

Министерство науки и высшего образования Российской Федерации

Федеральное государственное бюджетное образовательное учреждение
высшего образования

«ТОМСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ СИСТЕМ
УПРАВЛЕНИЯ И РАДИОЭЛЕКТРОНИКИ» (ТУСУР)

КАФЕДРА КОМПЬЮТЕРНЫХ СИСТЕМ В УПРАВЛЕНИИ И ПРОЕКТИРОВАНИЯ
(КСУП)

А. А. Изюмов

Спецкурс

Методические указания к лабораторным и самостоятельным
работам

Томск
2023

Изюмов А. А.

Спецкурс : методические указания к лабораторным и самостоятельным работам /

А. А. Изюмов – Томск : КСУП, ТУСУР, 2023. – 254 с.

Данное учебное пособие предназначено для бакалавров по направлениям 27.03.04 «Управление в технических системах», 27.04.04 «Управление в технических системах», 09.03.01 «Информатика и вычислительная техника», 09.04.01 «Информатика и вычислительная техника». Также пособие полезно для преподавателей и студентов высших учебных заведений технических направлений подготовки.

В пособии представлены методы построения UML-диаграмм, основы работы в 1С:Предприятии и построения на её базе конфигурации.

Содержание

Введение.....	6
Лабораторная работа №1	7
Цель работы	7
Краткое изложение теоретической части	7
Пример выполнения задания	30
Задание	36
Лабораторная работа №2	37
Цель работы	37
Краткое изложение теоретической части	37
Создание модели в Visual Paradigm	71
Пример выполнения задания	75
Задание	82
Лабораторная работа №3	83
Цель работы	83
Краткое изложение теоретической части	83
Пример выполнения задания	97
Задание	106
Лабораторная работа №4	106
Цель работы	106
Краткое изложение теоретической части	106
Пример выполнения задания	118
Задание	122
Лабораторная работа №5	122
Цель работы	122
Краткое изложение теоретической части	122
Диаграммы деятельности	125
Пример выполнения задания	135

Задание	136
Лабораторная работа №6	137
Цель работы	137
Краткое изложение теоретической части	137
Символы ERD-связей.....	142
Символы ERD-атрибутов	142
Символы физических ER-диаграмм.....	143
Нормальные формы	146
Нотация ER-диаграмм	148
Кардинальность и ординальность	148
Пример выполнения задания	150
Задание	153
Лабораторная работа №7	154
Цель работы	154
Краткое изложение теоретической части	154
Пример выполнения задания	167
Задание	168
Лабораторная работа №8	169
Цель работы	169
Краткое изложение теоретической части	169
Задание	200
9. Организация самостоятельной работы	200
9.1 Рабочая программа и план обучения	200
9.2 Теоретический материал	201
9.3 Рекомендации по работе с учебной и научной литературой.....	202
9.4 Консультации	205
9.5 Практические и лабораторные работы	205
9.6 Контрольные мероприятия.....	207
9.7 Итоговая аттестация	208
10. Рекомендуемая литература	208

Приложение 1 – Список тем для лабораторных работ.....	210
Приложение 2 – Краткий глоссарий терминов ООП	211
Приложение 3 – Листинг СУБД	225

Введение

Целью данного курса является формирование у обучающихся навыков формализации задачи построения комплекса автоматизации бизнес процессов. Эта задача подразделяется на следующие основные составляющие:

- построение модели бизнес процесса в SADT-диаграмме;
- выделение в модели тех участков, автоматизация которых повлечет значительный выигрыш в функционировании всей системы;
- разработка программного обеспечения по автоматизации и создание сопроводительной документации с использованием специализированного программного комплекса в соответствии с UML;
- построение ER-диаграмм и создание базы данных;
- создание СУБД.

В ходе курса предполагается сформировать у обучающегося следующие навыки:

- изучение основ языка UML;
- работа со средствами UML-проектирования;
- закрепление навыков построения ER-диаграмм;
- изучение основ SQL;
- формирование базовых навыков программирования на Python.

Выполнение лабораторных работ, таким образом, позволяет пройти полный путь от построения модели бизнес процесса до получения готового коммерческого приложения, открытого для дальнейшей доработки любым программистом, с полным комплексом документации, оформленным в соответствии с общемировыми стандартами.

Изучение данного курса заканчивается получением зачета или экзаменационной оценки (в зависимости от того, что предусмотрено учебным планом).

Лабораторная работа №1

Цель работы

Целью данной работы является построение SADT диаграммы выбранного бизнес процесса. Построение диаграммы происходит в среде BPWin. Список возможных тем приведен в приложении 1.

Краткое изложение теоретической части

SADT (акроним от англ. structured analysis and design technique) — методология структурного анализа и проектирования, интегрирующая процесс моделирования, управление конфигурацией проекта, использование дополнительных языковых средств и руководство проектом со своим графическим языком. Процесс моделирования может быть разделен на несколько этапов: опрос экспертов, создание диаграмм и моделей, распространение документации, оценка адекватности моделей и принятие их для дальнейшего использования. Этот процесс хорошо отлажен, потому что при разработке проекта специалисты выполняют конкретные обязанности, а библиотекарь обеспечивает своевременный обмен информацией.

Принципы построения модели IDEF0

На начальных этапах создания ИС необходимо понять, как работает организация, которую собираются автоматизировать. Никто в организации не знает, как она работает в той мере подробности, которая необходима для создания ИС. Руководитель хорошо знает работу в целом, но не в состоянии вникнуть в детали работы каждого рядового сотрудника. Рядовой сотрудник хорошо знает, что творится на его рабочем месте, но плохо знает, как работают коллеги. Поэтому для описания работы предприятия необходимо построить модель. Такая модель должна быть адекватна предметной

области, следовательно, она должна содержать в себе знания всех участников бизнес-процессов организации.

В IDEF0 система представляется как совокупность взаимодействующих работ или функций. Такая чисто функциональная ориентация является принципиальной – функции системы анализируются независимо от объектов, которыми они оперируют. Это позволяет более четко смоделировать логику и взаимодействие процессов организации.

Под *моделью* в IDEF0 понимают описание системы (текстовое и графическое), которое должно дать ответ на некоторые заранее определенные вопросы.

Моделируемая система рассматривается как произвольное подмножество Вселенной. Произвольное потому, что, во-первых, мы сами умозрительно определяем, будет ли некий объект компонентом системы, или мы будем его рассматривать как внешнее воздействие, и, во-вторых, оно зависит от точки зрения на систему. Система имеет границу, которая отделяет ее от остальной Вселенной. Взаимодействие системы с окружающим миром описывается как вход (нечто, что перерабатывается системой), выход (результат деятельности системы), управление (стратегии и процедуры, под управлением которых производится работа) и механизм (ресурсы, необходимые для проведения работы). Находясь под управлением, система преобразует входы в выходы, используя механизмы.

Процесс моделирования какой-либо системы в IDEF0 начинается с определения контекста, т. е. наиболее абстрактного уровня описания системы в целом. В контекст входит определение субъекта моделирования, цели и точки зрения на модель.

Под субъектом понимается сама система, при этом необходимо точно установить, что входит в систему, а что лежит за ее пределами, другими словами, мы должны определить, что мы будем в дальнейшем рассматривать как компоненты системы, а что как внешнее воздействие. На определение субъекта системы будет существенно влиять позиция, с

которой рассматривается система, и цель моделирования – вопросы, на которые построенная модель должна дать ответ. Другими словами, первоначально необходимо определить область (Scope) моделирования. Описание области как системы в целом, так и ее компонентов является основой построения модели. Хотя предполагается, что в течение моделирования область может корректироваться, она должна быть в основном сформулирована изначально, поскольку именно область определяет направление моделирования и когда должна быть закончена модель. При формулировании области необходимо учитывать два компонента – широту и глубину. Широта подразумевает определение границ модели – мы определяем, что будет рассматриваться внутри системы, а что снаружи. Глубина определяет, на каком уровне детализации модель является завершенной. При определении глубины системы необходимо не забывать об ограничениях времени – трудоемкость построения модели растет в геометрической прогрессии от глубины декомпозиции. После определения границ модели предполагается, что новые объекты не должны вноситься в моделируемую систему; поскольку все объекты модели взаимосвязаны, внесение нового объекта может быть не просто арифметической добавкой, но в состоянии изменить существующие взаимосвязи. Внесение таких изменений в готовую модель является, как правило, очень трудоемким процессом (так называемая проблема «плавающей области»).

Цель моделирования (Purpose). Модель не может быть построена без четко сформулированной цели. Цель должна отвечать на следующие вопросы:

- Почему этот процесс должен быть замоделирован?
- Что должна показывать модель?
- Что может получить читатель?

Формулировка цели позволяет команде аналитиков сфокусировать усилия в нужном направлении. Примерами формулирования цели могут

быть следующие утверждения: *«Идентифицировать и определить текущие проблемы, сделать возможным анализ потенциальных улучшений»*, *«Идентифицировать роли и ответственность служащих для написания должностных инструкций»*, *«Описать функциональность предприятия с целью написания спецификаций информационной системы»* и т. д.

