МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ
ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
ГРАЖДАНСКОЙ АВИАЦИИ

Л.А. Надейкина

ТЕХНОЛОГИИ ПРОГРАММИРОВАНИЯ

Учебно-методическое пособие
по выполнению лабораторных работ № 5. 6, 7, 8, 9

для студентов III курса
направления 09.03.01
очной формы обучения

Москва
2019

ФЕДЕРАЛЬНОЕ АГЕНТСТВО ВОЗДУШНОГО ТРАНСПОРТА

**ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ**

**«МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ
УНИВЕРСИТЕТ ГРАЖДАНСКОЙ АВИАЦИИ (МГТУ ГА)»**

Кафедра вычислительных машин, комплексов, систем и сетей
Л.А. Надейкина

ТЕХНОЛОГИЯ ПРОГРАММИРОВАНИЯ

Учебно-методическое пособие
по выполнению лабораторных работ № 5, 6, 7, 8, 9

для студентов III курса
направления 09.03.01
очной формы обучения

Москва
2019

ББК 6Ф7.3
Н-17

Рецензент:
Черкасова Н.И. – канд. физ.-мат. наук

Надейкина Л.А.
Н-17 Технология программирования: учебно-методическое пособие по выполнению лабораторных работ № 5, 6, 7, 8, 9./ Л.А. Надейкина. – Воронеж: ООО «МИР», 2019. – 48 с.

Данное учебно-методическое пособие издается в соответствии с рабочей программой учебной дисциплины «Технология программирования» по учебному плану для студентов III курса направления 09.03.01 очной формы обучения.

Рассмотрено и одобрено на заседании кафедры 23.04.2019 г. и методического совета 19.03.2019 г.

Учебно-методическое пособие издается в авторской редакции.

Подписано в печать 15.05.2019 г.
Формат 60x84/16 Печ.л. 3 Усл. печ. л. 2,79
Заказ 474/6590 Тираж 30 экз.

Московский государственный технический университет ГА
*125993 Москва, Кронштадтский бульвар, д.*20
Отпечатано ОООР «МИР»
394033, г. Воронеж, Ленинский пр-т 119 А, лит. Я, оф. 215

1 ЛАБОРАТОРНАЯ РАБОТА № 5

Генерация программного кода C++ на основе модели UML. CASE-технология.

1.1 Цель лабораторной работы

- Получение навыков построения «Диаграмма классов» являющейся, логической моделью, отражающей статические аспекты структурного построения сложной системы и «Диаграммы компонентов».

- Изучение возможностей генерации кода на основе диаграммы классов. Возможность прямых преобразований: модель-код и обратных: код-модель.

1.2 Теоретические сведения

Программные средства, поддерживающие язык UML

В настоящее время фирма Rational Software является безусловным лидером в области объектно-ориентированного анализа и проектирования информационных систем с компонентной архитектурой. Разрабатываемая этой фирмой методология, основанная на использовании унифицированного языка моделирования (UML) поддержана целым спектром инструментальных программных средств визуального моделирования, совместной разработки, автоматизированного тестирования и документирования, охватывающих жизненный цикл создания программных систем [1].

Rational Software Architect версии 9.0:

- Открытая и расширяемая система на основе Eclipse (интегрированной среде разработки) версии 4.2.2.

- Поддержка UML версии 2.1

- Поддерживает преобразования модель-код и код-модель. Вперед преобразования идут от:

  - UML в Java
  - UML в C#
  - UML в C++
  - UML в EJB (Enterprise JavaBeans)
  - UML в WSDL (Web Services Description Language)
  - UML-язык описания интерфейса CORBA (IDL)

Обратные преобразования идут от:

  - Java в UML
  - C++ в UML.
  - .NET в UML

- Позволяет управлять моделями для параллельной разработки и архитектурного рефакторинга (перепроектирование кода), например, разбивать, комбинировать, сравнивать и объединять модели и фрагменты моделей.

- Предоставляет визуальные строительные инструменты для ускорения разработки модели и разработки программного обеспечения.

- Интегрировано с другими инструментами Rational, такими как управление, версиями ClearCase и управление конфигурацией ClearQuest.

- Все программные продукты Rational, включая Rational Software Architect (RSA), проектируются как плагины, которые находятся на вершине платформы разработки Eclipse с открытым исходным кодом.

Так как RSA основан на Eclipse, он может воспользоваться преимуществами рынка сторонних плагинов для Eclipse, а также плагинов, предназначенных специально для Rational tools.

Диаграмма классов (class diagram)

Центральное место в ООАП занимает разработка логической модели системы в виде диаграммы классов. Нотация классов в языке UML проста и интуитивно понятна всем, кто когда-либо имел опыт работы с CASE-инструментариями.

Диаграмма классов служит для представления статической структуры модели системы в терминологии классов объектно-ориентированной программирования. Диаграмма классов может отражать, в частности, различные сущности предметной области, взаимосвязи между отдельными сущностями, такими как объекты и подсистемы, а также описывает их внутреннюю структуру и типы отношений. На данной диаграмме не указывается информация о временных аспектах функционирования системы.

Класс *(class)* в языке UML служит для обозначения множества объектов, которые обладают одинаковой структурой, поведением и отношениями с объектами из других классов.

Классы - *это строительные блоки любой объектно-ориентированной системы*. Они представляют собой описание совокупности объектов с общими атрибутами, операциями, отношениями и семантикой. При проектировании объектно-ориентированных систем диаграммы классов обязательны. Классы используются в процессе анализа предметной области для составления *словаря предметной области* разрабатываемой системы.

Информация с диаграммы классов напрямую отображается в исходный код приложения - в большинстве существующих инструментов UML-моделирования возможна ***генерация кода*** для определенных языков программирования (обычно это Java или C++).

Таким образом, диаграмма классов - конечный результат проектирования и отправная точка процесса разработки.

Графически класс изображается в виде прямоугольника, который дополнительно может быть разделен горизонтальными линиями на разделы или секции. В этих разделах могут указываться имя класса, атрибуты (переменные) и операции (методы). Обязательным элементов обозначения класса является его имя. Даже если секция атрибутов и операций является пустой, в обозначении класса она выделяется горизонтальной линией, чтобы сразу отличить класс от других элементов языка UML. Примеры графического изображения классов на диаграмме классов приведены на рис. 1. ***Имя класса*** должно быть уникальным в пределах пакета, который описывается некоторой совокупностью диаграмм классов и указывается в первой верхней секции прямоугольника. Рекомендуется в качестве имен классов использовать существительные, записанные по практическим соображениям без пробелов.

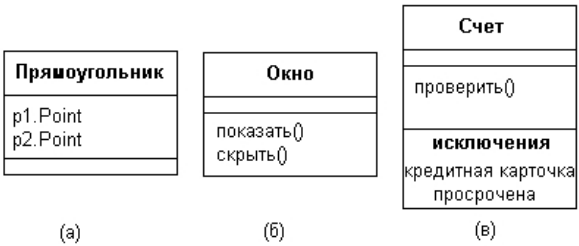


Рисунок 1. Примеры графического изображения класса

Атрибуты класса. Во второй сверху секции прямоугольника класса записываются его атрибуты (attributes) или свойства. В языке UML принята определенная стандартизация записи атрибутов класса. Каждому атрибуту класса соответствует отдельная строка текста, которая состоит из квантора видимости атрибута, имени атрибута, его кратности, типа значений атрибута и, возможно, его исходного значения:

<квантор видимости><имя атрибута>[кратность]:
<тип атрибута> = <исходное значение>{строка-свойство}

Квантор видимости может принимать одно из трех возможных значений и, соответственно, отображается при помощи специальных символов:

- Символ "+" обозначает атрибут с областью видимости типа общедоступный (public).
- Символ "#" обозначает атрибут с областью видимости типа защищенный (protected).
- знак "-" обозначает атрибут с областью видимости типа закрытый (private).

Отсутствие квантора видимости трактуется как public или private

Кратность атрибута характеризует общее количество конкретных атрибутов данного типа, входящих в состав отдельного класса. В общем случае кратность записывается в форме строки текста в квадратных скобках после имени соответствующего атрибута:

[нижняя_граница1 .. верхняя_граница1, нижняя_граница2.. верхняя_граца2, ..., нижняя_границаk .. верхняя_границаk]

Нижняя_граница и верхняя_граница являются положительными целыми числами, каждая пара которых служит для обозначения отдельного замкнутого интервала целых чисел. В качестве верхней границы может использоваться специальный символ "*". Если кратность атрибута не указана, то по умолчанию принимается ее значение равное 1..1, т. е. в точности 1. Так запись [1..3,7.. 10] означает, что кратность атрибута может принимать любое значение из чисел: 1, 2, 3, 7, 8, 9, 10.

Операция. В третьей сверху секции прямоугольника записываются операции или методы класса. Совокупность операций характеризует функциональный аспект поведения объектов класса. Запись операций класса в языке UML также стандартизована и подчиняется определенным

синтаксическим правилам. При этом каждой операции класса соответствует отдельная строка, которая состоит из квантора видимости операции, имени операции, выражения типа возвращаемого операцией значения и, возможно, строка-свойство данной операции:

<квантор видимости><имя операции>(список параметров):
<выражение типа возвращаемого значения>{строка-свойство}

Список параметров является перечнем разделенных запятой формальных параметров, каждый из которых может быть представлен в следующем виде:

<вид параметра><имя параметра>:<выражение типа>=<значение параметра по умолчанию>,

вид параметра — есть одно из ключевых слов in, out или inout со значением in по умолчанию, в случае если вид параметра не указывается. Имя параметра есть идентификатор соответствующего формального параметра. Выражение типа является зависимой от конкретного языка программирования спецификацией типа возвращаемого значения для соответствующего формального параметра. Наконец, значение по умолчанию в общем случае представляет собой выражение для значения формального параметра, синтаксис которого зависит от конкретного языка программирования и подчиняется принятым в нем ограничениям.

Отношения между классами

Кроме внутреннего устройства или структуры классов на диаграмме классов указываются различные отношения между классами. При этом совокупность типов таких отношений фиксирована в языке UML и предопределена семантикой этих типов отношений. Базовыми отношениями или связями в языке UML являются:

- Отношение зависимости (dependency relationship)
- Отношение ассоциации (association relationship)
- Отношение обобщения (generalization relationship)
- Отношение реализации (realization relationship)

Отношение зависимости

Отношение зависимости используется в такой ситуации, когда некоторое изменение одного элемента модели может потребовать изменения другого зависимого от него элемента модели. На диаграмме классов данное отношение связывает отдельные классы между собой, при этом пунктирная стрелка направлена от класса-клиента зависимости к независимому классу или классу-источнику. На рисунке изображены два класса: Класс_А и Класс_Б, при этом Класс_Б является источником некоторой зависимости, а Класс_А — клиентом этой зависимости (рис.2).

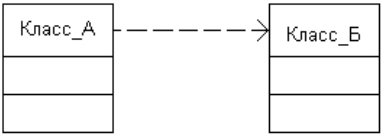


Рисунок 2. Пример отношения зависимости классов

Отношение ассоциации

Ассоциация соответствует наличию некоторого отношения между классами. Это отношение обозначается сплошной линией с дополнительными специальными символами, которые характеризуют отдельные свойства конкретной ассоциации. В качестве дополнительных специальных символов могут использоваться имя ассоциации, а также имена и кратность классов-ролей ассоциации (рис.3).

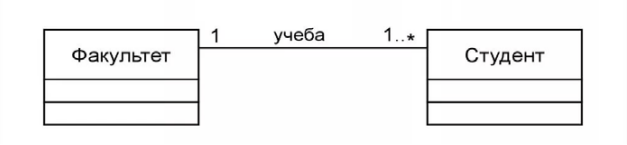


Рисунок 3. Отношение ассоциации

Отношение агрегации

Оно имеет место между несколькими классами в том случае, если один из классов представляет собой некоторую сущность, включающую в себя в качестве составных частей другие сущности. Это отношение имеет фундаментальное значение для описания структуры сложных систем, поскольку применяется для представления системных взаимосвязей типа "часть-целое" (рис.4).

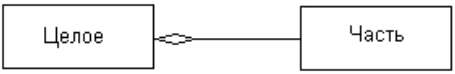


Рисунок 4. Отношение агрегации.

Отношение композиции

Является частным случаем отношения агрегации. Это отношение служит для выделения специальной формы отношения "часть-целое", при которой составляющие части в некотором смысле находятся внутри целого (рис.5). Специфика взаимосвязи между ними заключается в том, что части не могут выступать в отрыве от целого, то есть с уничтожением целого уничтожаются и все его составные части.

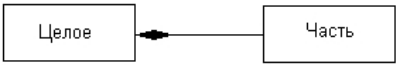


Рисунок 5. Отношение композиции

Отношение обобщения

Является обычным отношением между более общим элементом (родителем или предком) и более частным или специальным элементом. Применительно к диаграмме классов это отношение описывает иерархическое строение классов и наследование их свойств и поведения дочерним классом или потомком (рис.6).

