (despite parentheses)
```

```
{method} Some method
```

```
}
```

```
@enduml
```



Указание видимости

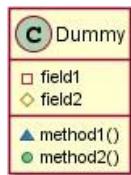
Определяя методы и поля, вы можете использовать символы указания видимости, приведённые в таблице ниже:

Character	Icon for field	Icon for method	Visibility
-	□	■	private
#			protected

~			Страница package private 105 из 105
+	○	●	public

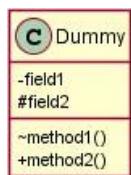
```
@startuml class Dummy {
    -field1 #field2
    ~method1()
    +method2()
}

@enduml
```



Убрать значки можно командой `skinparam classAttributeIconSize 0`:

```
@startuml skinparam classAttributeIconSize 0 class
Dummy {
    -field1 #field2
    ~method1()
    +method2()
}
```

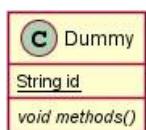


Абстрактные и статические

Вы можете определить статические или абстрактные методы и поля используя модификаторы `{static}` и `{abstract}` соответственно.

Эти модификаторы могут располагаться как в начале так и в конце строки. Вы так же можете использовать `{classifier}` как замену для `{static}`.

```
@startuml class Dummy {
    {static} String id
    {abstract} void methods()
}
@enduml
```



Расширенное тело класса

По умолчанию, методы и поля автоматически группируются PlantUML. Вы можете использовать разделители, чтобы определить собственный порядок полей и методов. Можно использовать следующие разделители:

```
-- .. == .
```

Вы также можете использовать заголовки внутри разделителей:

```
@startuml class Fool { You  
can use several lines  
.. as you want and  
group  
== things  
together.
```

You can have as many groups as you want

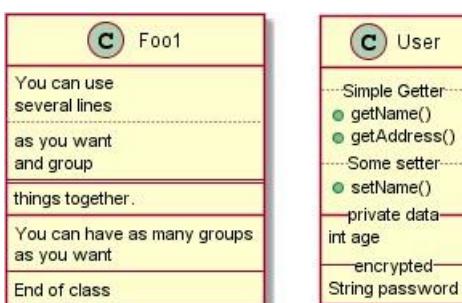
```
--
```

```
End of class
```

```
}
```

```
class User { ..  
Simple Getter ..  
+ getName() +  
getAddress() ..  
Some setter ..  
+ setName()  
    private data  
int age  
-- encrypted -- String password  
}
```

```
@enduml
```



Заметки и шаблоны

Шаблоны задаются ключевым словом `class, << и >>`.

Также вы можете создать заметку, используя ключевые слова `note left of`, '`note right of`', `note top of`, `note bottom of`'.

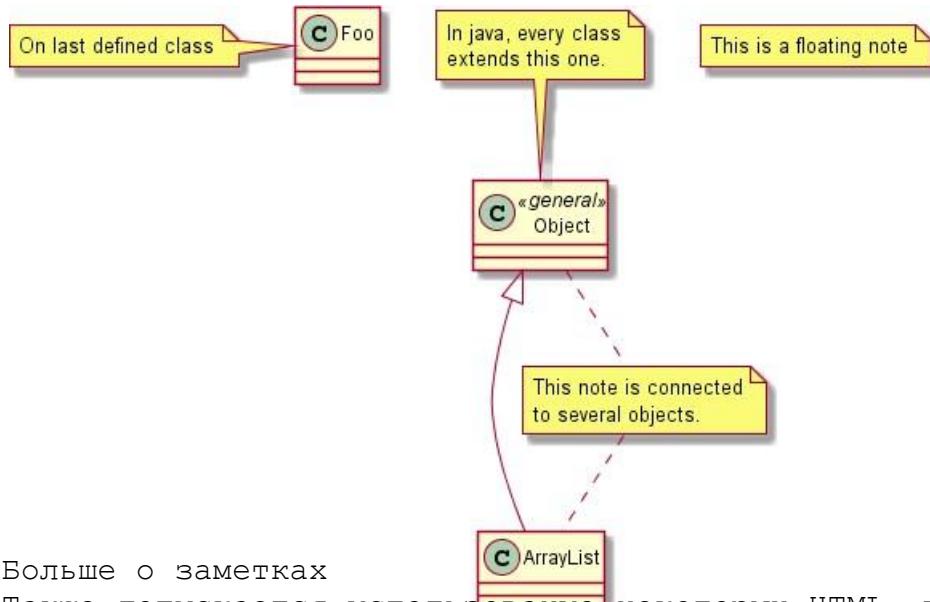
Вы также можете добавить заметку к последнему определённому классу, используя `note left`, `note right`, `note top`, `note bottom`.

Ключевым словом `note` легко создать заметку без привязи, а после, используя символ `..`, привязать её к другим объектам.

```

@startuml
    class Object << general >> Object <|--- ArrayList note top of
Object : In java, every class\nextends this one. note "This is a
floating note" as N1 note "This note is connected\nto several
objects." as N2
Object .. N2
    N2 .. ArrayList
        class
        Foo
        note left: On last defined class @enduml

```



Больше о заметках
Также допускается использование некоторых HTML -тегов, таких как:

<u>
<i>
<s>, , <strike>
 or
<color:#AAAAAA> or <color:colorName>
<size:nn> to change font size
 or <img:file>: the file must be accessible by the filesystem Заметка может быть из нескольких строк.

Можно определить заметку для класса, заданного последним, с помощью note left, note right, note top, note bottom.

```

@startuml
l class
Foo
note left: On last defined class

```

```

note top of Object
In java, <size:18>every</size> <u>class</u>
<b>extends</b>
<i>this</i> one. end note
note as
N1 This

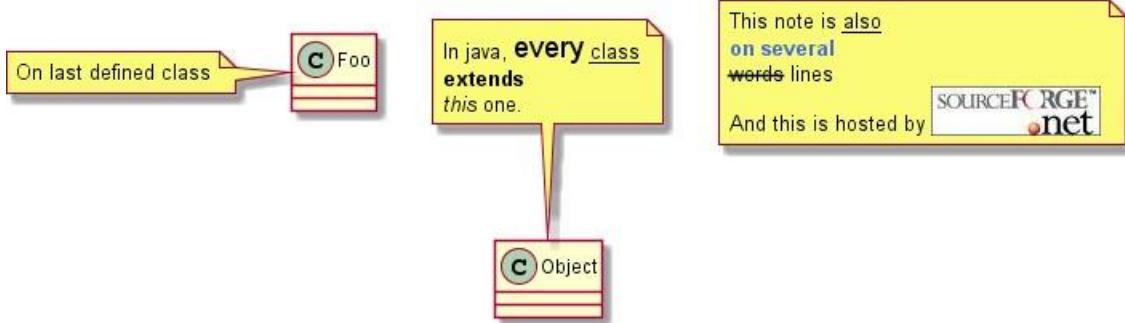
```

```

note is
<u>also</
u>
<b><color:royalBlue>on several</color>
<s>words</s> lines
And this is hosted by <img:sourceforge.jpg> end note

@enduml

```



Заметки на связях

Возможно добавить заметку на связь, сразу после определения связи, используя note on link.

Вы также можете использовать note left on link, note right on link, note top on link, note bottom on link если вы хотите изменить относительную позицию заметки с надписью.

```

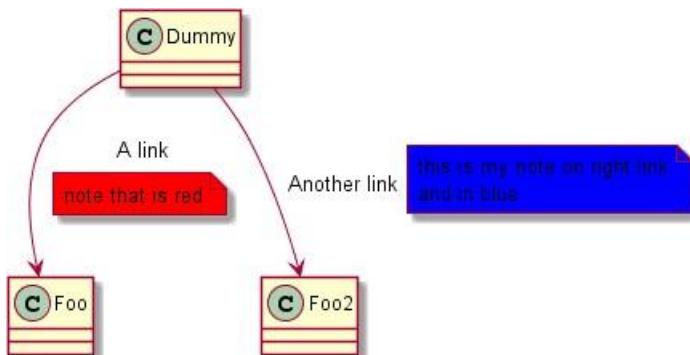
@startuml
class Dummy
Dummy --> Foo : A link
note on link #red: note that is red

```

```

Dummy --> Foo2 : Another link note right on link #blue this
is my note on right link and in blue end note @enduml

```



Абстрактные классы и интерфейсы

Вы можете определить класс как абстрактный, используя ключевые слова abstract или abstract class. Классы будут нарисованы курсивом.

Вы также можете использовать ключевые слова interface, annotation и enum. @startuml

```

abstract class AbstractList abstract Collection
interface List interface
Collection

List <|-- AbstractList
Collection <|-- Collection

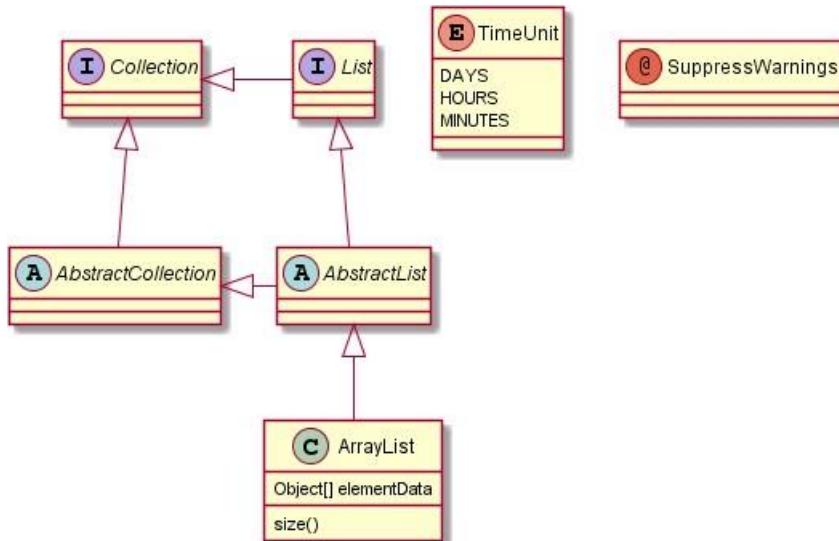
Collection <|- List AbstractCollection <|- AbstractList
AbstractList <|-- ArrayList

class ArrayList { Object[] elementData size()
}

enum TimeUnit { DAYS
HOURS MINUTES
}

annotation SuppressWarnings @enduml

```



Использование не буквенных символов

Если вы хотите использовать не буквенные символы в названии класса (или другого объекта), вы можете использовать 2 способа :

Использовать ключевое слово as в определении класса

Поставить кавычки "" вокруг имени класса

@startuml

```

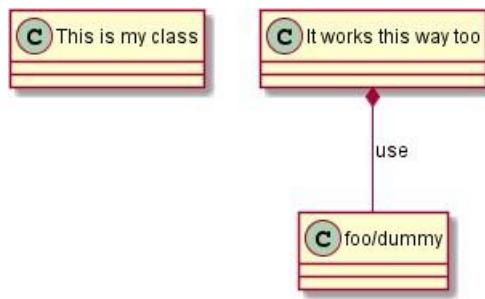
class "This is my class" as class1 class class2 as "It works
this way too"

```

```

class2 *-- "foo/dummy" : use @enduml

```



Скрытие атрибутов, методов...

Вы можете управлять видимостью классов с помощью команды `hide/show`.

Базовая команда это - `hide empty members`. Команда скроет атрибуты или методы, если они пусты. Вместо `empty members`, вы можете использовать:

`empty fields` или `empty attributes` для пустых полей, `empty methods` для пустых методов,
`fields` или `attributes`, которые скроют поля, даже если они были описаны, `methods`, которые скроют методы, даже если они были описаны, `members`, которые скроют поля и методы, даже если они были описаны, `circle` для круглых символов перед именем класса, `stereotype` для шаблона.

Вы также можете указать ключевое слово, сразу за `hide` или `show`:

`class` для всех классов,
`interface` для всех интерфейсов,
`enum` для всех перечислений,
`<<fool>>` для классов, к которым применен шаблон с помощью `fool`, имя существующего названия класса.

Для определения большого набора, состоящего из правил и исключений, можно использовать несколько команд `show/hide`.

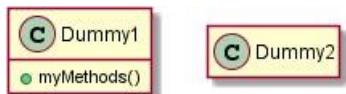
```

@startuml
class Dummy1 {
+myMethods()
}

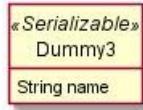
class Dummy2 {
+hiddenMethod()
}

class Dummy3 <<Serializable>> { String name
}
hide
members
hide <<Serializable>> circle show Dummy1 methods
show <<Serializable>> fields @enduml

```



Скрытие
классов



Вы также можете использовать команду `show/hide`, чтобы скрывать классы.

Это может быть полезно, если вы определяете большой подключенный файл, и если вы хотите скрыть некоторые классы после включения.

```

@startuml
class Foo1 class Foo2

Foo2 *-- Foo1 hide
Foo2 @enduml

```



Использование дженериков

Вы также можете использовать скобки < и > чтобы указать на использование дженериков в классе.

```

@startuml
class Foo<? extends Element> { int size()
}
Foo *- Element @enduml

```

Вы можете отображать эти параметры с помощью команды `skinparam`

Определение метки

Обычно, метка с буквой (C, I, E or A) используется для классов, интерфейсов, перечисления и абстрактных классов.

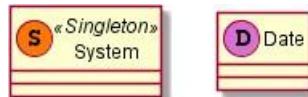
Но также вы можете использовать свою собственную метку для класса, когда создаёте шаблон, добавляя одну букву и цвет, как в этом примере:

```
@startuml
```

```

class System << (S,#FF7700) Singleton >> class Date <<
(D,orchid) >>
@enduml

```



Пакеты

Вы можете определить пакет, используя ключевое слово `package`, с возможностью объявить ещё и цвет его фона, (используя html-код цвета или его имя).

Обратите внимание, что определения пакета могут быть вложенными.

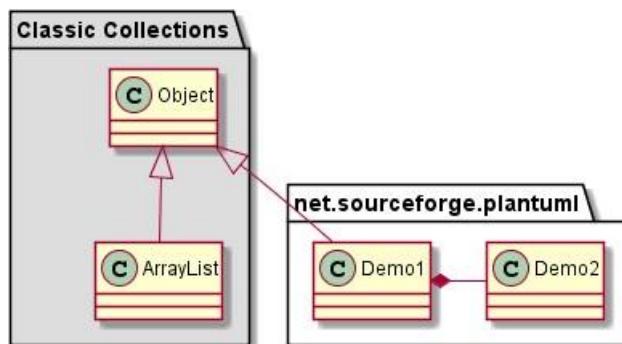
```

@startuml
package "Classic Collections" #DDDDDD { Object <|-- ArrayList
}

package net.sourceforge.plantuml { Object <|-- Demo1
Demo1 *-- Demo2
}

@enduml

```



Стили пакетов

Доступны различные стили для пакетов.

Можно задать стили по умолчанию с помощью команды: `skinparam packageStyle`, или применить шаблоны на пакет:

```

@startuml scale 750 width
package fool <<Node>> { class Class1
}

package foo2 <<Rectangle>> { class Class2
}

package foo3 <<Folder>> { class Class3
}

package foo4 <<Frame>> { class Class4
}

```

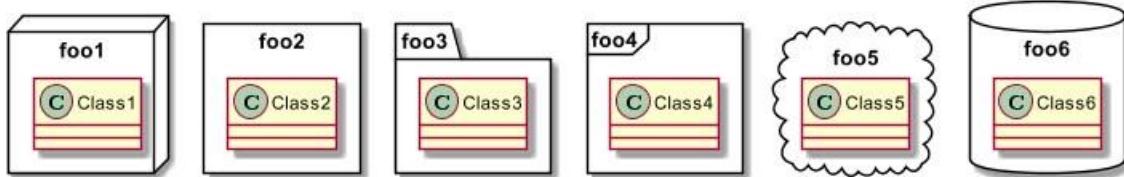
```

}

package foo5 <<Cloud>> { class Class5
}

package foo6 <<Database>> { class Class6
}
@enduml

```



Вы также можете определить связи между пакетами, как в данном примере: @startuml

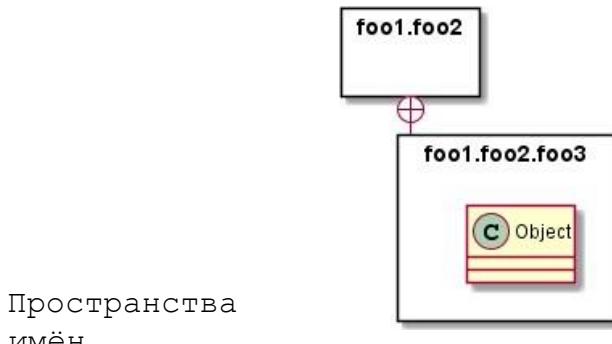
```

skinparam packageStyle rectangle package foo1..foo2 {
}

package foo1..foo2..foo3 { class Object
}

foo1..foo2 +-- foo1..foo2..foo3 @enduml

```



Пространства имен

В пакетах, имя класса является уникальным идентификатором этого класса. Это значит, что у вас не может быть двух одноименных классов в разных блоках.

В этом случае, вам следует использовать пространства имен вместо пакетов.

Вы можете ссылаться на классы из других пространств имён по их полному определению. Классы из пространства имён по умолчанию определяются ведущей точкой.

Обратите внимание, что вы не обязаны явно создавать пространство имен: полностью определенный класс автоматически попадает в правильное пространство имен.

```

@startuml class BaseClass namespace
net.dummy #DDDDDD {
    .BaseClass <|-- Person Meeting o-- Person
    .BaseClass <|- Meeting

```

```

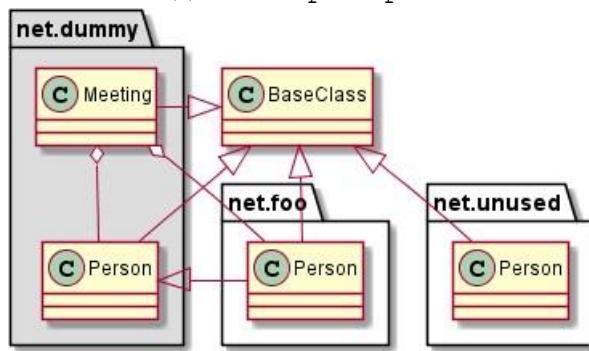
}

namespace net.foo { net.dummy.Person <|-- Person
    .BaseClass <|-- Person

    net.dummy.Meeting o-- Person
}

```

BaseClass <|-- net.unused.Person @enduml
Автоматическое создание пространств имён



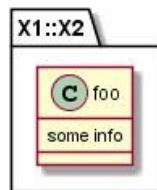
Вы также можете задать другой разделитель (не точку) используя команду : set namespaceSeparator

```

???..
@startuml
set namespaceSeparator :: class X1::X2::foo { some
info
}

@enduml

```



Вы можете отключить автоматическое создание пакетов используя команду set namespaceSeparator none.

```

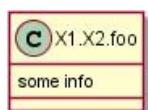
@startuml
set namespaceSeparator none class X1.X2.foo { some
info
}


```

3.19 Автоматическое создание пространств имён

3 ДИАГРАММА КЛАССОВ

@enduml

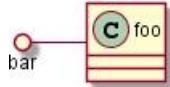


Lollipop интерфейс

Вы также можете задать lollipops интерфейсы на классах, используя следующий синтаксис:

```
bar ()- foo bar ()-- foo foo  
-() bar @startuml class foo  
bar ()- foo @enduml
```

Изменение направления стрелок

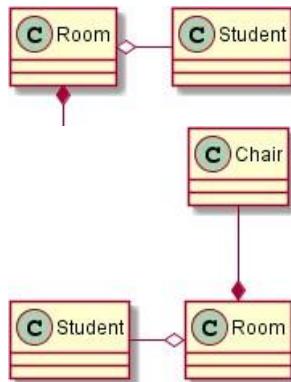


По умолчанию, связи между классами имеют два тире -- и вертикально ориентированы. Возможно создать горизонтальную связь, используя одно тире (or dot) вот так:

```
@startuml  
Room o- Student Room *-- Chair @enduml
```

Вы можете изменить направление перевернув связь:

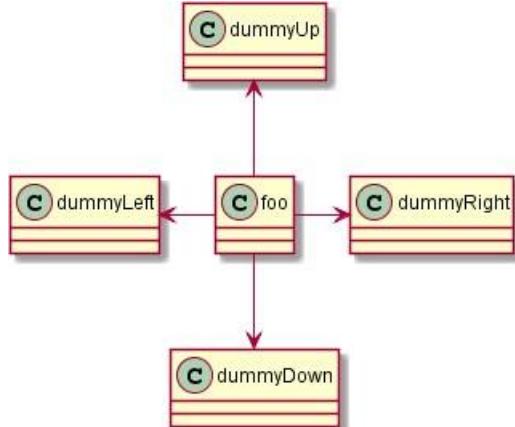
```
@startuml Student -o Room Chair --* Room @enduml
```



Также возможно изменять направление стрелок, добавляя ключевые слова left, right, up или down внутри стрелки:

```
@startuml  
foo -left-> dummyLeft foo -right-> dummyRight foo -up->  
dummyUp  
foo -down-> dummyDown @enduml
```

Вы можете укоротить запись, используя только первую букву



направления (например, -d- вместо -down-) или две первые буквы (do-).

Заметьте, что вам не стоит пользоваться этой функциональностью без особой надобности: Graphviz обычно предоставляет хорошие результаты без дополнительной настройки.

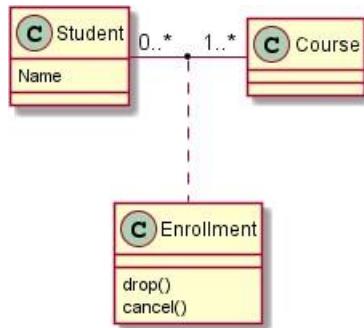
Ассоциация классов

Вы можете задать ассоциацию класса после того, как была задана связь между двумя классами, как в примере:

```
@startuml class Student {
    Name
}
Student "0..*" - "1..*" Course (Student, Course) .. Enrollment

class Enrollment { drop() cancel()
}

@enduml
```

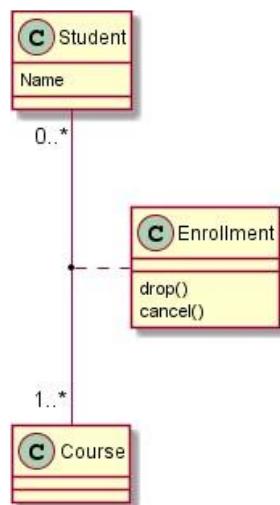


Вы можете задать это в другом направлении:

```
@startuml class Student {
    Name
}
Student "0..*" -- "1..*" Course (Student, Course) . Enrollment

class Enrollment { drop() cancel()
}

@enduml
```

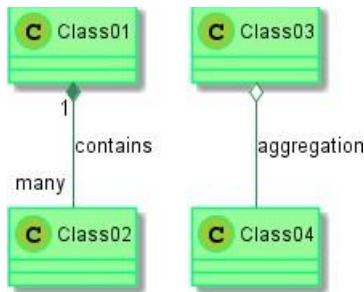


Skinparam

Вы можете использовать команду `skinparam` для изменения шрифтов и цветов диаграммы Вы можете использовать данную команду :

В определении диаграммы, как любую другую команду,
В подключенном файле,
В конфигурационном файле, указанном в командной строке в задании ANT.

```
@startuml  
skinparam class { BackgroundColor PaleGreen ArrowColor  
SeaGreen BorderColor SpringGreen  
}  
skinparam stereotypeBackgroundColor YellowGreen Class01 "1"  
*-- "many" Class02 : contains Class03 o-- Class04 : aggregation  
@enduml
```



Шаблоны со Skinparam

Вы можете задать цвет или шрифт для шаблонов классов.
@startuml

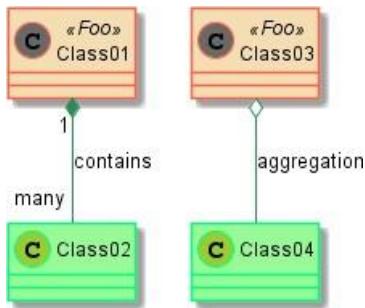
3.24 Шаблоны со Skinparam

3 ДИАГРАММА КЛАССОВ

```
skinparam class { BackgroundColor PaleGreen ArrowColor  
SeaGreen BorderColor SpringGreen BackgroundColor<<Foo>> Wheat  
BorderColor<<Foo>> Tomato  
}  
skinparam stereotypeBackgroundColor YellowGreen skinparam  
stereotypeBackgroundColor<< Foo >> DimGray  
  
Class01 <<Foo>> Class03 <<Foo>>  
Class01 "1" *-- "many" Class02 : contains Class03 o-- Class04  
: aggregation @enduml
```

Цветовой
Можно объявить
классов или
обозначения. Можно
стандартные
RGB-код.

Так же
градиента для фона,
символы
для



градиент
индивидуальный цвет для
примечаний, используя #
использовать как
названия цветов, так и

возможно использование
используя следующие
разделения пары цветов:

|,
/, \, or - в зависимости от направления градиента

Например так :

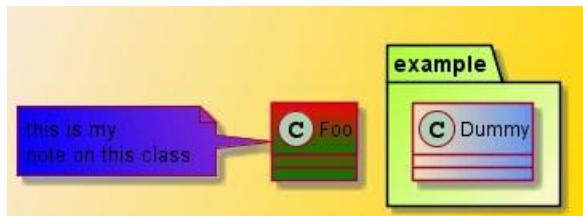
```

@startuml
skinparam backgroundColor AntiqueWhite/Gold skinparam
classBackgroundColor Wheat/CornflowerBlue

class Foo #red-green
note left of Foo #blue\9932CC this is my note
on this class end note

package example #GreenYellow/LightGoldenRodYellow { class
Dummy
}
  
```

@enduml



Помощь в расположении классов

Sometimes, the default layout is not perfect...

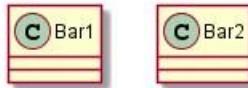
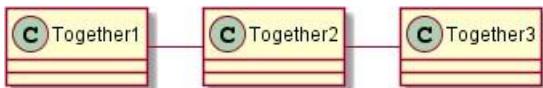
You can use together keyword to group some classes together : the layout engine will try to group them (as if they were in the same package).

You can also use hidden links to force the layout.

```

@startuml
class Bar1 class Bar2 together {
class Together1 class Together2 class Together3
}
Together1 - Together2 Together2 - Together3 Together2
[hidden]--> Bar1 Bar1 -[hidden]> Bar2
  
```

@enduml



Разделение больших файлов

Иногда могут получиться очень большие файлы изображений.

Вы можете использовать команду page (*hpages*)x(*vpages*) чтобы разделить создаваемое изображение на несколько файлов (страниц) :

- *hpages* - это задание числа горизонтальных страниц, и *vpages*
- это задание числа вертикальных страниц..

Здесь также можно использовать специфику *skinparam* настроек как цвета разделённых страниц, так и их границы (смотри пример).

```
@startuml
' Split into 4 pages page 2x2
skinparam pageMargin 10 skinparam pageExternalColor gray
skinparam pageBorderColor black

class BaseClass

namespace net.dummy #DDDDDD {
    .BaseClass <|-- Person Meeting o-- Person

    .BaseClass <|- Meeting
}

namespace net.foo { net.dummy.Person <|- Person
    .BaseClass <|-- Person

    net.dummy.Meeting o-- Person
}

BaseClass <|-- net.unused.Person @enduml
```

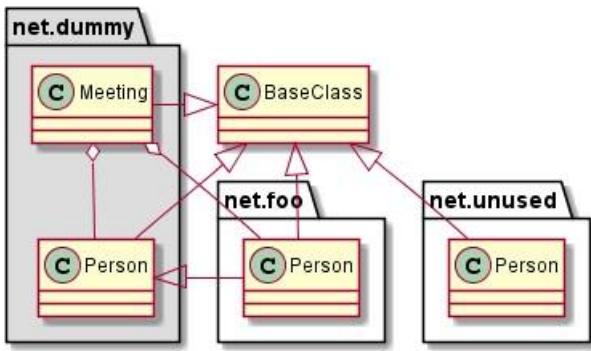


Диаграмма деятельности

Простая деятельность

Вы можете использовать (*) для начальных и конечных точек диаграммы деятельности.

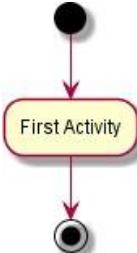
В некоторых случаях, вы можете использовать (*top) чтобы указать что начальная точка должна быть в верху диаграммы.

Используйте --> для стрелок.

@startuml

```
(*) --> "First Activity" "First Activity" --> (*)
```

@enduml



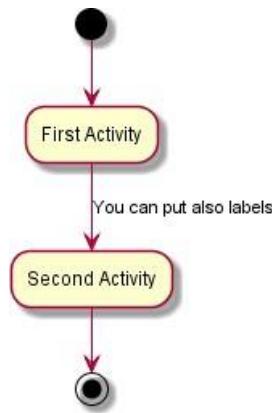
Метка на стрелках

По умолчанию, стрелка начинается с последней использованной активности.

Вы можете пометить стрелку при помощи скобок [и] сразу после определения стрелки. @startuml

```
(*) --> "First Activity"
-->[You can put also labels] "Second Activity"
--> (*)
```

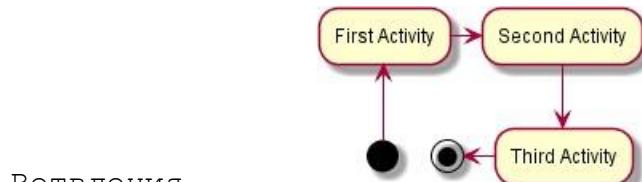
@enduml



Изменение направления стрелки

Вы можете использовать `->` для горизонтальных стрелок. Возможно задать направление стрелки используя следующий синтаксис:

- `-down->` (default arrow)
- `-right->` or `->`
- `-left->`
- `-up->` @startuml
- (*) `-up-> "First Activity"`
- `-right-> "Second Activity"`
- `--> "Third Activity"`
- `-left-> (*) @enduml`



Ветвления

Вы можете использовать ключевые слова `if/then/else` чтобы определять ветви.

```

@startuml
(*) --> "Initialization"

```

```

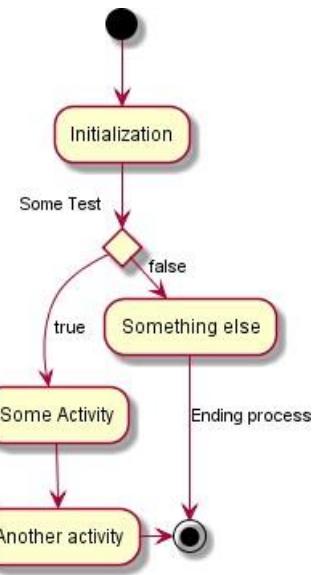
if "Some Test" then -->[true]
    "Some Activity"
--> "Another activity"
-right-> (*) else
->[false] "Something else"
-->[Ending process] (*) endif

```

```

@enduml

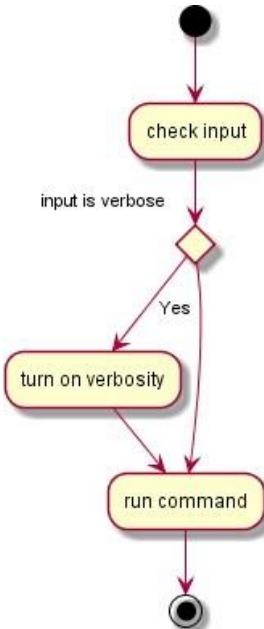
```



К сожалению, вам иногда придётся повторять ту же активность в тексте диаграммы:

```

@startuml
(*) --> "check input"
If "input is verbose" then
--> [Yes] "turn on verbosity"
--> "run command" else
--> "run command" Endif
--> (*)
@enduml
  
```



Больше о ветках

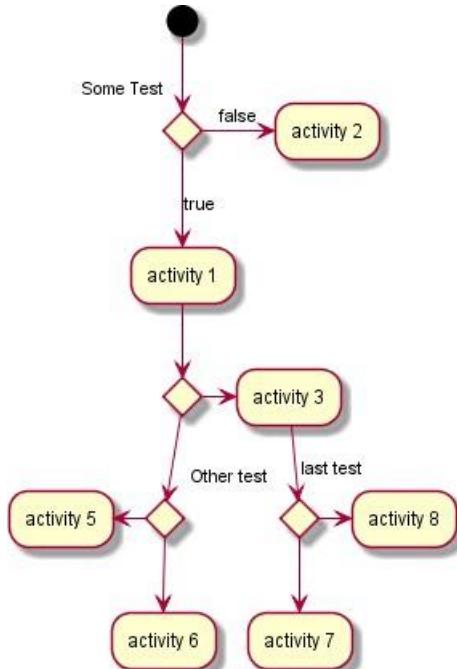
По умолчанию, ветка соединена к последней заданной активности, но возможно переопределить это и задать связь с помощью ключевого слова `if`.

Также возможно создавать вложенные ветки.

@startuml

```
(*) --> if "Some Test" then  
-->[true] "activity 1" if ""  
then -> "activity 3" as a3 else  
if "Other test" then -left->  
"activity 5" else --> "activity  
6" endif endif else  
->[false] "activity 2" endif a3  
--> if "last test" then  
--> "activity 7" else  
-> "activity 8" endif
```

@enduml



Синхронизация

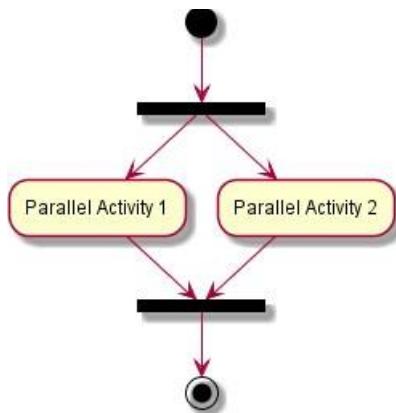
Вы можете использовать === code ===, чтобы отобразить барьеры синхронизации.

@startuml

```
(*) --> ===B1===  
--> "Parallel Activity 1"  
--> ===B2===  
  
====B1==== --> "Parallel Activity 2"
```

--> (*)

@enduml



Длинное описание активности

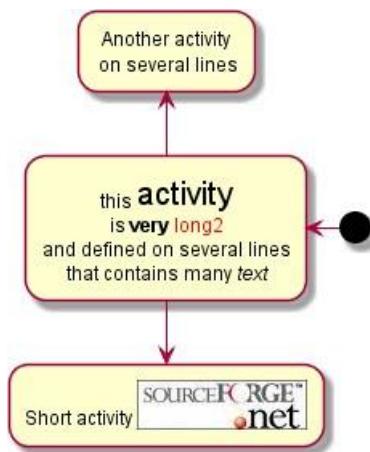
Когда вы задаёте активность, вы можете разделить её описание на несколько линий. Вы также можете добавить \n в описание.

Вы также можете задать короткий код активности в помощь ключевого слова as. Этот код может быть использован позже в описании диаграммы.