Точка зрения (Viewpoint). Хотя при построении модели учитываются мнения различных людей, модель должна строиться с единой точки зрения. Точку зрения можно представить как взгляд человека, который видит систему в нужном для моделирования аспекте. Точка зрения должна соответствовать цели моделирования. Очевидно, что описание работы предприятия с точки зрения финансиста и технолога будет выглядеть совершенно по-разному, поэтому в течение моделирования важно оставаться на выбранной точке зрения. Как правило, выбирается точка зрения человека, ответственного за моделируемую работу в целом. Часто при выборе точки зрения на модель важно задокументировать дополнительные альтернативные точки зрения.

IDEF0-модель предполагает наличие четко сформулированной цели, единственного субъекта моделирования и одной точки зрения. Не стоит пренебрегать их заданием в модели – ведь от того, какая точка зрения выбрана, меняется подход к проектированию: директору неинтересен процесс заправки картриджей, а инженеру-электронику будут неинтересны отчеты в государственные органы статистики, которые подает экономист. В программе BPWin в *Purpose* следует внести цель и точку зрения, а в закладку *Definition* – определение модели и описание области. При использовании онлайн средств проектирования можно использовать комментарии.

Также можно описать статус модели (черновой вариант, рабочий, окончательный и т. д.), время создания и последнего редактирования. Также, описываются источники информации (*Source*) для построения модели (например, *«Опрос экспертов предметной области и анализ*

документации»). Общая информация (*General*) служит для внесения имени проекта и модели, имени и инициалов автора и временных рамок модели — AS-IS и TO-BE: обычно сначала строится модель существующей организации работы — AS-IS (как есть). Найденные в модели AS-IS недостатки можно исправить при создании модели TO-BE (как будет) — модели новой организации бизнес-процессов. Модель TO-BE нужна для анализа альтернативных/лучших путей выполнения работы и документирования того, как компания будет делать бизнес в будущем.

Основу методологии IDEF0 составляет графический язык описания бизнес-процессов. Модель в нотации IDEF0 представляет собой совокупность иерархически упорядоченных и взаимосвязанных диаграмм. Каждая диаграмма является единицей описания системы и располагается на отдельном листе.

Модель может содержать четыре типа диаграмм:

- контекстную диаграмму (в каждой модели может быть только одна контекстная диаграмма);
- диаграммы декомпозиции;
- диаграммы дерева узлов;
- диаграммы только для экспозиции (FEO).

Контекстная диаграмма является вершиной древовидной структуры диаграмм и представляет собой самое общее описание системы и ее взаимодействия с внешней средой. После описания системы в целом проводится разбиение ее на крупные фрагменты. Этот процесс называется функциональной декомпозицией, а диаграммы, которые описывают каждый фрагмент и взаимодействие фрагментов, называются диаграммами декомпозиции. После декомпозиции контекстной диаграммы проводится декомпозиция каждого большого фрагмента системы на более мелкие и так далее, до достижения нужного уровня подробности описания. После каждого сеанса декомпозиции проводятся сеансы экспертизы — эксперты предметной области указывают на соответствие реальных бизнес-процессов

созданным диаграммам. Найденные несоответствия исправляются, и только после прохождения экспертизы без замечаний можно приступать к следующему сеансу декомпозиции. Так достигается соответствие модели реальным бизнес-процессам на любом и каждом уровне модели. Синтаксис описания системы в целом и каждого ее фрагмента одинаков во всей модели.

Диаграмма дерева узлов показывает иерархическую зависимость работ, но не взаимосвязи между работами. Диаграмм деревьев узлов может быть в модели сколь угодно много, поскольку дерево может быть построено на произвольную глубину и не обязательно с корня.

Диаграммы для экспозиции (FEO) строятся для иллюстрации отдельных фрагментов модели, для иллюстрации альтернативной точки зрения, либо для специальных целей.

Работы (Activity)

Работы обозначают поименованные процессы, функции или задачи, которые происходят в течение определенного времени и имеют распознаваемые результаты. Работы изображаются в виде прямоугольников. Все работы должны быть названы и определены. Имя работы должно быть выражено отглагольным существительным, обозначающим действие (например, «Посетить лекцию», «Сдать зачёт» и т.д.). Контекстная диаграмма с единственной работой, изображающей систему в целом, приведена на рис. 1.1.

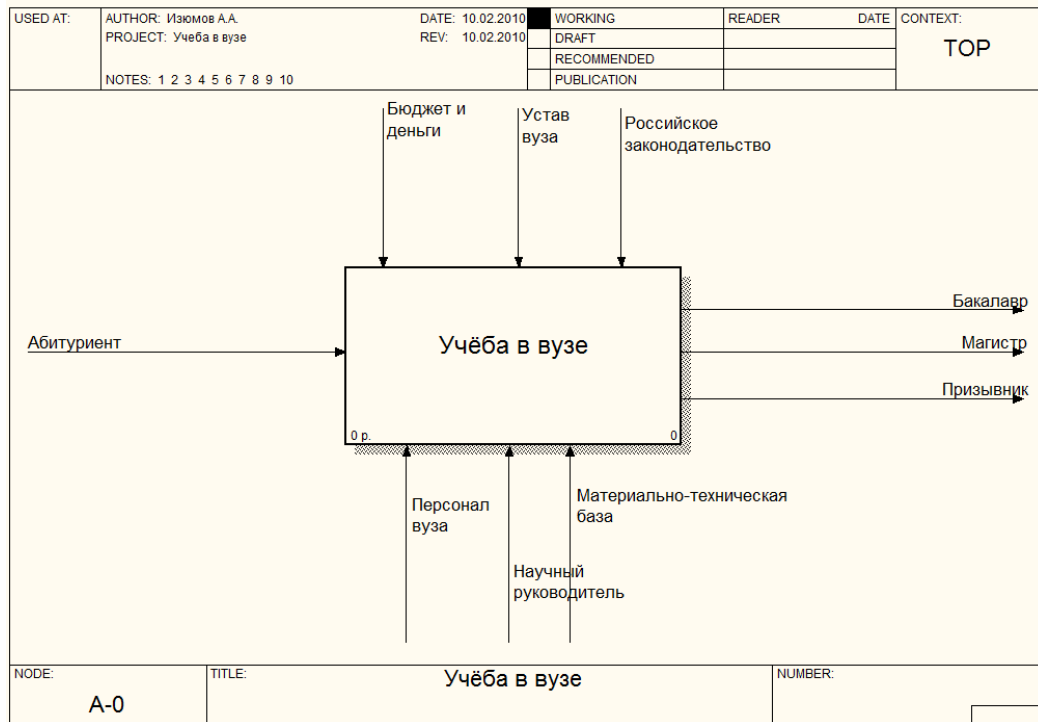


Рисунок 1.1 – Система в виде «Черного ящика»

Диаграммы декомпозиции содержат родственные работы, т.е. дочерние работы, имеющие общую родительскую работу. SADT-модели развиваются в процессе структурной декомпозиции сверху вниз. Сначала декомпозируется один блок, являющийся границей модели, на одной диаграмме, которая имеет от трех до шести блоков, затем декомпозируется один (или больше) из этих блоков на другой диаграмме с тремя-шестью блоками и т.д. Название диаграммы совпадает с названием декомпозируемого блока. Результатом этого процесса является модель, диаграмма верхнего уровня которой описывает систему в общих терминах "черного ящика", а диаграммы нижнего уровня описывают очень детализированные аспекты и операции системы.

Таким образом, каждая диаграмма представляет собой некоторую законченную часть всей модели. В методологии SADT идентифицируется каждая диаграмма данной модели посредством того, что называется «номер узла». Номер узла для контекстной диаграммы имеет следующий вид: название модели или аббревиатура, косая черта, заглавная буква A (Activity

в функциональных диаграммах), дефис и ноль. Например, номером узла для контекстной диаграммы, приведенной на рис.1.1. является А-0. Номером узла диаграммы, декомпозирующей контекстную диаграмму, является тот же номер узла, но без дефиса (например, А0). Все другие номера узлов образуются посредством добавления к номеру узла родительской диаграммы номера декомпозируемого блока. На рис.1.2 с номером узла А0 показана декомпозиция диаграммы А-0.

Работы на диаграммах декомпозиции обычно располагаются по диагонали от левого верхнего угла к правому нижнему.

Такой порядок называется порядком доминирования. Согласно этому принципу расположения в левом верхнем углу располагается самая важная работа или работа, выполняемая по времени первой. Далее вправо вниз располагаются менее важные или выполняемые позже работы. Такое расположение облегчает чтение диаграмм, кроме того, на нем основывается понятие взаимосвязей работ (см. ниже).