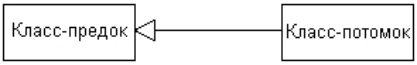


Рисунок 6. Отношение обобщения

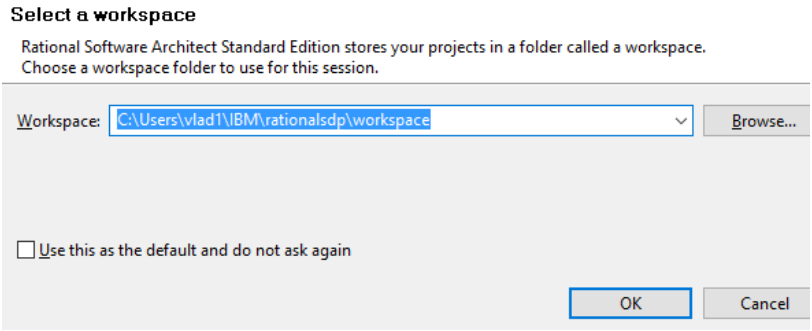
1.3 Задание на выполнение лабораторной работы

Задание:

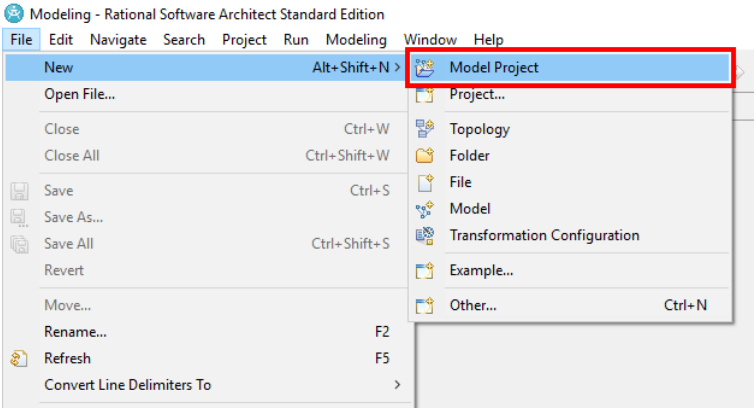
1) Изучить возможности языка UML.
2) Разработать диаграмму классов в соответствии с вариантом задания
3) На основе диаграммы классов сгенерировать программный код C++ в среде моделирования и разработки Rational Software Architect, версия 9.0.

1.4 Порядок выполнения лабораторной работы

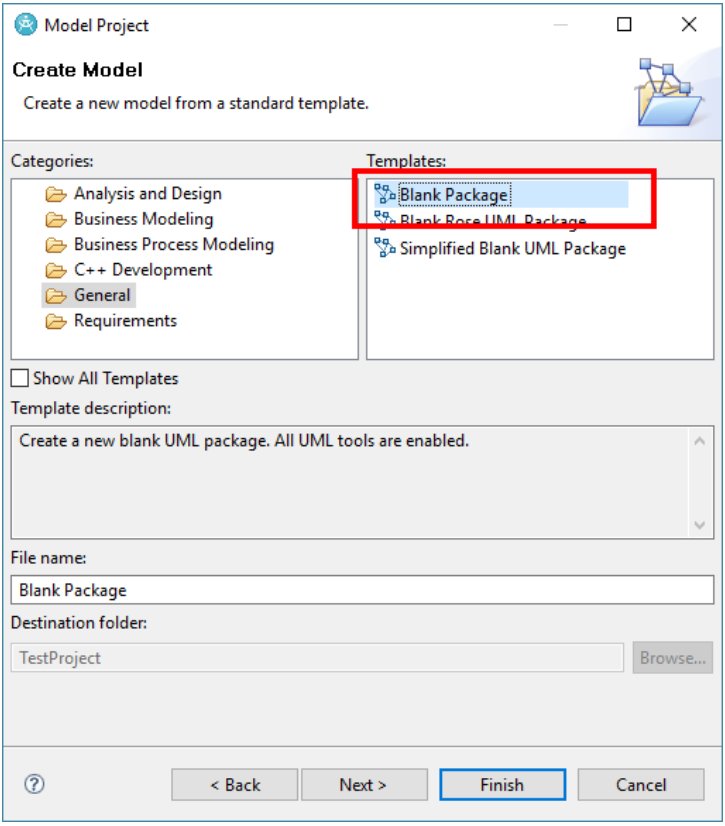
1) Запустить IBM Rational Software Architect Standard Edition. Выберать область где будут хранится проекты.



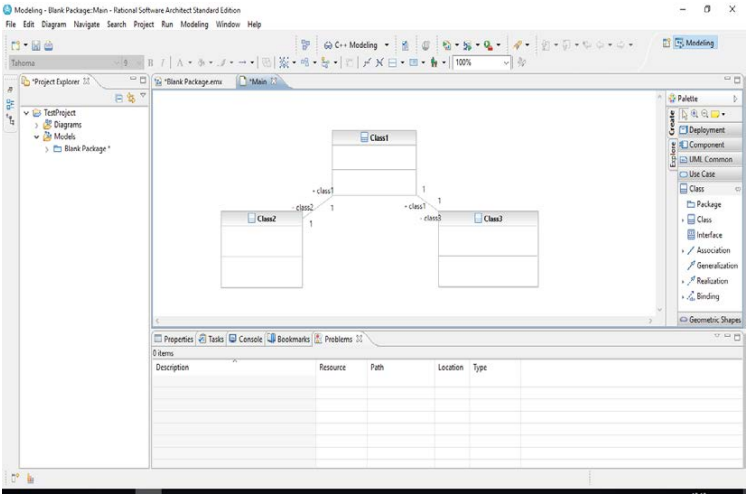
2) Создать новый пустой проект модели (File->New->Model Project).



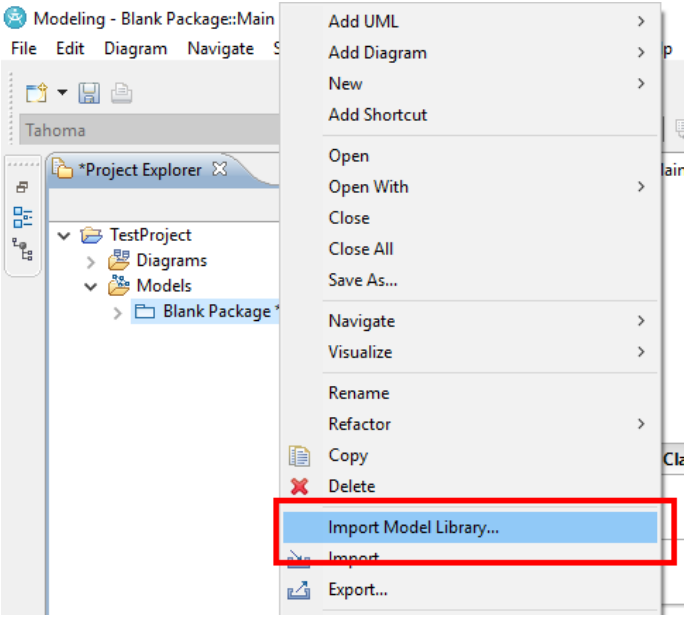
3) Выбрать категорию General и шаблон Blank Package. Нажать "Finish".



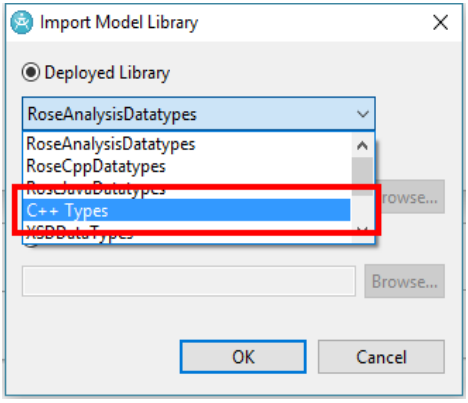
4) Разработать диаграмму классов в соответствии с вариантом



5) Добавить в проект типы C++. Для этого нажать правой кнопкой на Blank Package и выбрать Import Model Library.

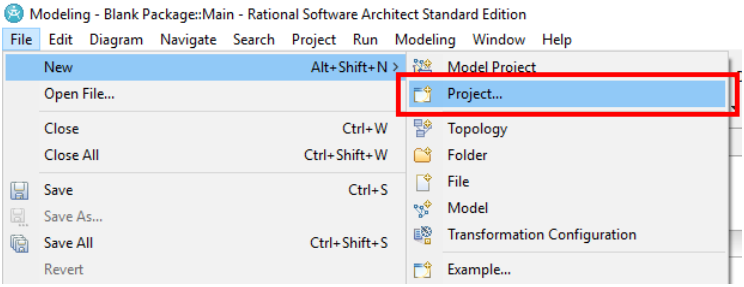


6) Выберем C++ Types.

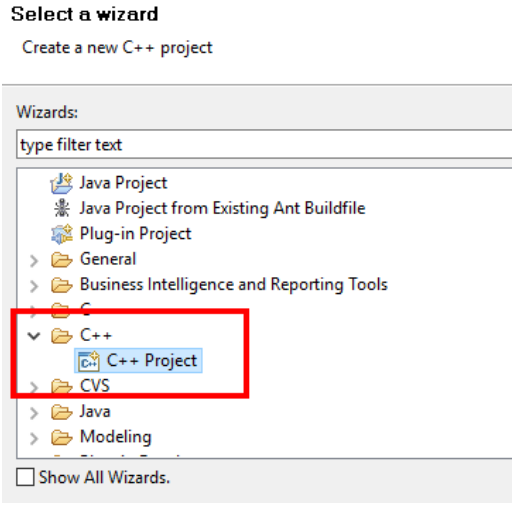


7) Для того чтобы трансформировать UML диаграмму в код C++ потребуется пустой C++ проект.

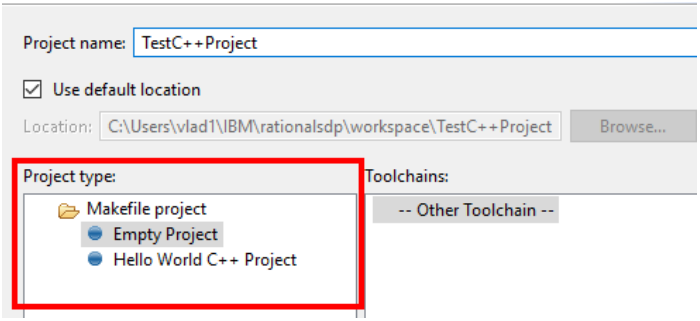
Выполним последовательность действий: File->New->Project



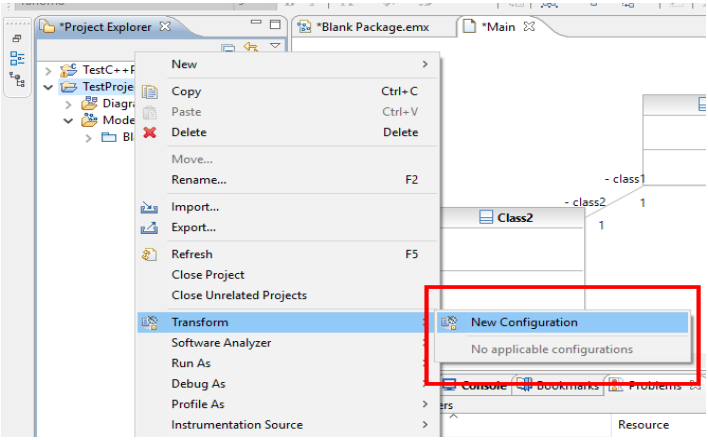
8) В New Project выбрать wizard – C++ -> C++ project.



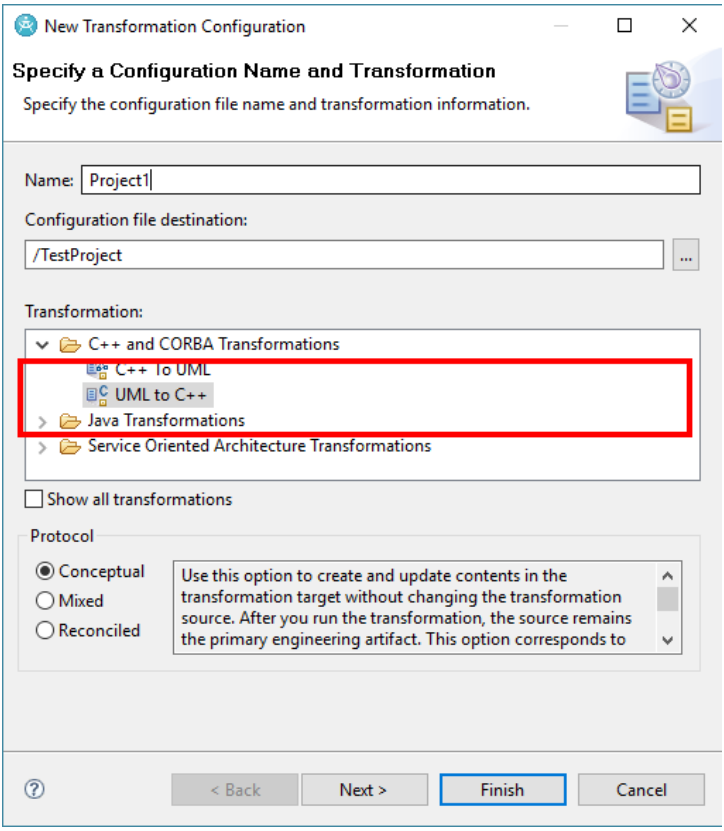
9) В следующем окне ввести имя C++ проекта и type Empty Project.