```
@startuml
(*) -left-> "this <size:20>activity</size> is <b>very</b>
<color:red>long2</color> and defined on several lines that
contains many <i>text</i>" as A1

-up-> "Another activity\n on several lines"

A1 --> "Short activity <img:sourceforge.jpg>" @enduml
```



Заметки

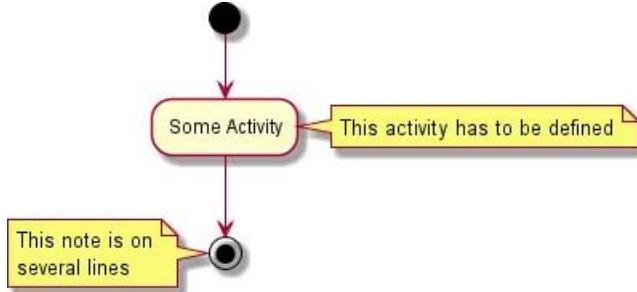
Вы можете добавить заметки к активности используя команды note left, note right, note top or note bottom, Сразу после описания активности, к которой вы хотите прикрепить заметку.

Если вы хотите прикрепить заметку к точке начала, задайте метку в самом начале описания диаграммы. Вы также можете создать заметку на нескольких линиях, используя ключевое слово endnote.

```
@startuml
```

```
(*) --> "Some Activity"  
note right: This activity has to be defined "Some Activity"  
--> (*) note  
left  
This note is on several lines end note
```

```
@enduml
```



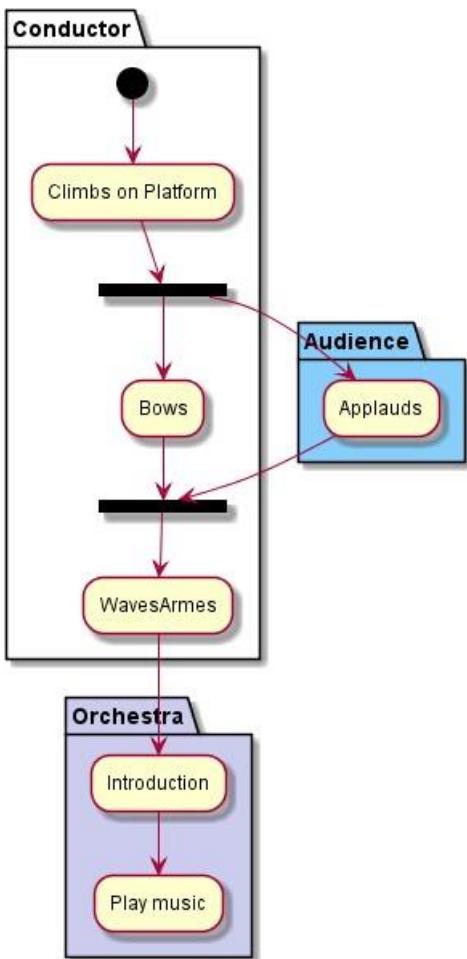
Разделы

Вы можете задать раздел используя ключевое слово partition, и опционально задать цвет фона для своего раздела (Используя код цвета html или название цвета)

Когда вы задаёте активность, они автоматически попадают в последнюю заданную активность. Вы можете закрыть раздел используя закрывающую скобку }.

```
@startuml
```

```
partition Conductor { (*) --> "Climbs on Platform"  
--> === S1 ===  
--> Bows  
}  
  
partition Audience #LightSkyBlue {  
=== S1 === --> Applauds  
}  
  
partition Conductor { Bows --> === S2 ===  
--> WavesArmes Applauds --> === S2 ===  
}  
  
partition Orchestra #CCCCEE { WavesArmes --> Introduction  
--> "Play music"  
}  
  
@enduml
```



Skinparam

Вы можете использовать команду `skinparam` чтобы изменить цвет и шрифт рисования. Вы можете использовать команду :

В определении диаграммы, как любую другую команду,
В подключаемом файле,
В конфигурационном файле, подставленный в командной строке
ANT задания. Вы можете задать определённый цвет и шрифт для
активностей с шаблоном. `@startuml skinparam backgroundColor #AAFFFF skinparam activity {`

```

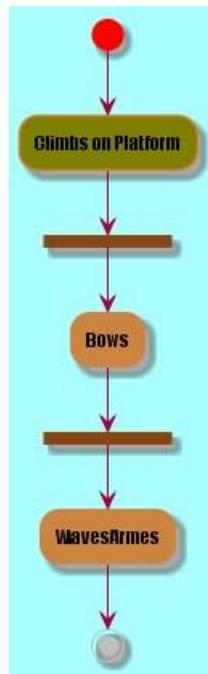
StartColor red BarColor SaddleBrown EndColor Silver
BackgroundColor Peru
BackgroundColor<< Begin >> Olive BorderColor Peru
FontName Impact
}

```

```

(*) --> "Climbs on Platform" << Begin >>
--> === S1 === -->
Bows
--> === S2 ===
--> WavesArmes
--> (*)
@enduml

```



Восьмиугольник

Вы можете изменить форму активностей на восьмиугольник, используя команду skinparam activityShape octagon.

```

@startuml
'Default is skinparam activityShape roundBox skinparam
activityShape octagon

(*) --> "First Activity" "First Activity" --> (*)

@enduml

```



Полноценный пример

```

@startuml
title Servlet Container
(*) --> "ClickServlet.handleRequest()"
--> "new Page"

if "Page.onSecurityCheck" then
->[true] "Page.onInit()"
if "isForward?" then ->[no]
"Process controls"

```

```

if "continue processing?"
then -->[yes] ===RENDERING===
else
-->[no] ===REDIRECT_CHECK==> endif

else
-->[yes] ===RENDERING==> endif

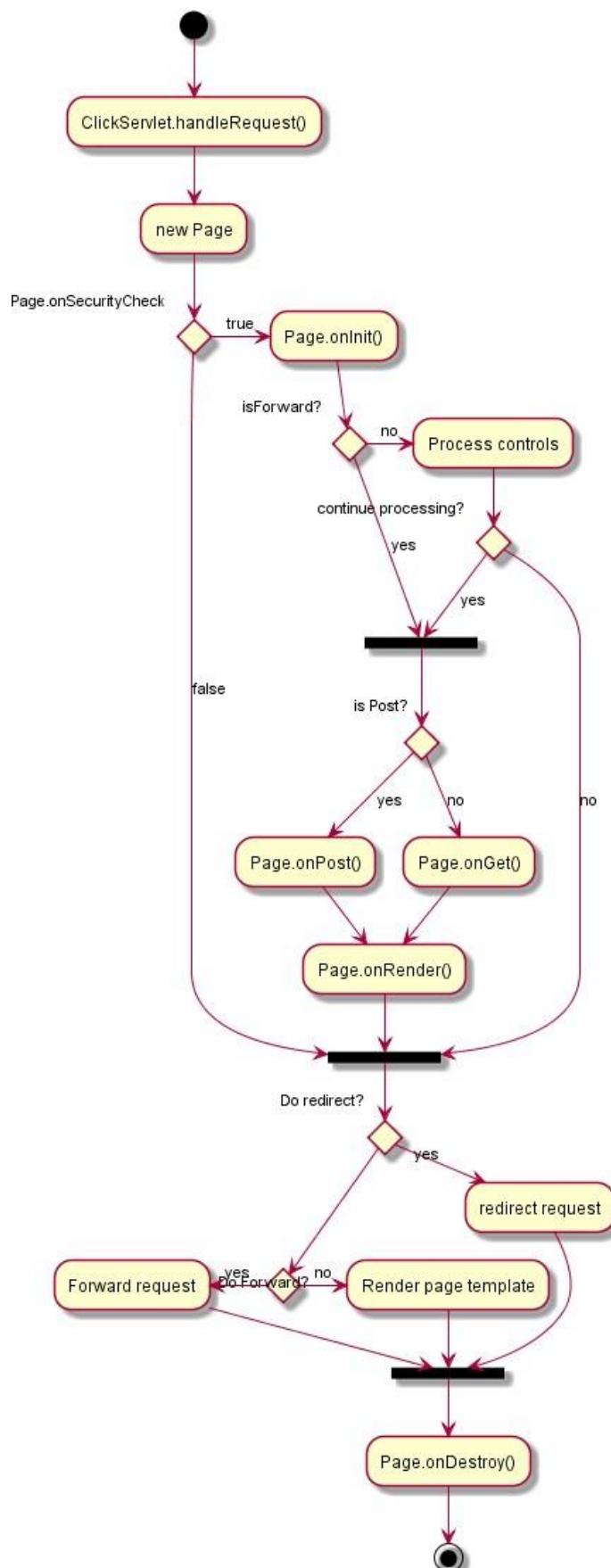
if "is Post?" then -->[yes]
"Page.onPost()"
--> "Page.onRender()" as
render -->
====REDIRECT_CHECK==> else
-->[no] "Page.onGet()"
--> render
endif else
-->[false] ===REDIRECT_CHECK==> endif
if "Do redirect?" then -
>[yes] "redirect request" -->
==BEFORE_DESTROY==> else if
"Do Forward?" then -left-
>[yes] "Forward request" -->
==BEFORE_DESTROY==> else
-right->[no] "Render page
template" --> ==BEFORE_DESTROY==>
endif endif

--> "Page.onDestroy()"
-->(*)

@enduml

```

Servlet Container





Практическое задание

Работа с MS SQL Server

Создание базы данных

Базу данных часто отождествляют с набором таблиц, которые хранят данные. Но это не совсем так. Лучше сказать, что база данных представляет хранилище объектов. Основные из них:

- Таблицы: хранят собственно данные
- Представления (Views): выражения языка SQL, которые возвращают набор данных в виде таблицы
- Хранимые процедуры: выполняют код на языке SQL по отношению к данным к БД (например, получает данные или изменяет их)
- Функции: также код SQL, который выполняет определенную задачу

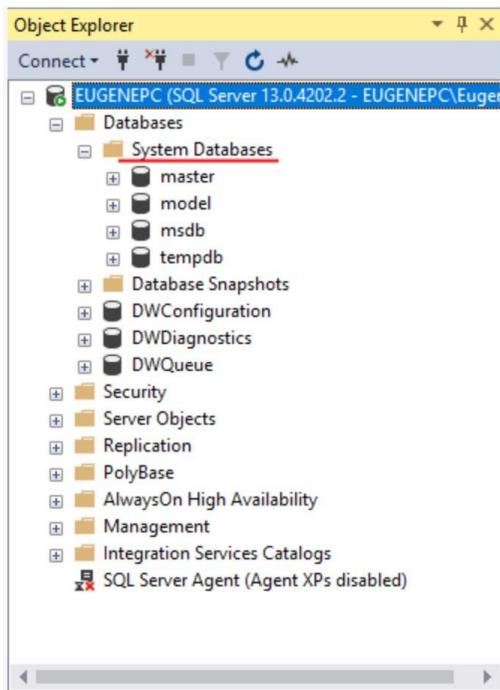
В SQL Server используется два типа баз данных: системные и пользовательские. Системные базы данных необходимы серверу SQL для корректной работы. А пользовательские базы данных создаются пользователями сервера и могут хранить любую произвольную информацию. Их можно изменять и удалять, создавать заново.

Системные базы данных

В MS SQL Server по умолчанию создается четыре системных баз данных:

- master: эта главная база данных сервера, в случае ее отсутствия или повреждения сервер не сможет работать. Она хранит все используемые логины пользователей сервера, их роли, различные конфигурационные настройки, имена и информацию о базах данных, которые хранятся на сервере, а также ряд другой информации.
- model: эта база данных представляет шаблон, на основе которого создаются другие базы данных. То есть когда мы создаем через SSMS свою БД, она создается как копия базы model.
- msdb: хранит информацию о работе, выполняемой таким компонентом как планировщик SQL. Также она хранит информацию о бэкапах баз данных.
- tempdb: эта база данных используется как хранилище для временных объектов. Она заново пересоздается при каждом запуске сервера.

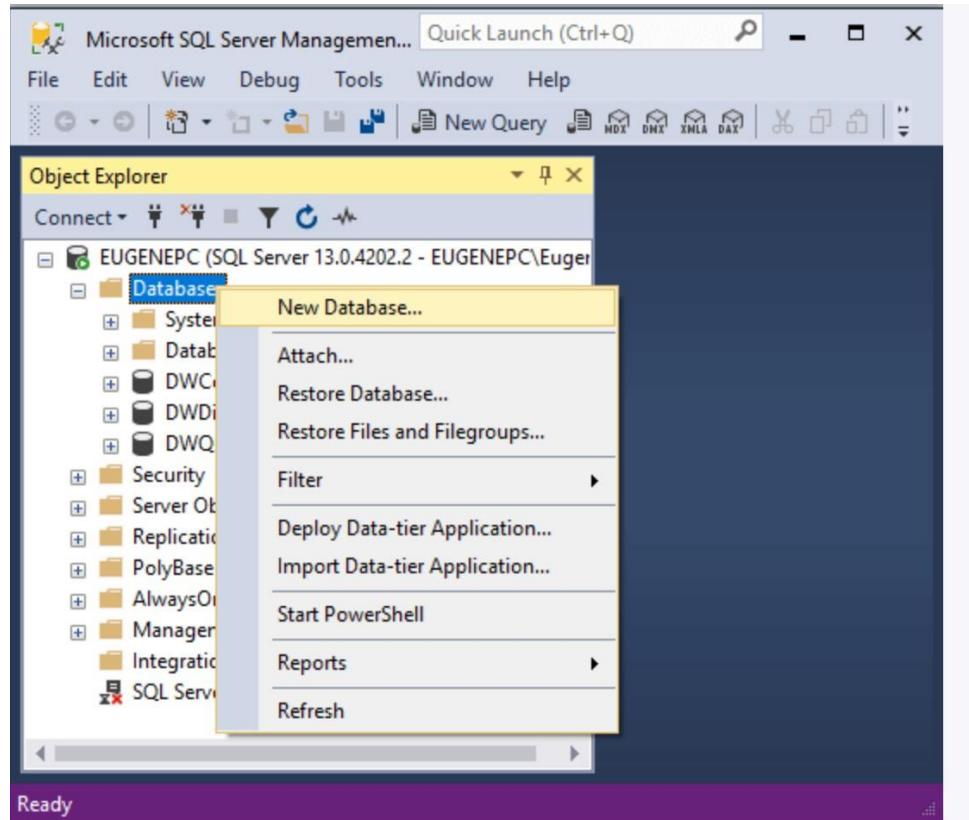
Все эти базы можно увидеть через SQL Server Management Studio в узле Databases -> System Databases:



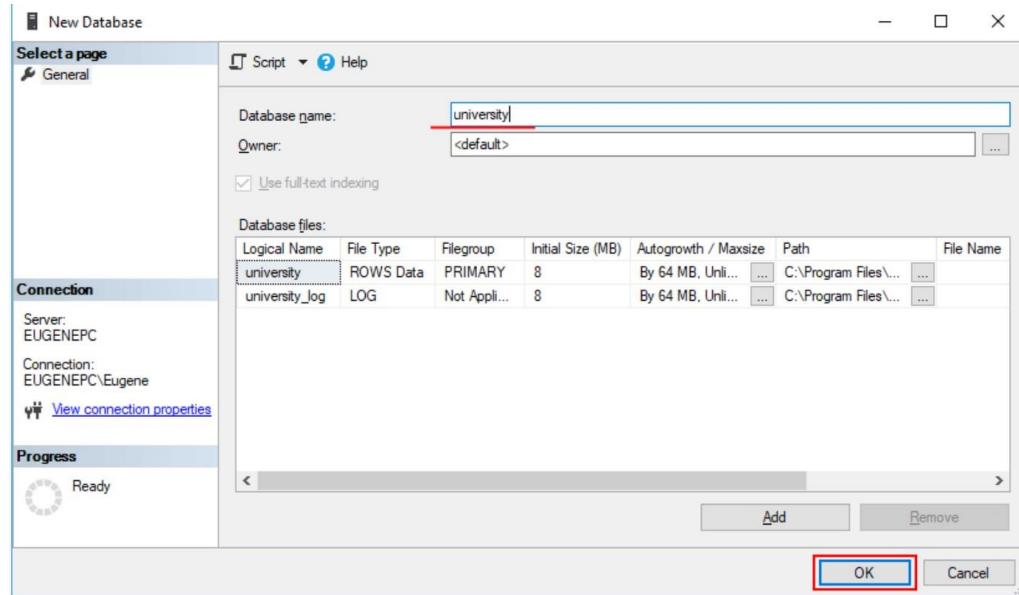
Эти базы данных не следует изменять.

Создание базы данных в SQL Management Studio

Теперь создадим свою базу данных. Для этого мы можем использовать скрипт на языке SQL, либо все сделать с помощью графических средств в SQL Management Studio. В данном случае мы выберем второй способ. Для этого откроем SQL Server Management Studio и нажмем правой кнопкой мыши на узел Databases. Затем в появившемся контекстном меню выберем пункт New Database:



После этого нам открывается окно для создания базы данных:



B

поле Database необходимо ввести название новой БД. Пусть у нас база данных называется university.

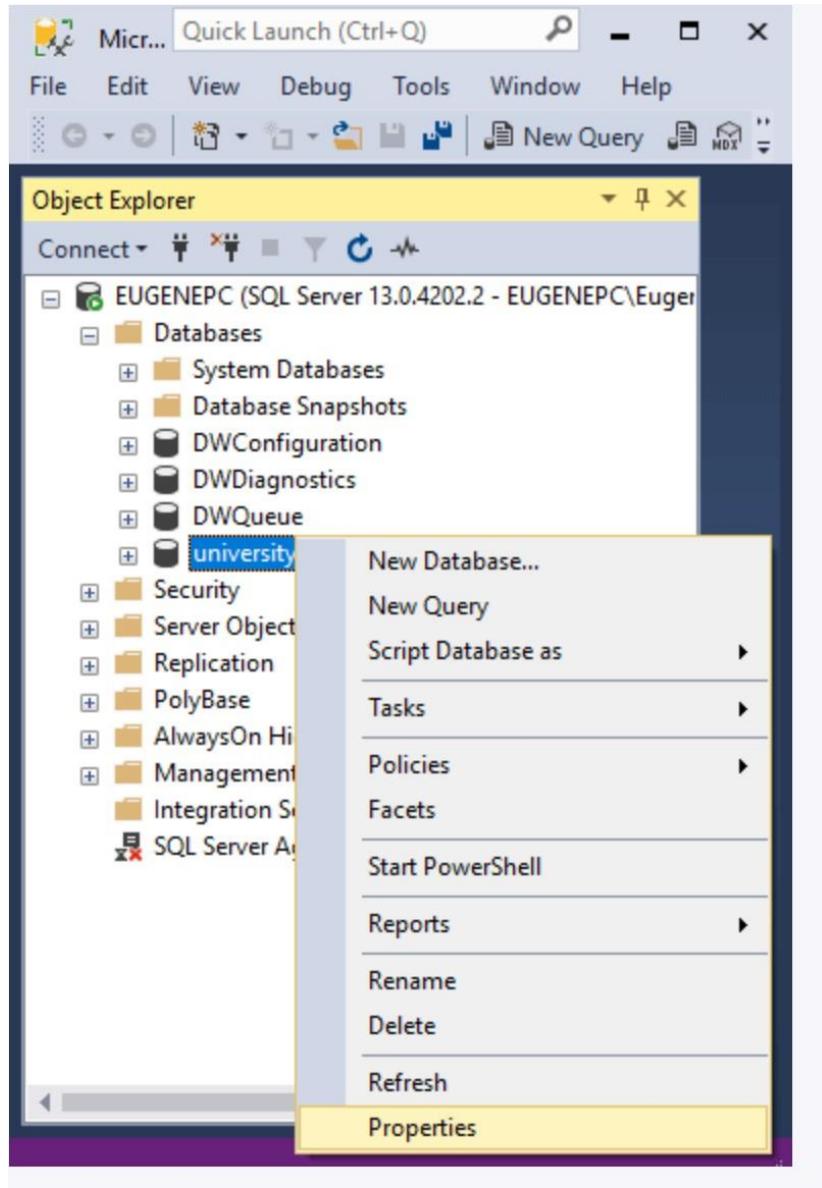
Следующее поле Owner задает владельца базы данных. По умолчанию оно имеет значение <default>, то есть владельцем будет тот, кто создает эту базу данных. Оставим это поле без изменений.

Далее идет таблица для установки общих настроек базы данных. Она содержит две строки – первая для установки настроек для главного файла, где будут храниться данные, и вторая строка для конфигурации файла логгирования. В частности, мы можем установить следующие настройки:

- Logical Name: логическое имя, которое присваивается файлу базы данных.
- File Type: есть несколько типов файлов, но, как правило, основная работа ведется с файлами данных (ROWS Data) и файлом лога (LOG)
- Filegroup: обозначает группу файлов. Группа файлов может хранить множество файлов и может использоваться для разбиения базы данных на части для размещения в разных местах.
- Initial Size (MB): устанавливает начальный размер файлов при создании (фактический размер может отличаться от этого значения).
- Autogrowth/Maxsize: при достижении базой данных начального размера SQL Server использует это значение для увеличения файла.
- Path: каталог, где будут храниться базы данных.
- File Name: непосредственное имя физического файла. Если оно не указано, то применяется логическое имя.

После ввода названия базы данных нажмем на кнопку ОК, и БД будет создана.

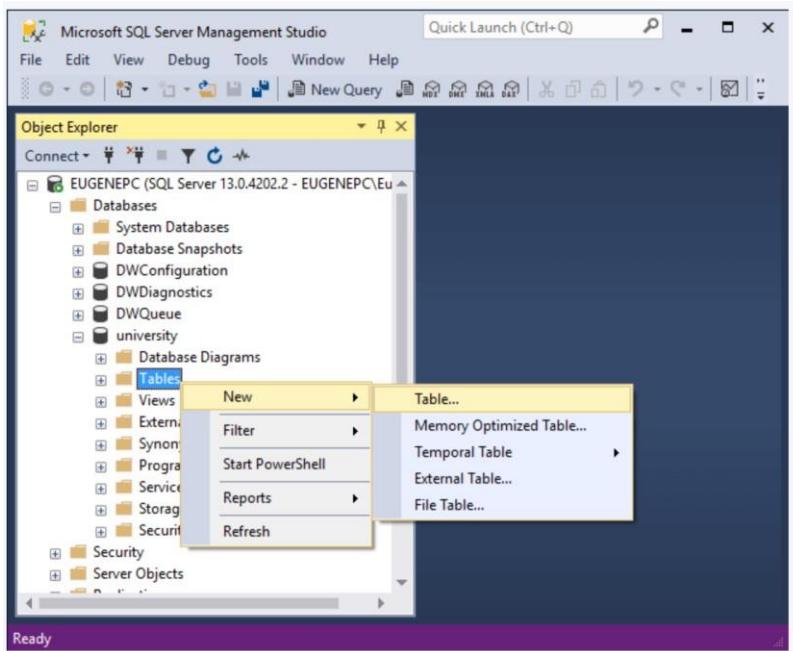
После этого она появится среди баз данных сервера. Если эта БД впоследствии не потребуется, то ее можно удалить, нажав на нее правой кнопкой мыши и выбрав в контекстном меню пункт Delete:



Ключевым объектом в базе данных являются таблицы. Таблицы состоят из строк и столбцов. Столбцы определяют тип информации, которая хранится, а строки содержат значения для этих столбцов.

Нами была создана база данных university. Теперь определим в ней первую таблицу. Опять же для создания таблицы в SQL Server Management Studio можно применить скрипт на языке SQL, либо воспользоваться графическим дизайнером. В данном случае выберем второе.

Для этого раскроем узел базы данных university в SQL Server Management Studio, нажмем на его подузел Tables правой кнопкой мыши и далее в контекстном меню выберем New -> Table



После этого нам откроется дизайнер таблицы. В центральной части в таблице необходимо ввести данные о столбцах таблицы. Дизайнер содержит три поля:

Column Name: имя столбца

Data Type: тип данных столбца. Тип данных определяет, какие данные могут храниться в этом столбце. Например, если столбец представляет числовой тип, то он может хранить только числа.

Allow Nulls: может ли отсутствовать значение у столбца, то есть может ли он быть пустым

Column Name	Data Type	Allow Nulls
Id	int	<input checked="" type="checkbox"/>
FirstName	nvarchar(50)	<input type="checkbox"/>
LastName	nvarchar(50)	<input type="checkbox"/>
Year	int	<input type="checkbox"/>

Допустим, нам надо создать таблицу с данными учащихся в учебном заведении. Для этого в дизайнере таблицы четыре столбца: Id,

`FirstName`, `LastName` и `Year`, которые будут представлять соответственно уникальный идентификатор пользователя, его имя, фамилию и год рождения. У первого и четвертого столбца надо указать тип `int` (то есть целочисленный), а у столбцов `FirstName` и `LastName` – тип `nvarchar(50)` (строковый).

Соответствие типов данных Microsoft Access и Microsoft SQL

№	Тип данных Microsoft Access	Тип данных Microsoft SQL	Описание типа данных Microsoft SQL
1	Текстовый	<code>nvarchar</code>	Тип данных для хранения текста до 4000 символов
2	Поле МЕМО	<code>ntext</code>	Тип данных для хранения символов в кодировке Unicode до 1 073 741 823 символов
3	Числовой	<code>int</code>	Численные значения (целые) в диапазоне от -2 147 483 648 до +2 147 483 647
4	Дата/время	<code>smalldatetime</code>	Дата и время от 1 января 1900 г. до 6 июня 2079 года с точностью до одной минуты
5	Денежный	<code>money</code>	Денежный тип данных, значения которого лежат в диапазоне от -922 337 203 685 477.5808 до +922 337 203 685 477.5807, с точностью до одной десятитысячной
6	Счетчик	<code>int</code>	См. пункт 3
7	Логический	<code>bit</code>	Переменная, способная принимать только два значения - 0 или 1
8	Поле объекта OLE	<code>image</code>	Переменная для хранения массива байтов от 0 до 2 147 483 647 байт
9	Гиперссылка	<code>ntext</code>	См. пункт 2
10	Мастер подстановок	<code>nvarchar</code>	См. пункт 1

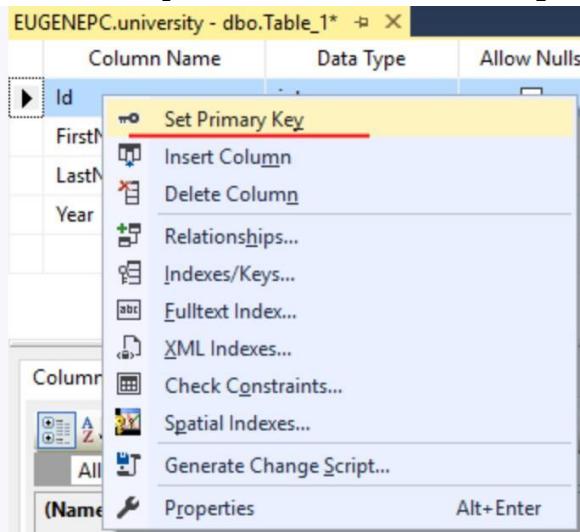
Затем в окне `Properties`, которая содержит свойства таблицы, в поле `Name` надо ввести имя таблицы – `Students`, а в поле `Identity` ввести `Id`, то есть тем самым указывая, что столбец `Id` будет идентификатором.

Имя таблицы должно быть уникальным в рамках базы данных. Как правило, название таблицы отражает название сущности, которая в ней хранится. Например, мы хотим сохранить студентов, поэтому таблица

называется Students (слово студент во множественном числе на английском языке). Существуют разные мнения по поводу того, стоит использовать название сущности в единственном или множественном числе (Student или Students). В данном случае вопрос наименования таблицы всецело ложится на разработчика базы данных.

И в конце нам надо отметить, что столбец Id будет выполнять роль первичного ключа (primary key). Первичный ключ уникально идентифицирует каждую строку. В роли первичного ключа может выступать один столбец, а может и несколько.

Для установки первичного ключа нажмем на столбец Id правой кнопкой мыши и в появившемся меню выберем пункт Set Primary Key.

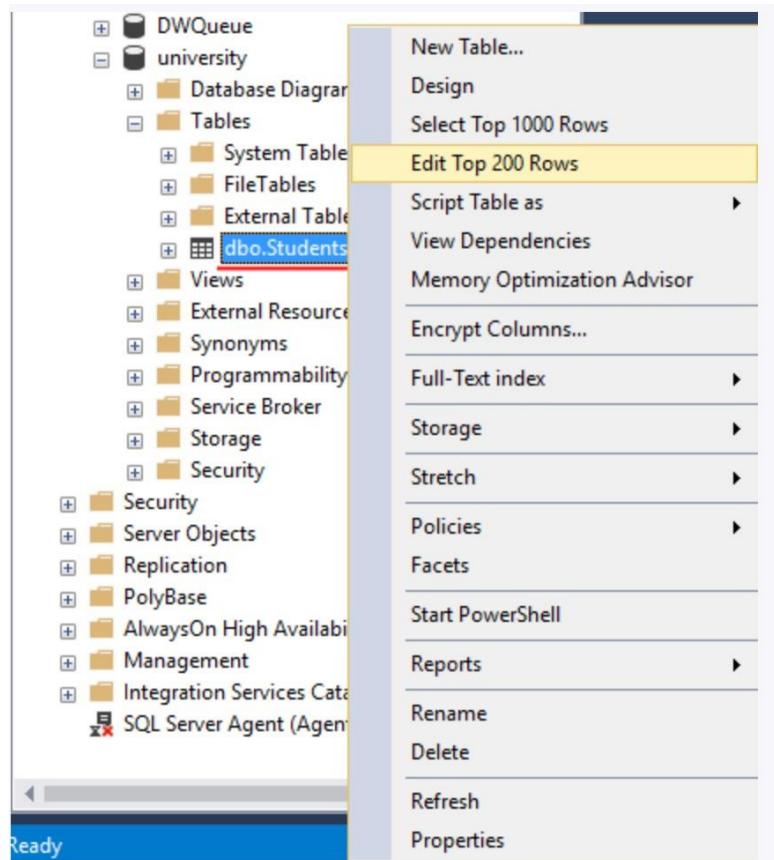


После этого напротив поля Id должен появиться золотой ключик. Этот ключик будет указывать, что столбец Id будет выполнять роль первичного ключа.

И после сохранения в базе данных university появится таблица Students:

Мы можем заметить, что название таблицы на самом деле начинается с префикса dbo. Этот префикс представляет схему. Схема определяет контейнер, который хранит объекты. То есть схема логически разграничивает базы данных. Если схема явным образом не указывается при создании объекта, то объект принадлежит схеме по умолчанию – схеме dbo.

Нажмем правой кнопкой мыши на название таблицы, и нам отобразится контекстное меню с опциями:



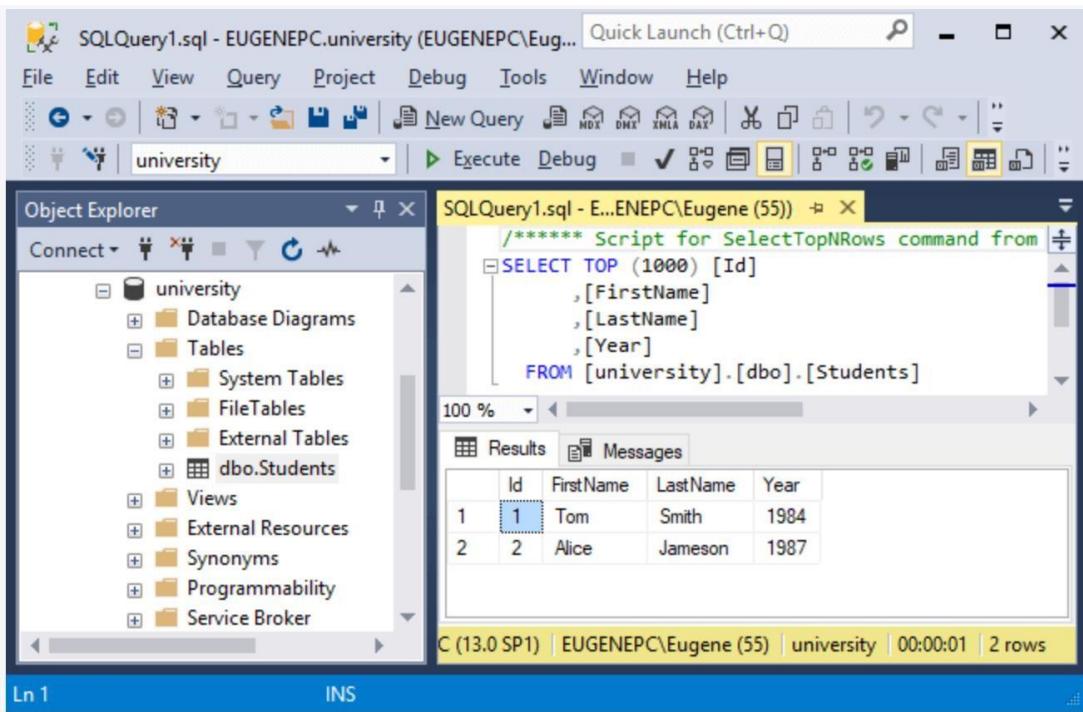
С помощью этих опций можно управлять таблицей. Так, опция Delete позволяет удалить таблицу. Опция Design откроет окно дизайнера таблицы, где мы можем при необходимости внести изменения в ее структуру.

Для добавления начальных данных можно выбрать опцию Edit Top 200 Rows. Она открывает в виде таблицы 200 первых строк и позволяет их изменить. Но так как у нас таблица только создана, то естественно в ней будет никаких данных. Введем пару строк – пару студентов, указав необходимые данные для столбцов:

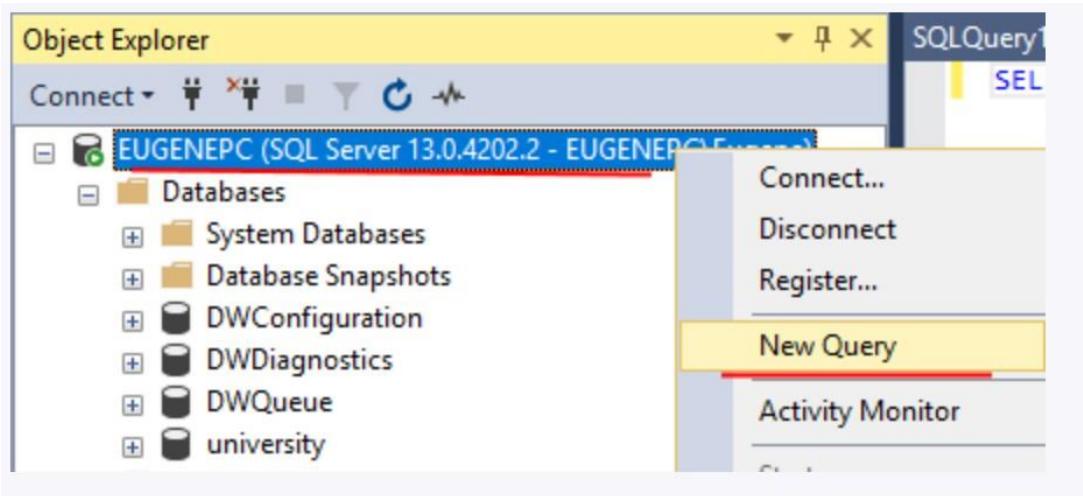
	Id	FirstName	LastName	Year
*	1	Tom	Smith	1984
*	2	Alice	Jameson	1987
*	NULL	NULL	NULL	NULL

данном случае мы добавили две строки.

Затем опять же по клику на таблицу правой кнопкой мыши мы можем выбрать в контекстном меню пункт Select To 1000 Rows, и будет запущен скрипт, который отобразит первые 1000 строк из таблицы:



Теперь определим и выполним первый SQL-запрос. Для этого откроем SQL Management Studio, нажмем правой кнопкой мыши на элемент самого верхнего уровня в Object Explorer (название сервера) и в появившемся контекстном меню выберем пункт New Query:



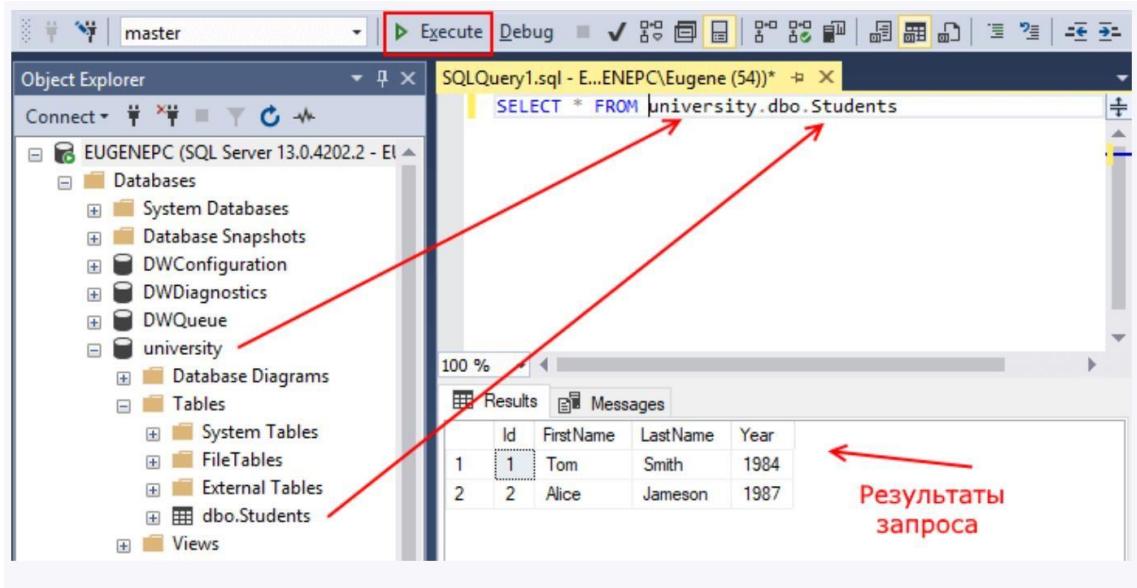
После этого в центральной части программы откроется окно для ввода команд языка SQL.

Выполним запрос к таблице, которая была создана в прошлой теме, в частности, получим все данные из нее. База данных у нас называется university, а таблица - dbo.Students, поэтому для получения данных из таблицы введем следующий запрос:

```
1      SELECT * FROM university.dbo.Students
```

Оператор SELECT позволяет выбирать данные. FROM указывает источник, откуда брать данные. Фактически этим запросом мы говорим "ВЫБРАТЬ все

Из таблицы university.dbo.Students". Стоит отметить, что для названия таблицы используется полный ее путь с указанием базы данных и схемы.



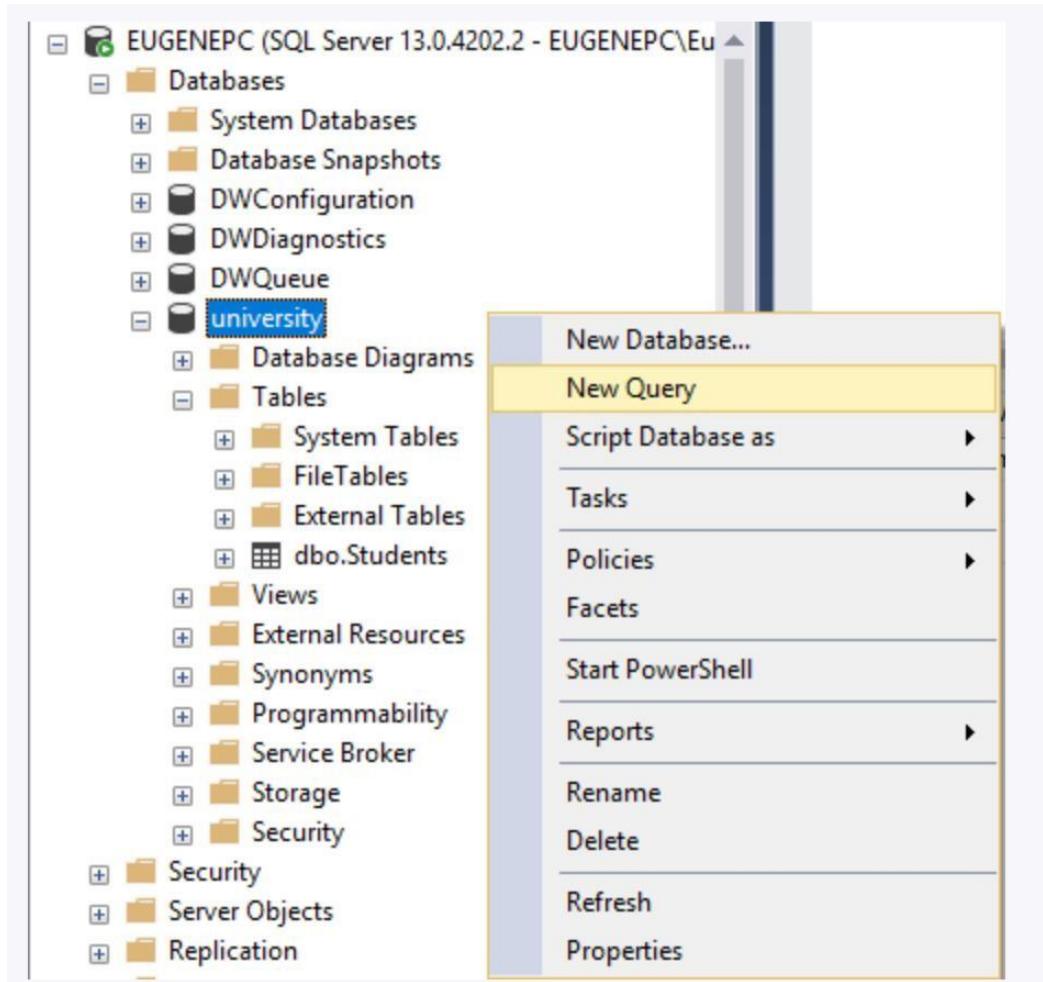
После ввода запроса нажмем на панели инструментов на кнопку Execute, либо можно нажать на клавишу F5.

В результате выполнения запроса в нижней части программы появится небольшая таблица, которая отобразит результаты запроса – то есть все данные из таблицы Students.

Если необходимо совершить несколько запросов к одной и той же базе данных, то мы можем использовать команду USE, чтобы зафиксировать базу данных. В этом случае при запросах к таблицам достаточно указать их имя без имени бд и схемы:

```
1      USE university
2      SELECT * FROM Students
```

В данном случае мы выполняем запрос в целом для сервера, мы можем обратиться к любой базе данных на сервере. Но также мы можем выполнять запросы только в рамках конкретной базы данных. Для этого необходимо нажать правой кнопкой мыши на нужную БД и в контекстном меню выбрать пункт New Query:



Если в этом случае мы захотим выполнить запрос к выше использованной таблице Students, то нам не пришлось бы указывать в запросе название базы данных и схему, так как эти значения итак уже были бы понятны:

```
1      SELECT * FROM Students
```

МОДУЛЬ КОМПЕТЕНЦИИ 2: «РАЗРАБОТКА ОКОННЫХ ПРИЛОЖЕНИЙ»

Платформы для разработки программных решений

Компьютерная платформа – в общем смысле, это любая существующая среда выполнения, в которой должен выполняться вновь разрабатываемый фрагмент программного обеспечения или объектный модуль с учётом накладываемых этой средой ограничений и предоставляемых возможностей. Термин платформа может применяться к разным уровням абстракции, включая определенную аппаратную архитектуру, операционную систему или библиотеку времени выполнения.

Нижний слой многоуровневой организации вычислительной системы (аппаратура, операционная система, прикладное программное обеспечение), на который опираются ОС и прикладное ПО. Аппаратные платформы отличаются друг от друга архитектурой центрального процессора и используемыми шинами связи функциональных блоков.

Каждой аппаратной платформе соответствуют совместимые с ней операционные системы и прикладные программы, которые могут на ней запускаться.

Программная платформа представляет собой общую организацию исполнения прикладных программ, задавая, например, порядок запуска программы, схему использования её адресного пространства, зафиксированные в архитектуре операционной системы плюс API на уровне операционной системы.

При рассмотрении совместимости, или сходства, на уровне операционных систем, например, системных вызовов, файловых систем и пользовательской среды, при сравнении родственных операционных систем (например, UNIX) или семейства (например, Microsoft Windows), речь идет о совместимости на уровне API операционной системы, например, в рамках семейства ОС, а не абстрактного понятия «платформы»

Примеры платформ ОС

- Win32 – Win32 API,
- API POSIX для ОС UNIX/Linux.

Кроссплатформенность программного обеспечения – возможность выполнять его, без перекомпиляции программы, как на различных аппаратных платформах, так и под управлением разных операционных систем (иначе говоря, возможность запуска исполняемого файла на платформах различных ОС).

Примерами программного обеспечения, выполняющегося на разных аппаратных платформах и под управлением разных операционных систем, являются разнообразные программы, написанные на языках программирования для виртуальных машин, таких, как, например, PHP, Perl, Python, Java, и многие другие, а также – кроссплатформенные среды разработки приложений.

Примеры

- Qt
- GTK
- Boost
- Java Virtual Machine
- .NET Framework
- Adobe AIR

В рамках компетенции «Программные решения для бизнеса» для разработки используются платформы Java или .NET. В рамках .NET чаще всего используют интерфейсы программирования Windows Forms или WPF.

Windows Forms – интерфейс программирования приложений (API), отвечающий за графический интерфейс пользователя и являющийся частью Microsoft .NET Framework. Данный интерфейс упрощает доступ к элементам

интерфейса Microsoft Windows за счет создания обёртки для существующего Win32 API в управляемом коде. Причём управляемый код – классы, реализующие API для Windows Forms, не зависят от языка разработки. То есть программист одинаково может использовать Windows Forms как при написании ПО на C#, C++, так и на VB.Net, J# и др.

С одной стороны, Windows Forms рассматривается как замена более старой и сложной библиотеке MFC, изначально написанной на языке C++. С другой стороны, WF не предлагает парадигму, сравнимую с MVC. Для исправления этой ситуации и реализации данной функциональности в WF существуют сторонние библиотеки. Одной из наиболее используемых подобных библиотек является User Interface Process Application Block, выпущенная специальной группой Microsoft, занимающейся примерами и рекомендациями, для бесплатного скачивания. Эта библиотека также содержит исходный код и обучающие примеры для ускорения обучения.

Внутри .NET Framework, Windows Forms реализуется в рамках пространства имён System.Windows.Forms.

Приложение Windows Forms представляет собой событийно-ориентированное приложение, поддерживаемое Microsoft .NET Framework. В отличие от пакетных программ, большая часть времени тратится на ожидание от пользователя какихлибо действий, как, например, ввод текста в текстовое поле или клика мышкой по кнопке.

Windows Presentation Foundation (WPF) – это система следующего поколения для построения клиентских приложений Windows с визуально привлекательными возможностями взаимодействия с пользователем. С помощью WPF можно создавать широкий спектр как автономных, так и размещенных в браузере приложений.

В основе WPF лежит векторная система визуализации, не зависящая от разрешения и созданная с расчетом на возможности современного графического оборудования. WPF расширяет базовую систему полным набором функций разработки приложений, в том числе Extensible Application Markup Language (XAML), элементами управления, привязкой данных, макетом, 2-D- и 3-D-графикой, анимацией, стилями, шаблонами, документами, мультимедиа, текстом и оформлением. WPF входит в состав Microsoft .NET Framework и позволяет создавать приложения, включающие другие элементы библиотеки классов .NET Framework.

Windows Presentation Foundation (WPF) – это система следующего поколения для построения клиентских приложений Windows с визуально привлекательными возможностями взаимодействия с пользователем. С помощью WPF можно создавать широкий спектр как автономных, так и размещенных в браузере приложений. На следующем рисунке показан пример одного из таких приложений Contoso Healthcare Sample Application.

Программирование с использованием WPF

WPF существует в качестве подмножества типов .NET Framework, которые занимают большую часть в пространстве имен System.Windows. Пользователи, которые ранее создавали приложения с помощью .NET Framework, используя такие управляемые технологии, как ASP.NET и Windows Forms, должны быть знакомы с основами программирования WPF; создание экземпляров классов, задание свойств, вызов методов и обработка событий осуществляется с помощью одного из хорошо знакомых языков программирования .NET Framework, таких как C# или Visual Basic.

Для поддержки некоторых более мощных возможностей WPF и для упрощения процесса программирования WPF включает дополнительные программные конструкции, которые расширяют свойства и события: свойства зависимостей и перенаправленные события. Дополнительные сведения о свойствах зависимостей см. в разделе Общие сведения о свойствах зависимости. Дополнительные сведения о перенаправленных событиях см. в разделе Общие сведения о перенаправленных событиях.

Разметка и код программной части

В WPF дополнительно совершенствуется процесс программирования для разработки клиентских приложений Windows. Одним очевидным усовершенствованием является возможность разрабатывать приложения с помощью разметки и кода программной части, с которыми разработчики ASP.NET должны быть уже знакомы. Разметка Extensible Application Markup Language (XAML) обычно используется для реализации внешнего вида приложения при реализации его поведения с помощью управляемых языков программирования (кода программной части). Это разделение внешнего вида и поведения имеет следующие преимущества:

Затраты на разработку и обслуживание снижаются, так как разметка определенного внешнего вида тесно не связана с кодом определенного поведения.

Разработка более эффективна, так как разработчики, реализующие внешний вид приложения, могут это делать одновременно с разработчиками, реализующими поведение приложения.

Для реализации и совместного использования разметки XAML применяется множество средств конструирования, чтобы удовлетворить требованиям участников разработки приложений. Microsoft Expression Blend предназначается для конструкторов, в то время как Visual Studio 2005 ориентируется на разработчиков.

Ниже приводится краткое описание разметки и кода программной части WPF. Дополнительные сведения об этой модели программирования см. в Общие сведения о языке XAML (WPF) и в Код программной части и XAML в WPF.

Разметка

XAML – это основанный на XML язык разметки, который используется для декларативной реализации внешнего вида приложения. Обычно он используется для создания окон, диалоговых окон, страниц и пользовательских элементов управления, а также для их заполнения элементами управления, фигурами и графикой.

Поскольку XAML основан на XML, UI, который формируется с его помощью, организуется в виде иерархии вложенных элементов, называемой деревом элементов. Дерево элементов предоставляет логический и интуитивно понятный способ для создания и управления UIs.

Код программной части

Приложение в основном предназначено для реализации функциональных возможностей, которые отвечают на взаимодействия с пользователем, включая обработку событий (например, нажатие меню, панели инструментов или кнопки) и вызов бизнес-логики и логики доступа к данным в ответ на события. В WPF такое поведение обычно реализуется в коде, который связан с разметкой. Этот тип кода называется кодом программной части.

Приложения

.NET Framework, System.Windows, разметка и выделенный код составляют основу разработки приложений WPF. Кроме того, WPF предоставляет полный набор средств для создания удобных и многофункциональных элементов пользовательского интерфейса. Чтобы упаковать разработанные элементы и предоставить их пользователю в виде приложений, WPF предоставляет типы и службы, вместе называемые моделью приложения. Модель приложения поддерживает разработку как автономных, так и размещенных в браузере приложений.

Автономные приложения

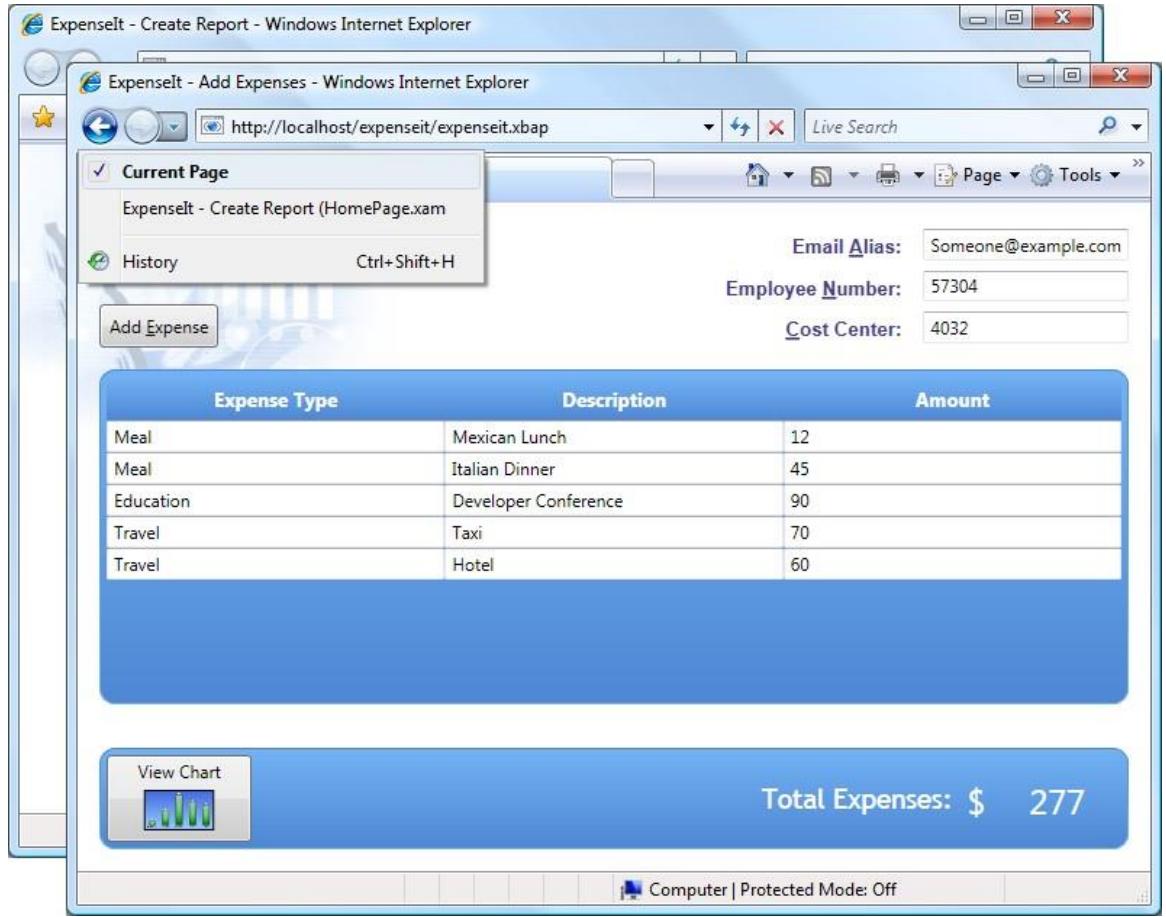
Для автономных приложений можно создавать доступные из меню и панелей инструментов окна и диалоговые окна с помощью класса Window. На следующем рисунке показано автономное приложение с главным и диалоговым окном.



Кроме того, можно использовать следующие диалоговые окна WPF: MessageBox, OpenFileDialog, SaveFileDialog и PrintDialog.

Приложения, размещенные в браузере

Для приложений, размещаемых в браузере, также называемых XAML browser applications (XBAPs), можно создавать страницы (Page) и страницочные функции (PageFunction<T>), по которым можно переходить с помощью гиперссылок (классы Hyperlink). На следующем рисунке показана страница в XBAP, размещенная в Internet Explorer 7.



Приложения WPF могут размещаться как в Microsoft Internet Explorer 6, так и в Internet Explorer 7. WPF предлагает два следующих параметра для альтернативных узлов переходов:

- Frame, чтобы размещать блоки содержимого для навигации в окнах или на страницах.
- NavigationWindow, чтобы размещать содержимое для навигации во всем окне.

Элементы управления

Взаимодействия с пользователем, предоставляемые моделью приложения, являются сконструированными элементами управления. В WPF "элемент управления" – это основное понятие, относящееся к категории классов WPF, которые расположены в окне или на странице, имеют user interface (UI) и реализовывают некоторое поведение.

Элементы управления WPF по функциям

Далее перечислены встроенные элементы управления WPF.

- Кнопки: Button и RepeatButton.
- Отображение данных: DataGridView, ListView и TreeView.
- Выбор и отображение дат: Calendar и DatePicker.
- Диалоговые окна: OpenFileDialog, PrintDialog и SaveFileDialog.
- Рукописный фрагмент: InkCanvas и InkPresenter.
- Документы: DocumentViewer, FlowDocumentPageViewer, FlowDocumentReader, FlowDocumentScrollView и StickyNoteControl.
- Ввод: TextBox, RichTextBox и PasswordBox.
- Структура: Border, BulletDecorator, Canvas, DockPanel, Expander, Grid,

`GridView`, `GridSplitter`, `GroupBox`, `Panel`, `ResizeGrip`, `Separator`, `ScrollBar`, `ScrollViewer`, `StackPanel`, `Thumb`, `Viewbox`, `VirtualizingStackPanel`, `Window` и `WrapPanel`.

- Мультимедиа: `Image`, `MediaElement` и `SoundPlayerAction`.
- Меню: `ContextMenu`, `Menu` и `ToolBar`.
- Переходы: `Frame`, `Hyperlink`, `Page`, `NavigationWindow` и `TabControl`.
- Выбор: `CheckBox`, `ComboBox`, `ListBox`, `RadioButton` и `Slider`.
- Информация пользователя: `AccessText`, `Label`, `Popup`, `ProgressBar`, `StatusBar`, `TextBlock` и `ToolTip`.

Ввод и команды

Элементы управления наиболее часто обнаруживают входные данные пользователя и отвечают на них. Система ввода WPF использует прямые и перенаправленные события для поддержки ввода текста, управления фокусом и позиционирования мыши. Дополнительные сведения см. в разделе Общие сведения о входных данных.

Приложения часто предъявляют сложные требования к вводу данных. WPF предоставляет систему команд, в которой действия пользователя по вводу данных отделяются от кода, который отвечает на эти действия. Дополнительные сведения см. в разделе Общие сведения о системе команд.

Макет

При создании UI происходит упорядочивание элементов управления по расположению и размеру для формирования структуры. Основным требованием любой структуры является адаптируемость к изменениям размера окна и параметрам отображения. Чтобы не пришлось создавать код для адаптации структуры в таких обстоятельствах, WPF предоставляет первоклассную расширяемую систему структуры.

Основой системы структуры является относительное позиционирование, что увеличивает способность адаптации к изменяющемуся окну и условиям отображения. Кроме того, система структуры управляет согласованием между элементами управления для определения структуры. Такое согласование состоит из двух этапов: сначала элемент управления сообщает родительскому элементу, какое расположение и размер требуется; затем родительский элемент сообщает элементу управления, какое пространство он может занять.

Дочерние элементы управления получают доступ к системе макета через базовые классы WPF. Для распространенных макетов, таких как сетки, вложение и закрепление, WPF включает несколько элементов управления макетом.

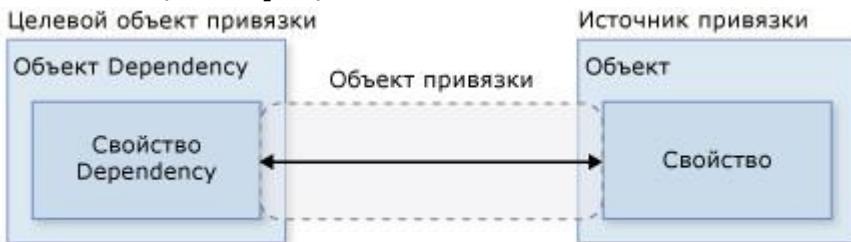
- `Canvas`: дочерние элементы управления предоставляют свои собственные макеты.
- `DockPanel`: дочерние элементы управления выравниваются по краям панели.
- `Grid`: дочерние элементы управления располагаются по строкам и столбцам.
- `StackPanel`: дочерние элементы управления располагаются либо горизонтально, либо вертикально.
- `VirtualizingStackPanel`: дочерние элементы управления являются виртуальными и располагаются в одной горизонтальной или вертикальной строке.
- `WrapPanel`: дочерние элементы управления располагаются в порядке слева-направо и переносятся на следующую строку, когда в текущей строке не хватает места.

Привязка данных

Большинство приложений создаются для предоставления пользователям средств просмотра и редактирования данных. В приложениях WPF работа по хранению и доступу к данным уже обеспечена такими технологиями, как Microsoft SQL Server и ADO.NET. После обращения к данным и загрузки данных в управляемые объекты приложения начинается основная тяжелая работа для приложений WPF. По существу, она включает в себя две вещи:

- копирование данных из управляемых объектов в элементы управления, где данные могут отображаться и редактироваться;
- проверка того, что изменения, внесенные в данные с помощью элементов управления, скопированы обратно в управляемые объекты.

Чтобы упростить разработку приложений, WPF предоставляет механизм привязки данных для автоматического выполнения этих этапов. Основной единицей механизма привязки данных является класс Binding, назначение которого привязать элемент управления (цель привязки) к объекту данных (источник привязки). Это отношение показано на следующем рисунке.



Механизм привязки данных WPF предоставляет дополнительную поддержку, включающую проверку, сортировку, фильтрацию и группировку. Кроме того, привязка данных поддерживает использование шаблонов данных для создания настраиваемого UI, чтобы связывать данные, когда UI отображается несоответствующими стандартными элементами управления WPF.

Графика

WPF представляет обширный, масштабируемый и гибкий набор графических возможностей, которые имеют следующие преимущества:

Графика, не зависящая от разрешения и устройства. Основной единицей измерения в графической системе WPF является аппаратно-независимая точка, которая составляет 1/96 часть дюйма независимо от фактического разрешения экрана и предоставляет основу для создания изображения, независимого от разрешения и устройства. Каждый аппаратно-независимый пиксель автоматически масштабируется в соответствии с числом точек на дюйм в системе, в которой он отображается.

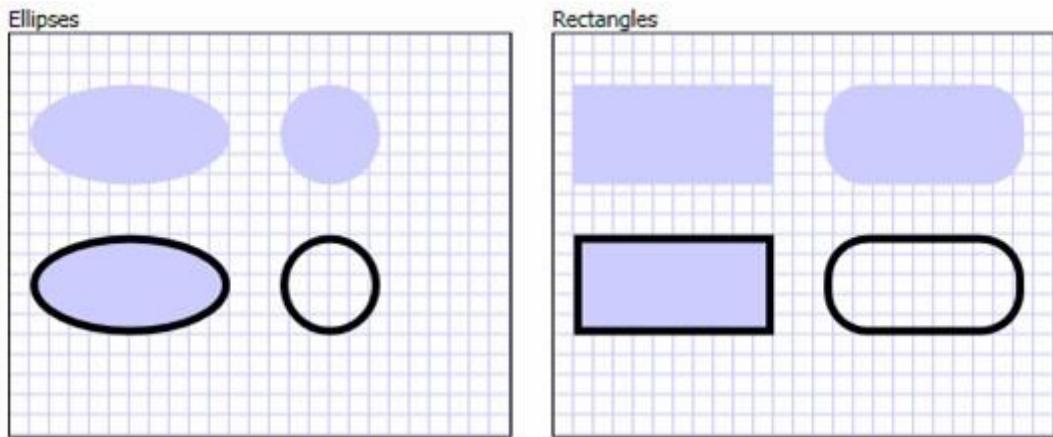
Повышенная точность. В системе координат WPF используются числа с плавающей запятой двойной точности, вместо одиночной точности. Значения преобразований и прозрачности также выражаются с помощью чисел двойной точности. Кроме того, WPF поддерживает широкую цветовую палитру (scRGB) и предоставляет встроенную поддержку для управления входными данными из различных цветовых пространств.

Дополнительная поддержка графики и анимации. WPF упрощает программирование графики за счет автоматического управления анимацией. Разработчик не должен заниматься обработкой сцен анимации, циклами визуализации и билинейной интерполяцией. Кроме того, WPF предоставляет поддержку проверки нажатия и полную поддержку альфа-компоновки.

Аппаратное ускорение. Графическая система WPF использует преимущества графического оборудования, чтобы уменьшить использование ЦП.

Двухмерные формы

WPF предоставляет библиотеку общих 2-Д фигур, нарисованных с помощью векторов, таких, как прямоугольники и эллипсы, показанные на следующем рисунке.



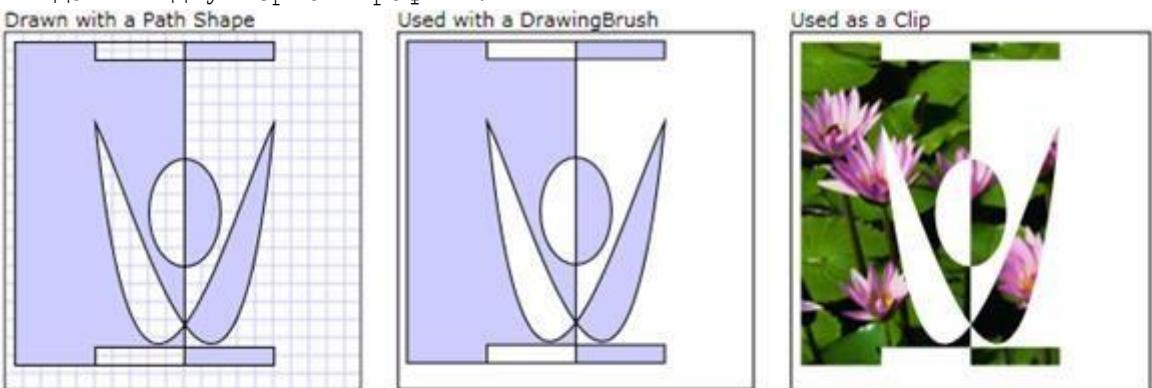
Интересная особенность фигур в том, что они могут не только отображаться; фигуры реализовывают многие возможности, ожидаемые от элементов управления, включая ввод с клавиатуры и ввод с помощью мыши.

Двухмерная геометрия

WPF предоставляет стандартный набор двухмерных (2-D) фигур. Однако, возможно, потребуется создать пользовательские фигуры для облегчения разработки настраиваемого UI. В этих целях WPF предоставляет геометрические объекты. На следующем рисунке показано использование геометрий для создания пользовательской фигуры, которая может быть нарисована непосредственно, использоваться в качестве кисти, или использоваться для отсечения других фигур и элементов управления.

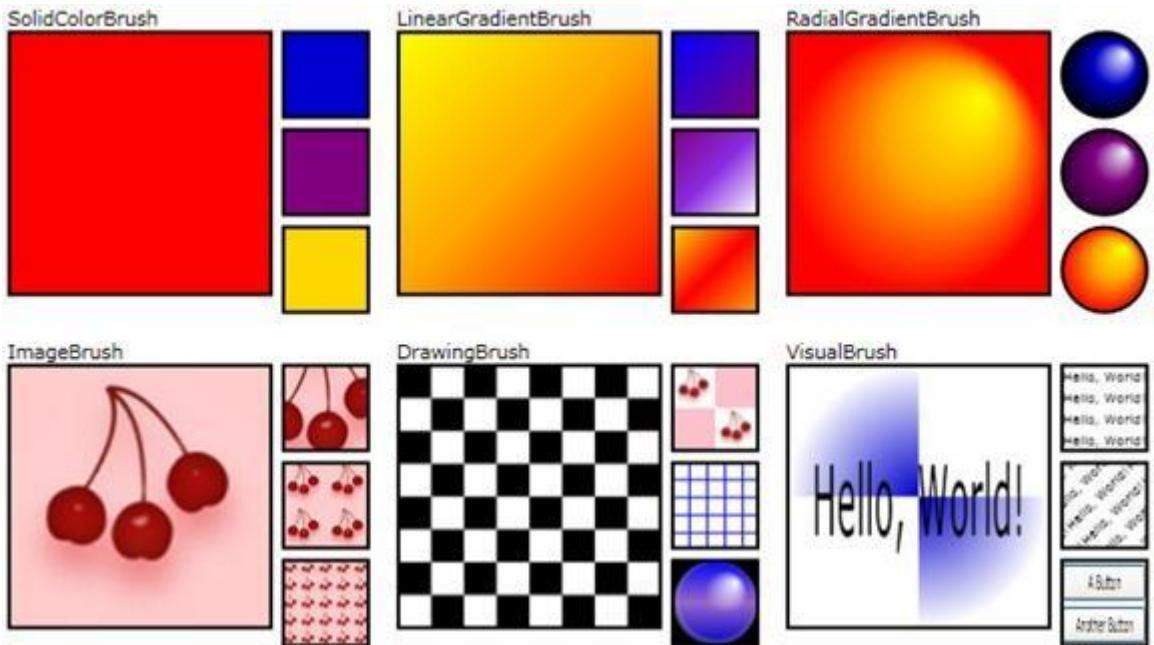
Объекты Path могут быть использованы для рисования замкнутых, открытых, составных фигур и даже кривых поверхностей.

Объекты Geometry могут быть использованы для отсечения, проверки нажатия и отрисовки данных двухмерной графики.



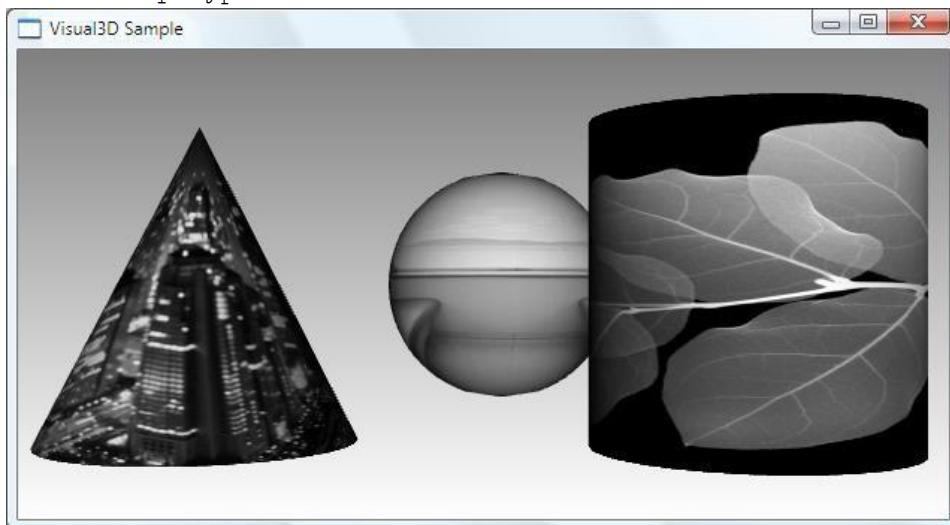
Двухмерные эффекты

Подмножество средств 2-D WPF включает визуальные эффекты, такие как градиенты, точечные рисунки, чертежи, рисунки с видео, поворот, масштабирование и наклон. Все это достигается с помощью кистей; на следующем рисунке показано несколько примеров.



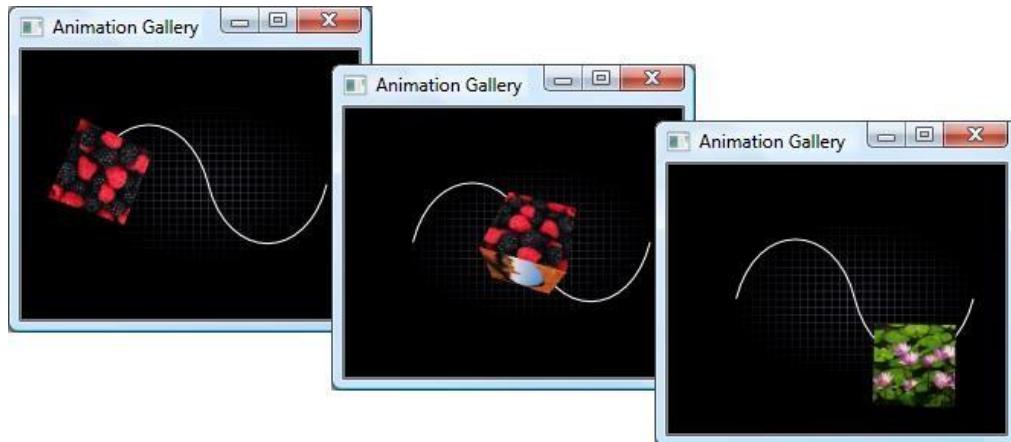
Трехмерная визуализация

WPF также включает возможности трехмерной (3-D) визуализации, интегрированные с двухмерной (2-D) графикой, что позволяет создавать более яркий и интересный UIs. Например, следующий рисунок показывает изображения 2-D, отображаемые в фигурах 3-D.



Анимация

Поддержка анимации WPF позволяет осуществлять рост, вибрацию, вращение и исчезновение элементов управления для создания интересных страниценных переходов и других эффектов. Можно анимировать большинство классов WPF, даже настраиваемые классы. На следующем рисунке показана простая анимация в действии.

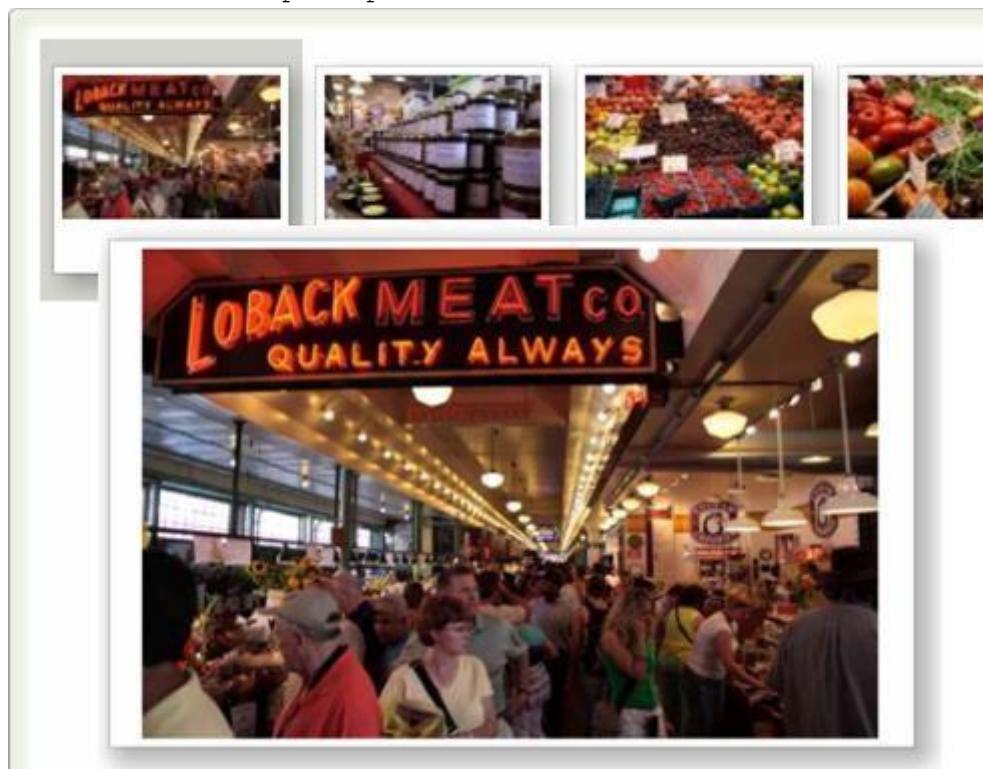


Мультимедиа

Одним из способов передачи богатого содержимого является использование аудиовизуальной среды. WPF предоставляет специальную поддержку для изображений, видео и аудио.

Изображения

Изображения присутствуют в большинстве приложений, и WPF предоставляет несколько способов их использования. На следующем рисунке показан UI со списком, в котором содержатся эскизные изображения. При выделении эскиза изображение показывается в полном размере.



Видео и аудио

Элемент управления `MediaElement` способен воспроизводить видео и аудио, и является достаточно гибким, чтобы служить основой для пользовательского проигрывателя.

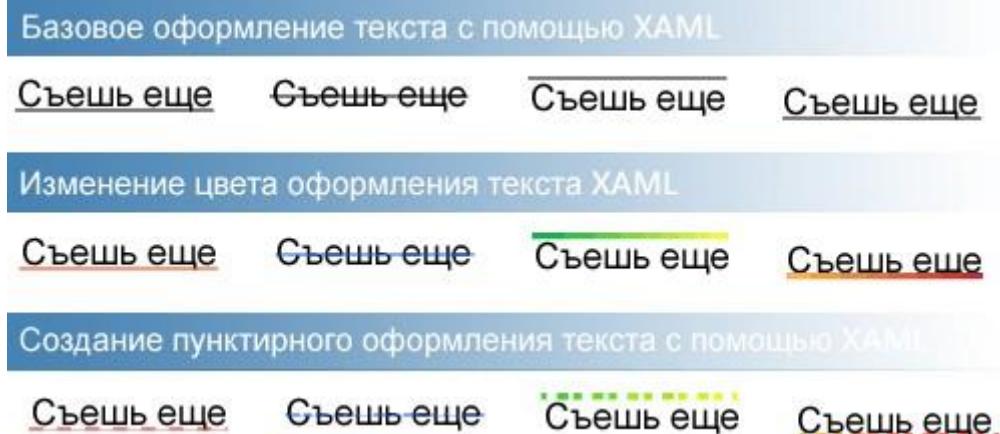
Текст и типография

Для облегчения отрисовки текста высокого качества WPF предоставляет следующие возможности:

- Поддержка шрифта OpenType.
- Улучшения ClearType.

- Высокая производительность, которая использует преимущества аппаратного ускорения.
- Интеграция текста с мультимедиа, графикой и анимацией.
- Механизмы резервирования и поддержки международного шрифта.

Для демонстрации интеграции текста с графикой на следующем рисунке показано применение художественного оформления текста.



Документы

WPF предоставляет встроенную поддержку работы с тремя типами документов: документами нефиксированного формата, документами фиксированного формата и документами XML Paper Specification (XPS). WPF также предоставляет службы для создания и просмотра документов, управления документами, добавления заметок, упаковки и печати документов.

Документы нефиксированного формата

Документы нефиксированного формата разработаны для оптимизации просмотра и читаемости посредством динамической настройки и обновления содержимого при изменении размера окна и параметров дисплея.

Документы фиксированного формата

Документы фиксированного формата предназначены для приложений, в которых требуется точное представление вида "что видишь, то и получишь" (режим полного соответствия изображения на экране и распечатки WYSIWYG), особенно по отношению к печати. Документы фиксированного формата обычно используются при подготовке публикаций с помощью настольных издательских средств, обработке текста и разметке формы, где строгое соблюдение исходного дизайна страницы является обязательным.

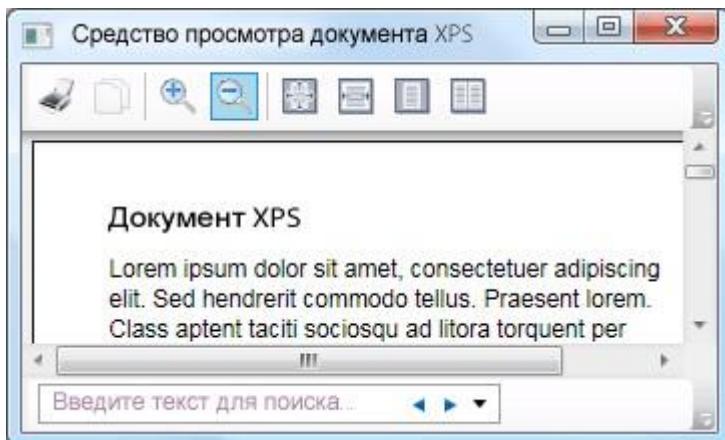
В документах фиксированного формата поддерживается точное размещение содержимого независимо от устройства. Например, документ фиксированного формата отображается на мониторе с разрешением 96 точек на дюйм точно так же, как при печати на лазерном принтере с разрешением 600 точек на дюйм или на фотонаборной машине с разрешением 4800 точек на дюйм. Макет документа остается одинаковым во всех случаях, хотя качество документа варьируется в зависимости от возможностей каждого устройства.

Документы XML Paper Specification (XPS) построены на основе документов фиксированного формата WPF. Документы XPS описываются схемой на основе XML, которая фактически представляет разбитый на страницы электронный документ. Открытый кросс-платформенный формат документов XPS предназначен для упрощения создания, печати и архивирования разбитых на страницы документов, а также организации совместного доступа. Технология XPS включает следующие важные возможности:

Упаковка документов XPS в файлы ZipPackage, соответствующие стандарту Open Packaging Conventions (OPC).

- Размещение в автономных и в размещенных в браузере приложениях.
- Создание документов XPS и управление ими из приложений WPF вручную.
- Высокоточная отрисовка путем выбора устройства вывода максимального качества.
- Очередь печати принтера Windows Vista.
- Прямая отправка документов на XPS-совместимые принтеры.
- Интеграция UI с DocumentViewer.

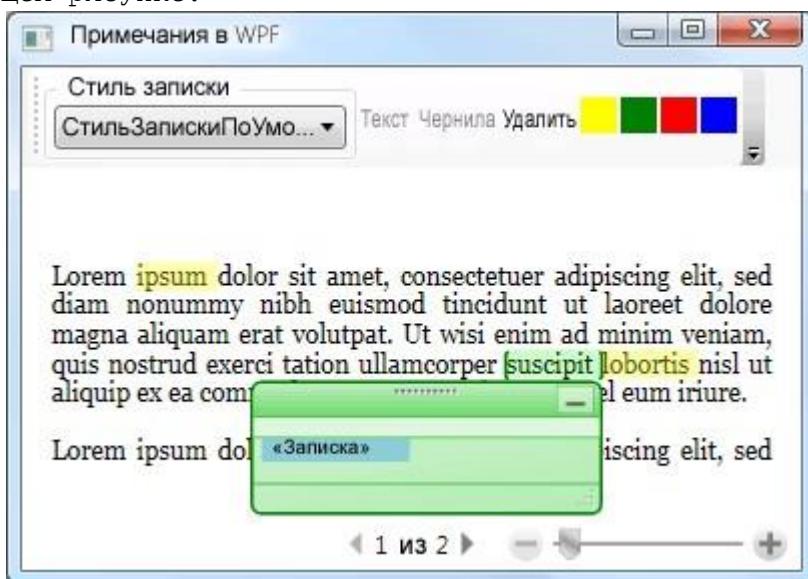
На следующем рисунке показан документ XPS, который отображается с помощью DocumentViewer.



DocumentViewer также дает возможность пользователям изменять просмотр, поиск и печать документов XPS.

Заметки

Заметки – это примечания или комментарии, которые добавляются к документу, чтобы отметить информацию или выделить интересующие элементы для дальнейшего использования. В напечатанных документах делать заметки просто, но в электронных документах возможность создания заметок часто ограничена или отсутствует. Однако в WPF для поддержки возможности создания комментариев-наклеек и выделений предоставляется система заметок. Кроме того, эти заметки можно применять к документам, размещенным в элементе управления DocumentViewer, как показано на следующем рисунке.



Упаковка

WPF System.IO.Packaging APIs позволяет приложениям организовывать данные, содержимое и ресурсы в единые, переносимые, удобные для распространения и для

доступа упакованные документы. Для проверки подлинности элементов, содержащихся в пакете, можно включать цифровые подписи, которые гарантируют, что подписанный элемент не был подделан или изменен. Кроме того, можно ограничить доступ к защищенной информации, зашифровав пакеты с помощью системы управления правами.

Печать

.NET Framework включает подсистему печати, которую WPF дополняет поддержкой для расширенного управления системой печати. Улучшения печати включают следующее:

- Установка удаленных серверов и очередей печати в режиме реального времени.
- Динамическое обнаружение возможностей принтера.
- Динамическая установка параметров принтера.
- Перенаправление и изменение приоритета заданий на печать.

В документах XPS также имеется ключевое улучшение производительности. Существующий путь печати Microsoft Windows Graphics Device Interface (GDI) обычно подразумевает два преобразования:

первое – преобразование документа в формат процессора печати, например в Enhanced Metafile (EMF); второе – преобразование в язык описания страниц принтера, например в Printer Control Language (PCL) или PostScript.

Однако документы XPS обходятся без этих преобразований, поскольку один компонент формата файла XPS является как языком обработчика заданий печати, так и языком описания страницы. Эта поддержка позволяет уменьшить как размер файла очереди, так и загрузки сетевых принтеров.

Главной задачей большей части элементов управления WPF является отображение содержимого. В WPF тип и количество элементов, которые могут составлять содержимое элемента управления, называется моделью содержимого элемента управления. Некоторые элементы управления могут содержать один элемент и один тип содержимого. Например, содержимое TextBox – строковое значение, которое присваивается свойству Text.

Триггеры

Хотя главной задачей разметки XAML и является реализация внешнего вида приложения, XAML также можно использовать для реализации некоторых действий, выполняемых приложением. Например, с помощью триггеров изменять внешний вид приложения в зависимости от действий пользователя.

Шаблоны элементов управления

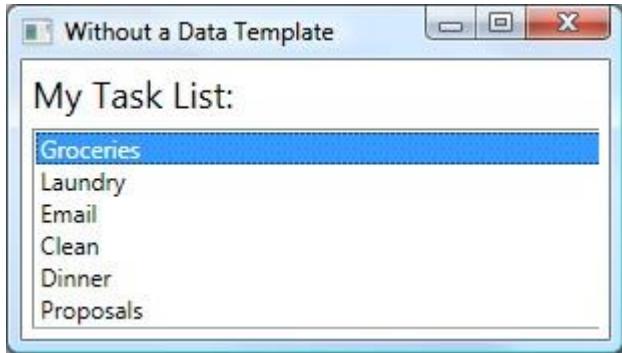
UIs по умолчанию для элементов управления WPF обычно создается из других элементов управления и фигур. Например, Button состоит из элементов управления ButtonChrome и ContentPresenter.

ButtonChrome обеспечивает стандартный внешний вид кнопки, в то время как ContentPresenter отображает содержимое кнопки, заданное свойством Content.

Иногда внешний вид элемента управления по умолчанию может не сочетаться с общим внешним видом приложения. В этом случае можно с помощью ControlTemplate изменить внешний вид UI элемента управления без изменения его содержимого и поведения.

Шаблоны данных

В то время как шаблон элемента управления позволяет задавать внешний вид элемента управления, шаблон данных позволяет задавать внешний вид содержимого элемента управления. Шаблоны данных часто используются для улучшения способа отображения данных, связанных с элементом управления. На следующем рисунке показан внешний вид по умолчанию элемента управления ListBox, который связан с коллекцией объектов Task, где каждый объект имеет имя, описание и приоритет.



Внешний вид по умолчанию – это вид, который ожидается от `ListBox`. Однако внешний вид по умолчанию каждого объекта коллекции содержит только имя. Чтобы отобразить имя, описание и приоритет объекта коллекции, внешний вид по умолчанию элементов связанного списка элемента управления `ListBox` должен быть изменен с помощью `DataTemplate`.

Ресурсы

Элементы управления в приложении должны использовать один и тот же внешний вид, который может включать любые шрифты и цвета фона для шаблонов элементов управления, шаблонов данных и стилей. С помощью поддержки WPF для ресурсов user interface (UI) можно инкапсулировать эти ресурсы в одном расположении для повторного использования.

Темы и обложки

С точки зрения визуального восприятия тема определяет глобальный внешний вид системы Windows и приложений, которые в ней запускаются. Windows поставляется с несколькими темами. Например, Microsoft Windows XP поставляется с темами Windows XP и Windows Classic, а Windows Vista поставляется с темами Windows Vista и Windows Classic. Внешний вид, определяемый темой, задает внешний вид по умолчанию для приложения WPF. Однако WPF не поддерживает прямую интеграцию с темами Windows. Поскольку внешний вид WPF определяется шаблонами, WPF включает по одному шаблону для каждой известной темы Windows, в том числе Aero (Windows Vista), Classic (Microsoft Windows 2000), Luna (Microsoft Windows XP) и Royale (Microsoft Windows XP Media Center Edition 2005). Эти темы упакованы в словари ресурсов, которые применяются, если ресурсы не найдены в приложении. Внешний вид многих приложений задается с помощью этих тем; сохраняющаяся согласованность с внешним видом Windows помогает пользователям быстрее освоиться с большинством приложений.

С другой стороны, опыт работы пользователя с некоторыми приложениями не обязательно связан с стандартными темами. Например, Microsoft Windows Media Player работает с аудио- и видеоданными, и здесь преимущество имеют пользователи с опытом работы в другом стиле. Такие UIs чаще предоставляют настраиваемые, специфичные для приложения темы. Такие темы называются "обложки", и приложения, которые их используют, часто предоставляют средства настройки различных аспектов обложек. Microsoft Windows Media Player имеет множество собственных обложек и обложек от сторонних производителей.

Пользовательские элементы управления

Хотя WPF и обеспечивает поддержку настройки, могут возникнуть ситуации, в которых существующие элементы управления WPF не удовлетворяют требованиям приложения или его пользователей. Это возможно в следующих ситуациях:

Нужный UI не может быть создан путем настройки внешнего вида и поведения существующих реализаций WPF.

Нужное поведение не поддерживается (или поддерживается частично) существующими реализациями WPF.

Тем не менее, в этом случае можно воспользоваться одной из трех моделей WPF для создания нового элемента управления. Каждая модель предназначена для

определенного скрипта и требует, чтобы пользовательский элемент управления был производным от конкретного базового класса WPF. Далее приводится описание этих трех моделей.

Модель пользовательского элемента управления. Пользовательский элемент управления производится из `UserControl` и состоит из одного или нескольких других элементов управления.

Модель элемента управления. Пользовательский элемент управления производится из `Control` и используется для построения реализаций, в которых внешний вид и поведение разделены с помощью шаблонов, подобно большей части элементов управления WPF. Создание элемента управления, производного от `Control`, предоставляет по сравнению с пользовательскими элементами управления большую свободу для создания нестандартного UI, но может потребовать дополнительных усилий.

Модель элемента .NET Framework. Пользовательский элемент управления производится от `FrameworkElement`, когда его внешний вид определяется пользовательской логикой визуализации (не шаблонами).

Советы и рекомендации по WPF

Как любая платформа разработки, WPF может использоваться множеством способов для достижения нужного результата. Для гарантий, что приложения WPF предоставляют требуемое взаимодействие с пользователем и удовлетворяют требованиям аудитории в целом, в данном разделе предлагаются советы и рекомендации по специальным возможностям, глобализации и локализации, а также производительности.

Рассмотрим примеры работы с платформой .NET на примере выполнения задания для демонстрационного экзамена 2016 года (WPF).

Ознакомиться с полной версией задания и использованными ресурсами можно на ресурсе Академии Ворлдскиллс Россия по ссылке:

<https://drive.google.com/drive/u/0/folders/1NrRGxRfGmf0HGT4fTCFvZpe1oCId-1kn>

Сессия 1 данного Конкурсного задания состоит из следующей документации / файлов:

1. `WSR2018_TP09_14+_C1.pdf` (Инструкция к первой сессии)
2. `marathon-skills-2018-database-mysql.sql` (Сценарий SQL для создания таблиц с данными для MySQL)
3. `marathon-skills-2018-database-mssql.sql` (Сценарий SQL для создания таблиц с данными для Microsoft SQL Server)
4. `marathon-skills-2018-staff-import.xlsx` (Данные для импорта)
5. `marathon-skills-2018-testing-data-s1.pdf` (Тестирование системы)

ВВЕДЕНИЕ

В этой сессии вы начнете разработку приложения и базы данных для MarathonSkills 2016. Дизайнер предоставил вам набор системной документации, так что вы можете построить систему в соответствии с потребностями клиента. Найдите время для знакомства с предоставленными материалами.

Создайте базу данных, а затем импортируйте туда необходимые данные. Затем создайте приложение: часть окон, которые будут доступны пользователям системы.

Файл: `marathon-skills-2018-testing-data-s1.pdf` предоставлен вам для того, чтобы вы смогли протестировать систему

ИНСТРУКЦИЯ УЧАСТНИКУ

К концу этой сессии, у вас должны быть достигнуты следующие результаты, необходимые для того, чтобы заказчик был спокоен, что система будет завершена вовремя. Убедитесь, что вы следуете предоставленному руководству по стилю во всех частях системы. Убедитесь, что вы предоставляете соответствующие проверки

и сообщения об ошибках во всех частях системы. Убедитесь, что все соответствующие кнопки / ссылки работают в конце сессии. Убедитесь, что вы используете соответствующие соглашения об именах для всех частей системы по мере необходимости.

ПРАКТИЧЕСКИЕ РЕЗУЛЬТАТЫ

1.1 СОЗДАНИЕ БАЗЫ ДАННЫХ Создайте базу данных, используя знакомую вам платформу (MySQL / MSSQLServer) на сервере баз данных, который вам предоставлен.

1.2 ЗАГРУЗКА ДАННЫХ Сценарий SQL предоставлен для вас, чтобы создать большинство таблиц и вставки данных в них. Все, что вам нужно сделать, это импортировать сценарий SQL в вашу базу данных. Выберите сценарий SQL, который подходит для вашей платформы: MySQL: marathon-skills-2016-database-mysql.sql СерверSQL: marathon-skills-2016-database-mssql.sql Таблица Сотрудники (персонал, должности) не включены в этот сценарий SQL. См результаты 1.3 и 1.4.

1.3 СОЗДАТЬ ТАБЛИЦЫ ДЛЯ ПЕРСОНАЛА СОГЛАСНО СПЕЦИФИКАЦИИ Обратитесь к диаграмме базы данных (ERD) и словарю данных. Создайте таблицы сотрудников (Position, Staff, Timesheet) согласно спецификации.

1.4 ИМПОРТ ДАННЫХ ПЕРСОНАЛА Все данные сотрудников были представлены в marathon-skills-2018-staff-import.xlsx. Эти данные не отформатированы для импортирования непосредственно в базу данных, вам необходимо отформатировать данные и загрузить их в таблицы, которые вы только что создали. Поле Summary Information не требуется. В поле "FullName" в формате "Имя Фамилия" используются разные символы разделителя. Убедитесь, что адреса электронной почты в правильном формате.

1.5 СОЗДАТЬ ПРИЛОЖЕНИЕ Создайте приложение, используя выбранную вами платформу .NET (или Java).

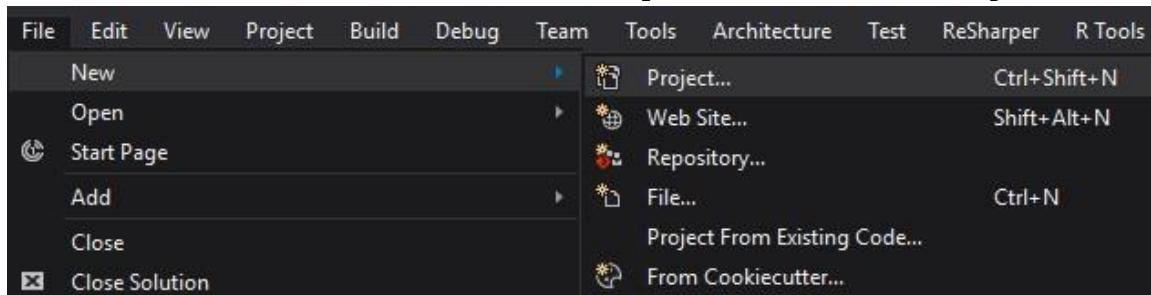
1.6 СОЗДАНИЕ "1. ГЛАВНЫЙ ЭКРАН СИСТЕМЫ" Создайте главное меню системы, как указано в "1. Главный экран системы" в презентации. Каждое окно / страница приложения, который имеет "? дней? часов и? минут до начала гонки в "в нижней части экрана должно автоматически обновляться в режиме реального времени. Рассчитать количество времени, оставшегося до начала первого MarathonSkills 2018 начинается (2018-11-24 6:00).

1.7 СОЗДАНИЕ ОКНА "7. ПОДРОБНАЯ ИНФОРМАЦИЯ" Создание окна подробная информация как указано в "7. Подробная информация" в презентации.

1.8 СОЗДАНИЕ ОКНА "10. СПИСОК БЛАГОТВОРИТЕЛЬНЫХ ОРГАНИЗАЦИЙ" Создать страницу, как описано в "10. Список благотворительных организаций" в презентации. Эта страница отображает все благотворительные организации, перечисленные в базе данных вместе с их логотипами (при условии, что они есть в общих ресурсах), чтобы показать бегунам благотворительные организации, которые их могут поддержать.

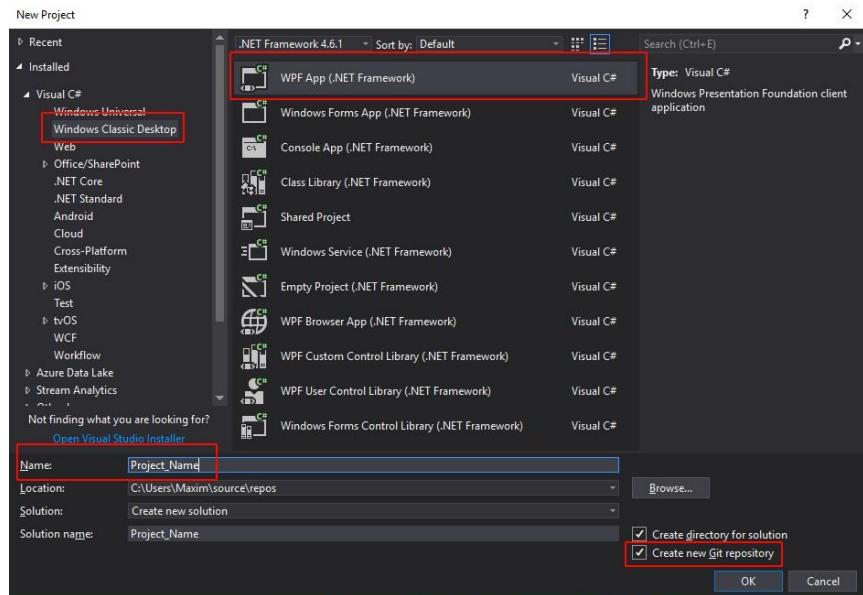
СОЗДАНИЕ ПРОЕКТА

Запустите Visual Studio 2017, перейдите File - Project.



Выберите Visual C# - Windows Classic Desktop - WPF App, укажите имя

проекта, а также поставьте галочку «Create new git repository» чтобы включить проект в систему контроля версий.

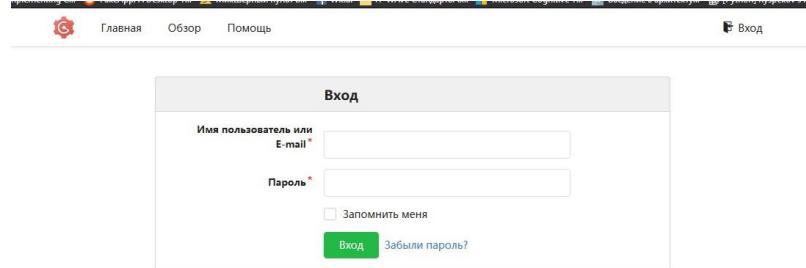


После

нажатия кнопки «OK» создастся проект с пустой формой.

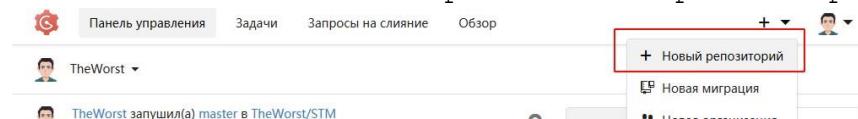
СОЗДАНИЕ РЕПОЗИТОРИЯ

Откройте в браузере GOOG. Авторизуйтесь под своей учетной записью.

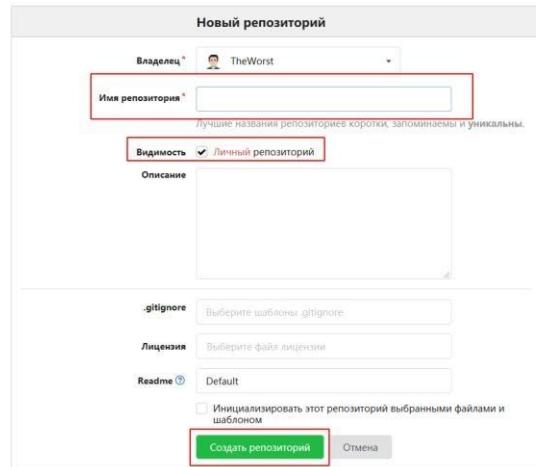


На

верхней панели нажмите «+» и выберите «Новый репозиторий».



После чего укажите имя репозитория, поставьте галочку что репозиторий будет являться личным. И создайте его путем нажатия на

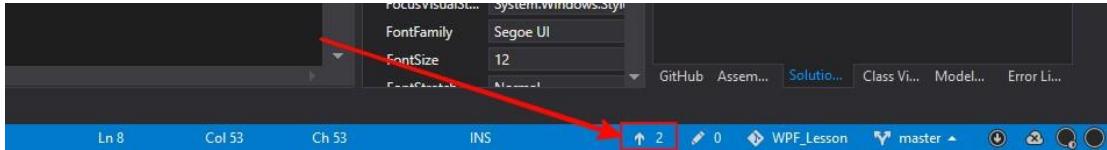


«Создать репозиторий».

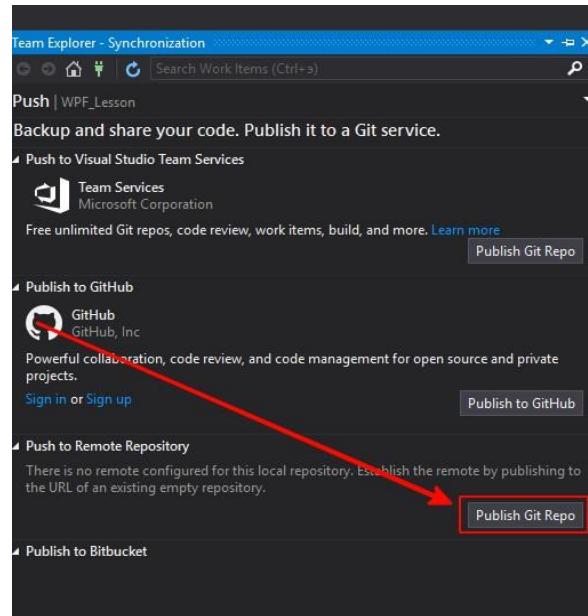
После создания репозитория появится следующее окно.

Создадим в проекте «Resource Dictionary».

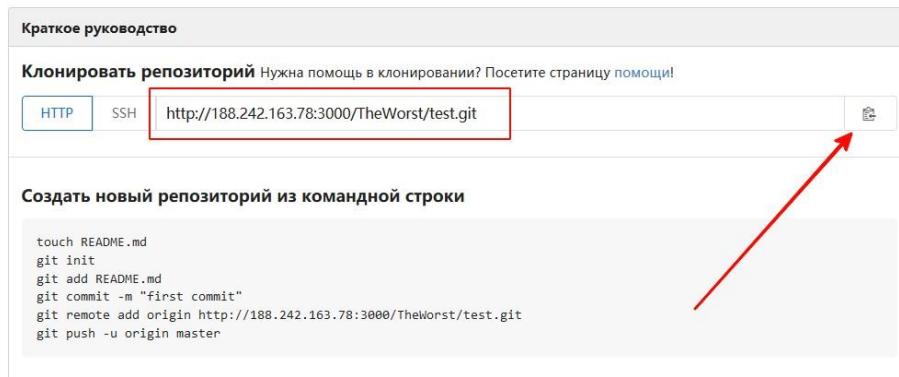
Для того чтобы подключить проект, откройте проект в Visual Studio, затем внизу нажмите на стрелку направленную вверх.



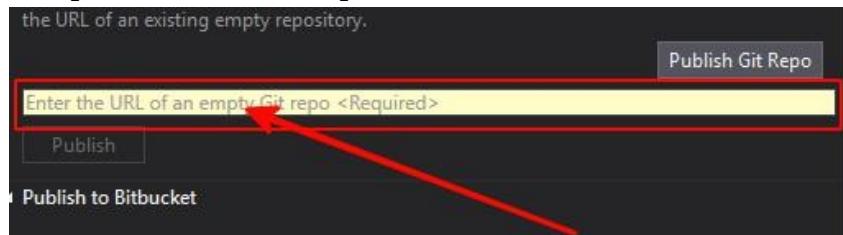
После нажатия у вас откроется окно «Team Explorer», в котором нажмите «Publish Git Repo»



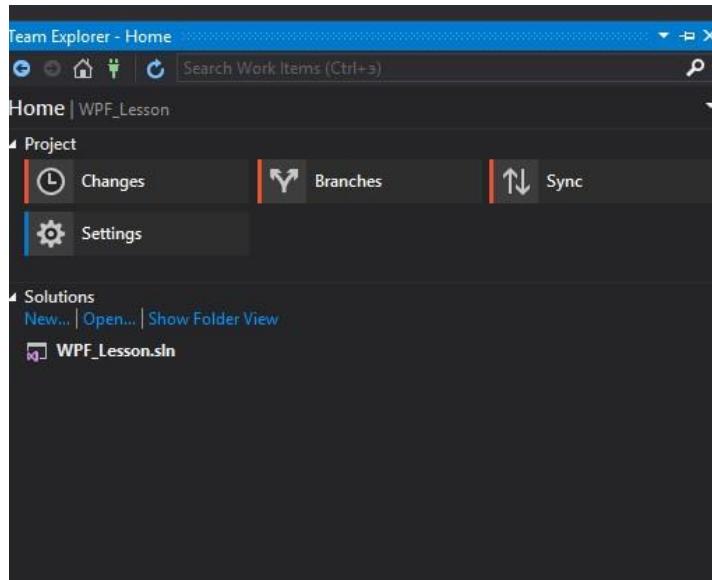
После чего у вас откроется окно в которое необходимо вставить ссылку на репозиторий. Для получения ссылки, откройте GOGS, авторизуйтесь, зайдите в созданный репозиторий (урок Создание репозитория) и там скопируйте на него ссылку.



После того как скопируете ссылку вставьте ее в поле подключения удаленного репозитория в «Team Explorer» и нажмите кнопку «Publish»



После нажатия кнопки, проект будет сохранен на удаленном репозитории, а также создастся связь между локальным и удаленным репозиторием.



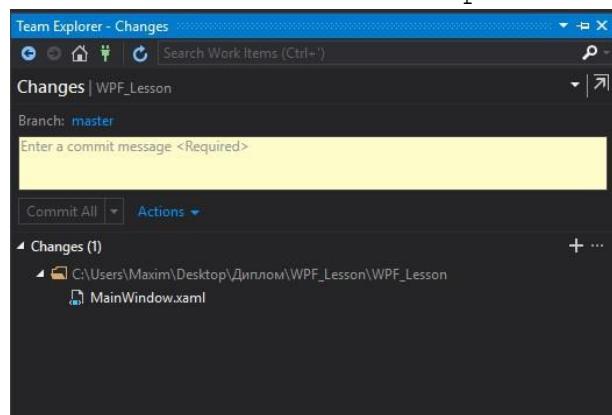
Проверить результат сохранения проекта можно на GOGS в репозитории.

СОХРАНЕНИЕ ИЗМЕНЕНИЙ В РЕПОЗИТОРИИ

По мере того как будут вноситься правки в проект, счетчик изменений будет расти (указывается внизу Visual Studio)

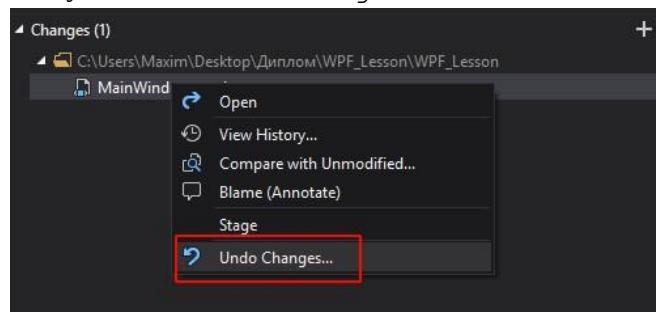


Чтобы сохранить проект в репозитории, необходимо нажать на счетчик (карандаш). После этого появится окно Team Explorer.

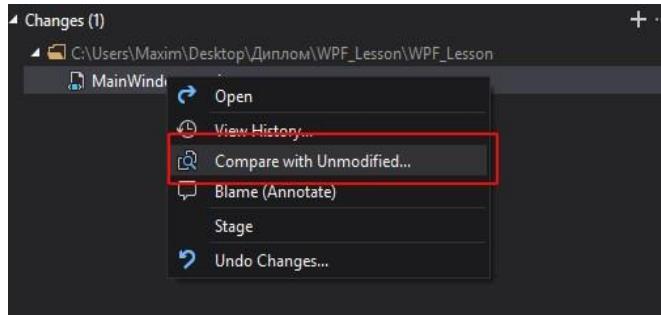


В данном окне показывается ветка (branch) в которую будет сохранятся изменения и файлы (Changes) в которых произошли изменения.

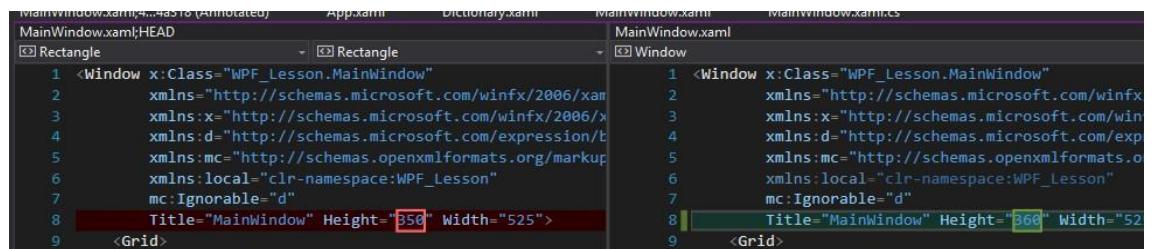
Если изменения были произведены ошибочно, то их можно откатить на предыдущую версию, путем нажатия на необходимом файле правой кнопкой мыши и выбрав пункт «Undo Changes»



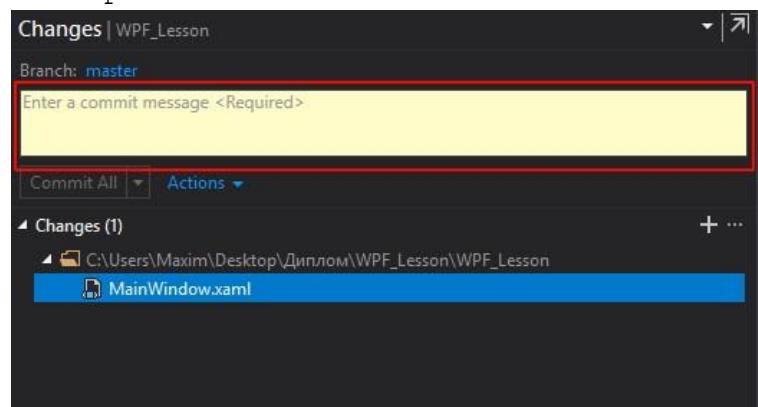
Если необходимо посмотреть изменения в файле (сравнить с предыдущей версией), то необходимо нажать правую кнопку мыши на нужном файле и выбрать пункт меню «Compare with Unmodified»



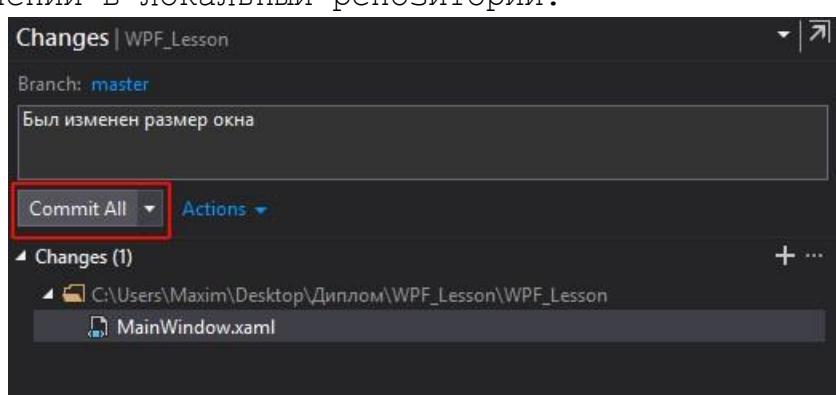
После чего откроется окно с изменениями, в одном окне предыдущая версия, в другом актуальная.



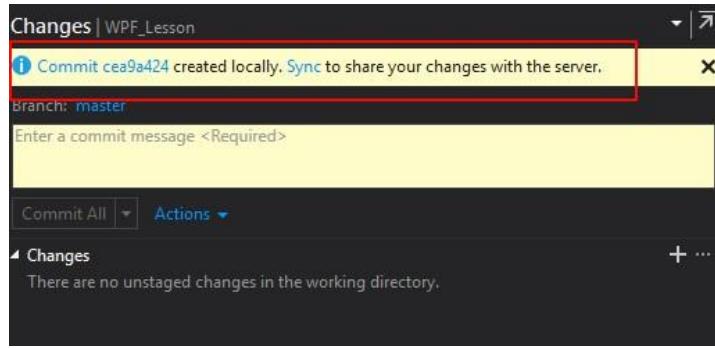
Для того чтобы сохранить изменения необходимо написать текст об изменениях в данной версии.



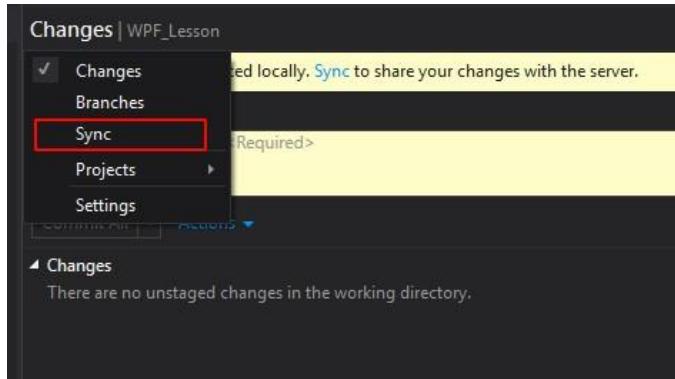
После ввода изменений, необходимо нажать кнопку «Commit All» для сохранения изменений в локальный репозиторий.



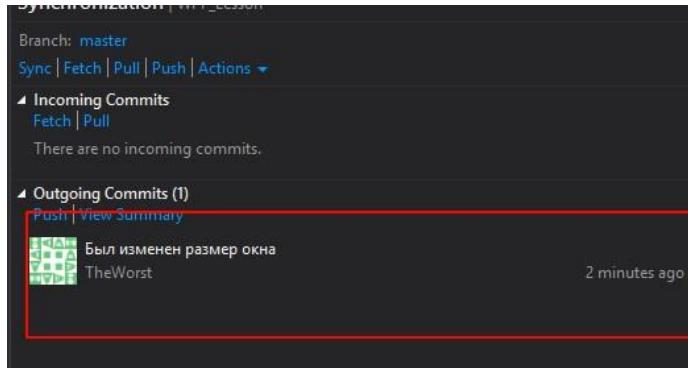
После чего будет выведено сообщение о успешном сохранении проекта. Так же будет предложено синхронизировать локальный репозиторий с удаленным.



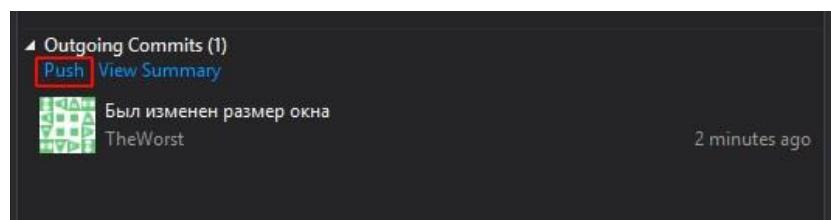
Теперь необходимо отправить изменения локального репозитория на удаленный репозиторий. Для этого необходимо выбрать пункт Sync из выпадающего меню.



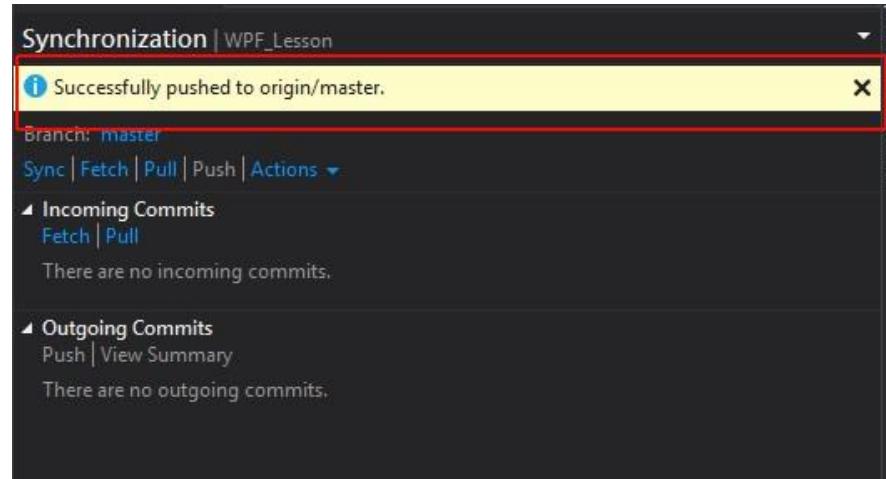
В данном пункте меню, внизу будет отображаться ваш коммит (изменения).



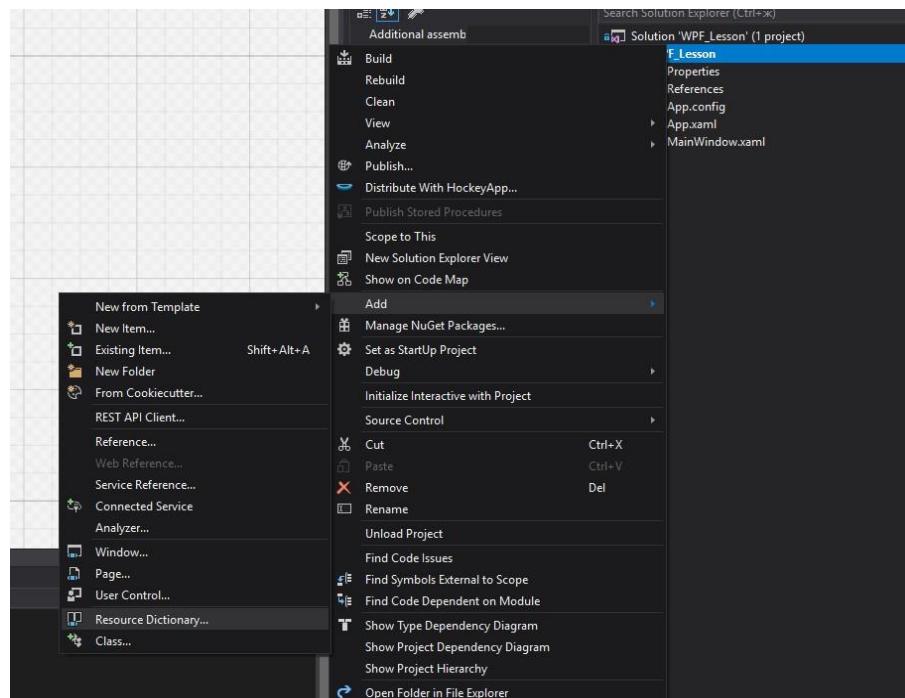
Для того чтобы отправить изменения необходимо нажать на кнопку «Push»



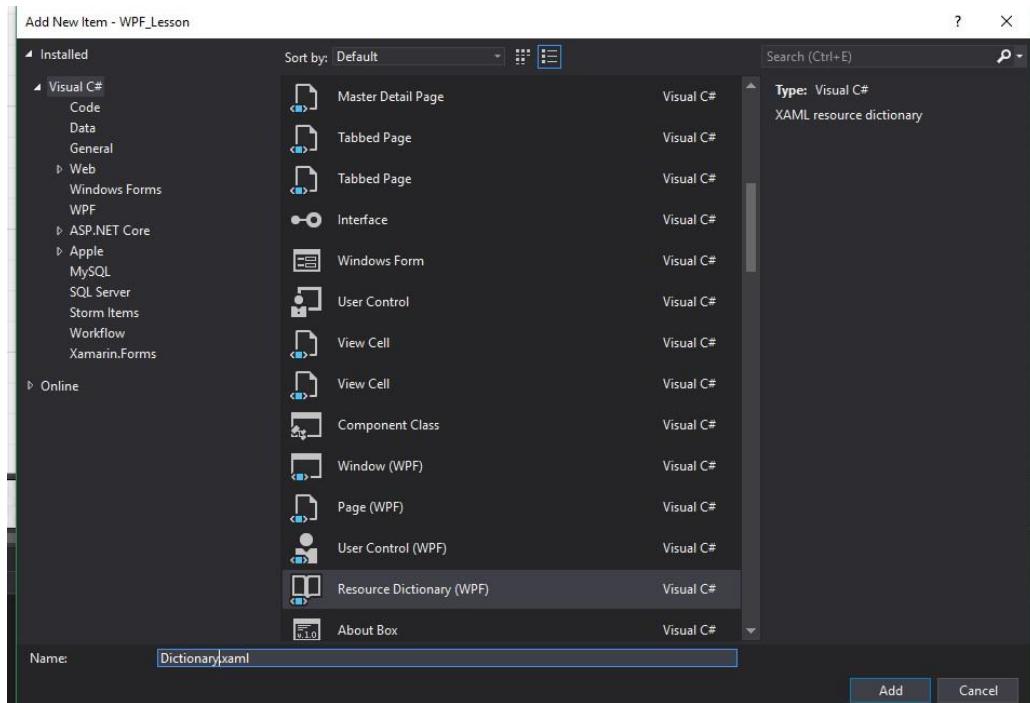
После нажатия кнопки локальный репозиторий будет сохранен на удаленном. И после успешного завершения будет выведено соответствующее сообщение.



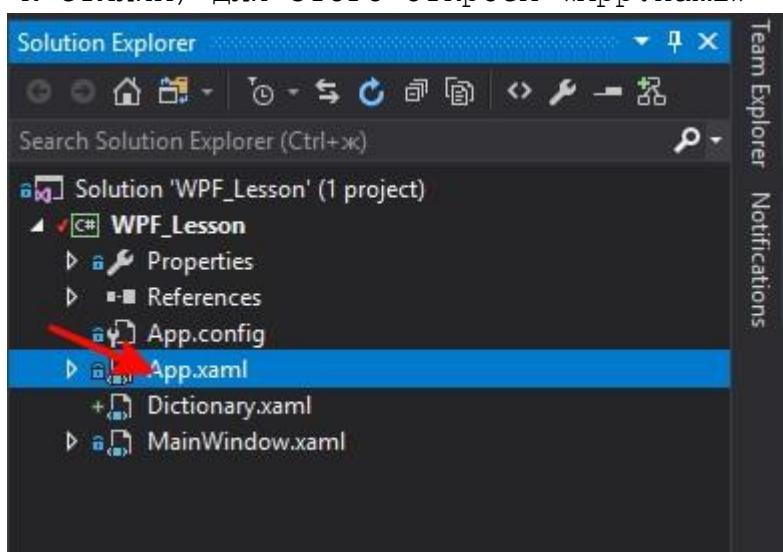
СОЗДАНИЕ СЛОВАРЯ СТИЛЕЙ



Укажем имя «Dictionary.xaml» и создадим его.



Теперь словарь необходимо подключить к проекту (чтобы компоненты могли обращаться к стилям) для этого откроем «App.xaml»



И впишем следующий код для подключения словаря.