Каждая из работ на диаграмме декомпозиции может быть в свою очередь декомпозирована. На диаграмме декомпозиции работы нумеруются автоматически слева направо. Номер работы показывается в правом нижнем углу. В левом верхнем углу изображается небольшая диагональная черта, которая показывает, что данная работа не была декомпозирована. Так, на рис. 1.3 работа «*Поступить в вуз*» имеет номер 1 и не была еще декомпозирована. Работа «*Сдать первую сессию*» (номер 2) имеет нижний уровень декомпозиции.

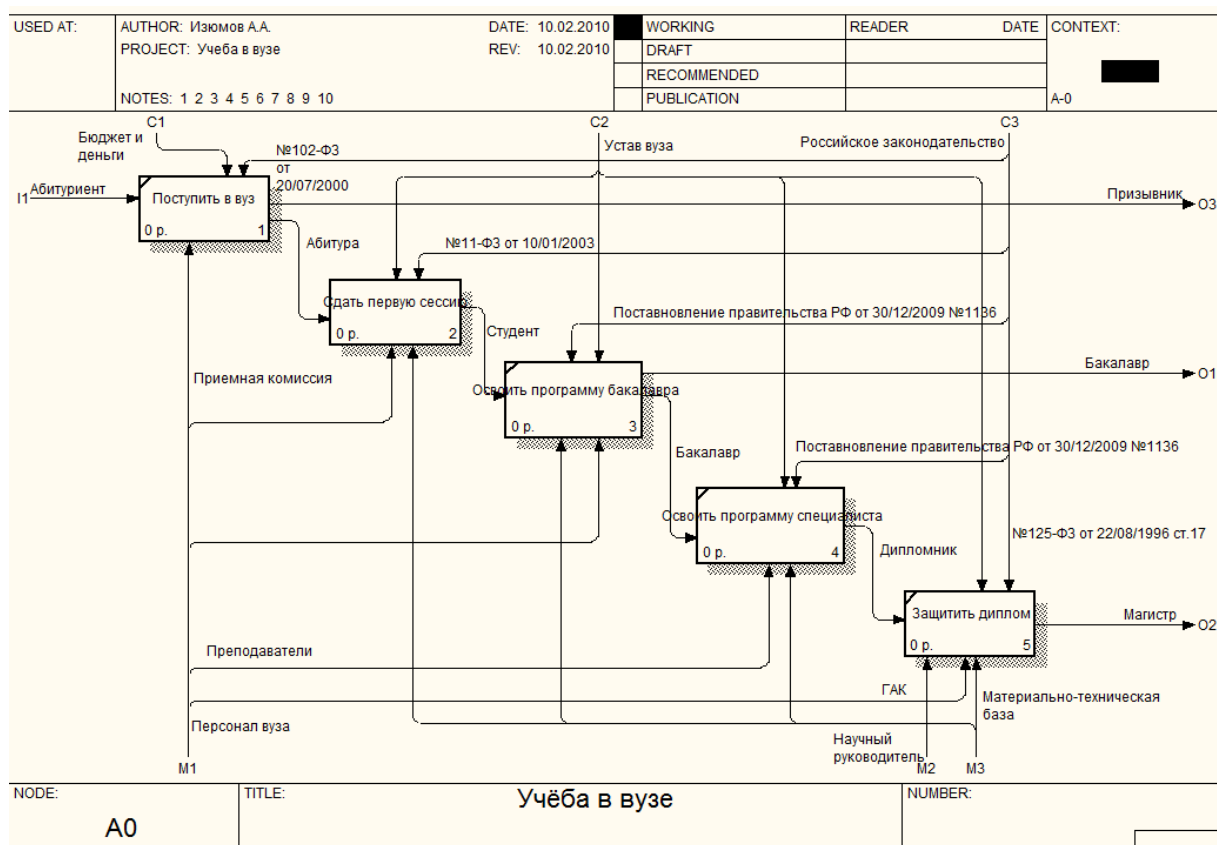


Рисунок 1.2 – Пример диаграммы декомпозиции

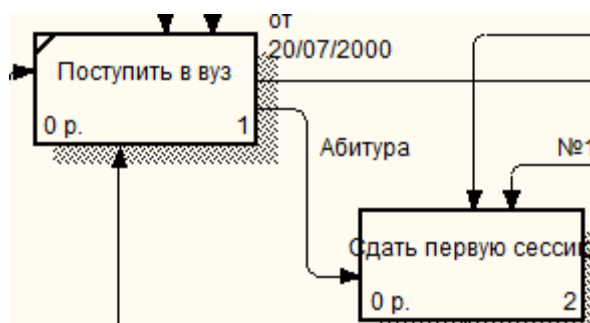


Рисунок 1.3 – Пример декомпозируемых работ

Стрелки (Arrow)

Взаимодействие работ с внешним миром и между собой описывается в виде стрелок. Стрелки представляют собой некую информацию и именуются существительными (например, «Дипломник», «Преподаватели», «Материально-техническая база»).

В IDEF0 различают пять типов стрелок:

Вход (Input) – материал или информация, которые используются или преобразуются работой для получения результата (выхода). Допускается, что работа может не иметь ни одной стрелки входа. Каждый тип стрелок подходит к определенной стороне прямоугольника, изображающего работу, или выходит из нее. Стрелка входа рисуется как входящая в левую грань работы (рис. 1.1 – «Абитуриент»). При описании технологических процессов (для этого и был придуман IDEF0) не возникает проблем определения входов. Очень часто сложно определить, являются ли данные входом или управлением. В этом случае подсказкой может служить то, перерабатываются/изменяются ли данные в работе или нет. Если изменяются, то скорее всего это вход, если нет – управление.

Управление (Control) – правила, стратегии, процедуры или стандарты, которыми руководствуется работа. Каждая работа должна иметь хотя бы одну стрелку управления. Стрелка управления рисуется как входящая в верхнюю грань работы. Управление влияет на работу, но не преобразуется работой

Выход (Output) – материал или информация, которые производятся работой. Каждая работа должна иметь хотя бы одну стрелку выхода. Работа без результата не имеет смысла и не должна моделироваться. Стрелка выхода рисуется как исходящая из правой грани работы.



Механизм (Mechanism) - ресурсы, которые выполняют работу, например персонал предприятия, станки, устройства и т. д. Стрелка механизма рисуется как входящая в нижнюю грань работы. По усмотрению аналитика стрелки механизма могут не изображаться в модели.

Вызов (Call) – специальная стрелка, указывающая на другую модель работы. Стрелка вызова рисуется как исходящая из нижней грани работы. Стрелка вызова используется для указания того, что некоторая работа выполняется за пределами моделируемой системы.

Стрелки на контекстной диаграмме служат для описания взаимодействия системы с окружающим миром. Они могут начинаться у

границы диаграммы и заканчиваться у работы, или наоборот. Такие стрелки называются граничными.

Для внесения граничной стрелки входа следует:

- щелкнуть по кнопке *Procedure Arrow Tool* ();
- в палитре инструментов перенести курсор к левой стороне экрана, пока не появится начальная штриховая полоска;
- щелкнуть один раз по полоске (откуда выходит стрелка) и еще раз в левой части работы со стороны входа (где заканчивается стрелка);
- вернуться в палитру инструментов и выбрать *Pointer Tool* ();
- щелкнуть правой кнопкой мыши на линии стрелки, во всплывающем меню выбрать *Name* и добавить имя стрелки в закладке *Name* диалога *IDEF0 Arrow Name*.

Стрелки управления, выхода, механизма и выхода изображаются аналогично. Для рисования стрелки выхода, например, следует щелкнуть по кнопке с символом стрелки в палитре инструментов, щелкнуть в правой части работы со стороны выхода (где начинается стрелка), перенести курсор к правой стороне экрана, пока не появится начальная штриховая полоска, и щелкнуть один раз по штриховой полоске.

Имена вновь внесенных стрелок автоматически заносятся в словарь (Arrow Dictionary). Словарь стрелок (*Model/Arrow Editor*) решает очень важную задачу. Диаграммы создаются аналитиком для того, чтобы провести сеанс экспертизы, т. е. обсудить диаграмму со специалистом предметной области. В любой предметной области формируется профессиональный жаргон, причем очень часто жаргонные выражения имеют нечеткий смысл и воспринимаются разными специалистами по-разному. В то же время аналитик – автор диаграмм должен употреблять те выражения, которые наиболее понятны экспертам. Поскольку формальные определения часто сложны для восприятия, аналитик вынужден употреблять профессиональный жаргон, а, чтобы не возникло неоднозначных трактовок,

в словаре стрелок каждому понятию можно дать расширенное и, если это необходимо, формальное определение.

Содержимое словаря стрелок можно распечатать в виде отчета (меню *<Tools/Reports/Arrow Report...>*) и получить тем самым толковый словарь терминов предметной области, использующихся в модели.