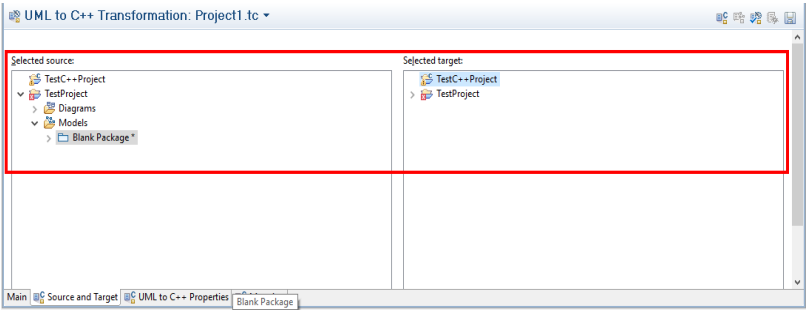
10) В Project Explorer нажмите правой кнопкой мыши на UML проект. В контекстном меню найти пункт Transform –> New Configuration.



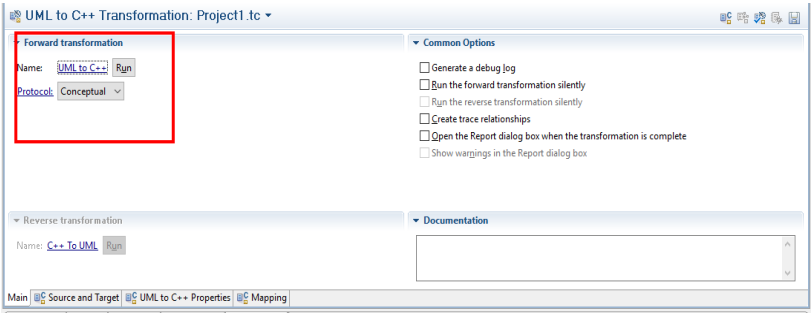
11)В открывшемся окне ввести для трансформации имя и её тип UML to C++.

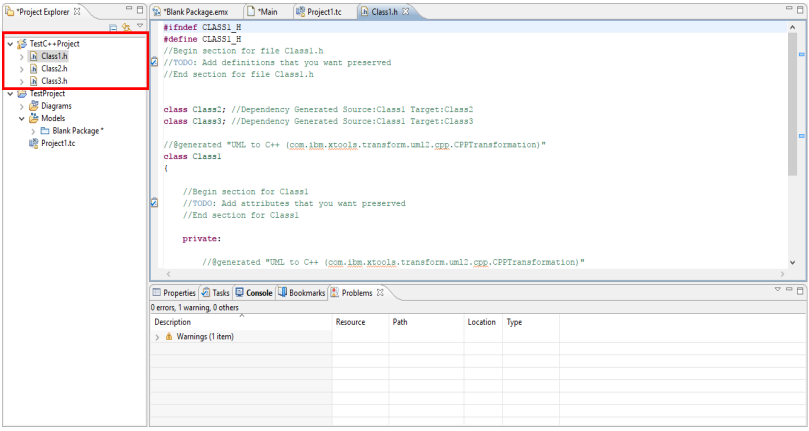


12) В созданной конфигурации перейдём во вкладку Source and Target. В ней выберем UML проект как источник(TestProject), а целью выберем пустой C++ проект (TestC++Project)



13) Перейдём в вкладку Main и нажмём кнопку Run. После чего RSA сгенерирует C++ код в пустом C++ проекте.





1.5. Контрольные вопросы

1) Методы и средства разработки программных продуктов. Различие. Case-средства.

2) Статические и динамические диаграммы UML.

3) Визуализация, прямое и обратное проектирование, специфицирование, документирование.

4) Диаграмма классов. Атрибуты класса. Отношения между классами.

5) Диаграмма компонентов. Диаграмма объектов.

6) Описание элементов главного меню Rational Software Architect.

7) Генерация программного кода C++.

2 ЛАБОРАТОРНАЯ РАБОТА № 6

Разработка программного проекта с использованием паттернов проектирования на C++ в среде Visual Studio

2.1 Цель лабораторной работы

Целью лабораторной работы является

- Познакомиться с известными шаблонами (паттернами) проектирования.
- Получение практических навыков применения паттернов при проектировании и разработке программного обеспечения.

2.2 Теоретические сведения

Шаблоны (паттерны) проектирования

При создании объектно-ориентированной программы требуется разработать классы, соответствующие предметам, процессам и понятиям предметной области, а также определить взаимоотношения этих классов и протоколы взаимодействия объектов во время выполнения программы.

Во многих объектно-ориентированных программных системах вы встретите повторяющиеся решения, состоящие из классов и взаимодействующих объектов.

Шаблон проектирования или паттерн (англ. design pattern) в разработке программного обеспечения — это повторяемая архитектурная конструкция, представляющая собой решение проблемы проектирования в рамках некоторого часто возникающего контекста [2].

Обычно шаблон не является законченным образцом, который может быть прямо преобразован в код; это лишь пример решения задачи, который можно использовать в различных ситуациях.

Объектно-ориентированные шаблоны показывают отношения и взаимодействия между классами или объектами, без определения того, какие именно конечные классы или объекты приложения будут использоваться.

Сообразное использование паттернов проектирования дает разработчику *ряд неоспоримых преимуществ.*

Приведем некоторые из них.

- Модель системы, построенная в терминах паттернов проектирования, фактически является структурированным выделением тех *элементов (компонентов) и связей*, которые значимы при решении поставленной задачи.

- Помимо этого, модель, построенная с использованием паттернов проектирования, *проста и наглядна* в изучении**,**

- Тем не менее, несмотря на простоту и наглядность, она позволяет *глубоко и всесторонне проработать архитектуру* разрабатываемой системы с использованием специального языка (UML)**.**

- Применение паттернов проектирования повышает *устойчивость системы к изменению требований и упрощает неизбежную последующую доработку системы.*

- Кроме того, трудно переоценить роль использования паттернов *при интеграции информационных систем* организации.

- Также следует упомянуть, что совокупность паттернов проектирования, по сути, представляет собой *единый словарь проектирования, который, будучи унифицированным средством, незаменим для общения разработчиков друг другом.*

- Но самое главное любой шаблон проектирования может стать палкой о двух концах: если он будет применен не к месту, это *может обернуться катастрофой* и создать много проблем в последующем.

- В то же время, реализованный в нужном месте, в нужное время, *он может стать настоящим спасителем [3]***.**

Есть три основных вида шаблонов (паттернов) проектирования:

- *структурные;*

- *порождающие;*

- *поведенческие***.**

В каждой группе можно, в свою очередь, выделить два уровня, определяющих, применяется паттерн к классам или объектам.

Описание каждого паттерна в книгах выполнено по одной и той же схеме, включающей его назначение, область применения, структурную схему (диаграмму классов), описание входящих в него объектов и их взаимодействий, а также пример реализации.

Такой уровень документирования делает возможным использование паттерна в различных конкретных случаях, возникающих при проектировании программных систем.

Паттерны позволяют повысить степень повторной используемости программного кода**,** повысить гибкость проекта и улучшить качество его документирования.

Применять паттерны целесообразно в тех случаях, когда гибкость действительно необходима. Однако, практически любой сколько-нибудь успешный программный продукт требует *сопровождения, модификации и,*

следовательно, перепроектирования, то есть должен обладать высокой степенью гибкости.

Структурные шаблоны определяют различные сложные структуры, которые изменяют интерфейс уже существующих объектов или его реализацию, позволяя облегчить разработку и оптимизировать программу.

Порождающие шаблоны проектирования, которые абстрагируют процесс *инстанцирования*, позволяют сделать систему *независимой от способа создания, композиции и представления объектов.*

Шаблон, порождающий классы, использует наследование, чтобы изменять *инстанцируемый* класс, а шаблон, порождающий объекты, делегирует *инстанцирование* другому объекту.

Поведенческие шаблоны определяют *взаимодействие* между объектами, увеличивая, таким образом, *его гибкость*.

Шаблонов проектирования много, кратко рассмотрим лишь некоторые из существующих паттернов.

Структурные шаблоны

Адаптер (Adapter)

Адаптер преобразует интерфейс класса в некоторый другой интерфейс, ожидаемый клиентами.

Обеспечивает совместную работу классов, которая была бы невозможна без данного паттерна из-за несовместимости интерфейсов.

Адаптер уровня класса использует множественное наследование (интерфейс наследуется от одного класса, а реализация — от другого),

В ***Адаптере*** уровня объекта применяется композиция объектов (как правило, в объекте определяется ссылка на другой объект).

Паттерн применяется:

- если требуется использовать существующий класс с интерфейсом, не подходящим к требованиям задачи;

- а также, если требуется создать повторно используемый класс, который должен взаимодействовать с заранее не известными или не связанными с ним классами, имеющими несовместимые интерфейсы.

Решаемая задача: *необходимо обеспечить взаимодействие несовместимых интерфейсов или создать единый устойчивый интерфейс для нескольких компонентов с разными интерфейсами.*

Решение*: преобразовать исходный интерфейс компонента к другому виду с помощью промежуточного объекта адаптера, то есть, добавить специальный объект с общим интерфейсом в рамках данного приложения и перенаправить связи от внешних объектов к этому объекту адаптеру.*

На рис. 7 представлена диаграмма классов, включающая адаптер. Класс ***Adapter*** приводит интерфейс класса ***Adaptee*** в соответствие с интерфейсом класса ***Target*** (наследником которого является ***Adapter***).

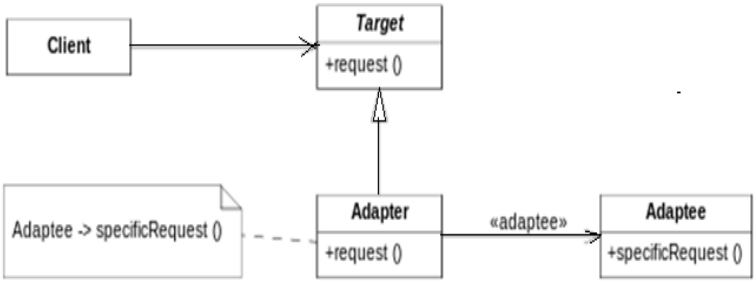


Рисунок 7. Диаграмма классов, включающая адаптер

Это позволяет объекту ***Client*** использовать объект ***Adaptee*** (посредством адаптера ***Adapter***) так, словно он является экземпляром класса ***Target.***

Таким образом, ***Client*** обращается к интерфейсу ***Target,*** реализованному в наследнике ***Adapter***, который *направляет* обращение к ***Adaptee.***

Шаблон Адаптер позволяет включать уже существующие объекты в новые объектные структуры, независимо от различий в их интерфейсах.

Этот шаблон позволяет в процессе проектирования не принимать во внимание возможные различия в интерфейсах уже существующих классов.

Если есть класс, обладающий требуемыми методами и свойствами (по крайней мере, концептуально), то при необходимости всегда можно воспользоваться шаблоном Адаптер для приведения его интерфейса к нужному виду.

Порождающие шаблоны

Абстрактная фабрика (Abstract Factory)

Абстрактная фабрика — порождающий шаблон проектирования, предоставляет интерфейс для создания семейств взаимосвязанных или взаимозависимых объектов, не специфицируя их конкретных классов.

От класса ***Абстрактная фабрика*** наследуются классы ***Конкретных Фабрик,*** которые содержат методы создания ***Конкретных объектов-продуктов***, являющихся наследниками класса ***Абстрактный продукт***, объявляющего интерфейс для их создания.

Клиент *пользуется только интерфейсами, заданными в классах* ***Абстрактная фабрика*** *и* ***Абстрактный продукт.***

Паттерн применяется в следующих случаях:

- когда программа должна быть независимой от процесса и типов создаваемых новых объектов.
- когда необходимо создать семейства или группы взаимосвязанных объектов, исключая возможность одновременного использования объектов из разных этих наборов в одном контексте.

Отметим следующие достоинства шаблона:

- изолирует конкретные классы;
- упрощает замену семейств продуктов;

- гарантирует сочетаемость продуктов.

На рисунке 8 представлена диаграмма классов, включающая ***Абстрактную фабрику.***

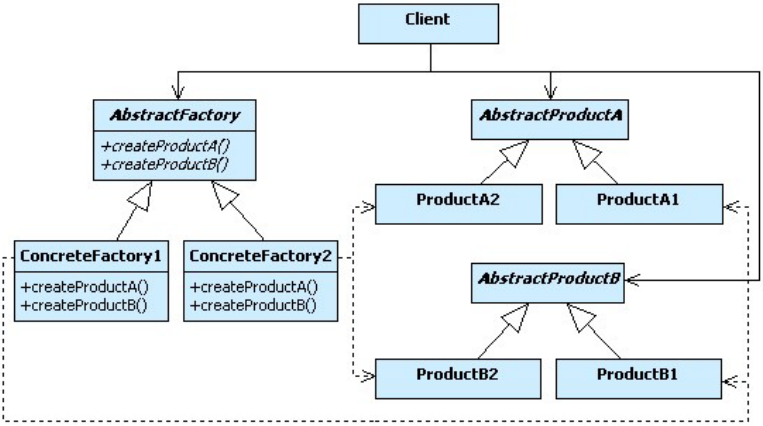


Рисунок 8. Диаграма классав шаблона *Абстрактная фабрика*

Поведенческие шаблоны

Наблюдатель (Observer)

Наблюдатель — поведенческий шаблон проектирования, также известен как «Подчинённые» (Dependents):

- Создает в классе механизм, который позволяет получать экземпляру объекта этого класса *оповещения от других объектов об изменении их состояния, тем самым наблюдая за ними.*

- Определяет *зависимость типа «один ко многим»* между объектами таким образом, что *при изменении состояния одного объекта все зависящие от него оповещаются об этом событии.*

При реализации шаблона «наблюдатель» обычно используются следующие классы и интерфейсы:

- ***Observable*** — интерфейс, определяющий методы для добавления, удаления и оповещения наблюдателей;

- ***Observer*** — интерфейс, с помощью которого наблюдатель получает оповещение;

- ***ConcreteObservable*** — конкретный класс, который реализует интерфейс ***Observable***;

- ***ConcreteObserver*** — конкретный класс, который реализует интерфейс ***Observer.***

Шаблон «Наблюдатель» применяется в тех случаях, когда система обладает следующими свойствами:

- существует, как минимум, один объект, рассылающий сообщения;

- имеются не менее одного получателя сообщений, причём их количество и состав могут изменяться во время работы приложения;

- нет надобности, очень сильно связывать взаимодействующие объекты, что полезно для повторного использования.