```
<Application x:Class="WPF_Lesson.App"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="clr-namespace:WPF_Lesson"
    StartupUri="MainWindow.xaml">
    <Application.Resources>
        <ResourceDictionary>
            <ResourceDictionary.MergedDictionaries>
                <ResourceDictionary Source="Dictionary.xaml"/>
            </ResourceDictionary.MergedDictionaries>
        </ResourceDictionary>
    </Application.Resources>
</Application>
```

После данных операций можно начинать писать стили и константы. Напишем первый стиль, который будет менять задний фон и внутренний отступ, и он применяется ко всем компонентам типа «Label»

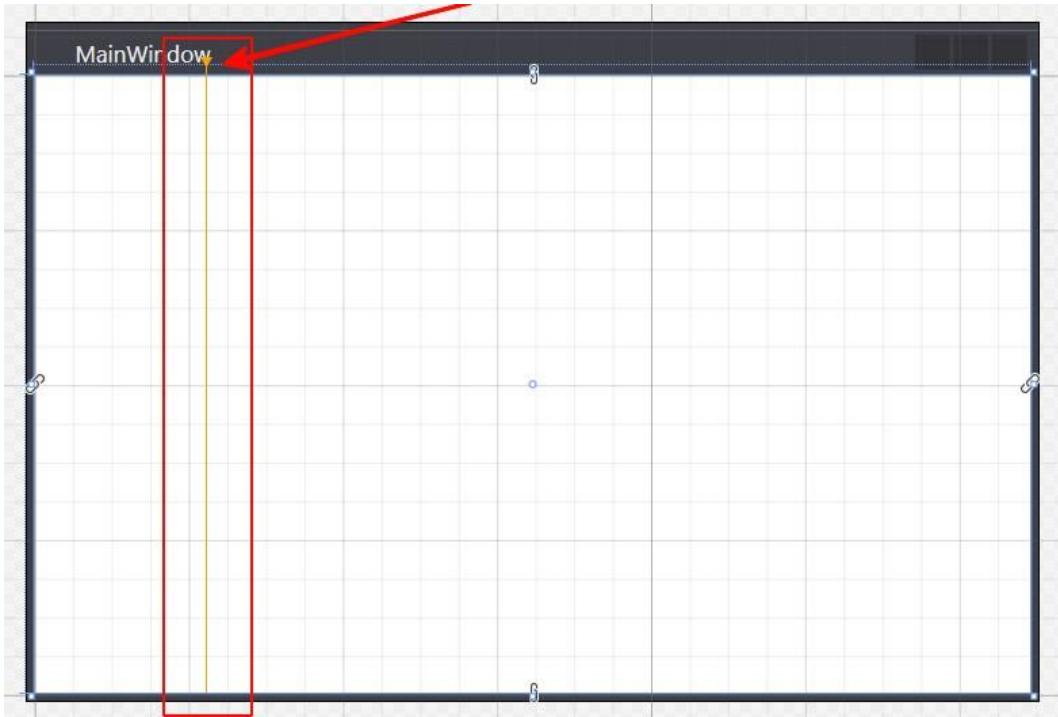
```
<ResourceDictionary xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="clr-namespace:WPF_Lesson">
    <Style TargetType="{x:Type Label}">
        <Setter Property="Background" Value="AntiqueWhite"/>
        <Setter Property="Padding" Value="0"/>
    </Style>
</ResourceDictionary>
```



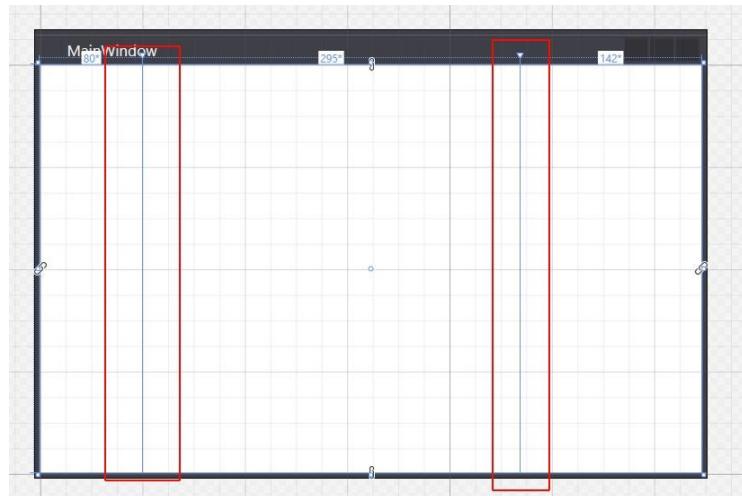
РАЗМЕЩЕНИЕ КОНТЕНТА ПО ЦЕНТРУ ФОРМЫ

Откройте форму или страницу. Выберите компонент Grid, после чего по краям (сверху и слева) появится возможность добавлять строки и столбцы.

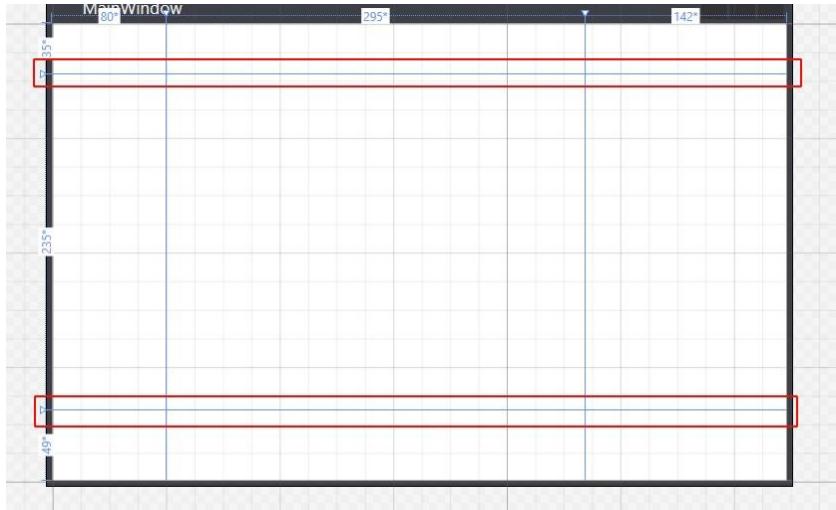
Добавим первый столбец.



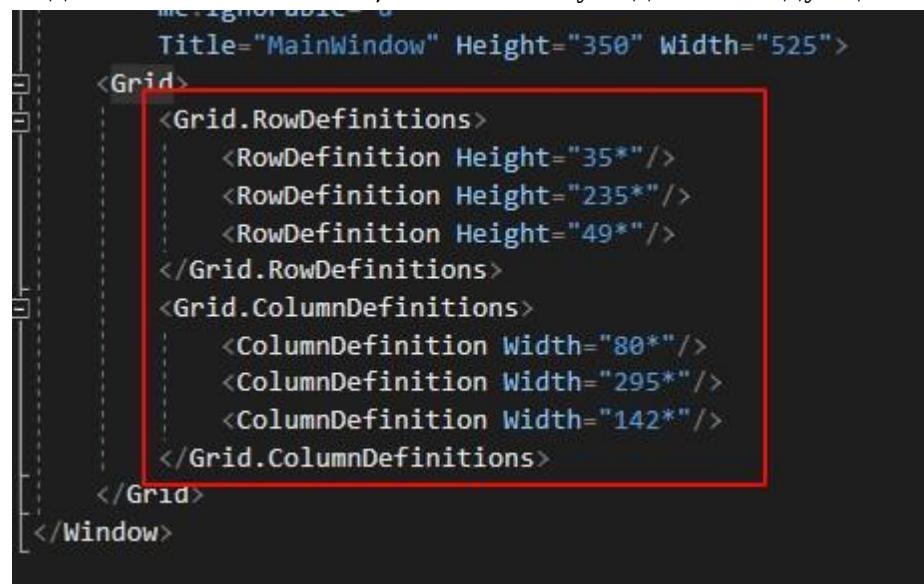
Добавим еще один столбец.



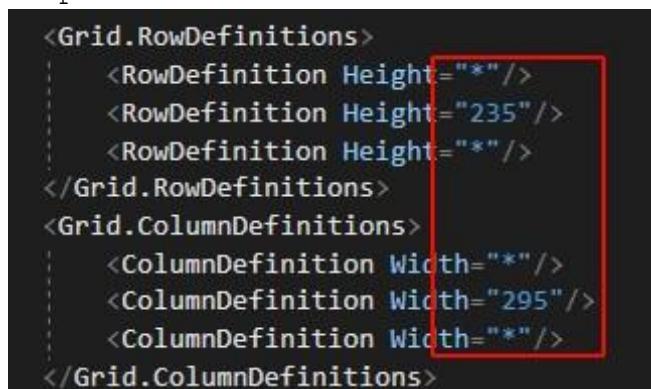
Таким же образом добавим две строки.



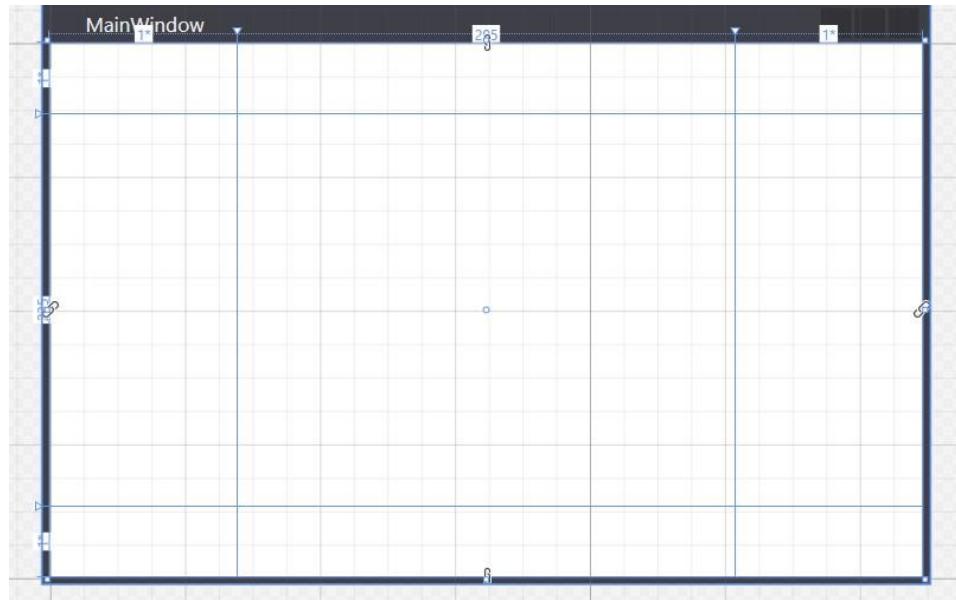
После всех добавлений мышкой, мы можем увидеть следующий код XAML.



Так же столбцы и строки можно добавлять, редактируя код XAML. Здесь же можно редактировать размер строк и столбцов. Чтобы разместить элементы по центру (которые будут во второй строке и во втором столбце) необходимо задать размеры строкам и столбцам. Зададим столбцам и строкам следующие размеры.



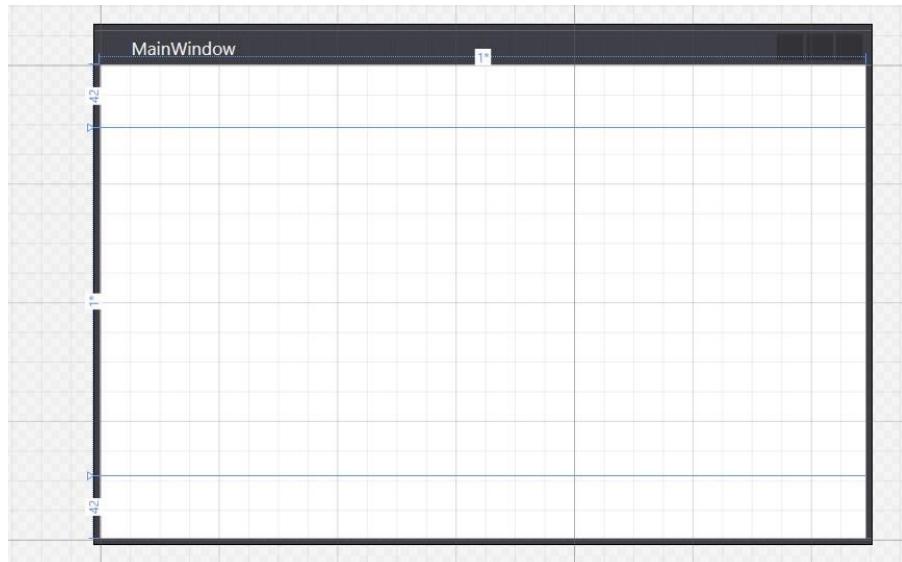
После чего на форме увидим следующее.



В результате средний столбец и средняя строка имеют фиксированный размер, в то время как последние и первые имеют динамический размер (меняется в зависимости от размера контейнера, окна).

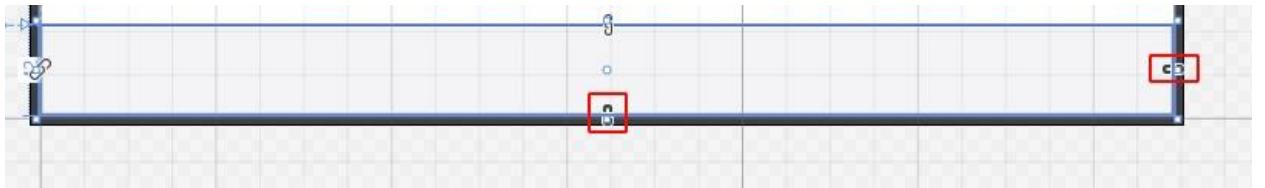
СОЗДАНИЕ БАЗОВОЙ ФОРМЫ

Создадим несколько строк с фиксированными размерами.