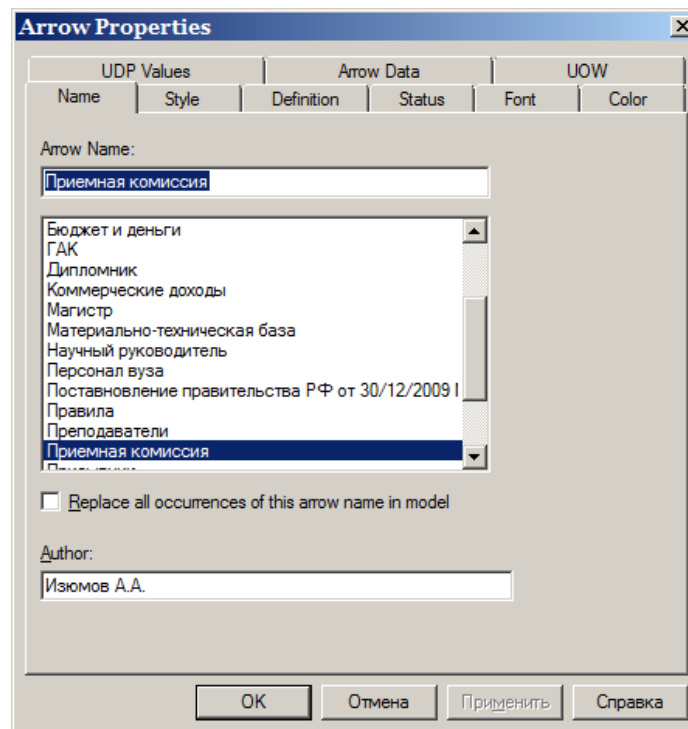


Рисунок 1.4 – Диалог IDEF0 Arrow Properties

Диаграмма декомпозиции предназначена для детализации работы. В отличие от моделей, отображающих структуру организации, работа на диаграмме верхнего уровня в IDEF0 – это не элемент управления нижестоящими работами. Работы нижнего уровня – это то же самое, что работы верхнего уровня, но в более детальном изложении. Как следствие этого границы работы верхнего уровня – это то же самое, что границы диаграммы декомпозиции. ICOM (аббревиатура от Input, Control, Output и Mechanism) – коды, предназначенные для идентификации граничных стрелок. Код ICOM содержит префикс, соответствующий типу стрелки (I, C, O или M), и порядковый номер (рис. 1.5).



Рисунок 1.5 – Фрагмент диаграммы декомпозиции с ICOM -кодами (I1 и C1)

ВРwin вносит ICOM-коды автоматически. Для отображения ICOM-кодов следует включить опцию Show ICOM codes на закладке *Display* диалога *Model Properties* (меню <Model/Model Properties>).

Несвязанные граничные стрелки (unconnected border arrow). При декомпозиции работы входящие в нее и исходящие из нее стрелки (кроме стрелки вызова) автоматически появляются на диаграмме декомпозиции (миграция стрелок), но при этом не касаются работ. Такие стрелки называются несвязанными и воспринимаются в ВРwin как синтаксическая ошибка (рис. 1.6).

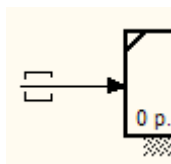


Рисунок 1.6 – Пример несвязанных стрелок

Разветвляющиеся и сливающиеся стрелки. Одни и те же данные или объекты, порожденные одной работой, могут использоваться сразу в нескольких других работах. С другой стороны, стрелки, порожденные в разных работах, могут представлять собой одинаковые или однородные данные, или объекты, которые в дальнейшем используются или перерабатываются в одном месте. Для моделирования таких ситуаций в IDEF0 используются разветвляющиеся и сливающиеся стрелки. Для разветвления стрелки нужно в режиме редактирования стрелки щелкнуть по фрагменту стрелки и по соответствующему сегменту работы. Для слияния

двух стрелок выхода нужно в режиме редактирования стрелки сначала щелкнуть по сегменту выхода работы, а затем по соответствующему фрагменту стрелки.

Смысл разветвляющихся и сливающихся стрелок передается именованием каждой ветви стрелок. Существуют определенные правила именования таких стрелок. Рассмотрим их на примере разветвляющихся стрелок. Если стрелка именована до разветвления, а после разветвления ни одна из ветвей не именована, то подразумевается, что каждая ветвь моделирует те же данные или объекты, что и ветвь до разветвления. Если стрелка именована до разветвления, а после разветвления какая-либо из ветвей не именована, то подразумевается, что эти ветви соответствуют именованию. Если при этом какая-либо ветвь после разветвления осталась неименованной, то подразумевается, что она моделирует те же данные или объекты, что и ветвь до разветвления (рис. 1.7).

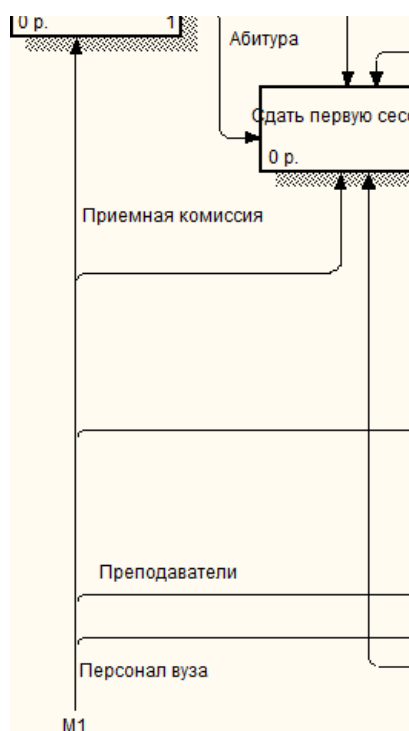


Рисунок 1.7 – Пример именования разветвляющейся стрелки

Все работы модели нумеруются. Номер состоит из префикса и числа. Может быть использован префикс любой длины, но обычно используют префикс А. Контекстная (корневая) работа дерева имеет номер А0. Работы 1 декомпозиции А0 имеют номера А1, А2, А3 и т. д. Работы декомпозиции нижнего уровня имеют номер родительской работы и очередной порядковый номер, например, работы декомпозиции А3 будут иметь номера А31, А32, А33, А34 и т. д. Работы образуют иерархию, где каждая работа может иметь одну родительскую и несколько дочерних работ, образуя дерево. Такое дерево называют деревом узлов, а вышеописанную нумерацию – нумерацией по узлам.

Диаграммы IDEF0 имеют двойную нумерацию. Во-первых, диаграммы имеют номера по узлу. Контекстная диаграмма всегда имеет номер А-0, декомпозиция контекстной диаграммы – номер А0, остальные диаграммы декомпозиции – номера по соответствующему узлу (например, А1, А2, А21, А213 и т. д.). ВРwin автоматически поддерживает нумерацию по узлам, т. е. при проведении декомпозиции создается новая диаграмма и ей автоматически присваивается соответствующий номер. В результате проведения экспертизы диаграммы могут уточняться и изменяться, следовательно, могут быть созданы различные версии одной и той же (с точки зрения ее расположения в дереве узлов) диаграммы декомпозиции. ВРwin позволяет иметь в модели только одну диаграмму декомпозиции в данном узле. Прежние версии диаграммы можно хранить в виде бумажной копии либо как FEO-диаграмму. (К сожалению, при создании FEO-диаграмм отсутствует возможность отката, т. е. можно получить из диаграммы декомпозиции FEO, но не наоборот.) В любом случае следует отличать различные версии одной и той же диаграммы. Для этого существует специальный номер – С-number, который должен присваиваться автором модели вручную. С-number – это произвольная строка, но рекомендуется придерживаться стандарта, когда номер состоит из буквенного префикса и порядкового номера, причем в качестве префикса

используются инициалы автора диаграммы, а порядковый номер отслеживается автором вручную, например, МСВ00021.

Диаграммы дерева узлов и ФЕО

Диаграмма дерева узлов (рис. 1.8) показывает иерархию работ в модели и позволяет рассмотреть всю модель целиком, но не показывает взаимосвязи между работами (стрелки). Процесс создания модели работ является итерационным, следовательно, работы могут менять свое расположение в дереве узлов многократно. Чтобы не запутаться и проверить способ декомпозиции, следует после каждого изменения создавать диаграмму дерева узлов. Впрочем, BPwin имеет мощный инструмент навигации по модели – Model Explorer, который позволяет представить иерархию работ и диаграмм в удобном и компактном виде, однако этот инструмент не является составляющей стандарта IDEF0.

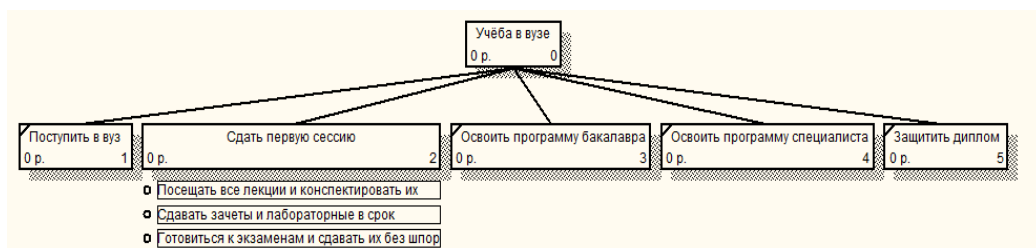


Рисунок 1.8 – Диаграмма дерева узлов

Для создания диаграммы дерева узлов следует выбрать в меню пункт *<Diagram/Add Node Tree>*. Возникает диалог формирования диаграммы дерева узлов *Node Tree Definition* (рис. 1.9).