Данный шаблон часто применяют в ситуациях, в которых отправителя сообщений не интересует, что делают получатели с предоставленной им информацией. На рис. 9 дана диаграмма **Наблюдателя.**

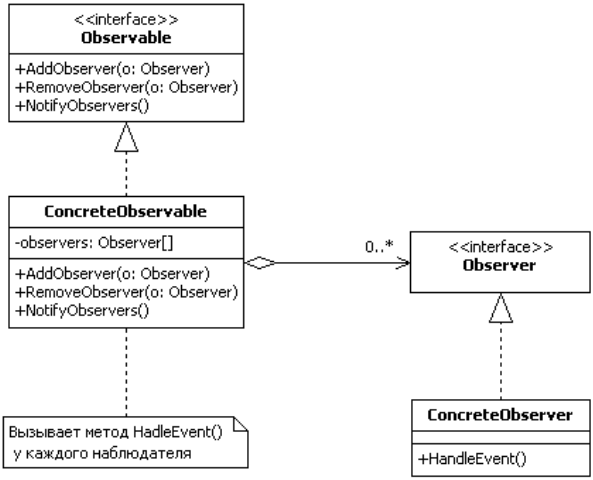


Рисунок 9. Диаграмма классов паттерна **Наблюдатель**

Паттерн ***Observer*** определяет зависимость "*один ко многим*" между объектами так, что при изменении состояния одного объекта все зависящие от него объекты *уведомляются и обновляются автоматически*;

Паттерн ***Observer*** инкапсулирует главный (независимый) компонент в абстракцию ***Subject*** и изменяемые (зависимые) компоненты в иерархию ***Observer***

Паттерн ***Observe****r* определяет часть ***"View"*** в архитектурной модели ***Model-View-Controller (MVC).***

2.3 Задание на выполнение лабораторной работы

- Проектировать программный проект в соответствие с вариантом задания, с использованием паттернов проектирования. Составить диаграмму классов.

- Реализовать проект в среде Visual Studio на языке C++ (Java, C#).

2.4. Пример выполнения лабораторной работы

Задание. Разработать проект, в котором реализуется следующий механизм: имеется некоторый объект и если этот объект обновляется, то все его зависимые объекты обновляются так же. Реализовать паттерн **Наблюдатель** (рис. 9)

#include <iostream>

```
#include <string>
#include <list>
using namespace std;
class IObserver
{  public:
   virtual void handleEvent (const SupervisedString&) = 0;
};
class SupervisedString // Observable class - определяющий методы для
                       //добавления, удаления и оповещения наблюдателей;
{   string _str;          // строка состояния
   list<IObserver* const> _observers; // список наблюдателей
   void _Notify() // метод оповещения всех наблюдателей о состоянии класса
   {   for(auto iter : _observers)
      {  iter->handleEvent(*this); }
   }
public:
   void add(IObserver& ref) //добавить наблюдателя
   { _observers.push_back(&ref); }
   void remove(IObserver& ref)// удалить наблюдателя
   {  _observers.remove(&ref);  }
   const string& get() const // возвращает строку состояния
   {   return _str;    }
   void reset(string str) //изменяет строку состояния и посылает оповещения
                          //всем наблюдателям о состоянии класса
   {  _str = str;
      _Notify();  }
};
class Reflector: public IObserver // наблюдатель первого типа - выводит
//наблюдаемую строку в cout
{ public:
   virtual void handleEvent(const SupervisedString& ref)
   {       cout << ref.get() << endl;    }
};
class Counter: public IObserver       //наблюдатель второго типа -
                          // выводит длину наблюдаемой строки в  cout
{ public:
  virtual void handleEvent(const SupervisedString& ref)
  {     cout << "length = " << ref.get().length() << endl;  }
};
int main()
{   SupervisedString str;
   Reflector refl;
   Counter cnt;
   str.add(refl);
   str.reset("Hello, World!");
```

```
    cout << endl;
    str.remove(refl);
    str.add(cnt);
    str.reset("World, Hello!");
    cout << endl;
    return 0;
}
```

Программа обладает следующими свойствами:

- существует один объект, который рассылает сообщения;
- имеется более одного получателя (два) сообщения и их количество может меняться в процессе работы системы;
- объекты не сильно связываются;
- стоит так же заметить, что отправителю не важно, что будут делать получатели с полученным сообщением;

2.5. Контрольные вопросы

1) Концепция создания программного обеспечения с использованием паттернов.
2) Архитектурные паттерны. Паттерны проектирования. Идиомы.
3) Паттерны проектирования.
4) Порождающие паттерны.
5) Структурные паттерны,
6) Паттерны поведения.
7) Что содержит описание паттерна. Преимущества применения паттернов.
8) Диаграмма классов паттерна.

2.6. Варианты заданий лабораторной работы

Вариант № 1, 9, 17, 25

Шаблон “Стратегия”. Проект “Принтеры”. В проекте должны быть реализованы разные модели принтеров, которые выполняют разные виды печати.

Вариант № 2, 10, 18, 26

Шаблон “Наблюдатель”. Проект “Оповещение постов ГАИ”. В проекте должна быть реализована отправка сообщений всем постам ГАИ.

Вариант № 3, 11, 19, 27

Шаблон “Декоратор”. Проект “Универсальная электронная карта”. В проекте должна быть реализована универсальная электронная карта, в которой есть функции паспорта, страхового полиса, банковской карты и т. д.

Вариант № 4, 12, 20, 28

Шаблон “Фабричный метод”. Проект “Фабрика смартфонов”. В проекте должно быть реализовано создание смартфонов с различными характеристиками.

Вариант № 5, 13, 21, 29

Шаблон “Абстрактная фабрика”. Проект “Заводы по производству автомобилей”. В проекте должно быть реализована возможность создавать автомобили различных типов на разных заводах.

Вариант №6, 14, 22, 30

Шаблон “Команда”. Проект “Клавиатура настраиваемого калькулятора”. Цифровые и арифметические кнопки имеют фиксированную функцию, а остальные могут менять своё назначение.

Вариант №7, 15, 23

Шаблон “Адаптер”. Проект “Часы”. В проекте должен быть реализован адаптер, который дает возможность пользоваться часами со стрелками так же, как и цифровыми часами. В классе “Часы со стрелками” хранятся повороты стрелок.

Вариант № 8, 16, 24

Шаблон “Фасад”. Проект “Компьютер”. В проекте должен быть реализован “компьютер”, который выполняет основные функции, к примеру, включение, выключение, запуск ОС, запуск программы, и т.д, не раскрывая клиенту деталей выполнения этой операции.

3. ЛАБОРАТОРНАЯ РАБОТА № 7

Разработка программного обеспечения с использованием архитектуры Model View Controller (MVC).

3.1 Цель лабораторной работы

Целью лабораторной работы является:

Получение навыков разработки приложений на базе архитектурного паттерна Model View Controller MVC.

3.2 Теоретические сведения

Архитектурный паттерн «Данные–представление–контроллер» (MVC).

При создании программных систем перед разработчиками часто встает проблема выбора тех или иных проектных решений. В этих случаях на помощь приходят *паттерны*.

Дело в том, что почти наверняка подобные задачи уже решались ранее и уже существуют хорошо продуманные элегантные решения, составленные экспертами, при этом паттерн дает не конкретное решение, а некий путь к решению. В настоящее время паттерны продолжают непрерывно развиваться. В области разработки программных систем существует множество паттернов, которые отличаются областью применения, масштабом, содержимым, стилем описания.

Одной из распространенных классификаций паттернов является классификация по степени детализации и уровню абстракции рассматриваемых систем.

Архитектурные паттерны, являясь наиболее высокоуровневыми паттернами, описывают структурную схему программной системы в целом.

В данной схеме указываются отдельные функциональные составляющие системы, называемые подсистемами, а также взаимоотношения между ними.

Примером архитектурного паттерна является хорошо известная программная парадигма "модель – представление - контроллер"

(model – view - controller - MVC). Нередко, архитектурные паттерны называют архитектурными стилями [3].

Архитектурный стиль определяет основные правила выделения компонентов и организации взаимодействия между ними в рамках системы или подсистемы в целом.

Рассмотрим детально архитектурный стиль (паттерн) «Данные–представление–контроллер».

Назначение - интерактивные приложения с гибким интерфейсом пользователя. Требования к пользовательскому интерфейсу в интерактивных приложениях меняются чаще всего. Разные пользователи имеют разные наборы требований. В несколько меньшей степени это касается методов обработки данных, лежащих в основе таких приложений, - визуальное представление управляющих элементов может меняться вместе с интерфейсом, а сами выполняемые действия зависят от бизнес-логики и предметной области, и поэтому более устойчивы. Наименее подвержена изменениям модель данных, с которыми работает приложение.

Поэтому для увеличения гибкости и удобства изменений в таких приложениях необходимо соответствующим образом разделить их компоненты. При этом нужно принимать во внимание следующие факторы:

- Одна и та же информация может быть представлена по-разному для удобства доступа к ней многих пользователей, имеющих разные привычки и разные навыки работы с информацией.
- Изменения в данных должны немедленно отображаться в различных представлениях этих данных.
- Внесение изменений в пользовательский интерфейс должно быть максимально простым, оно даже должно быть, возможным, прямо во время работы приложения.
- Поддержка различных стандартов пользовательского интерфейса и его перенос между платформами не должны влиять на код, связанный с методами работы с данными и структурой данных приложения.

Решение. Выделяется три набора компонентов.

Первый набор - *данные, модель данных* или просто *модель (model)* - соответствует структуре данных предметной области. Обязанности этих компонентов: представлять в системе данные и базовые операции над ними.

Компоненты второго набора — *представления* (*view*) - соответствуют различным способам представления данных в пользовательском интерфейсе. Для одних и тех же данных может иметься несколько представлений.

Каждому компоненту представления соответствует один компонент из третьего набора, *обработчик* (*controller*) — компонент, осуществляющий обработку действий пользователей. Такой компонент получает команды, чаще всего нажатия клавиш и нажатия кнопок мыши в областях, соответствующих визуальным элементам управления - кнопкам, элементам меню и пр. Эти команды он преобразует в действия над данными. В результате каждого действия требуется обновить представления всех данных, которые подверглись изменениям. На рис. 10 дана структура классов.

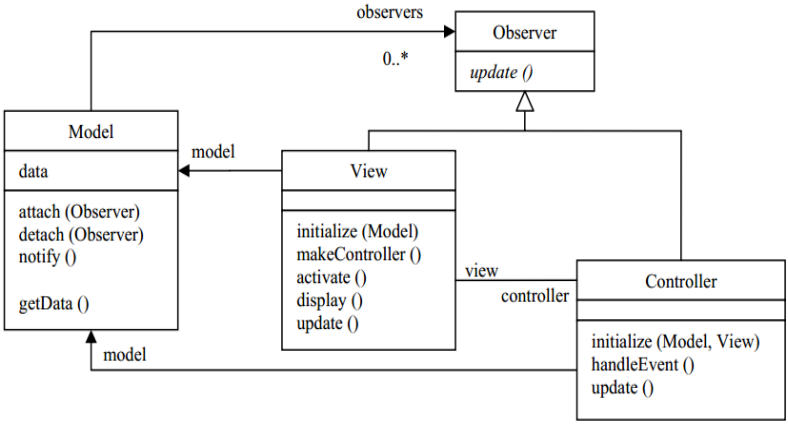


Рисунок 10. Структура классов модели, представления и обработчика

Компонент-модель моделирует данные приложения, реализует основные операции над ними и возможность регистрировать зависимые от него обработчики и представления. При изменениях в данных модель оповещает о них все зарегистрированные компоненты.

Компонент-представление представляет данные в некотором виде для пользователей, читая их из модели при необходимости, т.е. при инициализации и после сообщений о произошедших изменениях.

Кроме того, он инициализирует связанный с ним обработчик. На рис. 11 – сценарий обработки действия пользователя.

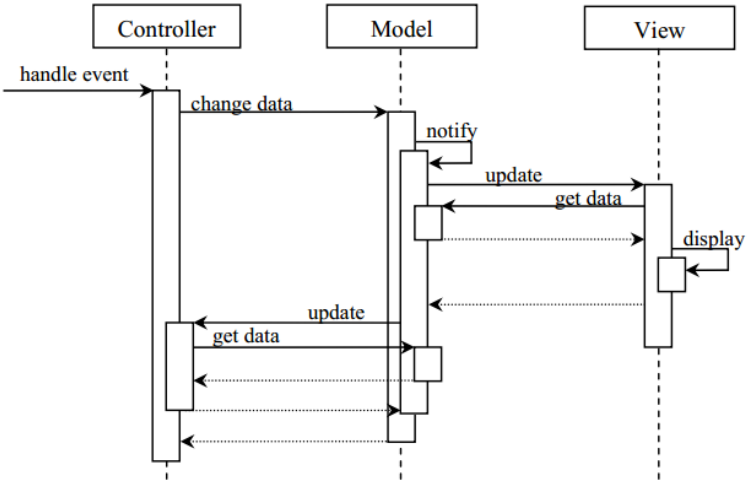


Рисунок 11. Сценарий обработки действия пользователя

Компонент-обработчик обрабатывает действия пользователя, транслируя их в операции над моделью или запросы на показ некоторых элементов

представлений. При оповещении об изменениях в модели он соответствующим образом изменяет собственное состояние, например, делая активными или отключая какие-нибудь кнопки и пункты меню.