```
<Grid>
  <Grid.RowDefinitions>
    <RowDefinition Height="42"/>
    <RowDefinition/>
    <RowDefinition Height="42"/>
  </Grid.RowDefinitions>
</Grid>
```

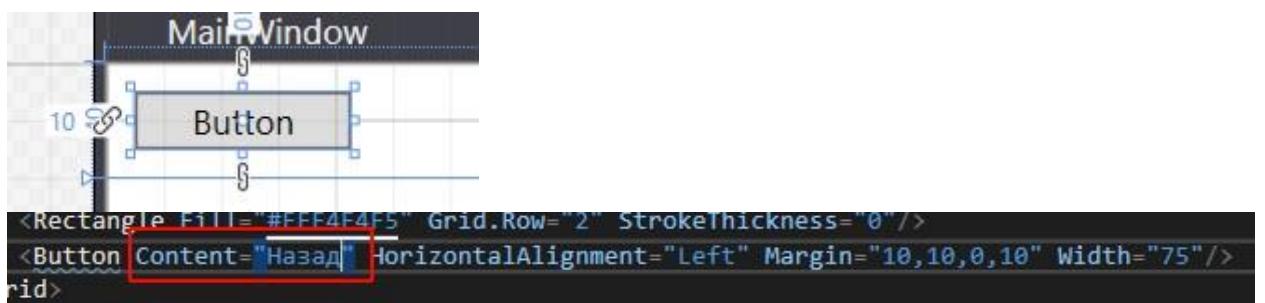
Добавим компонент «Rectangle» на форму и разместим его в 3 строке. Компонент необходимо растянуть на всю ширину и высоту строки, а затем закрепить высоту и ширину (чтобы он мог растягиваться)



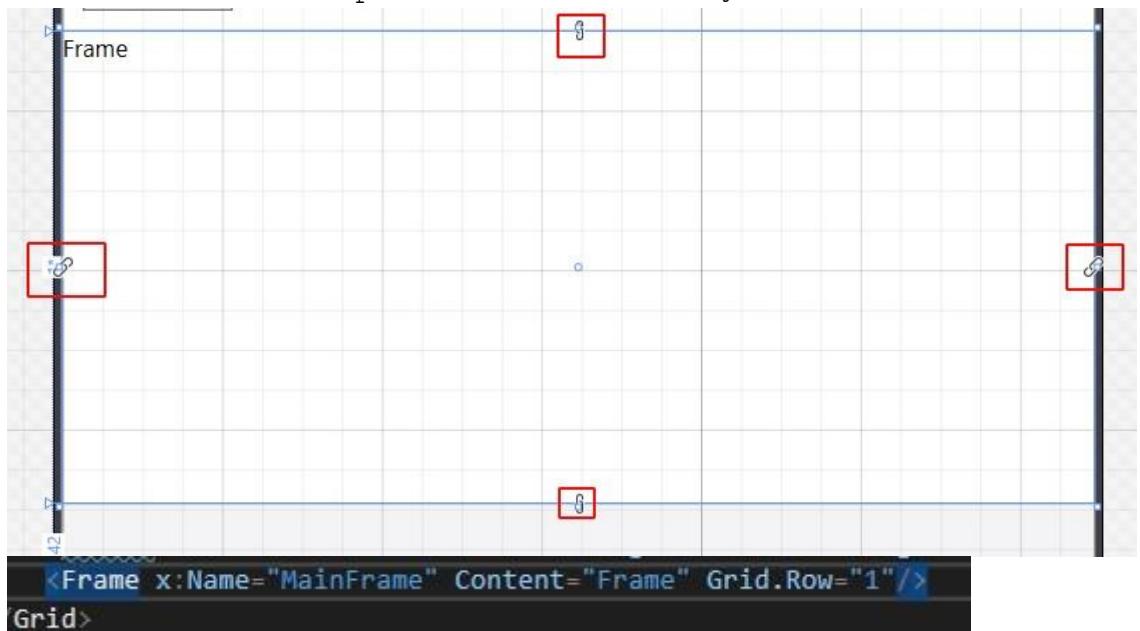
Укажем у компонента «Rectangle» свойство «StrokeThickness» равное 0, это задаст размер границы прямоугольника.

```
</Grid.RowDefinitions>
<Rectangle Fill="#FFF4F4F5" Grid.Row="2" StrokeThickness="0" />
```

Добавим на форму кнопку и разместим в левом углу первой строки. И укажем имя кнопки «Назад», которая в последствии будет выполнять соответствующую функцию.



Добавим на форму компонент «Frame» во вторую строку, в него будут загружать страницы (Авторизация, Регистрация и другие). Закрепим компонент по высоте и ширине. Укажем имя ему «MainFrame»

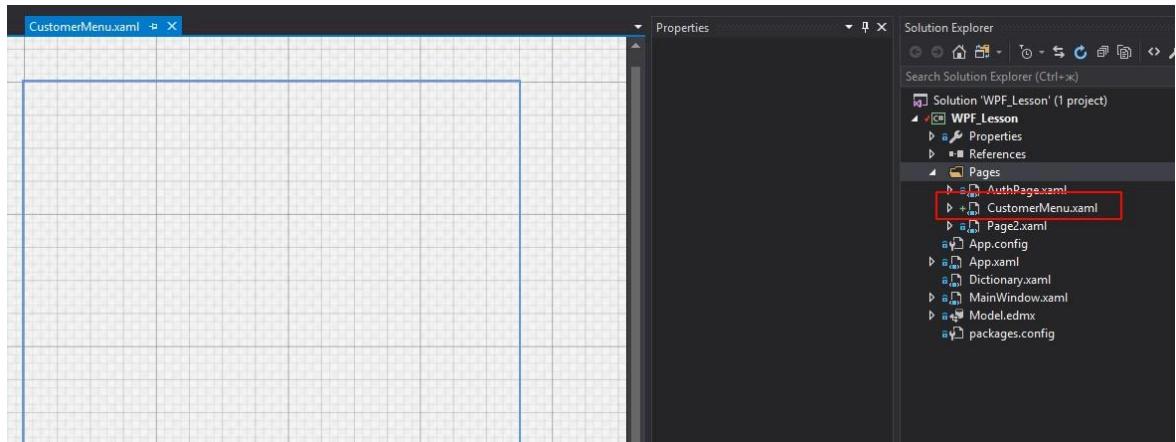


Запустим проект и увидим примерно следующее.

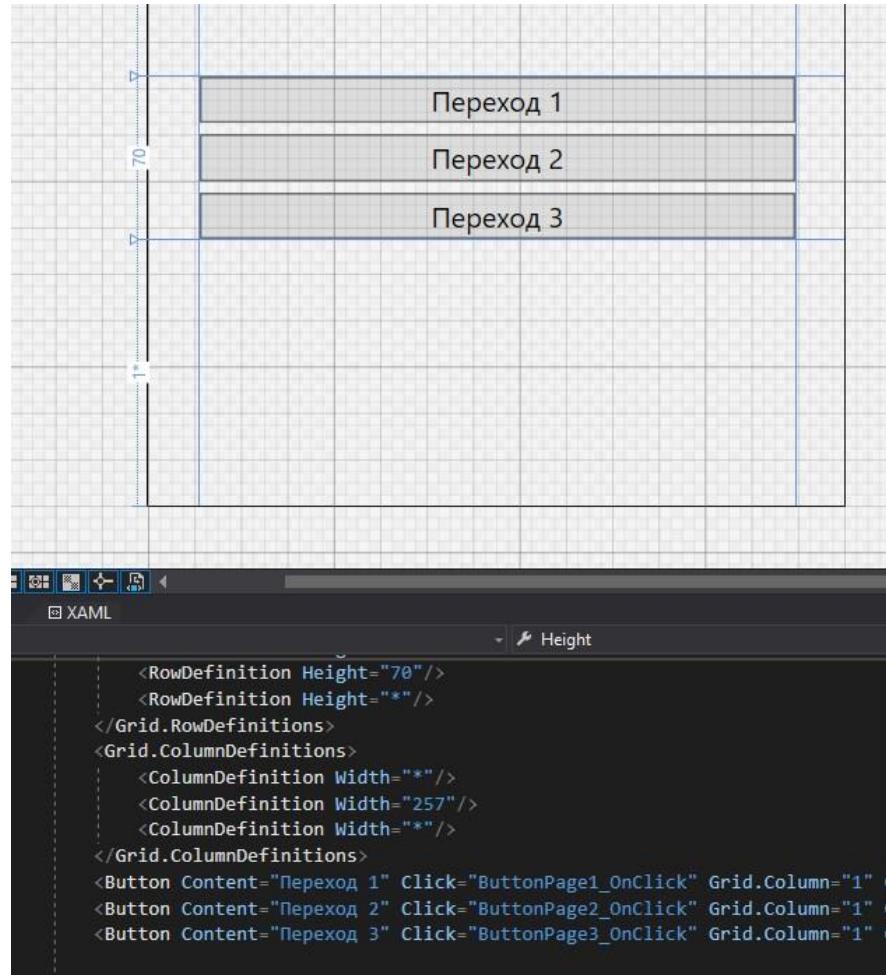


СОЗДАНИЕ МЕНЮ ПОЛЬЗОВАТЕЛЯ

Создадим новую страницу, назовем ее CustomerMenu.



Добавим несколько кнопок, разместим их по центру и добавим им обработчик события клик.



И в коде каждому обработчику укажем свой переход

```

private void ButtonPage1_OnClick(object sender, RoutedEventArgs e)
{
    NavigationService?.Navigate(new Page2());
}

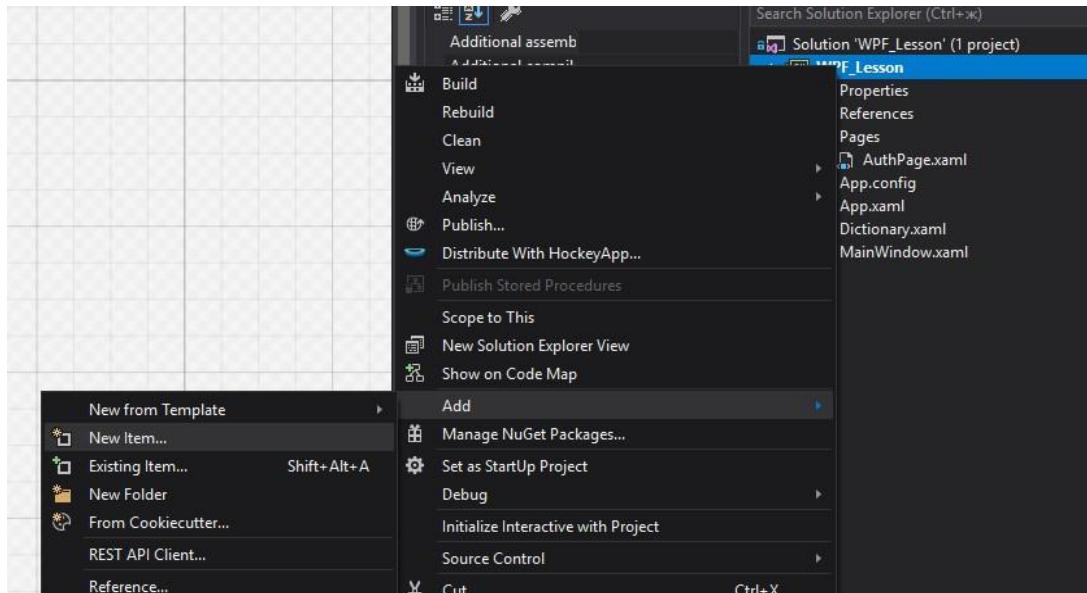
private void ButtonPage2_OnClick(object sender, RoutedEventArgs e)
{
    NavigationService?.Navigate(new AuthPage());
}

private void ButtonPage3_OnClick(object sender, RoutedEventArgs e)
{
    NavigationService?.Navigate(new Menu());
}

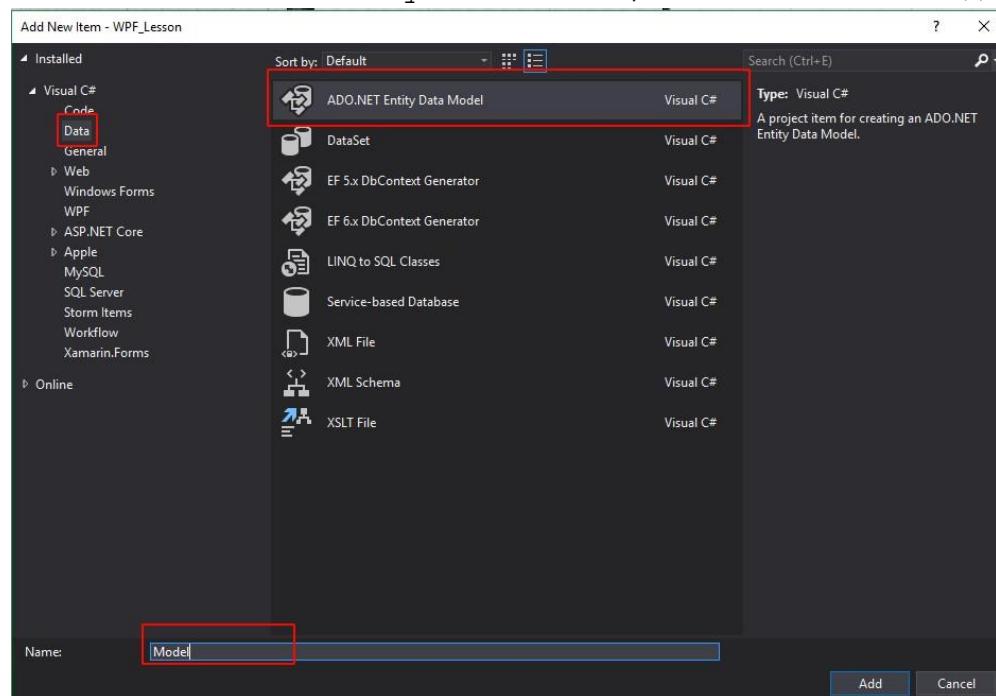
```

СОЗДАНИЕ ПОДКЛЮЧЕНИЯ К БАЗЕ ДАННЫХ

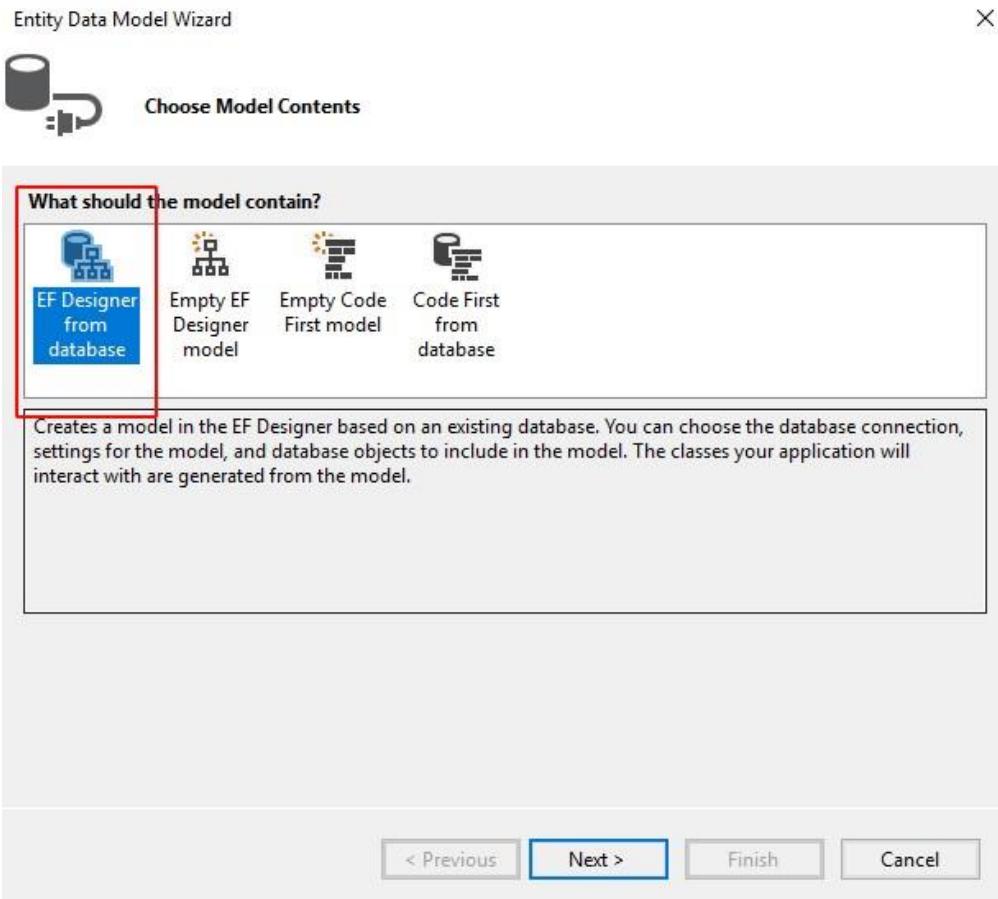
Добавим в проект новый объект



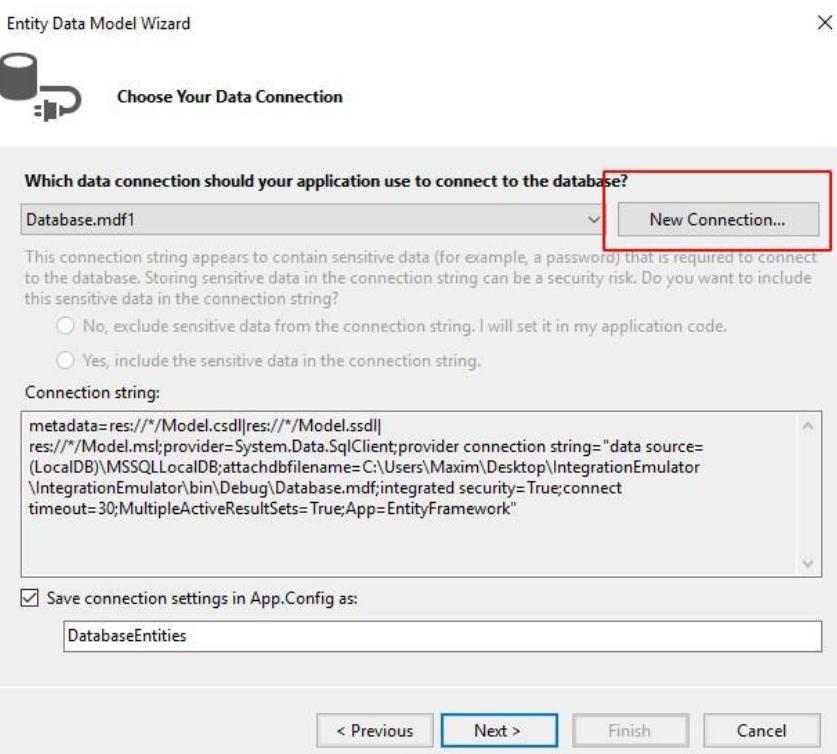
Выберем Data - ADO.NET Entity Data Model, назовем Model и добавим.



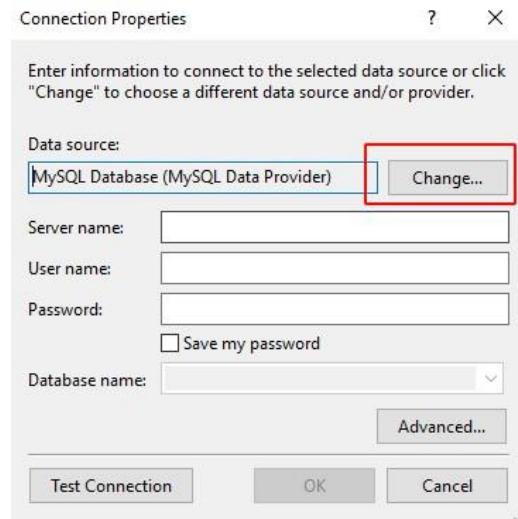
Появится диалоговое окно и выберем пункт «EF Designer from database»



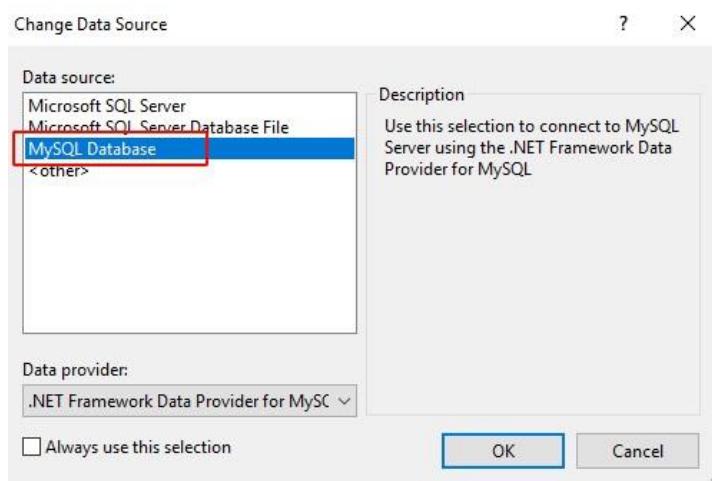
После появится следующее окно. Нажмем «New Connection»



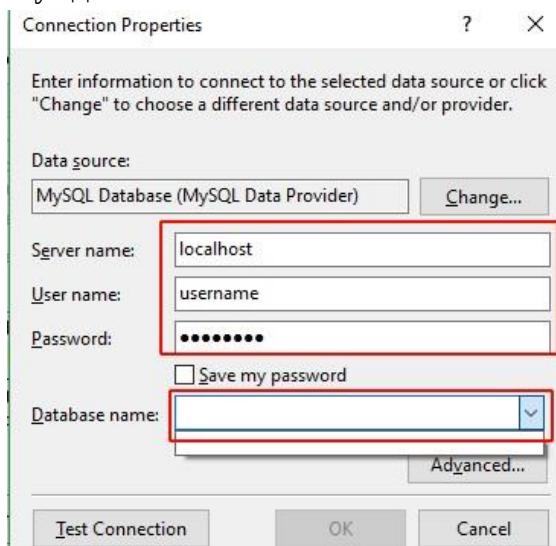
В данном окне выберем «Change».



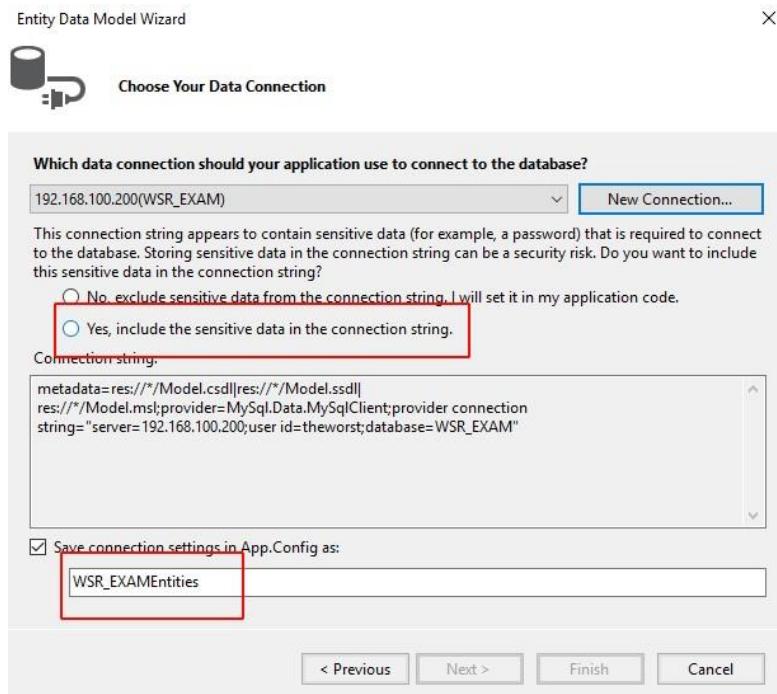
В следующем окне выберем «MySQL Database» и нажмем «OK».



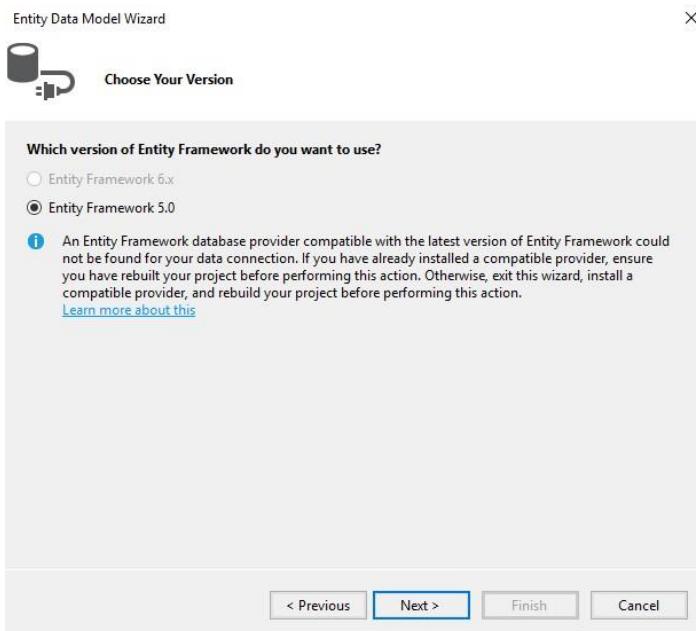
После выбора введем данные сервера: IP адрес, логин и пароль.
После выберем нужную базу данных.



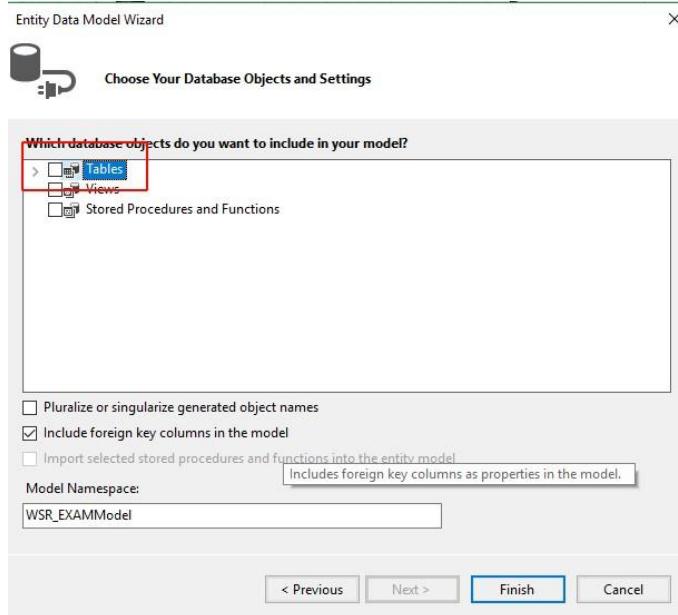
На следующем окне, выберем созданное подключение и поставим «Yes, include the sensitive data in the connection string», а также укажем имя нашего объекта базы данных «Entitites».



В следующем окне оставим по умолчанию:



В этом окне выберем таблицы, которые хотим использовать в проекте:



ПОЛУЧЕНИЕ ДАННЫХ ИЗ БАЗЫ ДАННЫХ

Для подключения к базе данных необходимо создать контекст.

```
using (var db = new Entities())
{
}
```

Теперь загрузим всю таблицу пользователей.