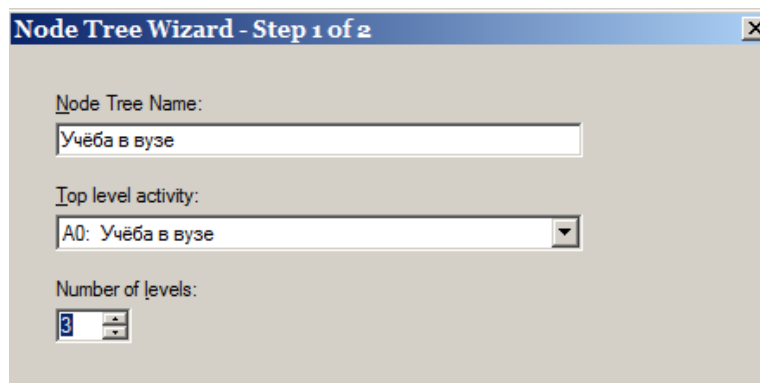


Рисунок 1.9 – Диалог настройки диаграммы дерева узлов

В диалоге *Node Tree Definition* следует указать глубину дерева – *Number of Levels* (по умолчанию 3) и корень дерева (по умолчанию – родительская работа текущей диаграммы). По умолчанию нижний уровень декомпозиции показывается в виде списка, остальные работы – в виде прямоугольников. Для отображения всего дерева в виде прямоугольников следует выключить опцию *Bullet Last Level*. При создании дерева узлов следует указать имя диаграммы, поскольку, если в нескольких диаграммах в качестве корня на дереве узлов использовать одну и ту же работу, все эти диаграммы получат одинаковый номер (номер узла + постфикс N, например AON) и в списке открытых диаграмм (пункт меню *<Window>*) их можно будет различить только по имени.

Диаграммы «только для экспозиции» (FEO) часто используются в модели для иллюстрации других точек зрения, для отображения отдельных деталей, которые не поддерживаются явно синтаксисом IDEF0. Диаграммы FEO позволяют нарушить любое синтаксическое правило, поскольку по сути являются просто картинками – копиями стандартных диаграмм и не включаются в анализ синтаксиса. Например, работа на диаграмме FEO может не иметь стрелок управления и выхода. С целью обсуждения определенных аспектов модели с экспертом предметной области может быть создана диаграмма только с одной работой и одной стрелкой, поскольку стандартная диаграмма декомпозиции содержит множество деталей, не относящихся к теме обсуждения и дезориентирующих эксперта. Но если FEO используется для иллюстрации альтернативных точек зрения (альтернативный контекст), рекомендуется все-таки придерживаться синтаксиса IDEF0. Для создания диаграммы FEO следует выбрать пункт меню *<Diagram/Add FEO Diagram.>*

Слияние и расщепление моделей

Возможность слияния и расщепления моделей обеспечивает коллективную работу над проектом. Так, руководитель проекта может создать декомпозицию верхнего уровня и дать задание аналитикам продолжить декомпозицию каждой ветви дерева в виде отдельных моделей. После окончания работы над отдельными ветвями все подмодели могут быть слиты в единую модель. С другой стороны, отдельная ветвь модели может быть отщеплена для использования в качестве независимой модели, для доработки или архивирования.

ВРwin использует для слияния и разветвления моделей стрелки вызова. Для слияния необходимо выполнить следующие условия:

- обе сливаемые модели должны быть открыты в ВРwin;
- имя модели-источника, которое присоединяют к модели-цели, должно совпадать с именем стрелки вызова работы в модели-цели;
- стрелка вызова должна исходить из недекомпозируемой работы (работа должна иметь диагональную черту в левом верхнем углу);
- имена контекстной работы подсоединяемой модели-источника и работы на модели-цели, к которой мы подсоединяем модель-источник, должны совпадать;
- Модель-источник должна иметь по крайней мере одну диаграмму декомпозиции.

Для слияния моделей нужно щелкнуть правой кнопкой мыши по работе со стрелкой вызова в модели-цели и во всплывающем меню выбрать пункт *<Merge Model>*.

Проведение экспертизы

Цикл автор-читатель предназначен для обеспечения обратной связи при построении модели. Он включает определенные формализованные процедуры, предписывающие правила координации деятельности участников создания модели. В работе над моделью принимают участие

специалисты разных специальностей – аналитики (авторы), эксперты предметной области (читатели), библиотекари и комитет технического контроля. На практике зачастую сеанс экспертизы проводится в форме устного собеседования между автором и экспертом. В этом случае особенно важно вносить замечания эксперта и комментарии автора в диаграмму для документирования всех идей, возникших в результате моделирования.

Если это необходимо, проводится дополнительная экспертиза у того же или у другого эксперта.

После прохождения нескольких циклов число замечаний обычно уменьшается и диаграмма становится стабильной. В процессе изменения диаграмма может менять свой статус, который должен быть отражен в каркасе. Когда автор считает, что диаграмма уже достаточно проработана и достигла уровня «*Recommended*», он пересылает ее на утверждение в комитет технического контроля, где она проходит окончательную экспертизу. После внесения замечаний и окончательных изменений диаграмма (или набор диаграмм) окончательно утверждается, получает статус «*Publication*» и может быть распечатана и распространена среди участников проекта.



Диаграммы потоков данных (Data Flow Diagramming)

Диаграммы потоков данных (Data flow diagramming, DFD) используются для описания документооборота и обработки информации. Подобно IDEF0, DFD представляет модельную систему как сеть связанных между собой работ. Их можно использовать как дополнение к модели IDEF0 для более наглядного отображения текущих операций документооборота в корпоративных системах обработки информации. DFD описывает:

- функции обработки информации (работы);
- документы (стрелки, arrow), объекты, сотрудников или отделы, которые участвуют в обработке информации;

- внешние ссылки (external references), которые обеспечивают интерфейс с внешними объектами, находящимися за границами моделируемой системы;
- таблицы для хранения документов (хранилище данных, data store).

Для того чтобы дополнить модель IDEF0 диаграммой DFD, нужно в процессе декомпозиции в диалоге Activity Box Count «кликнуть» по радио-кнопке DFD. В палитре инструментов на новой диаграмме DFD появляются новые кнопки:

- добавить в диаграмму внешнюю ссылку (*External Reference*). Внешняя ссылка является источником или приемником данных извне модели ();
- добавить в диаграмму хранилище данных (*Data store*). Хранилище данных позволяет описать данные, которые необходимо сохранить в памяти прежде, чем использовать в работах (.

В отличие от стрелок IDEF0, которые представляют собой жесткие взаимосвязи, стрелки DFD показывают, как объекты (включая данные) двигаются от одной работы к другой. Это представление потоков совместно с хранилищами данных и внешними сущностями делает модели DFD более похожими на физические характеристики системы – движение объектов (data flow), хранение объектов (data stores), поставка и распространение объектов (external entities).

В DFD работы представляют собой функции системы, преобразующие входы в выходы. Хотя работы изображаются прямоугольниками со скругленными углами, смысл их совпадает со смыслом работ IDEF0 и IDEF3. Так же как работы IDEF3, они имеют входы и выходы, но не поддерживают управления и механизмы, как IDEF0.

Внешние сущности изображают входы в систему и/или выходы из системы. Внешние сущности изображаются в виде прямоугольника с тенью и обычно располагаются по краям диаграммы. Одна внешняя сущность может быть использована многократно на одной или нескольких

диаграммах. Обычно такой прием используют, чтобы не рисовать слишком длинных и запутанных стрелок.

Стрелки описывают движение объектов из одной части системы в другую. Поскольку в DFD каждая сторона работы не имеет четкого назначения, как в IDEF0, стрелки могут подходить и выходить из любой грани прямоугольника работы. В DFD также применяются двунаправленные стрелки для описания диалогов типа «команда-ответ» между работами, между работой и внешней сущностью и между внешними сущностями.

В отличие от стрелок, описывающих объекты в движении, хранилища данных изображают объекты в покое (рис. 1.10).

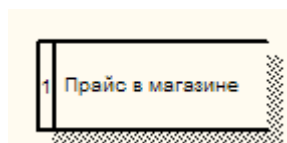


Рисунок 1.10 – Хранилище данных

В материальных системах хранилища данных изображаются там, где объекты ожидают обработки, например, в очереди. В системах обработки информации хранилища данных являются механизмом, который позволяет сохранить данные для последующих процессов.