У системы два базовых сценария работы — инициализация всех компонентов (рис. 12) и обработка некоторого действия пользователя с изменением данных и обновлением соответствующих им представлений и обработчиков.

Реализация.

- Отделить взаимодействие человека с системой от базовых функций самой системы. Для этого необходимо выделить структуру данных, с которыми система работает, и набор необходимых для функционирования системы операций над ними.
- Реализовать механизм передачи изменений. Для этого можно воспользоваться паттерном проектирования Наблюдатель (Observer).
- Спроектировать и реализовать необходимые представления.
- Реализовать необходимые обработчики действий пользователя.
- Спроектировать и реализовать связь между обработчиком и представлением.
- Реализовать построение системы из компонентов и инициализацию компонентов.

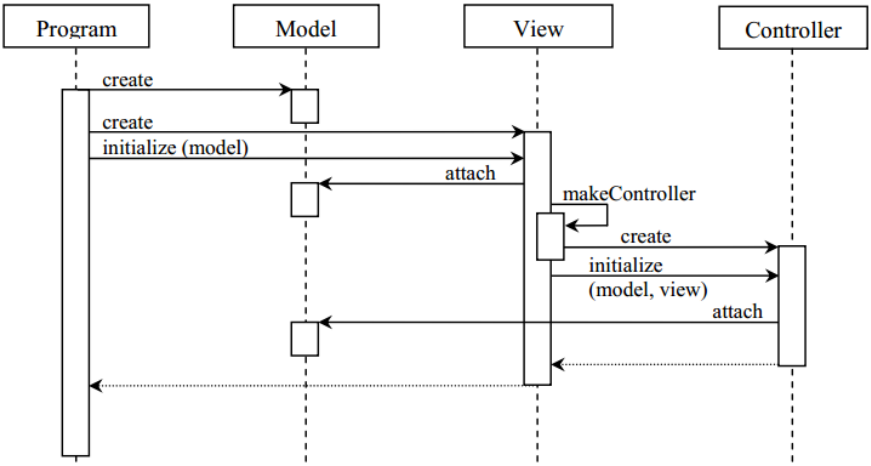


Рисунок 12. Сценарий инициализации системы

Достоинства.

- Возможность иметь несколько представлений одних данных, обновляемых по результатам воздействий пользователя.
- Поддержка подключаемых и динамически изменяемых представлений и обработчиков.

- Возможность изменения стилей пользовательского интерфейса во время работы.

Недостатки.

- Возрастание сложности разработки.

- Потери в производительности из-за необходимости обработки запросов пользователей сначала в обработчиках, затем в моделях, а затем во всех обновляемых компонентах.

- Представления и обработчики связаны очень тесно, из-за чего эти компоненты почти никогда нельзя переиспользовать по отдельности.

- И представления, и обработчики достаточно тяжело использовать без соответствующей им модели.

3.3 Задание на выполнение лабораторной работы

Разработать Windows-приложение в соответствием с вариантом задания.

Для создания графического интерфейса пользователя с помощью платформы .NET использовать технологию - Window Forms.

Для создания проекта использовать среду разработки Visual Studio и язык программирования C#.

Архитектура приложения должна быть спроектирована в соответствие с архитектурной моделью «model-view-controller» - MVC.

3.4. Пример разработки программы.

Наилучший способ вникнуть в суть шаблона проектирования — «изучить» его на практике. Был выбран язык C++, так как, с ним знакомы практически все, кто занимается программированием. Рассмотрим простейшее консольное приложение — конвертер температуры, используя классическую схему MVC.

Модель

Начнем с модели. Создадим класс TemperatureModel, задача которого — инкапсулировать всю логику конвертации градусов из шкалы Цельсия в шкалу Фаренгейта, и наоборот.

```
class TemperatureModel
{   public:
  TemperatureModel(float tempF)
  {     _temperatureF = tempF;   }
  float getF()
  {     return _temperatureF;   }
  float getC()
  {     return (_temperatureF - 32.0) * 5.0 / 9.0;    }
  void setF(float tempF)
  {     _temperatureF = tempF;    }
  void setC (float tempC )
  {     _temperatureF = tempC * 9.0 / 5.0 + 32.0;  }
}
```

Температура будет храниться во внутреннем поле ***_temperatureF***, а установить новую или получить её значение в любой из двух систем измерения помогут методы ***setF(), setC() и getF(), getC()*** соответственно.

Теперь наша задача — адаптировать этот класс под шаблон Observer. Шаблон «Наблюдатель» (Observer) определяет зависимость типа «один ко многим» между объектами таким образом, что при изменении состояния одного объекта все зависящие от него оповещаются об этом событии.

Это нужно для того, чтобы остальные классы приложения «знали» о любых изменениях в модели. Для этого создадим еще два класса, которые будут базовыми для остальных — это ***Observable*** (определяет методы для добавления, удаления и оповещения «наблюдателей») и ***Observer*** (класс, с помощью которого наблюдаемый объект оповещает наблюдателей).

```
class Observer
{   public:
   virtual void update() = 0;
};
 class Observable
{  public:
  void addObserver(Observer *observer)
  {  _observers.push_back(observer); }
  void notifyUpdate()
  {  int size = _observers.size();
     for (int i = 0; i < size; i++)
     {  _observers[i]->update();   }
  }
    private:
    std::vector<Observer*> _observers;
};
```

Использован шаблонный STL класс vector.

Класс ***Observevable*** содержит список всех «наблюдателей». Нового «наблюдателя» можно будет добавить с помощью метода ***addObserver().*** При вызове метода ***notifyUpdate()*** класс ***Observable*** пройдется по списку «наблюдателей» и вызовет их методы ***update(),*** а они, в свою очередь, смогут каким-то образом на это отреагировать.

Теперь нужно сделать класс ***TemperatureModel*** «оповещателем», чтобы у него впоследствии могли быть «слушатели», следящие за его изменениями. Так как мы уже создали для этого все необходимые классы, нет ничего проще, чем сделать класс TemperatureModel наследником класса Observable.

```
class TemperatureModel : public Observable
{  public:
  float getF()
  {     return _temperatureF;   }
  float getC()
  {    return (_temperatureF - 32.0) * 5.0 / 9.0;    }
  void setF(float tempF)
```

```
  { _temperatureF = tempF;
    notifyUpdate();  }
  void setC(float tempC)
  {  _temperatureF = tempC * 9.0 / 5.0 + 32.0;
     notifyUpdate();  }
private:
  float _temperatureF;
};
```

Обратите внимание, что добавились вызовы ***notifyUpdate()*** в методах ***setF() и setC().*** Таким образом, мы достигаем нашей цели: «слушатели» будут оповещены в случае любых изменений в модели.

Представление

Теперь нам нужно создать View — класс, выводящий изменения модели на консоль. Назовем его ***ConsoleView***.

```
class ConsoleView: public Observer
{  public:
 ConsoleView(TemperatureModel *model)
  {  _model = model;
     _model->addObserver(this);
  }
  virtual void update()
  {  system("cls");
     printf("Temperature in Celsius: %.2f\n", _model->getC());
     printf("Temperature in Farenheit: %.2f\n", _model->getF());
     printf("Input temperature in Celsius: ");
  }
   private:
   TemperatureModel *_model;
};
```

Класс ***ConsoleView*** является наследником класса ***Observer,*** потому он сможет получать сообщения от модели. ***ConsoleView*** хранит в себе указатель на модель, который передается в конструкторе. Обратите внимание, что там же ***ConsoleView*** «подписывает» себя на изменения модели вызовом метода ***addObserver()***. Метод ***update()*** очищает экран консоли, выводит текущие данные, а также текст «Введите температуру в цельсиях» (для простоты мы ограничимся только этим режимом).

Контроллер

Осталось «оживить» созданные классы и добавить ***Controller*** — управляющий класс, который будет отслеживать введенные пользователем данные и соответственно изменять модель.

```
  class Controller
  {  public:
     Controller(TemperatureModel *model)
     {  _model = model;  }
   void start()
     {     _model->setC(0);
```

```
        float temp;
        do
       {  scanf("%f", &temp);
         _model->setC(temp);
       }
       while (temp != 0);
    }
      private:
     TemperatureModel *_model;
};
```

Класс ***Controller***, так же как и ***ConsoleView***, получает ссылку на модель в конструкторе. Метод ***start()*** сбрасывает модель в первоначальное состояние и запускает цикл ввода данных до тех пор, пока пользователь не введет 0.

Конечная реализация

```
int main()
{  TemperatureModel model;
  ConsoleView view(&model);
  Controller controller(&model);
  controller.start();
  return 0;
}
```

Ее содержание, как и предполагалось, оказалось максимально простым. Все что нужно, — это создать экземпляры классов TemperatureModel (модель), ConsoleView (представление) и Controller (управление), а затем запустить управление вызовом метода controller.start().
Вывод на консоль будет выглядеть так:

```
Temperature in Celsius: 36.60
Temperature in Farenheit: 97.88
Input temperature in Celsius: 36.6_
```

Заключение

Можно модифицировать написанную программу, добавив режим ввода температуры в шкале Фаренгейта: для этого понадобится дополнительный метод ***setMode()*** в классе ***ConsoleView***, который будет устанавливать текущий режим отображения (их должно быть два: с предложением ввести градусы в Цельсиях и в Фаренгейтах), а сам режим будет устанавливать контроллер. Можно изменить поведение модели, или способ ввода данных, либо добавить графический интерфейс — каждое из этих изменений затронет только соответствующий слой, и вам не придется каждый раз переписывать приложения заново. Этот пример поможет быстро освоить основные принципы ***MVC*** и позволит создавать намного более сложные системы.

3.5. Контрольные вопросы.

1) Архитектура ПО, влияние архитектуры на его свойства.

2) Особенности разработки сложных программных систем. Алгоритмическая и объектная декомпозиция программного кода. Рефакторинг.

3) Повторное использование кода.

4) Архитектурные паттерны.

5) Паттерн: «Модель-Представление-Контроллер» (model–view–controller, MVC). Диаграмма.

6) Паттерны проектирования, используемые при реализации MVC.

4. ЛАБОРАТОРНАЯ РАБОТА № 8

Технология создания программной системы в Visual Studio.NET. Создание базы данных на SQL Server из Visual Studio.

4.1 Цель лабораторной работы

Целью лабораторной работы является:

- Получение практических навыков создания программных систем в Visual Studio.NET.

- Получение навыков создание базы данных на SQL Server из Visual Studio.Net, навыков работы с классами DataSet и DataAdapter.

4.2 Теоретические сведения

Технология .NET разработки и развертывания программного обеспечения

Технология **.NET** предназначена для разработки приложений под Windows с новым интерфейсом программирования.

Платформа **.NET** состоит из различных продуктов, которые можно условно разделить на четыре группы:

- ***средства разработки*** - языки программирования (*Visual C++, C#, Visual Basic.NET, Visual Java*), среда выполнения Common Language Runtime (*CLR, общеязыковая среда выполнения*), библиотека классов для создания разнообразных приложений (***FCL***), а также инструментальная среда разработки *Visual Studio.NET*;

- ***web-сервисы*** - возможность применения коммерческих web-сервисов (таких, например, как *.NET MyServices*), которые необходимы для создания web-приложений, требующих идентификации пользователей;

- ***специализированные серверы*** - набор серверов *SQL Server, Exchange Server, BizTalk* и др., объединенных в одно семейство серверов *.NET Enterprise Servers*. Эти серверы обеспечивают работу с базами данных, с электронной почтой, и многое другое;

- ***поддержка устройств*** — встроенная поддержка устройств, которые могут работать с технологиями .NET (например, мобильные телефоны).

Архитектура платформы .NET

В упрощенном виде платформа .NET состоит из пяти основных компонентов (рис. 13).

На нижнем уровне платформы находится операционная система Windows.

На уровне, расположенном выше уровня операционных систем, находятся сразу три компонента:

специализированные серверы .NET Enterprise Servers — набор серверных продуктов, таких как Application Center, BizTalk Server, Commerce Server, Exchange Server, Host integration Server, Internet Security Acceleration Server и SQL Server;

набор web-сервисов .NET MyServices — представляющих собой готовые блоки кода, которые разработчик может включать в свои проекты;

.NET Framework - новая инфраструктура разработки и исполнения Windows-приложений, которая включает в себя общеязыковую среду выполнения **CLR**, а также общую структуру классов, которые можно использовать в любом языке программирования семейства .**NET.**

Среда разработки Visual Studio .NET		
Специализированные серверы .NET Enterprise Servers	**.NET Framework**	**.NET MyServices**
Операционная система		

Рисунок 13. Архитектура платформы .NET

На верхнем уровне архитектуры .NET располагается среда разработки приложений Visual Studio .NET.

Архитектура .NET Framework

.NET Framework состоит из двух основных частей:

- *CLR - Common Language Runtime*, общеязыковой среды выполнения;
- библиотеки классов .NET Framework (*FCL - Framework Class Library*).