```
using (var db = new Entities())
{
    var users = db.User.AsNoTracking().ToList();
}
```

Получим пользователей по определенному критерию

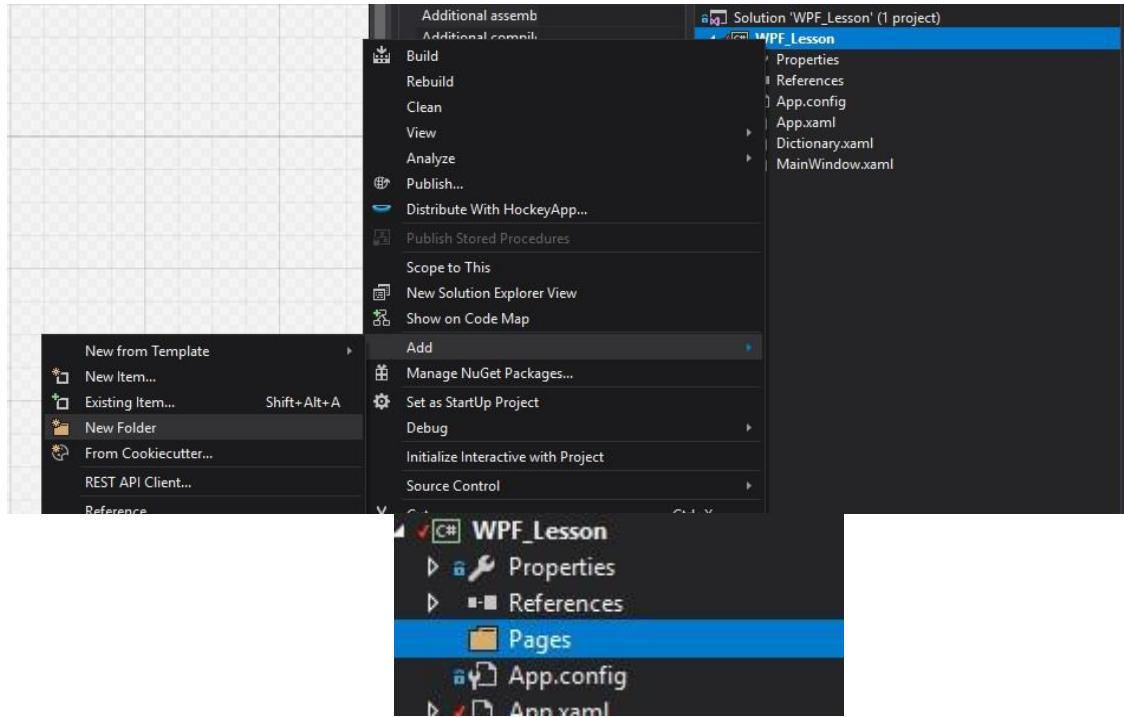
```
using (var db = new Entities())
{
    var users = db.User.AsNoTracking().Where(u => u.Login.StartsWith("max")).ToList();
}
```

Получим пользователя по определенным критериям

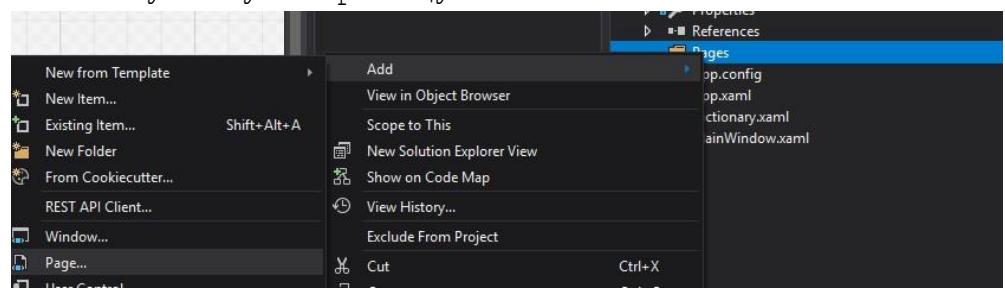
```
using (var db = new Entities())
{
    var user = db.User.AsNoTracking().FirstOrDefault(u => u.Login == "max" && u.Password == "test");
}
```

Создание формы авторизации

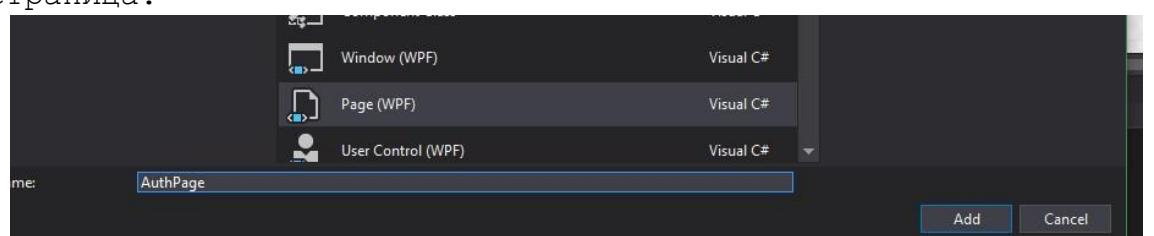
Добавим в проект новую папку и назовем ее «Pages», в этой папке будут находиться страницы (авторизация, регистрация и другие)

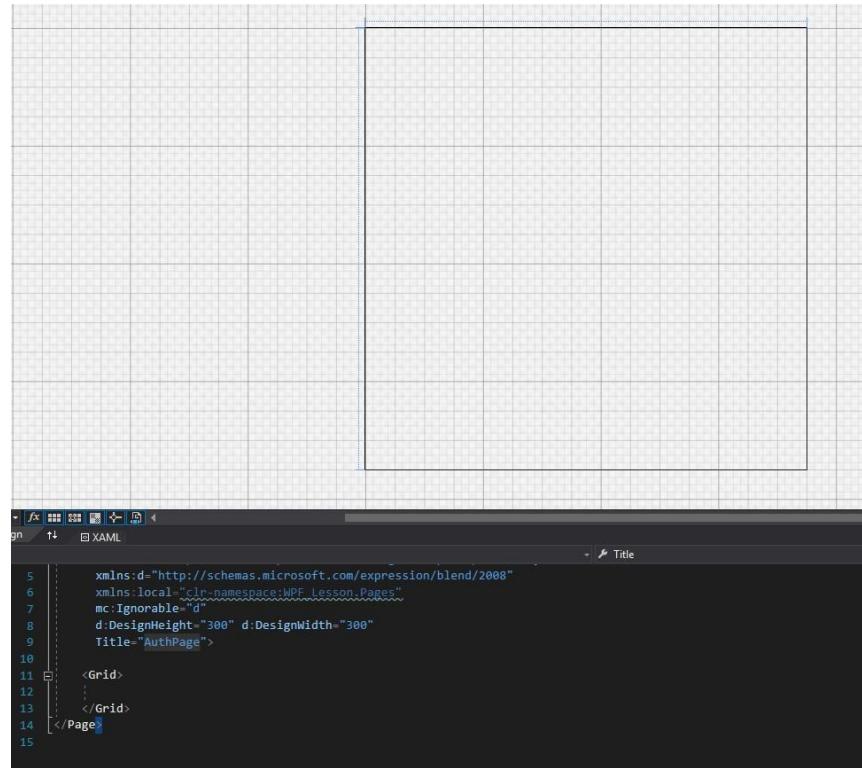


Добавим в папку новую страницу.

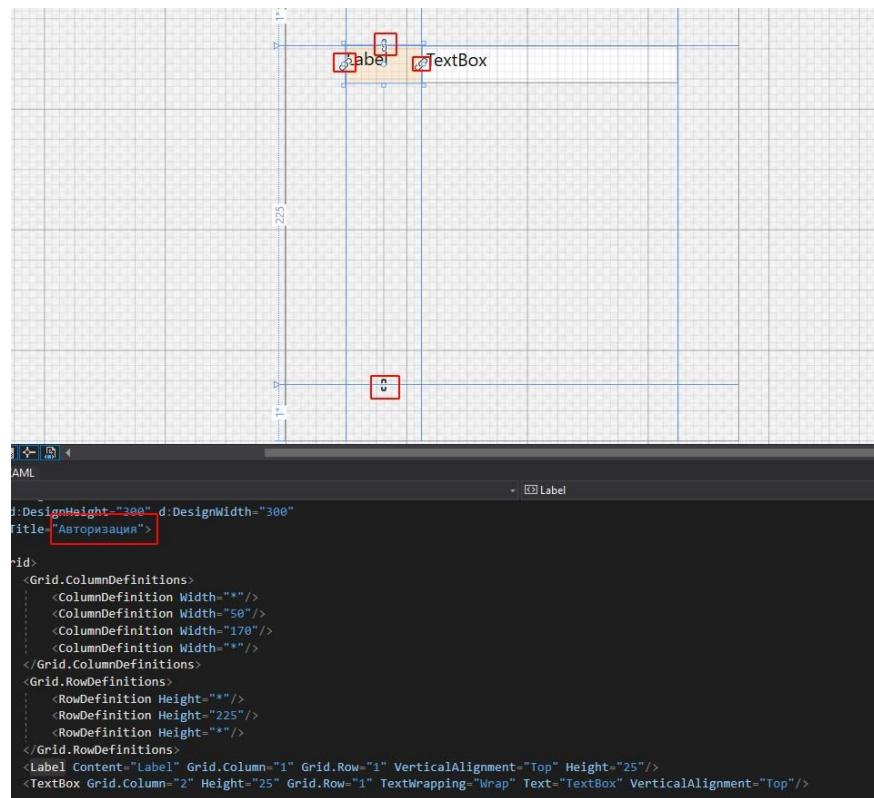


Назовем страницу AuthPage и создадим ее. После чего появится пустая страница.

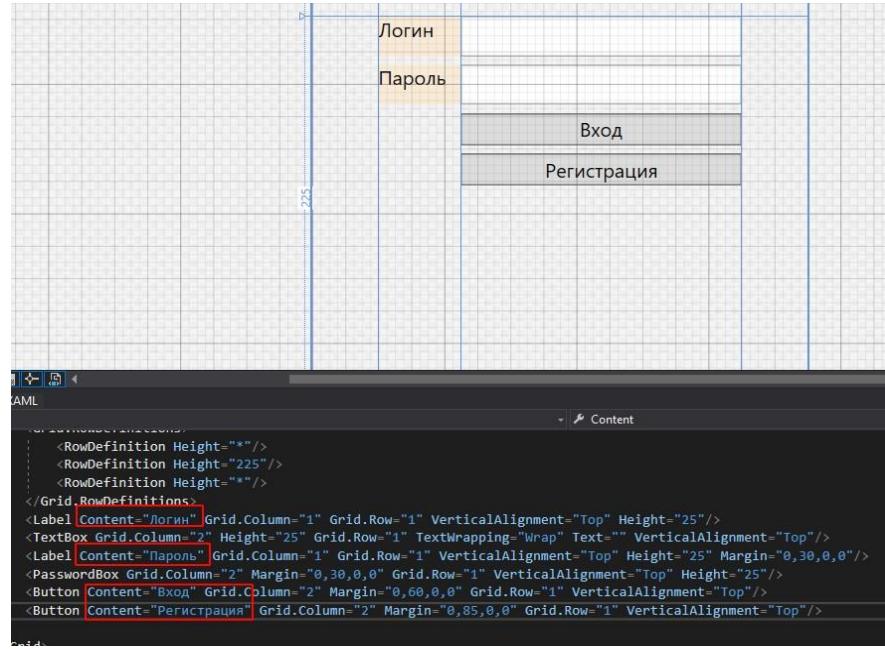




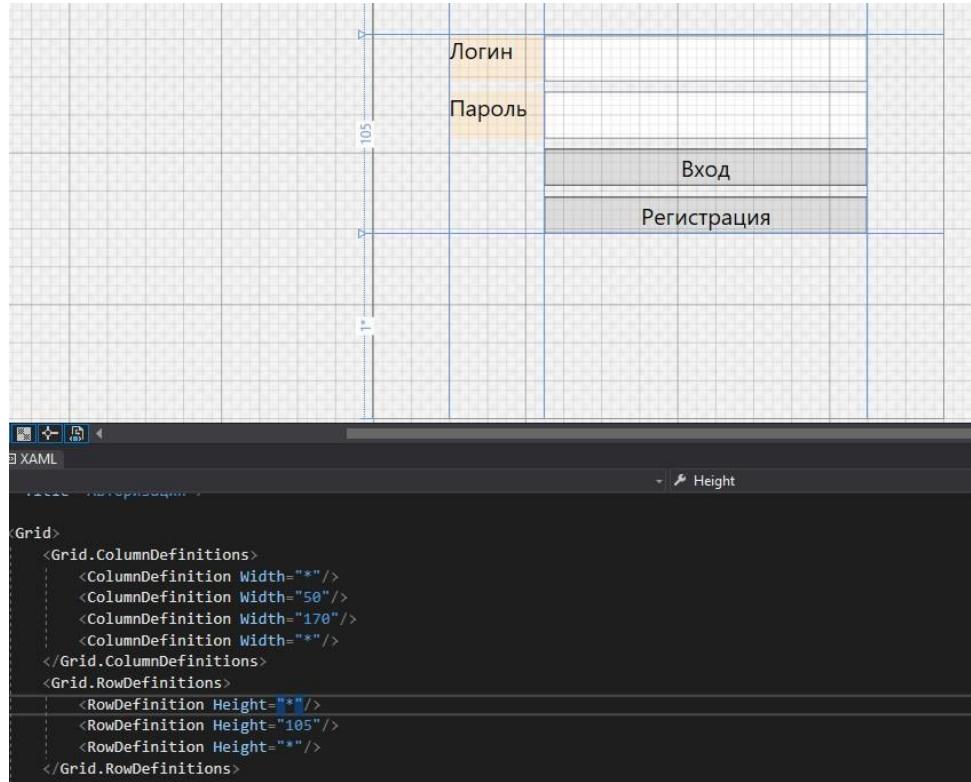
Добавим на форму компоненты «Label» и «TextBox», а затем отцентруем их и добавим еще один столбец (для размещения лейблов). Также не забудем переименовать страницу, а затем закрепить лейбл и текстбокс, чтобы они растягивались по ширине.



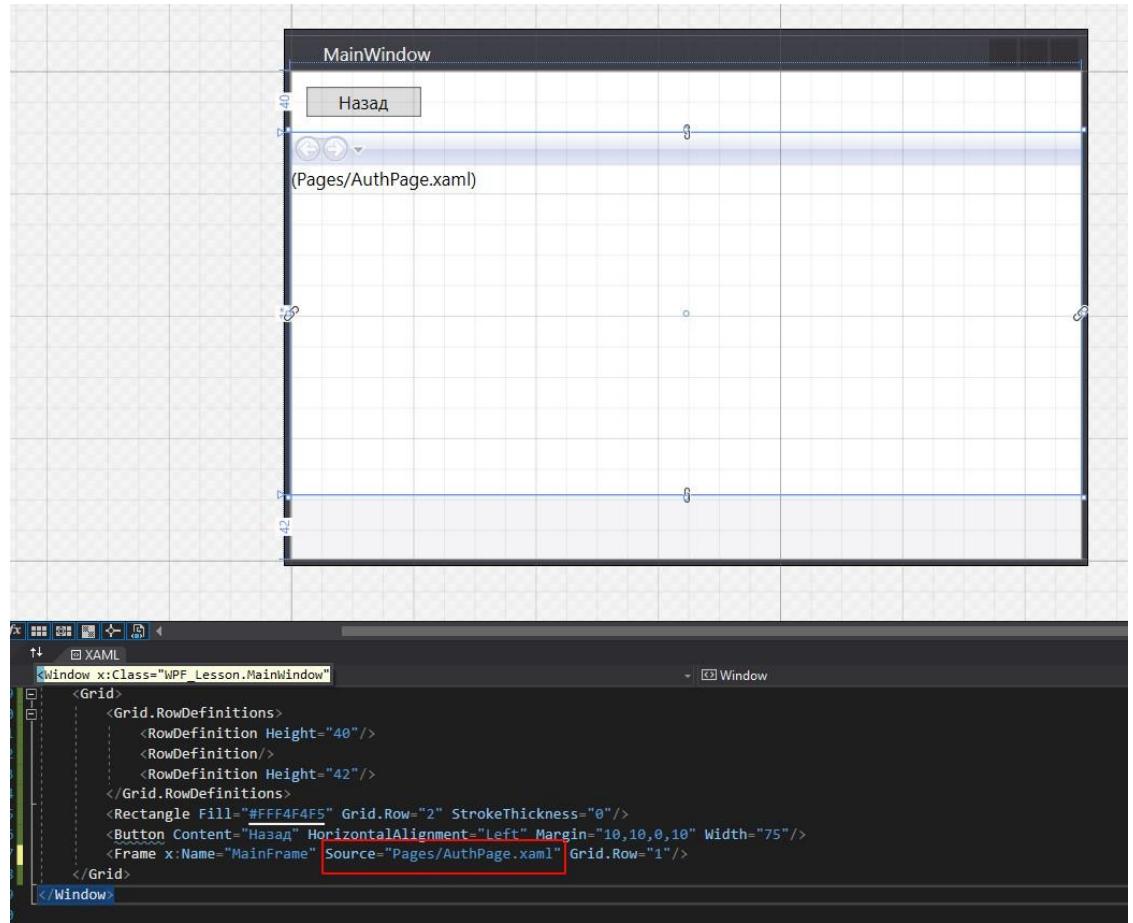
Добавим остальные компоненты по аналогии и переименуем их.



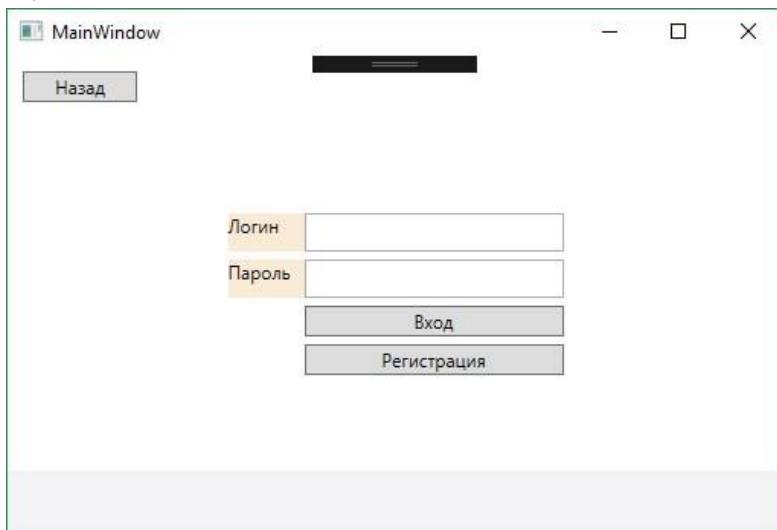
В результате получится примерно так.



Перейдем на главную форму и в компоненте «Frame» укажем в свойстве «Source» нашу страницу «AuthPage.xaml»



Запустим проект и увидим, что при запуске теперь отображается страница авторизации.



Теперь добавим функционал. Добавим обработчик события на кнопку ВХОД.

```
<PasswordBox Grid.Column="2" Margin="0,85,0,0" Grid.Row="1" VerticalAlignment="Top" Height="20"/>
<Button Click="ButtonEnter_OnClick" Content="Вход" Grid.Column="2" Margin="0,60,0,0" Grid.Row="1" VerticalAlignment="Top" Height="20"/>
<Button Content="Регистрация" Grid.Column="2" Margin="0,85,0,0" Grid.Row="1" VerticalAlignment="Top" Height="20"/>
```

0 references | 0 changes | 0 authors, 0 changes

```
private void ButtonEnter_OnClick(object sender, RoutedEventArgs e)
```

Добавим полям имена:

```
</Grid.RowDefinitions>
<Label Content="Логин" Grid.Column="1" Grid.Row="1" VerticalAlignment="Top" Height="20"/>
<TextBox x:Name="TextBoxLogin" Grid.Column="2" Height="20"/>
<Label Content="Пароль" Grid.Column="1" Grid.Row="1" VerticalAlignment="Top" Height="20"/>
<PasswordBox x:Name="PasswordBox" Grid.Column="2" Margin="0,60,0,0" Grid.Row="1" VerticalAlignment="Top" Height="20"/>
<Button Click="ButtonEnter_OnClick" Content="Вход" Grid.Column="2" Margin="0,85,0,0" Grid.Row="1" VerticalAlignment="Top" Height="20"/>
<Button Content="Регистрация" Grid.Column="2" Margin="0,85,0,0" Grid.Row="1" VerticalAlignment="Top" Height="20"/>
```

</Grid>
ge>

Добавим в код базовую проверку

```
0 references | 0 changes | 0 authors, 0 changes
```

```
private void ButtonEnter_OnClick(object sender, RoutedEventArgs e)
```

{

```
    if (string.IsNullOrEmpty(TextBoxLogin.Text) || string.IsNullOrEmpty>PasswordBox.Password))
```

{

```
        MessageBox.Show("Введите логин и пароль!");
```

return;

}

}

Добавим запрос к базе данных:

```
0 references | 0 changes | 0 authors, 0 changes
```

```
private void ButtonEnter_OnClick(object sender, RoutedEventArgs e)
```

{

```
    if (string.IsNullOrEmpty(TextBoxLogin.Text) || string.IsNullOrEmpty>PasswordBox.Password))
```

{

```
        MessageBox.Show("Введите логин и пароль!");
```

return;

}

```
    using (var db = new Entities())
```

{

```
        var user = db.User
            .AsNoTracking()
            .FirstOrDefault(u => u.Login == TextBoxLogin.Text && u.Password == PasswordBox.Password);
```

```
        if (user == null)
        {
            MessageBox.Show("Пользователь с такими данными не найден!");
            return;
        }
    }
```

И теперь добавим переходы в зависимости от роли на меню пользователя (для этого необходимо создать страницы меню для каждого типа пользователя, CustomerMenu или DirectorMenu и тд)

```
private void ButtonEnter_OnClick(object sender, RoutedEventArgs e)
{
    if (string.IsNullOrEmpty(TextBoxLogin.Text) || string.IsNullOrEmpty(PasswordBox.Password))
    {
        MessageBox.Show("Введите логин и пароль!");
        return;
    }

    using (var db = new Entities())
    {
        var user = db.User
            .AsNoTracking()
            .FirstOrDefault(u => u.Login == TextBoxLogin.Text && u.Password == PasswordBox.Password);

        if (user == null)
        {
            MessageBox.Show("Пользователь с такими данными не найден!");
            return;
        }

        MessageBox.Show("Пользователь успешно найден!");
        // Переход на меню пользователя в зависимости от роли

        switch (user.Role)
        {
            case "Заказчик":
                NavigationService?.Navigate(new Menu());
                break;
            case "Директор":
                NavigationService?.Navigate(new Menu());
                break;
        }
    }
}
```

ПЕРЕХОД МЕЖДУ СТРАНИЦАМИ

Перейдем на базовую форму и создадим обработчик события Navigated у Frame.

```
<Rectangle Fill="#FFF4F4F5" Grid.Row=2 StrokeThickness=0 />
<Button Content="Назад" HorizontalAlignment="Left" Margin="10,10,0,10" Width="75"/>
<Frame x:Name="MainFrame" Source="Pages/AuthPage.xaml" Grid.Row=1 Navigated="MainFrame_OnNavigated"/>
```

В обработчик события напишем следующий код. Сначала мы проверяем что получили ли мы страницу на вход, затем устанавливаем заголовок формы в соответствии с шаблоном, после в зависимости от страницы отображаем или скрываем кнопку «Назад».

```

- references | 0 changes | 0 authors, 0 changes
private void MainFrame_OnNavigated(object sender, NavigationEventArgs e)
{
    if (!(e.Content is Page page)) return;
    this.Title = $"LESSON - {page.Title}";

    if (page is AuthPage)
    {
        ButtonBack.Visibility = Visibility.Hidden;
    }
    else
    {
        ButtonBack.Visibility = Visibility.Visible;
    }
}

```

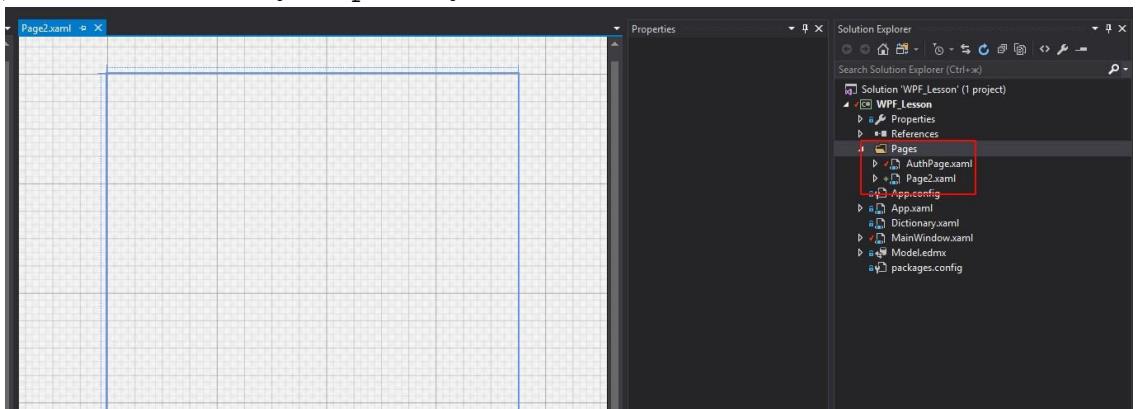
А в обработчик события нажатия кнопки «Назад». При нажатии на кнопку будет выполнен переход назад, если такой возможен.

```

- references | 0 changes | 0 authors, 0 changes
private void ButtonBack_OnClick(object sender, RoutedEventArgs e)
{
    if(MainFrame.CanGoBack) MainFrame.GoBack();
}

```

Добавим еще одну страницу.



Теперь выполним переход со страницы AuthPage на Page2. Перейдем на страницу AuthPage и добавим обработчик события на кнопку «Регистрация»:

```
<Button Content="Регистрация" Grid.Column="2" Margin="0,85,0,0" Grid.Row="1" VerticalAlignment="Top" Height="20" Click="ButtonRegistration_OnClick"/>
```

Напишем следующий код в обработчике:

```

- references | 0 changes | 0 authors, 0 changes
private void ButtonRegistration_OnClick(object sender, RoutedEventArgs e)
{
    NavigationService?.Navigate(new Page2());
}

```

После этого переход по кнопке будет осуществляться на страницу Page2, а по кнопке «Назад» обратно на AuthPage.

Тестирование программных решений

Тестирование ПО – процесс проверки соответствия заявленных к продукту требований и реально реализованной функциональности, осуществляется путем наблюдения за его работой в искусственно созданных ситуациях и на ограниченном наборе тестов, выбранных определенным образом. Тестирование (software testing) – деятельность, выполняемая для оценки и улучшения качества программного обеспечения [1]. Эта деятельность, в общем случае, базируется на обнаружении дефектов и проблем в программных системах.

В соответствие с IEEE Std 829-1998 тестирование – это процесс анализа ПО, направленный на выявление отличий между его реально существующими и требуемыми свойствами (дефект) и на оценку свойств ПО.

По ГОСТ Р ИСО МЭК 12207-99 в жизненном цикле ПО определены среди прочих вспомогательные процессы верификации, аттестации, совместного анализа и аудита.

Процесс верификации является процессом определения того, что программные продукты функционируют в полном соответствии с требованиями или условиями, реализованными в предшествующих работах. Данный процесс может включать анализ, проверку и испытание (тестирование).

Процесс аттестации предусматривает определение полноты соответствия требований и системы их конкретному функциональному назначению. Под аттестацией понимается подтверждение и оценка достоверности проведенного тестирования ИС. Аттестация должна гарантировать полное соответствие спецификациям, требованиям и документации. Аттестацию выполняют путем тестирования во всех возможных ситуациях и используют при этом независимых специалистов.

Процесс совместного анализа является процессом оценки состояний и, при необходимости, результатов работ (продуктов) по проекту.

Процесс аудита является процессом определения соответствия требованиям, планам и условиям договора. В сумме эти процессы и составляют то, что обычно называют тестированием.

Тестирование основывается на тестовых процедурах с конкретными входными данными, начальными условиями и ожидаемым результатом, разработанными для определенной цели, такой, как проверка отдельной программы или верификация соответствия на определенное требование [2]. Тестовые процедуры могут проверять различные аспекты функционирования программы – от правильной работы отдельной функции до адекватного выполнения бизнес-требований.

Основные цели тестирования:

- проверить взаимодействие между объектами;
- проверить корректную интеграцию всех модулей системы;
- проверить, что все требования были корректно реализованы;
- идентифицировать дефекты и убедиться, что они максимально выявлены еще до развертывания системы.

Этапы процесса тестирования

На рисунке 1 представлены этапы процесса тестирования программного обеспечения.



Рисунок 1 – Этапы процесса тестирования

Планирование и подготовка процесса тестирования состоит в анализе требований, предъявляемых к программному продукту, выборе стратегии тестирования, целей и приоритетов.

Создание тест-кейсов. Тест-кейсы должны быть основаны на требованиях к программному продукту, должны покрывать все эти требования и иметь приоритет.

Проверка на критические ошибки, блокирующие процесс тестирования. Если такие ошибки были обнаружены, то программу сразу отправляют на исправление разработчикам. Обычно такие ошибки не заносятся в базу данных дефектов.

Выполнение тест-кейсов – проверка соответствия результатов работы программы ожидаемым результатам.

Описание дефектов. Дефект – выявленное в процессе тестирования несоответствие полученных и ожидаемых результатов.

Проверка и устранение дефектов. Жизненный цикл дефекта представлен на рисунке 2.

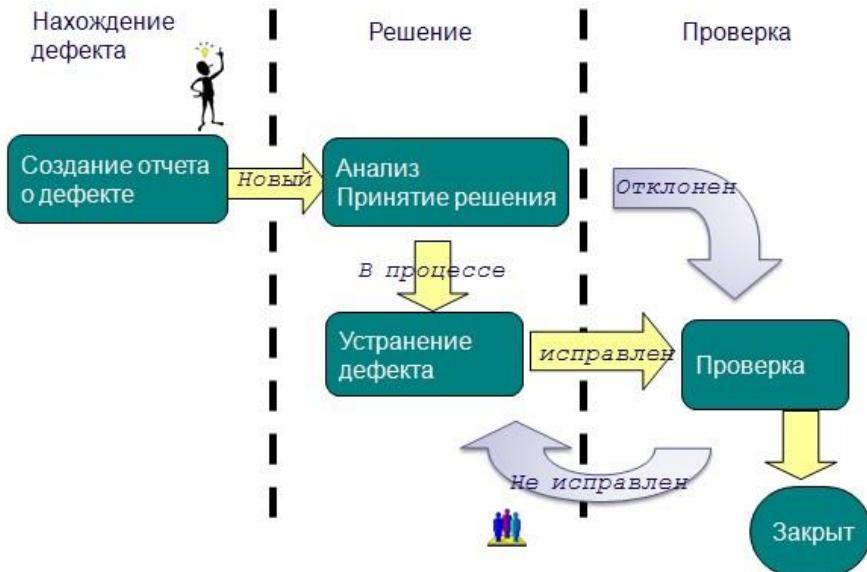


Рисунок 2 – Жизненный цикл дефекта

Автоматизация тестирования выполняется для того, чтобы уже написанные и проверенные один раз тест-кейсы выполнялись автоматически. Повторное выполнение тестов необходимо, чтобы убедиться, что во время исправления дефектов не было внесено новых ошибок. В настоящее время существует много программных продуктов, предназначенных для автоматизации тестирования. Идея этих продуктов заключается в создании централизованного репозитория для хранения, доступа и управления всеми составляющими компонентами процесса тестирования. Именно с использованиями такого инструментария, как правило, и начинается переход от тестирования вручную к внедрению автоматизированных средств. Одно из важных требований к инструменту подобного класса – возможность использования обычного браузера в качестве клиентской части, что упрощает установку, настройку и последующую поддержку продукта [3].

Отчет о тестировании. Отчеты о тестировании могут формироваться в различных точках в течение процесса тестирования. Отчеты о тестировании будут суммировать результаты тестирования и документировать любой анализ. Отчет о приемочном тестировании часто является договорным документом, подтверждающим приемку ПО.

Существует несколько признаков, по которым принято производить классификацию видов тестирования. Обычно выделяют следующие признаки [4] :

По объекту тестирования:

- Функциональное тестирование (functional testing);
- Нагрузочное тестирование:
 - Тестирование производительности (perfomance/stress testing);
 - Тестирование стабильности (stability/load testing);
- Тестирование удобства использования (usability testing);
- Тестирование интерфейса пользователя (UI testing);

- Тестирование безопасности (security testing); □ Тестирование локализации (localization testing); □ Тестирование совместимости (compatibility testing).

По знанию системы:

- Тестирование чёрного ящика (black box)

При тестировании чёрного ящика, инженер по тестированию имеет доступ к ПО только через те же интерфейсы, что и заказчик или пользователь, либо через внешние интерфейсы, позволяющие другому компьютеру либо другому процессу подключиться к системе для тестирования. Например, тестирующий модуль может виртуально нажимать клавиши или кнопки мыши в тестируемой программе с помощью механизма взаимодействия процессов, с уверенностью в том, все ли идёт правильно, что эти события вызывают тот же отклик, что и реальные нажатия клавиш и кнопок мыши. Как правило, тестирование чёрного ящика ведётся с использованием спецификаций или иных документов, описывающих требования к системе.

- Тестирование белого ящика (white box)

При тестировании белого ящика (англ. *white-box testing*, также говорят – прозрачного ящика), разработчик теста имеет доступ к исходному коду программ и может писать код, который связан с библиотеками тестируемого ПО. Это типично для юнит-тестирования (англ. *unit testing*), при котором тестируются только отдельные части системы. Оно обеспечивает то, что компоненты конструкции – работоспособны и устойчивы, до определённой степени. При тестировании белого ящика используются метрики покрытия кода.

- Тестирование серого ящика (gray box) . По степени автоматизированности:
- Ручное тестирование (manual testing);
- Автоматизированное тестирование (automated testing); □ Полуавтоматизированное тестирование (semiautomated testing).

По степени изолированности компонентов:

- Компонентное (модульное) тестирование (component/unit testing);
- Интеграционное тестирование (integration testing); □ Системное тестирование (system/end-to-end testing).

По времени проведения тестирования:

- Альфа тестирование (alpha testing):
 - Тестирование при приёмке (smoke testing);
 - Тестирование новых функциональностей (new feature testing);
 - Регрессионное тестирование (regression testing); ○ Тестирование при сдаче (acceptance testing).

В альфа-тестировании программу тестирует разработчик с точки зрения пользователя. Ближе к выходу продукта наступает следующая стадия тестирования – бета-тестирование, когда на тестирование программу отдают реальным, конечным пользователям.

- Бета тестирование (beta testing)

В бета-версии обычно ограничивают функциональность, время использования программы, взамен обратной связи, а также найденных ошибок.

По признаку позитивности сценариев:

- Позитивное тестирование (positive testing)
- Негативное тестирование (negative testing) **По степени подготовленности к тестированию:**
- Тестирование по документации (formal testing)
- Эд Хок (интуитивное) тестирование (ad hoc testing)

Статическое и динамическое тестирование

Описанные выше техники – тестирование белого ящика и тестирование чёрного ящика – предполагают, что код исполняется, и разница состоит лишь в той информации, которой владеет инженер по тестированию. В обоих случаях это динамическое тестирование.

При статическом тестировании программный код не выполняется – анализ программы происходит на основе исходного кода, который вычитывается вручную, либо анализируется специальными инструментами. В некоторых случаях, анализируется не исходный, а промежуточный код (такой как байт-код или код на MSIL) [5].

На рисунке 3 представлена классификация техник тестирования программного обеспечения.



Рисунок 3 – Техники тестирования программного обеспечения

Также к статическому тестированию относят тестирование требований, спецификаций, документации.

Тестирование обычно производится на протяжении всей разработки и сопровождения на разных уровнях. Уровень тестирования определяет «над чем» производятся тесты: над отдельным модулем, группой модулей или системой, в целом. При этом ни один из уровней тестирования не может считаться приоритетным. Важны все уровни тестирования, вне зависимости от используемых моделей и методологий [6].

Модульное тестирование (Unit testing). Этот уровень тестирования позволяет проверить функционирование отдельно взятого элемента системы. Что считать элементом – модулем системы определяется контекстом. Наиболее полно данный вид тестов описан в стандарте IEEE

1008-87 «Standard for Software Unit Testing», задающем интегрированную концепцию систематического и документированного подхода к модульному тестированию.

Интеграционное тестирование (Integration testing). Данный уровень тестирования является процессом проверки взаимодействия между программными компонентами/модулями.

Системное тестирование (System testing). Системное тестирование охватывает целиком всю систему. Большинство функциональных сбоев должно быть идентифицировано еще на уровне модульных и интеграционных тестов. В свою очередь, системное тестирование, обычно фокусируется на нефункциональных требованиях - безопасности, производительности, точности, надежности т.п. На этом уровне также тестируются интерфейсы к внешним приложениям, аппаратному обеспечению, операционной среде и т.д.

Рассмотрим подробнее каждый из уровней тестирования:

Модульное (unit) тестирование [1] – проверка функционирования первого компонента системы, самого элементарного. Обычно берется самый минимально возможный для тестирования компонент, например, одна функция программы.

В данном виде тестирования используется метод «белого ящика». Обычно модульное тестирование выполняется программистами.

Цель этого вида тестирования – изолировать отдельные части программы, протестировать их и показать, что в отдельности они работоспособны.

Преимущество этого вида тестирования в том, что программисты довольно легко идут на изменение программы, не сопротивляясь нововведениям. Это объясняется тем, что протестировать отдельный модуль после изменения достаточно просто. В одном модуле получается достаточно маленький набор вариантов развития событий, и достаточно легко рассмотреть их все.

Этот вид тестирования помогает локализовать ошибку.

Локализовать ошибку – значит определить место, где содержится ошибка, т.е. в каком модуле, в какой функции она произошла. Когда мы находим ошибку, мы пытаемся установить, при каком наборе действий она возникает, и обобщить его. Затем пытаемся локализовать, т.е. определить в каком компоненте программы происходит сбой.

Юнит-тестирование не решит проблемы производительности, качества, безопасности, надежности. Поэтому остается еще целый ряд непроверенных параметров, для которых заданы определенные требования при разработке продукта. Проблемы взаимодействия компонентов этот вид тестирования также не решает.

Этот вид тестирования отдельно никогда не используют, только с другими видами тестирования.

На выходе этого вида тестирования мы получаем протестированные модули программы. Они в свою очередь подаются на вход следующего уровня тестирования – интеграционного тестирования.

Этот вид тестирования проверяет взаимодействие компонентов, взаимодействие модулей, которые общаются между собой. Модули передают

информацию друг другу и каким-то образом связаны. Вот на этой стыковке и могут быть сбои в передаче этой информации. Либо она может передаваться не точно, либо вообще не передаваться, либо искажаться при передаче. Поэтому тестировать взаимосвязь компонентов нужно обязательно.

Таким образом, мы получаем на входе протестированные модули, затем объединяем их в группы, тестируем эти группы. И на выходе получаем протестированные группы модулей.

Этот вид тестирования проводится через интерфейс программы – методом «черного ящика».

Также с помощью метода «черного ящика» тестируется следующий уровень.

На вход системного тестирования мы подаем выходные данные интеграционного тестирования, т.е. протестированные группы модулей.

На выходе же системного тестирования мы получаем полностью протестированную программу.

В системном тестировании мы проверяем всю систему целиком.

Какие ошибки обычно выявляются на этом уровне тестирования?

Это ошибки надежности, безопасности, производительности. Также на этом уровне тестируется интерфейс для внешнего окружения, например, доступ к другим программам, доступ к операционной системе, доступ к «железу» компьютера. Знания внутреннего устройства работы программы не требуются.

Таким образом, модульное тестирование – проводим методом «белого ящика», обнаруживаем ошибки функциональности. Затем проводим интеграционное тестирование – методом «черного ящика», обнаруживаем ошибки взаимодействия модулей программы. И, наконец, системное тестирование – методом «черного ящика», выявляем ошибки производительности, надежности, безопасности и других параметров.

В системном тестировании можно выделить два этапа, которые будут являться стадиями разработки продукта: альфа-тестирование, бетатестирование.

Выполнение задач процесса тестирования программных комплексов сопровождается разработкой различных артефактов (документов, моделей и других материалов проекта). Обычно разработка артефактов может проводиться в разной форме с разными требованиями к способу выполнения, рецензированию и качеству оформления.

Ниже представлены основные рабочие артефакты тестировщиков, в той или иной форме связанные со сценариями использования. Эти документы необходимо передавать заказчику или группе сопровождения и технической поддержки системы в случае необходимости.

План тестирования (Test plan). План тестирования определяется международным стандартом IEEE 829-1998. В нем должны быть предусмотрены как минимум три раздела содержащие, следующие описания:

- что будет тестируться (тестовые требования, тестовые варианты);

- какими методами и насколько подробно будет тестироваться система;
- план-график работ и требуемые ресурсы (персонал, техника) (*Schedule*).

Дополнительно описываются критерии удачного/неудачного завершения тестов, критерии окончания тестирования, риски, непредвиденные ситуации, приводятся ссылки на соответствующие разделы в основных документах проекта - план управления требованиями, план конфигурации [8].

Сценарий тестирования (*Test case, тест кейс*). Это один из основных документов, с которыми имеет дело тестировщик. По сути, упрощенное описание теста. То есть входной информации, условий и последовательности выполнения действий и ожидаемого выходного результата. Учитывая, что даже успешно прошедшие тесты выполняются неоднократно в ходе регрессионного тестирования, наличие таких описаний необходимо. Однако уровень формальных требований к их оформлению может меняться в очень широких пределах. Одно дело, если вы собираетесь использовать тесты в ходе приемочных испытаний, проводимых заказчиком, и другое – в ходе внутреннего тестирования коробочного продукта.

Тест скрипт (*Test script*). Обычно говорят о программной реализации теста, хотя скрипт может описывать и ручные действия, необходимые для выполнения конкретного тест кейса.

Набор тестов (*Test set*). Как правило, сценарии тестирования объединяются в пакеты или наборы. Во-первых, это просто способ группирования тестов со сходными задачами, а, во-вторых, в такой набор можно включать зависимые тесты, которые должны выполняться в определенном порядке (поскольку последующие тесты используют данные, сформированные в ходе выполнения предыдущих).

Список идей тестов. Использование списка идей тестов для анализа и проектирования системы сценариев использования существенно упрощает задачу разработки необходимого набора тестов. Основной объем тестов строится как проверка различных вариантов выполнения каждого сценария использования. Однако тесты не сводятся к сценариям использования, как и задачи тестирования не сводятся только лишь к проверке функциональных требований к системе. Проверка нефункциональных требований может потребовать использования специальных приемов и подходов. Соответствующие тесты не всегда очевидны. Для таких ситуаций и создается список идей тестов. В него все желающие могут записать «что и как» стоит еще проверить. Этот список является внутренним рабочим документом группы тестирования. Наиболее разумная форма его ведения – электронный документ с минимальными формальными требованиями к оформлению.

Модель нагрузки. Сценарии использования, как правило, описывают взаимодействие с системой одного пользователя, часто этого бывает мало. При тестировании систем необходимо учитывать возможность параллельной работы большого числа пользователей, решая различные задачи. Модель реальной нагрузки описывает характеристики типового

«потока заявок», которые должны использоваться для нагрузочного тестирования, имитирующего работу системы в реальных условиях. Также могут быть созданы стрессовые модели нагрузки для тестирования отказоустойчивости системы.

Дефекты(Defects). Основополагающие артефакты процесса тестирования – описывают обнаруженные факты несоответствия системы предъявляемым требованиям [9]. Являются одним из подтипов запросов на изменение, описывающих найденную ошибку или несоответствие на всех этапах тестирования. Хотя базу данных дефектов можно вести в текстовом файле или Excel таблице, предпочтительным является использования специализированного инструментального средства, которое позволяет передавать информацию об обнаруженных дефектах от тестировщиков к разработчикам, а в обратную сторону – сведения об устранении дефектов. А также формировать необходимые отчеты о тенденциях изменения количества обнаруживаемых и устраниемых дефектов.

Журнал тестирования. Каждое выполнение теста должно быть зарегистрировано в журнале тестирования. Журнал тестирования будет содержать записи о том, когда запускался каждый тест, итог выполнения каждого теста и может также содержать важные наблюдения, сделанные при выполнении теста. Зачастую журнал тестирования не ведут для нижних уровней тестирования (тестов компонент и интеграции ПО).

Отчеты о тестировании. Отчеты о тестировании могут формироваться в различных точках в течение процесса тестирования. Отчеты о тестировании будут суммировать результаты тестирования и документировать любой анализ. Отчет о приемочном тестировании часто является договорным документом, подтверждающим приемку ПО.

Функциональность

Функциональное тестирование объекта-тестирования планируется и проводится на основе требований к тестированию, заданных на этапе определения требований. В качестве требований выступают диаграммы usecase, бизнес-функции и бизнес-правила. Цель функциональных тестов состоит в том, чтобы проверить соответствие разработанных графических компонентов установленным требованиям. В основе функционального тестирования лежит методика «черного ящика» [3]. Идея тестирования сводится к тому, что группа тестировщиков проводит тестирование, не имея доступа к исходным текстам тестируемого приложения. При этом во внимание принимается только входящие требования и соответствие им тестируемым приложением.

Цель тестирования:

Убедиться в надлежащем функционировании объекта тестирования. Тестируется правильность навигации по объекту, а также ввод, обработка и вывод данных.

Методика:

Необходимо исполнить (проиграть) каждый из use-case, используя как верные значения, так и заведомо ошибочные, для подтверждения правильного функционирования, по следующим критериям:

- продукт адекватно реагирует на все вводимые данные (выводятся ожидаемые результаты в ответ на правильно вводимые данные);

- продукт адекватно реагирует на неправильно вводимые данные (появляются соответствующие сообщения об ошибках);
- каждое бизнес-правило реализовано надлежащим (установленным) образом.

Критерии Завершения: Все запланированные действия по тестированию выполнены.

Все найденные дефекты были соответствующим образом обработаны (документированы и помещены в базу дефектов). **Целостность данных и баз данных Цель Тестирования:**

Убедиться в надежности методов доступа к базам данных, в их правильном исполнении, без нарушения целостности данных.

Методика:

Необходимо последовательно испробовать максимально возможное число способов обращения к базе. Используется подход, при котором тест составляется таким образом, чтобы «нагрузить» базу последовательностью, как верных значений, так и заведомо ошибочных.

После этого необходимо оценить правильность внесения данных и убедиться в корректной обработке базой входящих значений.

Критерии Завершения:

Все способы доступа функционируют, в соответствии с требованиями.

Действия скрипта не приводят к потере данных или нарушению целостности базы, либо к другим неадекватным реакциям.

Пользовательский интерфейс

Цель Тестирования:

Проверить правильность навигации по объекту тестирования (в том числе межоконные переходы, переходы между полями, правильность обработки клавиш «enter» и «tab», работа с мышью, функционирование клавиш-акселераторов и полное соответствие индустриальным стандартам);

Проверить объекты и их характеристики (меню, размеры, положения, состояния, фокус ввода и др.) на соответствие общепринятым стандартам на графический интерфейс пользователя.

Методика:

Создаются или дорабатываются тесты для каждого из окон, на предмет соответствия навигации и состояний каждого из объектов.

Критерии завершения:

Каждое окно протестировано и удовлетворяет, базовой линии поведения, требованиям стандартов и не противоречит проектным требованиям. Все выявленные дефекты обработаны и документированы.

Описание процесса тестирования как этапа разработки программного обеспечения

Процесс тестирования - один из основных процессов общего процесса разработки программного обеспечения. Для того чтобы выработать правильный алгоритм процесса тестирования, определить стратегии тестирования, запланировать процесс тестирования заданного программного комплекса необходимо знать место процесса тестирования в общем процессе разработки программного обеспечения. Покажем место тестирования в общем процессе разработки программного обеспечения,

определим объекты тестирования, уровни тестирования на карте процесса тестирования, которая приведена на рисунке 4.

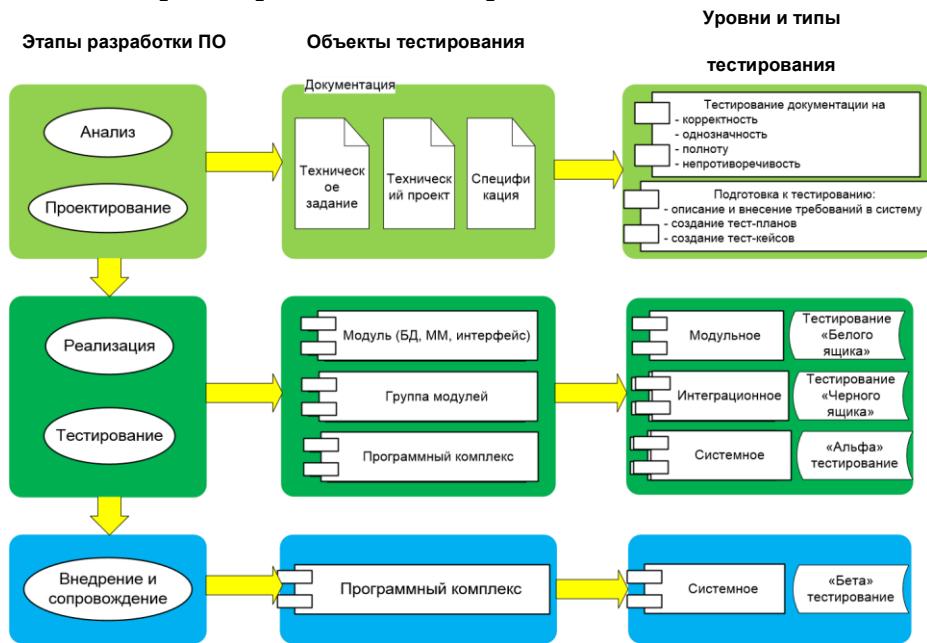


Рисунок 4 – Место тестирования в процессе разработки ПО

Анализируя карту процесса тестирования, мы видим, что процесс тестирования начинается на этапе проектирования и разработки технической документации на программный комплекс. Правильно построенный процесс тестирования не может начинаться на этапе программирования или внедрения и сопровождения. Только таким образом построенный процесс тестирования позволит на ранних стадиях разработки, а именно, на стадии разработки требований к программным комплексам, выявить дефекты, которые в будущем могут негативно сказаться на разрабатываемом программном комплексе. Объектами тестирования на этапе разработки требований к программному комплексу является техническая документация в виде технических заданий, спецификаций на отдельные модули или систему в целом, руководства пользователей. Этап тестирования заключается в регистрации и анализе требований к программному комплексу в системе поддержки процесса тестирования через модуль управления требованиями к ПО. Тестирование документации проводится на корректность, полноту, однозначность, непротиворечивость, тестируемость, упорядоченность, модифицируемость, отслеживаемость. На данном этапе процесса тестирования инженер по тестированию обращается к базе данных документов. В результате тестирования документации и ее анализа необходимо перейти к этапу планирования процесса тестирования конкретного модуля, интеграции модулей или системы в целом. Руководитель группы тестирования составляет план тестирования, определяет стратегии тестирования, при этом он работает с базой данных заданий на тестирование. Группа тестирования начинает проектирование тестов, подготовку тестовых данных, тестовой среды и окружения. Как только группа тестирования получает от разработчиков модуль, группу модулей или программный комплекс начинается этап выполнения тестов и регистрации дефектов.

При этом идет пополнение базы данных тестов и дефектов. Руководитель группы тестирования, в свою очередь, работая с базами данных тестов и дефектов через подсистему генерации отчетности, получает информацию о ходе процесса тестирования и анализирует состояние тестируемого программного комплекса. Как только выполняется условие завершения работ (закончилось время, отведенное на тестирование; закончились денежные средства, выделенные на тестирование проекта; найдены дефекты, неисправленными остались лишь незначительные дефекты), руководитель группы тестирования составляет отчет о тестировании и программный комплекс передается в эксплуатацию. Таким образом, происходит тестирование программного комплекса на всех трех уровнях тестирования: модульном, интеграционном, системном, изменяется лишь объект тестирования.

Модель работы с дефектами

Каждый обнаруженный дефект программного комплекса должен быть зарегистрирован и с ним должна быть проведена работа, которая в конечном итоге должна привести к исправлению дефекта. Работа с дефектами осуществляется по выбранной модели работы с дефектами. Выбор модели работы с дефектами зависит от конкретного проекта, структуры подразделения разработчиков и инженеров по тестированию. Для разработанной системы поддержки процесса тестирования была спроектирована модель работы с дефектами, которая представлена на рисунке 5:

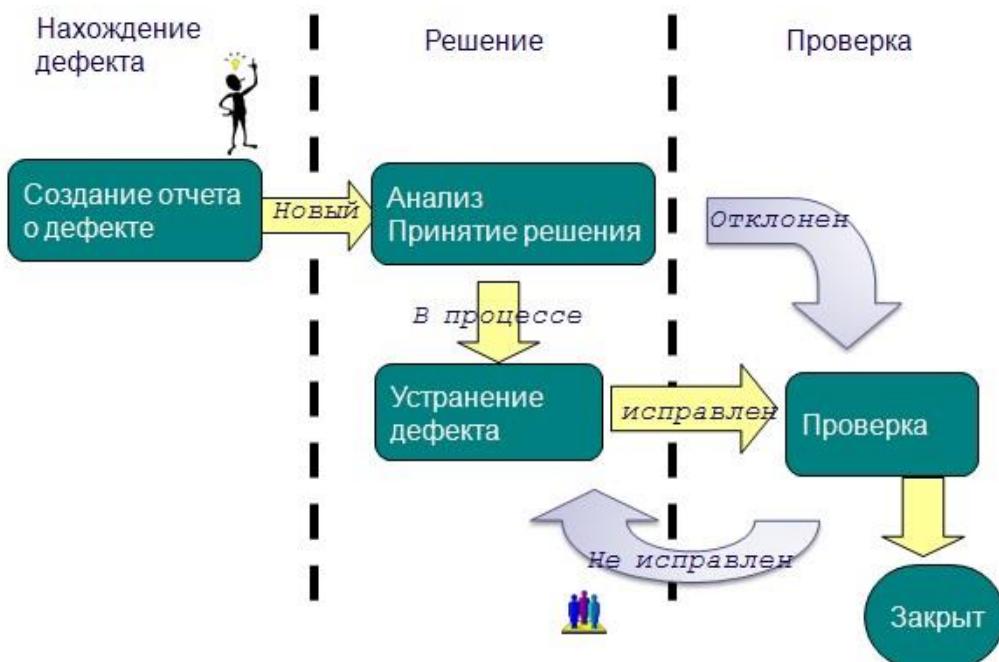


Рисунок 5 – Модель работы с дефектами

Как только дефект обнаружен, тестировщик регистрирует его в подсистеме работы с дефектами, присваивая ему статус «Новый» и направляет его на руководителя группы тестирования. Руководитель группы тестирования подтверждает дефект и направляет его на

руководителя группы разработчиков или направляет дефект на инженера по тестированию на доработку. Далее руководитель группы разработчиков направляет дефект на программиста, отвечающего за дефект, программист изучает суть дефекта и проставляет статус «В процессе». Если в данный момент необходимо отложить исправление дефекта, то разработчик может присвоить дефекту статус «Отложен». После исправления дефекта разработчик присваивает ему статус «Исправлен» и направляет дефект на руководителя группы тестировщиков, а тот направляет дефект на тестировщика. Инженер по тестированию проверяет дефект и если он исправлен, присваивает ему статус «Исправлен». Если дефект не исправлен, то ему присваивается статус «Не исправлен» и направляется на программиста для исправления.

Всем дефектам со статусом «Исправлен» руководитель группы тестирования присваивает статус «Закрыт».

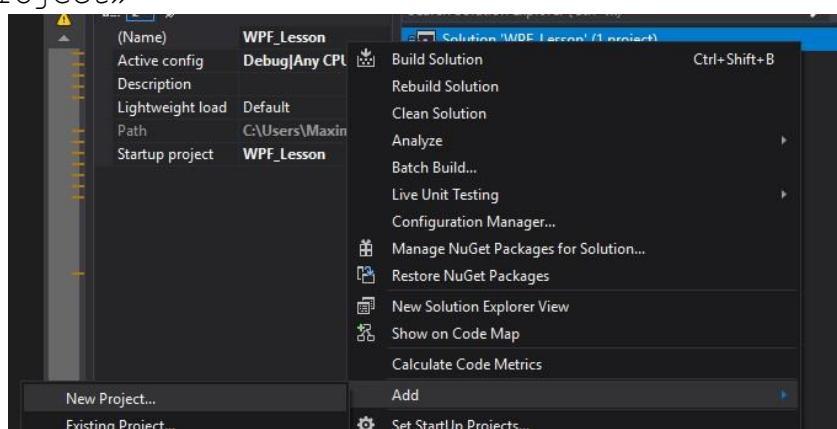
Кроме статусов атрибутами дефекта являются приоритет и важность. Приоритет дефекта показывает насколько быстро нужно исправить дефект, по приоритету определяется очередность выполнения задачи. Важность характеризует критичность дефекта по отношению к тестируемому приложению. Например, по признаку «важность» дефект может быть описан как «блокирующий» - не позволяющий приложению запуститься, «критичный» - некоторые функции приложения недоступны, «крупный» - какая-то функция не работает, «средний» - часть функции не работает и «второстепенный» - не влияющий на работу приложения (например, неправильное написание или неудобное расположение элементов управления). Приоритет в свою очередь может быть «высоким», «средним» и «низким».

Такая модель работы с дефектами позволяет полностью контролировать состояние тестируемого проекта, распределяет роли между участниками процесса тестирования и делает процесс работы с дефектами открытым и отслеживаемым как для тестировщиков, так и для разработчиков.

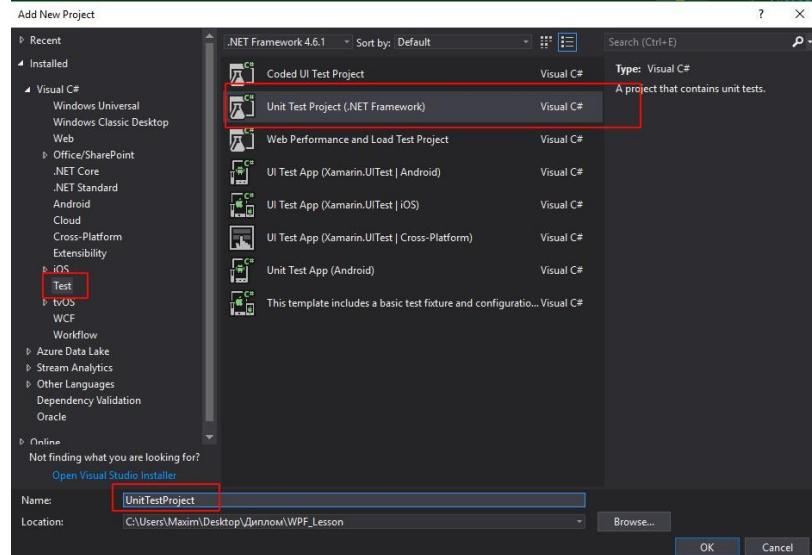
Рассмотрим пример создания unit-test.

СОЗДАНИЕ ПРОЕКТА ТЕСТИРОВАНИЯ

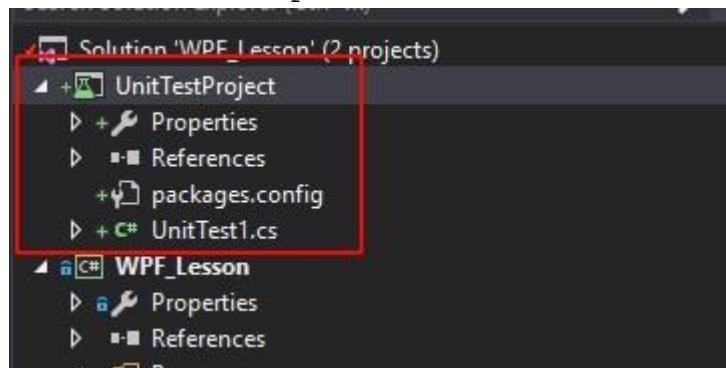
Откроем созданное решение, нажмем правой кнопкой мыши на решении и выберем «New Project»



Далее выберем Test - Unit Test Project.

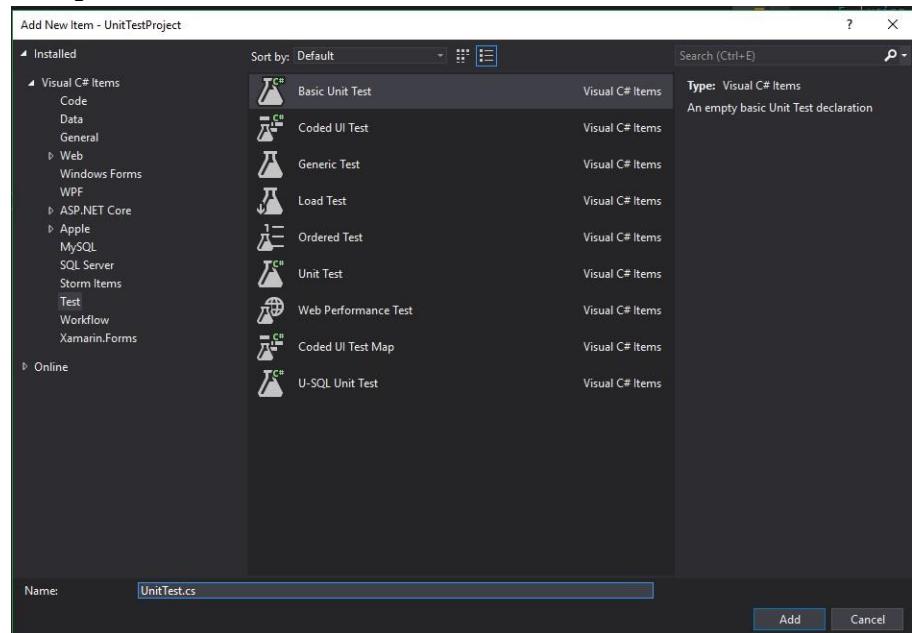


В решение добавится новый проект.



СОЗДАНИЕ ЮНИТ ТЕСТА

Добавим в проект базовый юнит-тест.



напишем первый тест. Тест проверяет различные варианты ошибок.

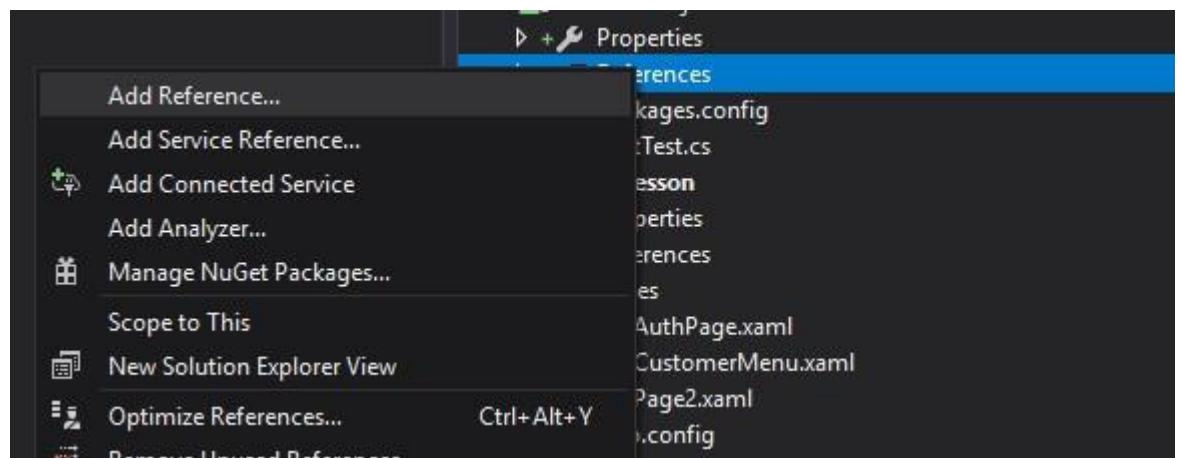
Страница 140 из 146

```
[TestClass]
0 references | 0 changes | 0 authors, 0 changes
public class UnitTest
{
    [TestMethod]
    0 references | 0 changes | 0 authors, 0 changes
    public void TestMethod1()
    {
        int res = 2 + 2;

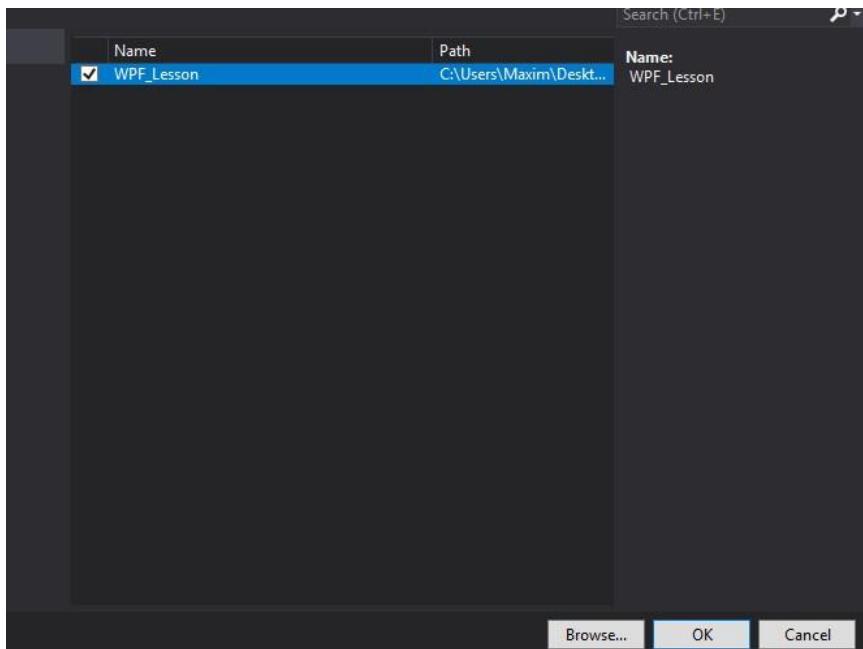
        Assert.AreEqual(res, 4);
        Assert.AreNotEqual(res, 5);
        Assert.IsFalse(res > 5);
        Assert.IsTrue(res < 5);

    }
}
```

Для тестирования проекта, подключим его к тестовому проекту. Для этого необходимо нажать правой кнопкой мыши по «References» и выбрать «Add Reference».



После чего выбрать нужный проект:



Для тестирования авторизации, нужно немного видоизменить форму авторизации:

```
1 reference | TheWorst, 1 hour ago | 1 author, 1 change
private void ButtonEnter_OnClick(object sender, RoutedEventArgs e)
{
    Auth(TextBoxLogin.Text, PasswordBox.Password);
}

1 reference | 0 changes | 0 authors, 0 changes
private bool Auth(string login, string password)
{
    if (string.IsNullOrEmpty(login) || string.IsNullOrEmpty(password))
    {
        MessageBox.Show("Введите логин и пароль!");
        return false;
    }

    using (var db = new Entities())
    {
        var user = db.User
            .AsNoTracking()
            .FirstOrDefault(u => u.Login == TextBoxLogin.Text && u.Password == PasswordBox.Password);

        if (user == null)
        {
            MessageBox.Show("Пользователь с такими данными не найден!");
            return false;
        }

        MessageBox.Show("Пользователь успешно найден!");
        // Переход на меню пользователя в зависимости от роли

        switch (user.Role)
        {
            case "Заказчик":
                NavigationService?.Navigate(new Menu());
                break;
            case "Директор":
                NavigationService?.Navigate(new Menu());
                break;
        }
    }

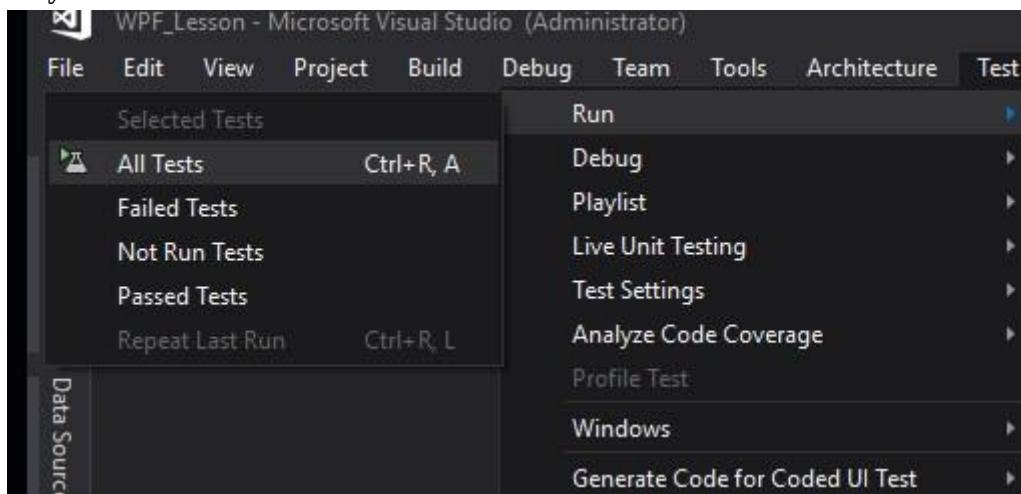
    return true;
}
```

Теперь можно начинать тестировать авторизацию. Создадим новый метод для проверки авторизации и напишем следующий код:

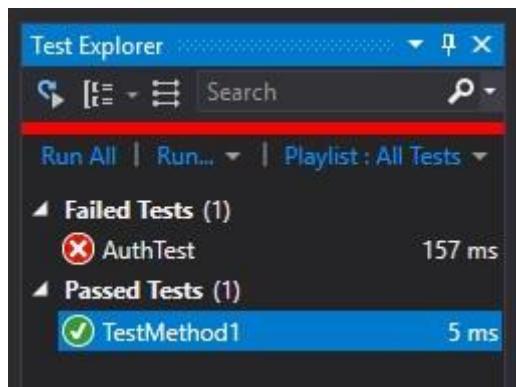
```
[TestMethod]
0 references | 0 changes | 0 authors, 0 changes
public void AuthTest()
{
    var page = new AuthPage();

    Assert.IsTrue(page.Auth("test", "test"));
    Assert.IsFalse(page.Auth("Test", "test"));
    Assert.IsFalse(page.Auth("test", "Test"));
    Assert.IsFalse(page.Auth("test1", "test"));
    Assert.IsFalse(page.Auth("test", "test1"));
    Assert.IsFalse(page.Auth("", ""));
    Assert.IsFalse(page.Auth(" ", ""));
    Assert.IsFalse(page.Auth("", " "));
    Assert.IsFalse(page.Auth("^^", "&&"));
    Assert.IsFalse(page.Auth("^^", "^^"));
    Assert.IsFalse(page.Auth("346456457456745745745", "12345464567658657"));
}
```

Запустим все тесты на выполнение:



После запуска можем увидеть какие тесты были пройдены, а какие нет.



Задание: Представьте себе, что ваша цель – тестирование приложения или сервиса, указанного в вашем варианте работы. Необходимо указать, какие тесты необходимы для покрытия различных видов, типов и областей тестирования,

представленных в таблице 1. При этом нет необходимости перечислять все тесты. Необходимо привести 2-3 конкретных примера тестов (см. пример выполнения работы). Таблица 1.

Тесты	Пример тестов
Различные виды тестирования	
Функциональное тестирование (Functional testing)	
Тестирование производительности (Performance testing)	
Нагрузочное тестирование (Load testing)	
Тестирование совместимости (Compatibility testing)	
Различные типы тестов	
Позитивные тесты	
Негативные тесты	
Исследовательские тесты	
Различные области тестирования	
Модульное тестирование	
Интеграционное тестирование	
Системное тестирование	

Пример выполнения работы: Тестирования форума тестировщиков (<http://software-testing.ru/forum>)

Тесты	Пример тестов
Различные виды тестирования	
Функциональные тесты (Functional testing)	Переход по разделам форума. Поиск по сайту. Подписка на рассылку – письма с информацией приходят.
Тесты производительности (Performance testing)	Скорость перехода по вкладкам Скорость поиска по ключевым словам
Нагрузочные тесты (Load testing)	Большое количество пользователей обращаются к разделам форума. Большой апдейт нескольких разделов

Тестирование совместимости (Compatibility testing)	Корректная работа форума в разных браузерах
Различные типы тестов	
Позитивные тесты	<p>Правильность ссылок – ведут куда предполагалось.</p> <p>Правильность поиска по ключевым словам – находят нужные темы.</p>
Негативные тесты	<p>Не авторизованный пользователь не может оставлять комментарии на форуме</p> <p>Не модератор не может закрыть тему на форуме</p>
Исследовательские тесты	Ввод информации символами с диакритическими знаками
Различные области тестирования	
Модульное тестирование	<p>Тестирование каждого раздела в отдельности</p> <p>Тестирование модуля регистрации</p>
Интеграционное тестирование	<p>Авторизация: залогиниться в одном разделе, перейти в другой, система не выкинула – «продолжает» узнавать</p> <p>Правильно ли работают вместе модуль учета статистики и модуль добавления сообщений.</p>
Системное тестирование	<p>Основные использований сценарии форума соответствуют ожиданию.</p> <p>Встроенное видео проигрывается, отображаются картинки отображается корректно.</p>

Варианты:

- 1) Текстовый редактор Notepad.
- 2) Почтовый сервис Mail.Ru (www.mail.ru).
- 3) Графический редактор Paint.
- 4) Сервис хранения файлов Яндекс.Диск (<http://disk.yandex.ru/>).
- 5) Проигрыватель Windows Media Player.
- 6) Картографический сервис Google-карты

- (<https://maps.google.ru/>) . 7)
Браузер Internet Explorer.
- 8) Торрент-клиент µTorrent.
- 9) Архиватор WinRAR.
- 10) Сервис прогноза погоды от Рамблер
(<http://weather.rambler.ru/>)