В DFD стрелки могут сливаться и разветвляться, что позволяет описать декомпозицию стрелок. Каждый новый сегмент сливающейся или разветвляющейся стрелки может иметь собственное имя.

Диаграммы DFD могут быть построены с использованием традиционного структурного анализа, подобно тому как строятся диаграммы IDEF0. Сначала строится физическая модель, отображающая текущее состояние дел. Затем эта модель преобразуется в логическую модель, которая отображает требования к существующей системе. После этого строится модель, отображающая требования к будущей системе. И наконец, строится физическая модель, на основе которой должна быть построена новая система.

Альтернативным подходом является подход, популярный при создании программного обеспечения, называемый событийным разделением (event partitioning), в котором различные диаграммы DFD выстраивают модель системы. Во-первых, логическая модель строится как совокупность работ и документирования того, что они (эти работы) должны делать.

Затем модель окружения (environment model) описывает систему как объект, взаимодействующий с событиями из внешних сущностей. Модель, окружения обычно содержит описание цели системы, одну контекстную диаграмму и список событий. Контекстная диаграмма содержит один прямоугольник работы, изображающий систему в целом, и внешние сущности, с которыми система взаимодействует.

Наконец, модель поведения (behavior model) показывает, как система обрабатывает события. Эта модель состоит из одной диаграммы, в которой каждый прямоугольник изображает каждое событие из модели окружения. Хранилища могут быть добавлены для моделирования данных, которые необходимо запоминать между событиями. Потоки добавляются для связи с другими элементами, и диаграмма проверяется с точки зрения соответствия модели окружения.

Полученные диаграммы могут быть преобразованы с целью более наглядного представления системы, в частности работы на диаграммах могут быть декомпозированы.

В DFD номер каждой работы может включать префикс, номер родительской работы (A) и номер объекта. Номер объекта – это уникальный номер работы на диаграмме. Например, работа может иметь номер A.12.4. Уникальный номер имеют хранилища данных и внешние сущности независимо от их расположения на диаграмме. Каждое хранилище данных имеет префикс D и уникальный номер, например, D5. Каждая внешняя сущность имеет префикс E и уникальный номер, например E5.

Метод описания процессов IDEF3

Наличие в диаграммах DFD элементов для описания источников, приемников и хранилищ данных позволяет более эффективно и наглядно описать процесс документооборота. Однако для описания логики взаимодействия информационных потоков более подходит IDEF3, называемая также *workflow diagramming* – методологией моделирования, использующая графическое описание информационных потоков, взаимоотношений между процессами обработки информации и объектов, являющихся частью этих процессов. Диаграммы Workflow могут быть использованы в моделировании бизнес-процессов для анализа завершенности процедур обработки информации. С их помощью можно описывать сценарии действий сотрудников организации, например, последовательность обработки заказа или события, которые необходимо обработать за конечное время. Каждый сценарий сопровождается описанием процесса и может быть использован для документирования каждой функции.

IDEF3 – это метод, имеющий основной целью дать возможность аналитикам описать ситуацию, когда процессы выполняются в определенной последовательности, а также описать объекты, участвующие совместно в одном процессе.

Техника описания набора данных IDEF3 является частью структурного анализа. В отличие от некоторых методик описаний процессов IDEF3 не ограничивает аналитика чрезмерно жесткими рамками синтаксиса, что может привести к созданию неполных или противоречивых моделей.

IDEF3 может быть также использован как метод создания процессов. IDEF3 дополняет IDEF0 и содержит все необходимое для построения моделей, которые в дальнейшем могут быть использованы для имитационного анализа.

Пример выполнения задания

1. Сформулировать цели и точку зрения, описать модель на уровне «черного ящика» (рис. 1.5)
2. Декомпозируя модель до 1-го уровня, добиться полной ее ясности (рис. 1.11).
3. Декомпозировать все элементы модели, полученные на 1-м уровне (рис. 1.12).

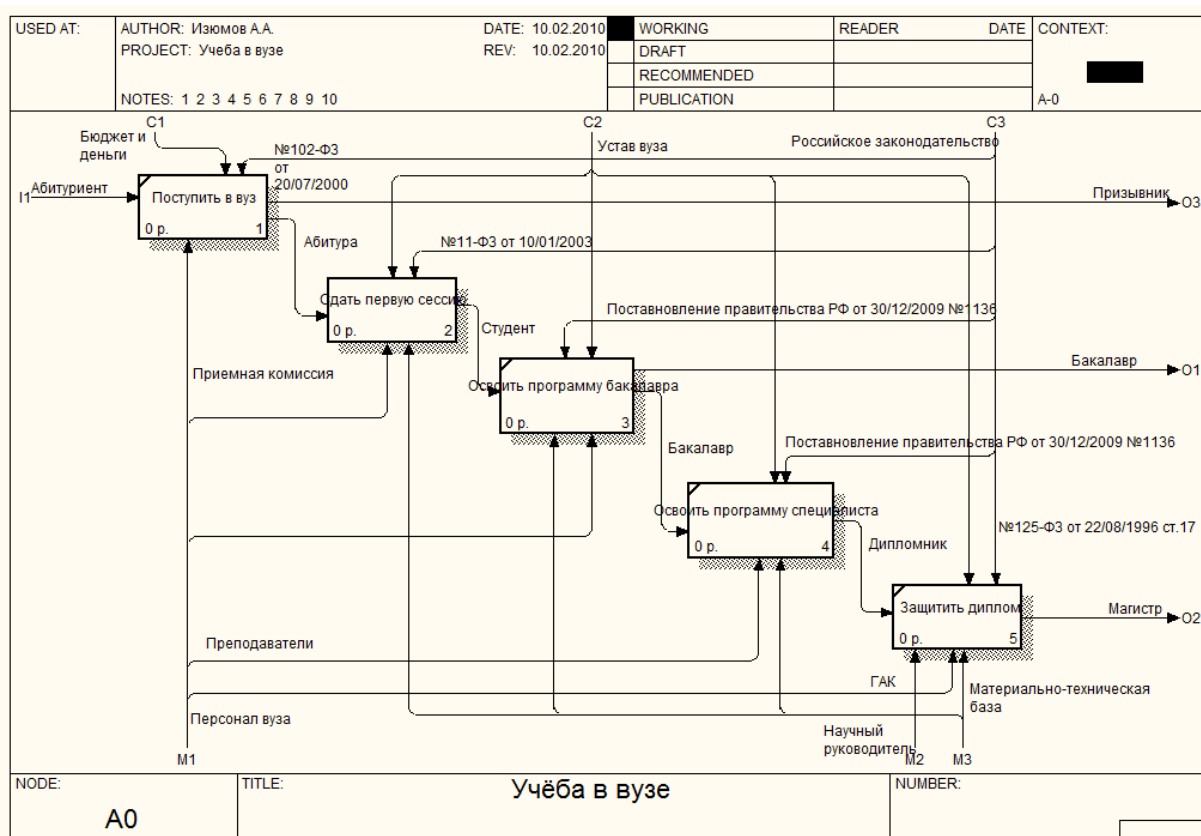


Рисунок 1.11 – Модель на 1-м уровне декомпозиции

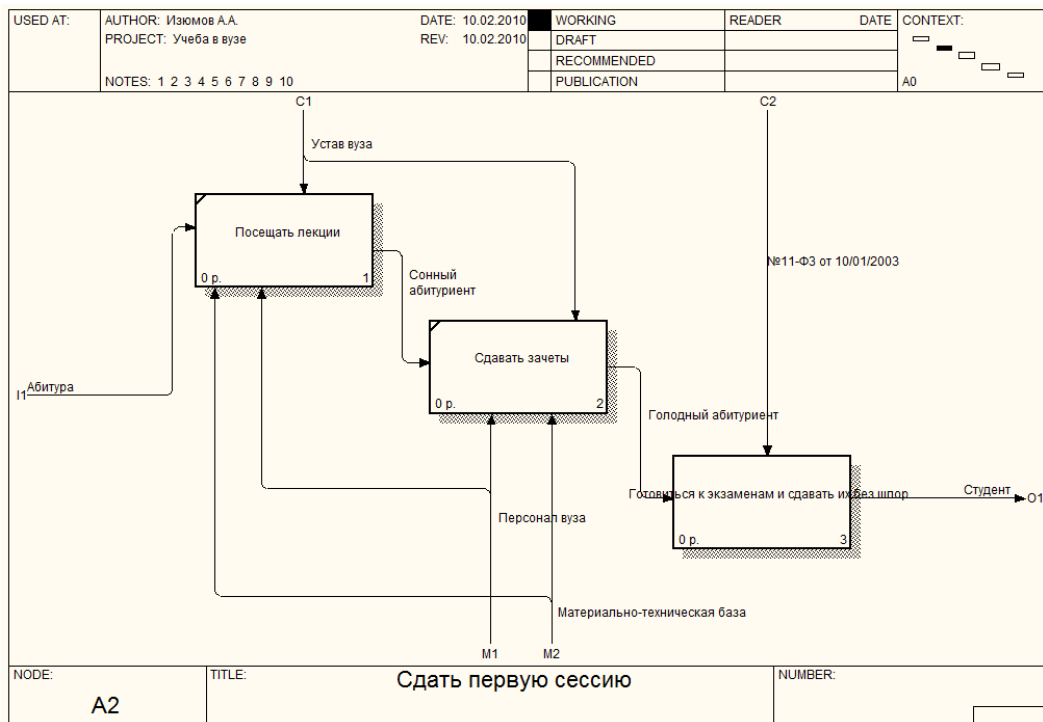


Рисунок 1.12 – Модель на на 2-м уровне декомпозиции для работы «Сдать первую сессию»

4. Декомпозировать все элементы модели, полученные на 2-м уровне (рис. 1.13).