Главным компонентом *.NET Framework* является *CLR*.

Среда ***CLR*** лежит в основе технологии **.NET**. Среда **CLR** управляет выполнением кода, написанного на любом из языков семейства .**NET**.

Такая возможность обеспечена ***общей системой типов (CTS –Common Type System)***, которую используют все языки, ориентированные на ***CLR.***

Для обеспечения межъязыкового взаимодействия необходимо придерживаться ***общеязыковой спецификации (CLS – Common Language Specification)***, предложенной Microsoft. Эта спецификация ограничивает все многообразие типов того или иного языка программирования тем подмножеством, которое присутствует одновременно во всех языках

Спецификация ***CTS*** определяет правила определения типов и особенности их поведения.

Среда CLR оперирует так называемыми ***сборками***.

Сборки (assemblies) — представляют собой .NET-компоненты, которые являются переносимыми исполняемыми файлами (Portable Executable, PE). Эти файлы являются файлами с расширениями exe или dll и состоят из метаданных и кода.

Весь код для платформы .**NET** преобразуется ***CLR*** в промежуточный код на языке ***Common Intermediate Language (CIL).*** Поэтому разработчики могут

легко интегрировать код, написанный для .NET на различных языках программирования. Все, что можно сделать на одном .NET-совместимом языке, можно сделать на любом другом. Код на этих языках компилируется в код на одном языке — языке ***CIL.***

Код для CLR представляет собой команды псевдомашинного языка ***CIL***. Эти команды компилируются в машинный код соответствующего типа процессора по запросу в период выполнения.

Обычно компиляция метода происходит один раз во время его первого вызова. Затем результат кэшируется в памяти, чтобы избежать задержек при повторных вызовах. ***JIT-компилятор*** выполняет оптимизацию кода специально для процессора, на котором исполняется этот код.

JIT-компиляция (*англ. Just-in-time compilation, компиляция «на лету»), — технология увеличения производительности программных систем, использующих байт-код, путём компиляции байт-кода в машинный код непосредственно во время работы программы. При этом среда исполнения справляется с поздним связыванием типов и гарантирует безопасность исполнения.*

Преобразуя команды CIL в команды процессора, JIT-компилятор выполняет верификацию кода на предмет безопасности типов.

Ресурсы, выделяемые управляемым кодом, освобождаются сборщиком мусора. Иначе говоря, программист только выделяет память, но не освобождает ее — это делает **CLR**.

NET Framework Class library (FCL)

В **.NET** включены сборки библиотеки классов **.NET - Framework Class library (FCL),** содержащие определения нескольких тысяч типов, каждый из которых предоставляет некоторую функциональность.

Наборы "родственных" типов собраны в отдельные пространства имен.

Так, пространство имен **System** содержит базовый класс **Object**, из которого, в конечном счете, порождаются все остальные типы.

Кроме того, **System** содержит типы для целых чисел, символов, строк, обработки исключений, консольного ввода/вывода, группу типов для безопасного преобразования одних типов в другие, форматирования данных, генерации случайных чисел и выполнения математических операций и др..

ADO.NET — технология, предоставляющая доступ к данным для приложений, основанных на Microsoft .NET.

В отличие от классической технологии ADO, которая была в основном предназначена для тесно связанных клиент-серверных систем, ADO.NET больше нацелена на автономную работу с помощью объектов ***DataSet.***

Эти типы представляют *локальные копии* любого количества взаимосвязанных таблиц данных.

*Объекты **DataSet** позволяют вызывающей сборке (наподобие web-страницы или программы, выполняющейся на настольном компьютере) работать с содержимым **DataSet**, изменять его, не требуя подключения к источнику данных, и отправлять обратно блоки измененных данных для обработки с помощью соответствующего адаптера данных.*

Типы, составляющие ADO.NET, используют протокол управления памятью *CLR*, и доступ к ним возможен с помощью любого языка .NET.

Классы ADO.NET находятся в сборке *System.Data.dll*. Классы, которые составляют провайдер данных, включают следующие:

- Connection - используется для установки соединения с источником данных.

- Command - используется для выполнения команд SQL и хранимых процедур.

- DataReader - предоставляет быстрый доступ к данным только для чтения.

- DataAdapter. Этот объект выполняет две задачи:

1) наполнение объекта *DataSet* информацией, извлеченной из источника данных (автономные данные).

2) применение изменений данных к источнику данных в соответствии с модификациями данных в объекте *DataSet.*

Функционально классы ADO.NET можно разбить на два уровня: подключенный и отключенный:

В технологии ADO.NET предусмотрена эффективная и гибкая поддержка приложений, распределенных между несколькими компьютерами (серверами баз данных, серверами приложений и клиентскими рабочими станциями). В частности, особая поддержка предусмотрена для отсоединенных приложений *с минимизацией нагрузки при работе с параллельными операциями доступа к данным и блокировкой ресурсов сервера базы данных.* В результате повышаются возможности *масштабирования приложений*.

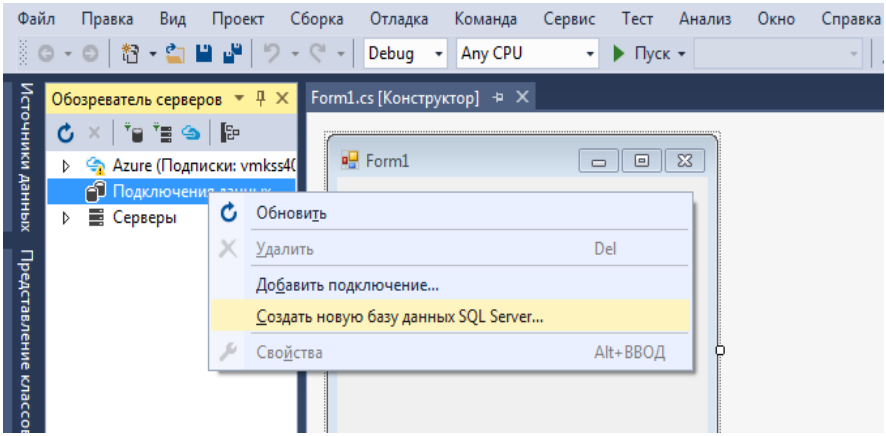
4.3 Задание на выполнение лабораторной работы

Разработать распределенное Windows-приложение в соответствием с вариантом задания. Разработку приложения строить на основе технологии *.NET Framework 4.5.*

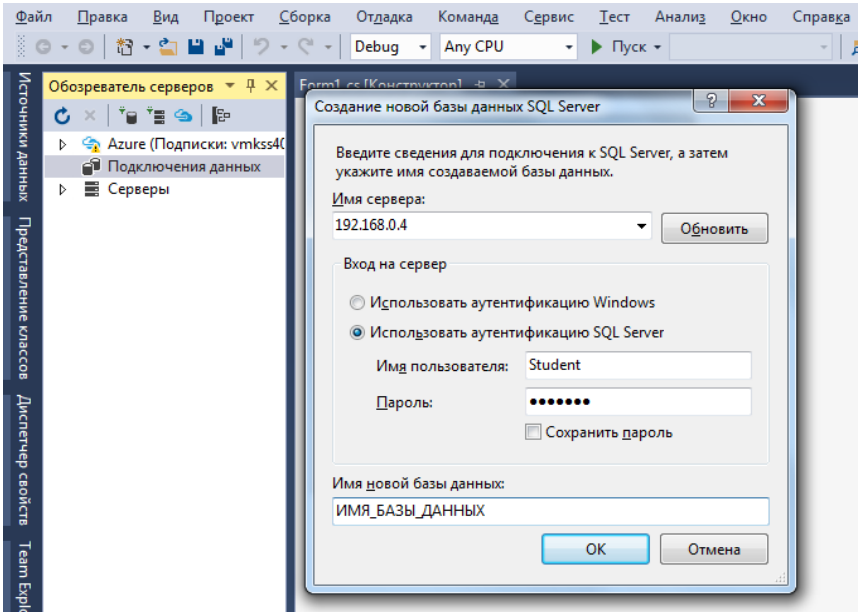
4.4. Порядок выполнения работы

1. Запустите Visual Studio.
2. Создайте проект, в котором вы будете работать. Для создания нового проекта выберите Файл -> Создать -> Проект… или нажмите Ctrl + Shift + N. Выберите язык Visual C# [4], тип приложения: Windows Forms.
3. Перейдите к обозревателю серверов и создайте базу данных. Для открытия обозревателя серверов выберите Вид -> Другие окна -> Обозреватель

серверов или нажмите Ctrl + Alt + S. Выберите - «Подключения данных» и в контекстном меню перейдите к созданию новой базы данных.

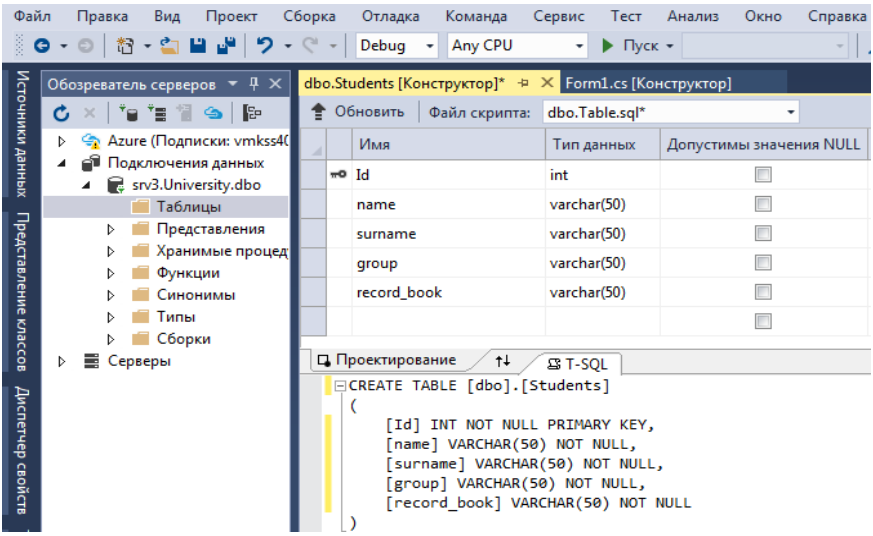


Укажите адрес сервера, имя пользователя и пароль и придумайте название базы данных. Пароль и адрес сервера для доступа уточните у преподавателя.



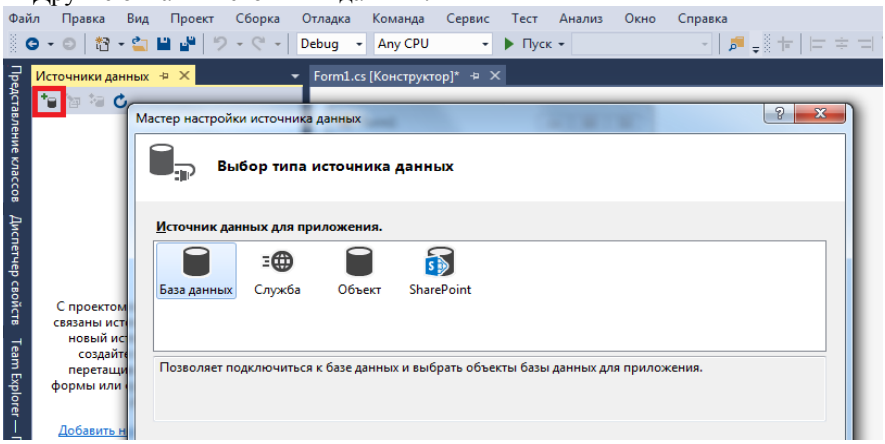
4. Создайте таблицы в вашей базе данных согласно полученному заданию. Для перехода к конструктору таблиц выберите вашу базу данных в

подключениях данных обозревателя серверов и в контекстном меню пункта «Таблицы» выберите «Добавить новую таблицу».



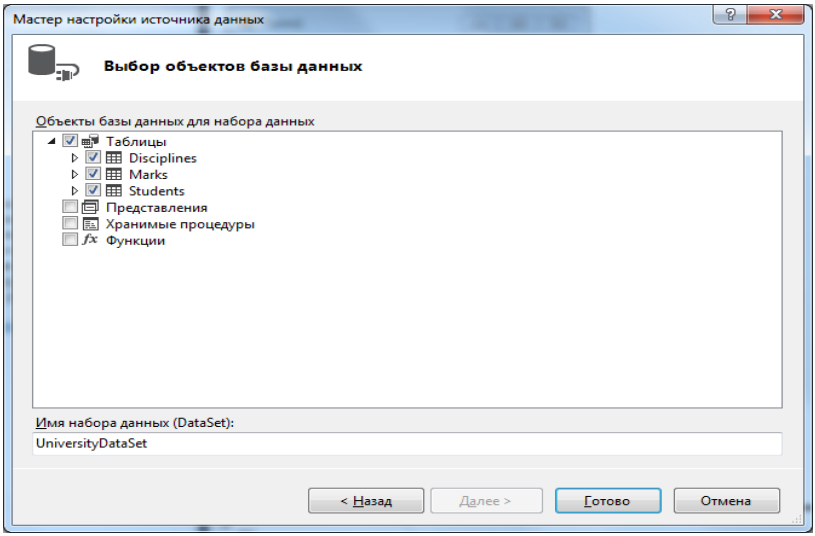
После заполнения структуры таблицы нажмите кнопку «Обновить» для передачи данных на сервер.

5. Для работы с базой данных в приложении вам нужно представить её в виде источника данных. Для перехода к окну «Источники данных» выберите Вид -> Другие окна -> Источники данных:

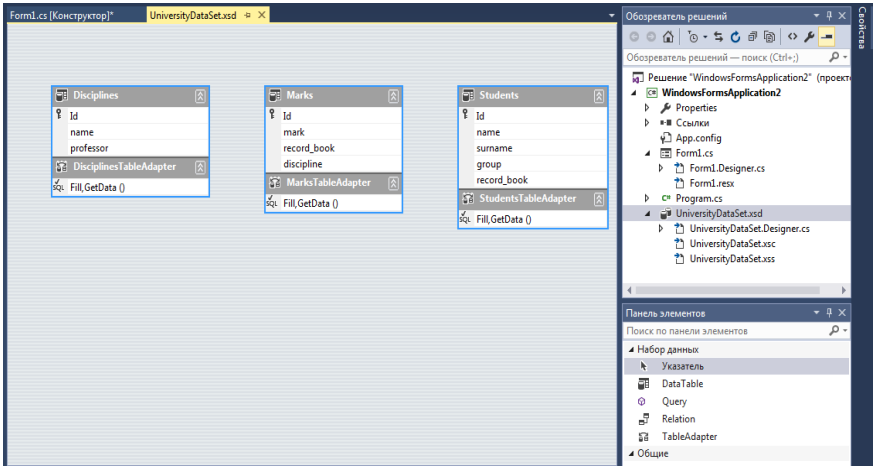


Выберите форму, к которой будет создан источник данных. Создайте источник данных. В типе источника данных укажите «База данных», в модели – «Набор данных», используйте уже существующее подключение. При первичном создании проекта разрешите сохранение конфиденциальных

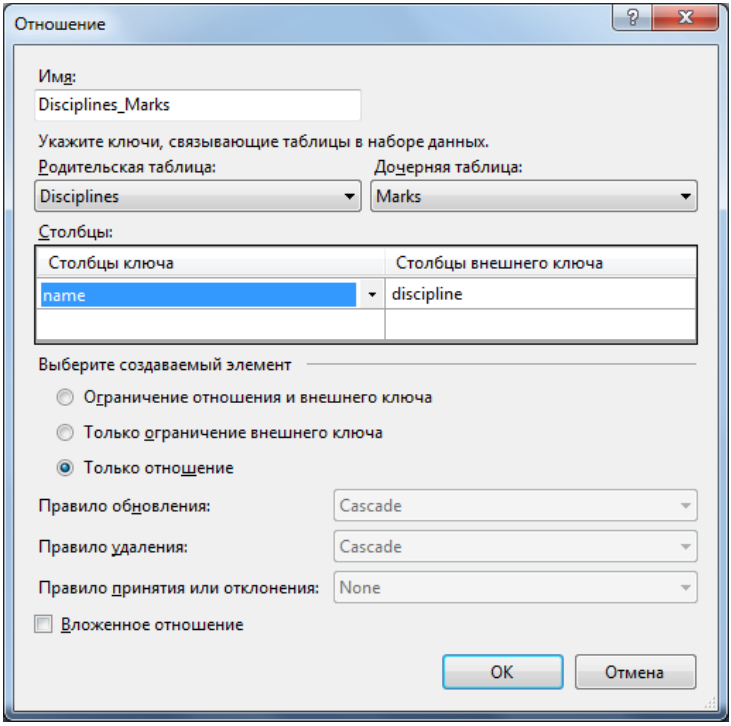
данных в строке подключения. Позднее модифицируйте проект на передачу данных авторизации вручную. Укажите все используемые таблицы:



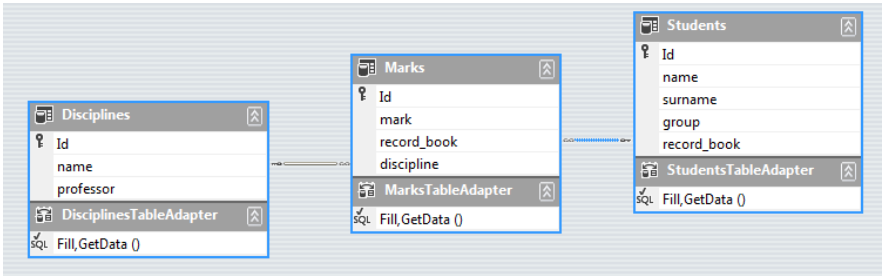
6. Настройте связи между таблицами. В обозревателе решений выберите ваш источник данных и перетяните элемент Relation из панели элементов на необходимые таблицы.



Для примера была создана связь по названию дисциплины между таблицами Marks и Disciplines:

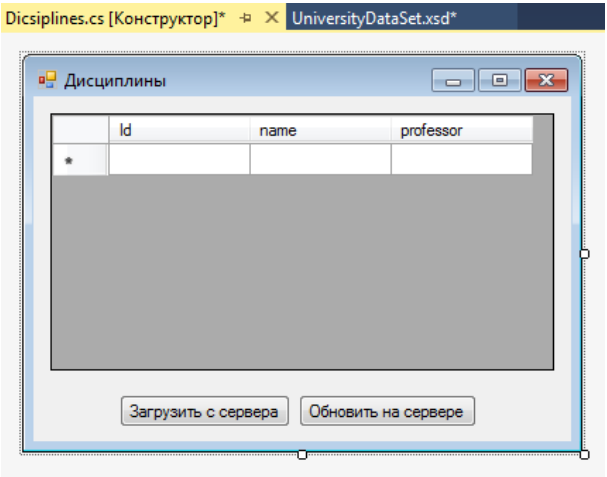


После создания всех необходимых связей схема примет следующий вид:



7. Добавьте представление таблиц базы данных и способы взаимодействия с ними в интерфейс формы. Для этого перетащите нужную таблицу из источников данных на окно формы. Из панели элементов добавьте нужные управляющие и информационные элементы (кнопки, меню, текстовые поля).

Возможный результат представлен ниже.

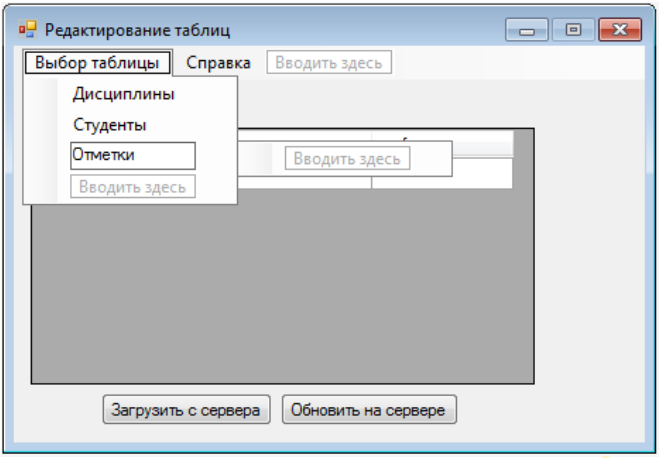


Для предотвращения деформации интерфейса таблицы удобно отключить изменение размера формы - свойства формы: MaximizeBox:false и FormBorderStyle:Single.

8. Добавьте в форму реализацию функций взаимодействия с базой данных на сервере. Для этого дважды кликните мышью по элементу управления и в открывшемся окне задайте логику работы программы.

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;
namespace UniversityDB {
    public partial class Dicsiplines : Form {
    public Dicsiplines() { InitializeComponent();}
    private void Disciplines_Load(object sender, EventArgs e) {}
    private void loadFromServer_Click(object sender, EventArgs e){
      // Загрузка данных с сервера
    this.disciplinesTableAdapter.Fill (this.universityDataSet.Disciplines);}
    private void updateOnServer_Click(object sender, EventArgs e){
    this.Validate ();
    this.disciplinesBindingSource.EndEdit ();
          // Передача данных на сервер
   this.disciplinesTableAdapter.Update (this.universityDataSet.Disciplines); }
   }}
```

9. Кроме уже сгенерированного кода необходимо указать методы взаимодействия с базой данных. Для обновления таблицы используется метод ***Fill*** адаптера добавленной таблицы базы данных (***disciplinesTableAdapter***):
   this.disciplinesTableAdapter.Fill(this.universityDataSet.Disciplines);
   Для отправки данных на сервер – метод ***Update***:
   this.disciplinesTableAdapter.Update(this.universityDataSet.Disciplines);
   Для проверки правильности заполнения формы используется метод ***Validate*** компонента Form: ***this.Validate();***
10. Для создания логически завершённого приложения-клиента для работы с базой данных, необходимо внести дополнения:
    - Авторизацию на произвольном сервере;
    - Работу со всеми таблицами базы данных;
11. Работа с несколькими таблицами.
    Для работы с несколькими таблицам необходимо добавить несколько адаптеров для обмена данными и реализовать навигацию между графическим представлением данных адаптеров.
12. Добавьте все необходимые представления таблиц базы данных на форму (пункт 7) и сделайте их скрытыми (свойство Visible:false).
13. Создайте главное меню (MenuStrip) для формы редактирования таблиц базы данных. И добавьте пункты переключения, которые будут предоставлять возможность работы с различными таблицами.



Элемент управления ***DataGridView (Windows Forms)*** предоставляет мощный и гибкий способ отображения данных в табличном формате.
На событие по клику по каждому из пунктов добавьте вызов метода, который будет реализовывать смену вида. Реализуйте метод переключения вида, например, так:

```
private void displayTableView(int view) {
    this.disciplinesDataGridView.Hide();
```

```
this.studentsDataGridView.Hide();
this.marksDataGridView.Hide();
this.selectedView = view;
switch (selectedView) {
   case 1: //1 - дисциплины
      this.disciplinesDataGridView.Show(); break;
   case 2: //2 - студенты
      this.studentsDataGridView.Show(); break;
   case 3: //3 - отметки
      this.marksDataGridView.Show(); break;
}}
```

А в тело метода клика поместите вызов данной функции с параметром вида нужной таблицы ***displayTableView(HOMEP);*** В дальнейшем потребуется значение текущего отображаемого вида при работе с базой данных. Сохраните его в закрытой переменной (в примере это ***selectedView***) класса работы с таблицами.

14. Модифицируйте методы обновления данных на сервере и загрузки данных с сервера. Добавьте выбор метода для текущей формы.

```
private void loadFromServer_Click(object sender, EventArgs e) {
   // Загрузка данных с сервера
   switch (selectedView) {
      case 1: //1 - дисциплины
    this.disciplinesTableAdapter.Fill (this.universityDataSet.Disciplines); break;
      case 2: //2 - студенты
        this.studentsTableAdapter.Fill (this.universityDataSet.Students); break;
      case 3: //3 - отметки
            this.marksTableAdapter.Fill (this.universityDataSet.Marks); break;
   }}

private void updateOnServer_Click(object sender, EventArgs e) {
this.Validate();
   // Передача данных на сервер
switch (selectedView) {
case 1: //1 - дисциплины
this.disciplinesBindingSource.EndEdit();
this.disciplinesTableAdapter.Update (this.universityDataSet.Disciplines); break;
case 2: //2 - студенты
this.studentsBindingSource.EndEdit();
this.studentsTableAdapter.Update(this.universityDataSet.Students); break;
case 3: //3 - отметки
this.marksBindingSource.EndEdit();
this.marksTableAdapter.Update(this.universityDataSet.Marks); break;
   }}
```

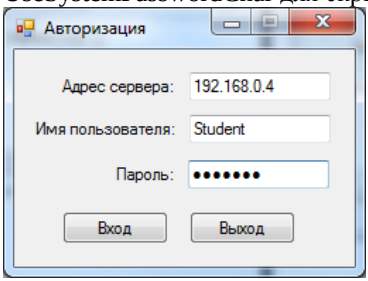
15. Для добавления авторизации на сервере необходимо дать возможность пользователю изменять строку подключения.

Для этого создадим дополнительную форму, открывающуюся по умолчанию при запуске приложения и имеющую поля для ввода данных авторизации.

16. Создайте новую форму по умолчанию. Для этого создайте новую форму в обозревателе решений и отредактируйте программный код функции Main (файл program.cs), изменив название формы по умолчанию.

```
static void Main() {
        Application.EnableVisualStyles();
        Application.SetCompatibleTextRenderingDefault(false);
        Application.Run(new Auth());
```

17. Добавьте на созданную форму соответствующие управляющие элементы (текстовые поля с описанием, поля ввода, кнопки подключения и выхода). При настройке поля для ввода пароля установить свойство UseSystemPasswordChar для скрытия пароля при вводе.



18. Создайте форму работы с таблицами в классе формы авторизации. Добавьте метод передачи данных авторизации из формы авторизации в форму работы с таблицами. Вызовете данный метод (setConnectionDetails) при нажатии кнопки «Вход» и убедитесь в успешности подключения.

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;
namespace UniversityDB {
public partial class Auth : Form {
TableEdit tableEdit = new TableEdit();
public Auth() { InitializeComponent();}
private void loginButton_Click(object sender, EventArgs e) {
try {tableEdit.setConnectDetails(serverInput.Text, userInput.Text,
      passwordInput.Text); }
catch { MessageBox.Show("Произошла ошибка авторизации!"); return;}
tableEdit.Show();
      }
```

private void exitButton_Click(object sender, EventArgs e) {
this.Close();}}

TableEdit – тип (имя) объекта формы, в котором производится работа с таблицами, ***tableEdit*** – экземпляр данной формы, который отображается при успешности авторизации, ***setConnectionDetails*** – метод авторизации на сервере, который необходимо добавить в класс формы ***TableEdit***.