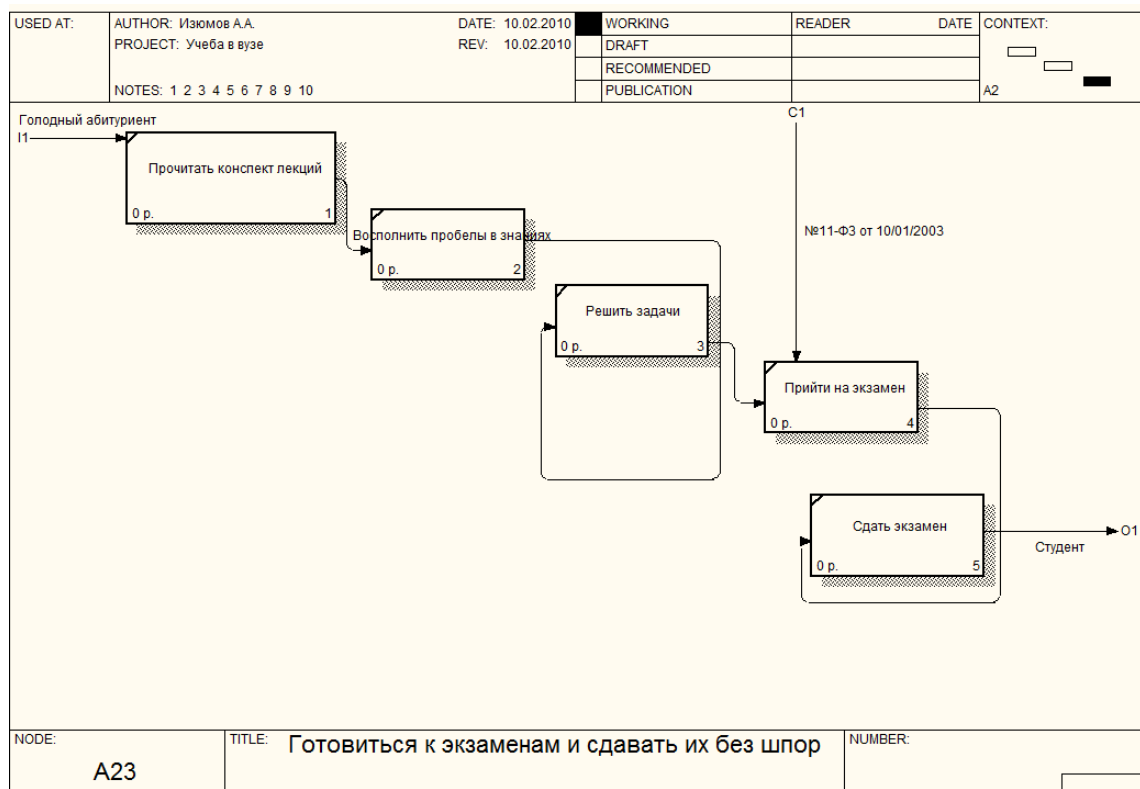


Рисунок 1.13 – Модель на на 3-м уровне декомпозиции для работы «Готовиться к экзаменам и сдавать их без шпор»

3. Написать спецификацию по следующему шаблону

Образец спецификации модели "Учёба в вузе"

Цель

Описать, как должен осуществляться процесс обучения в вузе, чтобы процесс получения знаний завершился защитой дипломного проекта.

Точка зрения

Документ написан студентом для абитуриентов.

Содержание

Документ содержит неполную спецификацию, поскольку он не детализирует многие важные функции процесса обучения. Тем не менее, он детализирует процесс успешной сдачи экзаменов. Следующий список раскрывает структуру этого документа:

Диаграмма	Название
A-0	Учёба в вузе (контекст)
A0	Учёба в вузе (верхний уровень)
A2	Сдать первую сессию
A23	Готовиться к экзаменам и сдавать их без шпор

A-0 Учёба в вузе (контекст)

Выпускник школы, используя знания, полученные в результате освоения программы среднего образования, поступает в выбранное учебное заведение, применяя собственные средства для оплаты дополнительных занятий (в случае, если таковые занятие необходимы). Идеальным вариантом данной подготовки является поступление на бюджетное отделение. Наиболее трудным для абитуриента является первая сессия, которая отделяет тех, кто будет учиться дальше от тех, кто составит основной контингент защитников Родины. В соответствии с Болонским

процессом, высшее образование подразделяется на две ступени, высшим итогом которой является защита магистерской диссертации на иностранном языке.

A0 Учёба в вузе (верхний уровень)

Поступить в вуз

Подготовка начинается сразу после окончания 9 классов. Необходимо выбрать специальность, посещать специальные курсы по выбранному направлению, стремиться максимально углубить свои познания и успешно пройти выпускной экзамен в школе. В 11 классе желательно общение с людьми, уже получившими данную специальность. После сдачи ЕГЭ на максимально возможный балл, пройти дополнительные вузовские испытания.

Сдать первую сессию

Перестроить своё сознание. Привыкать к самостоятельности, учиться жить по средствам. Стремиться наладить контакты с коллективом и с преподавателями, которые в дальнейшем могут стать руководителем научной деятельности. Тщательно и внимательно слушать преподавателей и приучать себя к дисциплине. Постараться максимальное количество предметов сдать «автоматом».

Освоить программу бакалавра

Продолжая обучение, постараться понять – была ли острая необходимость в высшем образовании. Ответить для себя на вопрос – «буду ли я заниматься научной деятельностью». Постараться устроиться на работу по специальности в качестве дешевой рабочей силы и набираться опыта.

Освоить программу специалиста

Программа специалиста наиболее важный этап обучения. Стремится максимально насытить время своего обучения практическими занятиями по специальности. Участвовать в работе ГПО, выполнять реальные задачи, получать опыт. Внимательно относится к фундаментальным для своей специализации наукам – без понимания сути профессии, от навыков толку мало. Сдать государственные экзамены и подтвердить свою квалификацию.

Защитить диплом

Внимательно относится к теме научно-исследовательской работы, стремиться получить максимум от научного руководителя. Желательно выбирать тему, связанную с последующей темой диссертации. При прохождении практики стремится выполнять все указания и запоминать все советы руководителя практики. Дипломный проект должен отражать суть, не будучи чрезмерно затянутым. Оптимальный объём 100-120 страниц. Перед комиссией держать себя уверенно.

A2. Сдать первую сессию

Обзор

Сдача сессии – закономерный результат обучения в семестре. При должном старании и прилежании, сессия проходит для студента незаметно.

Посещать лекции

Вставать рано утром – даже если в данный день нет первой пары. Тем самым приучить себя к расписанию. Внимательно слушать преподавателя и стремиться максимальный объём знаний усвоить на месте. Непонятные моменты не стесняться переспрашивать. Выполнять домашние задания преподавателя.

Сдавать зачеты

Обязательно посещать дополнительные занятия перед зачетами. Внимательно читать конспект лекций и методические пособия по лабораторным работам. Составлять примерный план подготовки к зачету загодя – выделив в течении дня отдельный промежуток времени для самосовершенствования.

Готовиться к экзаменам и сдавать их без шпор

Сдавать со шпаргалкой проще, но опаснее и вреднее. В реальной жизни неожиданные проблемы валятся без предварительного уведомления о своём сроке наступления. Поэтому лучше тщательнее подготовиться и сдать сессию с обязательными предметами. Остальные сессии пройдут легче.

А23. Готовиться к экзаменам и сдавать их без шпор

Обзор

Сдав все зачеты и долги, приступить к подготовке к экзаменам. Не надо чрезмерно увлекаться студенческой самодеятельностью. Постараться объяснить соседям по комнате, что вы находитесь на важном для вас этапе, на рубеже, где решается ваша дальнейшая судьба. Систематизировать знания по максимуму – вот цель экзамена и подготовки к нему.

Прочитать конспект лекций

Внимательно читать то, что было записано в ходе посещения лекций. Непонятные или неаккуратно записанные места переписать у коллег. Постараться вспомнить, что и как говорил преподаватель, диктуя конкретное предложение.