19. Убедитесь, что в программе не сохранён пароль для доступа, так как это противоречит политике безопасности Для этого откройте ***Properties -> Settings*** в обозревателе решений, найдите свойство ***ConnectionString*** и запомните его значение (для использования в функции ***setConnectionDetails***). В дальнейшем значение оставить пустым.
20. Реализуйте метод ***setConnectionDetails*** для соединения со всеми адаптерами, используемыми в форме.

```
public void setConnectDetails(string server, string username, string password)
{ string connectionStr = "Data Source=" + server +  ";
Initial Catalog=University; User ID=" + username + ";
Password=" + password + "; Pooling=False";
this.disciplinesTableAdapter.Connection.ConnectionString =connectionStr;
this.disciplinesTableAdapter.Connection.Open();
this.studentsTableAdapter.Connection.ConnectionString =connectionStr;
this.studentsTableAdapter.Connection.Open();
this.marksTableAdapter.Connection.ConnectionString =connectionStr;
this.marksTableAdapter.Connection.Open();
}
```

4.5. Контрольные вопросы.

1) Назначение .NET Framework.NET Framework. CLR. Class library (FCL).

2) Технология ADO.NET как часть Фреймворка .NET.

3) Провайдеры данных ADO.NET. Объекты Connection, Command, DataReader и DataAdapter.

4) DataSet - ключевой объект ADO.NET. Схема взаимодействия объекта DataSet с подчиненными объектами.

5) Объект DataAdapter. Назначение объекта.

6) Схема работы объекта DataAdapter как моста между источником данных и объектом DataSet.

7) Как происходит передача данных из источника данных в объект DataSet и наоборот?

5. ЛАБОРАТОРНАЯ РАБОТА № 9

Создание хранимых процедур программной системы

5.1 Цель лабораторной работы

Целью лабораторной работы является:

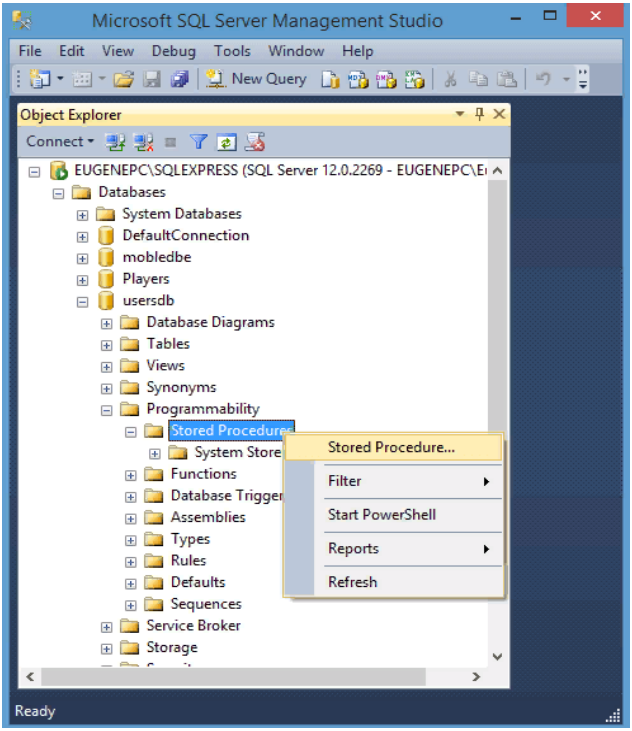
- Получение практического навыка создания и совместной работы хранимой процедуры с базой данных

5.2 Теоретические сведения

Хранимые процедуры являются еще одной формой выполнения запросов к базе данных. Но по сравнению с ранее рассмотренными запросами, которые посылаются из приложения базе данных, хранимые процедуры определяются на сервере и предоставляют большую производительность и являются более безопасными. Для создания хранимых процедур применяется язык Transact-SQL (T-SQL). В этой работе используются только простые операторы. Более подробное описание можно найти в литературе.

Объект ***SqlCommand*** имеет встроенную поддержку хранимых процедур. В частности у него определено свойство ***CommandType,*** которое в качестве значения принимает значение из перечисления ***System.Data.CommandType***. И значение ***System.Data.CommandType.StoredProcedure*** как раз указывает, что будет использоваться хранимая процедура.

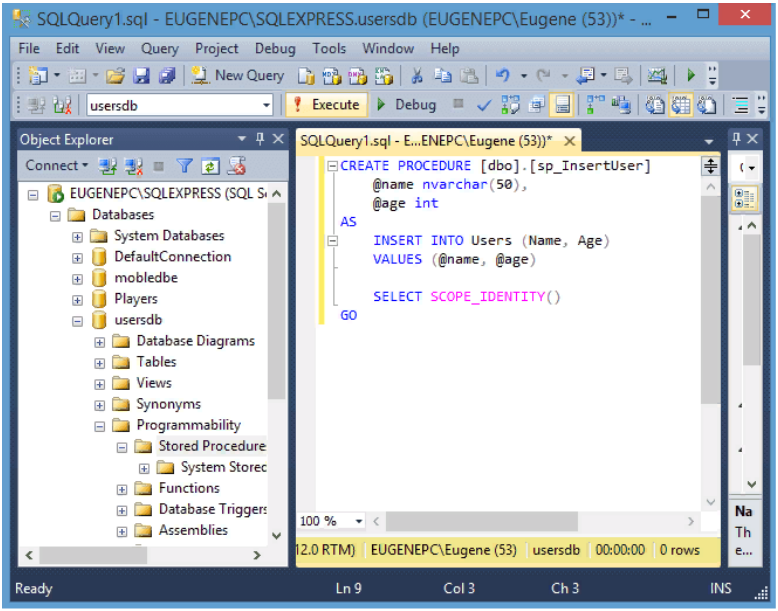
Но чтобы использовать хранимые процедуры, нам надо их вначале создать. Для этого перейдем в SQL Server Management Studio к нашей базе данных, раскроем ее узел и далее выберем ***Programmability->Stored Procedures.*** Нажмем на этот узел правой кнопкой мыши и в контекстном меню выберем пункт ***Stored Procedure***:



В центральной части программы открывает код процедуры, который генерируется по умолчанию. Заменим этот код следующим:

```
CREATE PROCEDURE [dbo].[sp_InsertUser]
   @name nvarchar(50),
   @age int
AS
   INSERT INTO Users (Name, Age)
  VALUES (@name, @age)
  SELECT SCOPE_IDENTITY()
GO
```

Эта процедура выполняет добавление данных. После выражения ***CREATE PROCEDURE*** идет название процедуры. Процедура называется ***"sp_InsertUser"***, и по этому названию мы ее будем вызывать в коде C#. Название процедуры может быть любое. Процедура имеет два входных параметра: ***@name и @age***. Через эти параметры будут передаваться значения для имени и возраста пользователя. В самом теле процедуры после выражения ***AS*** идет стандартное sql-выражение ***INSERT***, которое выполняет добавление данных. И в конце с помощью выражения ***SELECT*** возвращается результат. Выражение ***SCOPE_IDENTITY()*** возвращает ***id*** добавленной записи, поэтому на выходе из процедуры мы получим id новой записи. И завершается процедура ключевым словом ***GO***.



И затем нажмем на кнопку ***Execute.*** После этого в базу данных добавляется хранимая процедура. Подобным образом добавим еще одну процедуру, которая будет возвращать объекты:

```
CREATE PROCEDURE [dbo].[sp_GetUsers]
AS
```

```
SELECT * FROM Users
GO
```

И также для ее добавления нажмем на кнопку Execute.
Теперь перейдем к коду C# [4] и определим следующую программу:

```
class Program
{  static string connectionString = @"Data Source=.\SQLEXPRESS;Initial
   Catalog=usersdb;Integrated Security=True";
   static void Main(string[] args)
   { Console.Write("Введите имя пользователя:");
     string name = Console.ReadLine();
     Console.Write("Введите возраст пользователя:");
      int age = Int32.Parse(Console.ReadLine());
      AddUser(name, age);
      Console.WriteLine();
      GetUsers();
      Console.Read();
   }
   // добавление пользователя
   private static void AddUser(string name, int age)
   { // название процедуры
      string sqlExpression = "sp_InsertUser";
     using (SqlConnection connection = new SqlConnection(connectionString))
      { connection.Open();
         SqlCommand command = new SqlCommand(sqlExpression, connection);
         // указываем, что команда представляет хранимую процедуру
         command.CommandType = System.Data.CommandType.StoredProcedure;
         // параметр для ввода имени
         SqlParameter nameParam = new SqlParameter
         {  ParameterName = "@name", Value = name  };
         // добавляем параметр
         command.Parameters.Add(nameParam);
         // параметр для ввода возраста
         SqlParameter ageParam = new SqlParameter
         {  ParameterName = "@age", Value = age };
         command.Parameters.Add(ageParam);
         var result = command.ExecuteScalar();
         // если нам не надо возвращать id
         //var result = command.ExecuteNonQuery();
          Console.WriteLine("Id добавленного объекта: {0}", result);
      } }

   // вывод всех пользователей
   private static void GetUsers()
   {   // название процедуры
      string sqlExpression = "sp_GetUsers";
```

```
using (SqlConnection connection = new SqlConnection(connectionString))
{   connection.Open();
    SqlCommand command = new SqlCommand(sqlExpression, connection);
    // указываем, что команда представляет хранимую процедуру
    command.CommandType = System.Data.CommandType.StoredProcedure;
    var reader = command.ExecuteReader();
     if (reader.HasRows)
    { Console.WriteLine("{0}\t{1}\t{2}", reader.GetName(0),
      reader.GetName(1), reader.GetName(2));
      while (reader.Read())
        {  int id = reader.GetInt32(0);
           string name = reader.GetString(1);
           int age = reader.GetInt32(2);
           Console.WriteLine("{0} \t{1} \t{2}", id, name, age);
        }
    }
    reader.Close();
} }}
```

Для упрощения кода, обращения к процедурам здесь вынесены в отдельные методы. В методе ***AddUser*** вызывается процедура ***sp_InsertUser***. Ее название передается в конструктор объекта ***SqlCommand*** также, как и обычное ***sql-выражение***. И с помощью выражения

command.CommandType = System.Data.CommandType.StoredProcedure

устанавливается, что это выражение система будет рассматривать как хранимую процедуру.

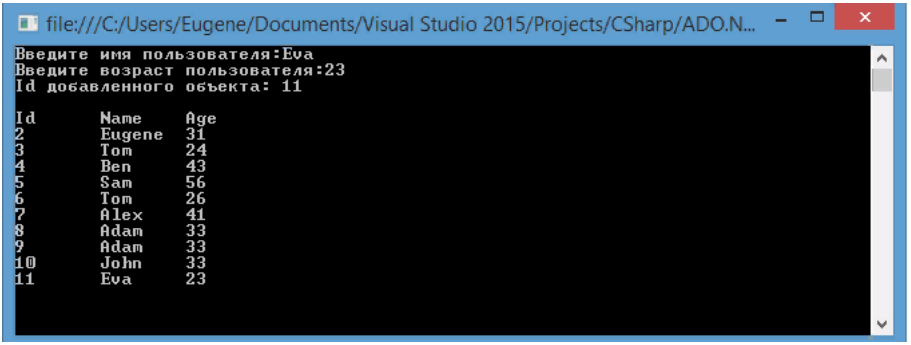
Поскольку процедура получает данные через параметры, то соответственно надо определить эти параметры с помощью объекта ***SqlParameter***. Ему передается название параметра и значение. Названия параметров должны соответствовать тем названиям, которые были определены в коде процедуры.

С помощью метода ***command.Parameters.Add()*** параметры добавляются к процедуре. И затем происходит ее выполнение.

Так как в коде процедуры добавления определено возвращение ***id*** новой записи, то есть возвращение скалярного значения, то для выполнения команды и его получения можно использовать метод ***ExecuteScalar().*** Также можно использовать и метод ***ExecuteNonOuery(),*** только он вернет количество добавленных записей, а не ***id***.

В случае второго метода все еще проще: объекту команды просто передается название процедуры, и так как процедура фактически выполняет выражение SELECT и возвращает набор данных, то для выполнения команды мы можем использовать метод ***ExecuteReader().*** И с помощью ридера получить все данные.

Запустим программу и введем какие-либо данные на добавление:



5.3 Задание на выполнение лабораторной работы

Создайте хранимые процедуры для проекта, выполненного в лабораторной работе № 8.

5.4. Контрольные вопросы.

1) Архитектура .NET Framework.
2) Что такое хранимые процедуры
3) Использование C# Windows Application.
4) Использование средства Transact –SQL для создания хранимых процедур.
5) Хранимые процедуры как еще одна форма выполнения запросов к базе данных.

6. СПИСОК ЛИТЕРАТУРЫ

1. Мартин Р. Чистый код: создание, анализ и рефакторинг. – Санкт Петербург: "Питер", 2016.

2. Э. Фридман, К. Сиерра, Б. Бейтс. Паттерны проектирования. 2-е издание: Пер с англ. - Издательство "Питер", 2016.

3. Э. Гамма, Р. Хелм, Р. Джонсон, Д. Влиссидес. Приемы объектно-ориентированного проектирования. Паттерны проектирования. Пер с англ. - Издательство "Питер", 2015.

4. Подбельский В.В. Язык C#. Базовый курс, М: Финансы и статистика, 2015.

СОДЕРЖАНИЕ