Восполнить пробелы знаний

Лекции никогда не охватывают всего богатство предмета. Даже такие постоянные предметы как физика и математика не находятся в статичном состоянии. Необходимо приучить себя получать знания самостоятельно –

ведь преподаватель будет только рад тому, что студент на экзамене знает больше него. Восполнять пробелы необходимо через чтение журналов по данной тематике, через посещения тематических конференций и чтение статей в Интернет.

Решить задачи

Учитывая, что экзамен – это не только теория, но и практика, необходимо подготовиться к сдаче практической части. Для этого необходимо выполнять все те задания, которые преподаватель оставил для самостоятельного и дополнительного обучения. Постараться запомнить не конкретный ответ, а логику решения схожих проблем.

Прийти на экзамен

Накануне экзамена лучше всякую подготовку прекратить. Выспаться, хорошо покушать – даже если на это придётся потратить больше обычного. Не играть весь день в динамичные игры, постараться расслабиться, не напрягаться. Лучше всего – погулять на свежем воздухе. Поставить будильник на 2-2.5 часа раньше начала экзамена. Утром сходить в туалет, легко позавтракать, аккуратно одеться, пробежаться глазами по конспекту лекций и смело идти сдаваться.

Сдать экзамен

Не нервничать и не переживать, если вопрос не совсем ясен. Постараться по максимуму рассказать о темах, в которых вы уверены. Обязательно решить практические задания – по возможности быстро, чтобы была возможность их перепроверить.

Задание

Выполните построение SADT-диаграммы процесса, используя полученные навыки. Список тем приведен в Приложении 1. Выполните

написание спецификации для построенной диаграммы по приведенному примеру. Результат работы – документ с SADT-диаграммой (*.bp1) и текстовый документ со спецификацией (*.odt/*.rtf).

Лабораторная работа №2

Цель работы

Целью данной работы является изучение основных типов диаграмм, которые возможно построить в методологии UML, знакомство с интерфейсом графических приложений для UML моделирования и построение Use Case диаграммы выбранной предметной области.

Краткое изложение теоретической части

UML (сокр. от англ. Unified Modeling Language — унифицированный язык моделирования) — язык графического описания для объектного моделирования в области разработки программного обеспечения. UML является языком широкого профиля, это открытый стандарт, использующий графические обозначения для создания абстрактной модели системы, называемой UML моделью. UML был создан для определения, визуализации, проектирования и документирования в основном программных систем. UML не является языком программирования, но в средствах выполнения UML-моделей как интерпретируемого кода возможна кодогенерация.

Использование UML не ограничивается моделированием программного обеспечения. Его также используют для моделирования бизнес-процессов, системного проектирования и отображения организационных структур.

UML позволяет также разработчикам программного обеспечения достигнуть соглашения в графических обозначениях для представления

общих понятий (таких как класс, компонент, обобщение (generalization), объединение (aggregation) и поведение) и больше сконцентрироваться на проектировании и архитектуре.

В UML используются следующие виды диаграмм:

Structure Diagrams:

- Class diagram
- Component diagram
- Composite structure diagram
- Collaboration (UML2.0)
- Deployment diagram
- Object diagram
- Package diagram

Структурные диаграммы:

- Классов
- Компонентов
- Композитной/составной структуры
- Кооперации (UML2.0)
- Развёртывания
- Объектов
- Пакетов

Behavior Diagrams:

- Activity diagram
- State Machine diagram
- Use case diagram
- *Interaction Diagrams:*
 - Communication diagram (UML2.0) / Collaboration (UML1.x)
 - Interaction overview diagram (UML2.0)
 - Sequence diagram
 - Timing diagram (UML2.0)

Диаграммы поведения:

- Деятельности
- Состояний
- Вариантов использования
- *Диаграммы взаимодействия:*
 - Коммуникации (UML2.0) / Кооперации (UML1.x)
 - Обзора взаимодействия (UML2.0)
 - Последовательности
 - Синхронизации (UML2.0)

Рассмотрим Rational Rose – мощное CASE-средство для проектирования программных систем любой сложности. Одним из достоинств этого программного продукта будет возможность использования диаграмм на языке UML (*Unified Modeling Language*).

Можно сказать, что Rational Rose является графическим редактором UML диаграмм. Для работы с диаграммами UML используются графические редакторы UML диаграмм: Rational Rose и Sparx Enterprise Architect (платформа windows), Visual Paradigm, Umbrello UML Modeller (платформа nix). Все программы оперируют некоторым набором диаграмм, отличаются интерфейсом, но главным объединяющим является тот факт, что все они позволяют строить диаграммы UML.

Class diagram (диаграммы классов)

Этот тип диаграмм позволяет создавать логическое представление системы, на основе которого создается исходный код описанных классов.

Существуют разные точки зрения на построение диаграмм классов в зависимости от целей их применения:

- концептуальная точка зрения — диаграмма классов описывает модель предметной области, в ней присутствуют только классы прикладных объектов;
- точка зрения спецификации — диаграмма классов применяется при проектировании информационных систем;
- точка зрения реализации — диаграмма классов содержит классы, используемые непосредственно в программном коде (при использовании объектно-ориентированных языков программирования).

Значки диаграммы позволяют отображать сложную иерархию систем, взаимосвязи классов (Classes) и интерфейсов (Interfaces). Данный тип диаграмм противоположен по содержанию диаграмме Collaboration, на котором отображаются объекты системы. Существует несколько типов нотаций по созданию диаграмм данного типа. В нотации, предложенной Г. Бучем, которая так и называется Booch, классы изображаются в виде чего-то нечеткого (рис. 2.1), похожего на облако. Таким образом Г.Буч пытается

показать, что класс – это лишь шаблон, по которому в дальнейшем будет создан конкретный объект.

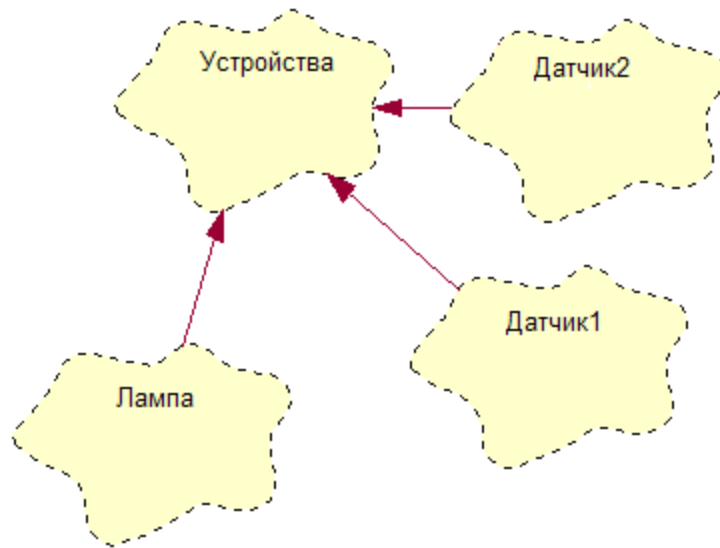


Рисунок 2.1 – Диаграмма классов в нотации Гради Буча

Также, возможно создавать диаграмму классов в нотации UMT и унифицированной нотации (рис. 2.2). В Rational Rose переключение режимов отображения осуществляется в меню <View/As Booch (As OMT, As Unified)>

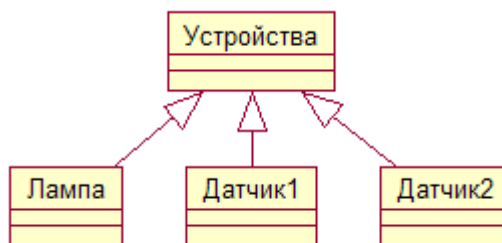


Рисунок 2.2 – Диаграмма классов в унифицированной нотации

Component diagram (диаграммы компонентов)

Этот тип диаграмм (рис. 2.3) предназначен для распределения классов и объектов по компонентам при физическом проектировании системы. Часто данный тип диаграмм называют диаграммами модулей.

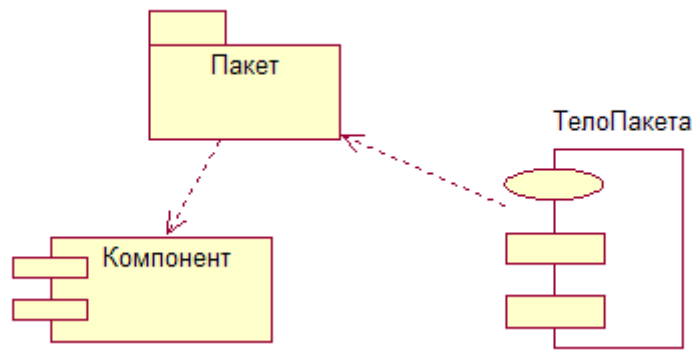


Рисунок 2.3 – Диаграмма компонентов

При проектировании больших систем может оказаться, что система должна быть разложена на несколько сотен или даже тысяч компонентов, и этот тип диаграмм позволяет не потеряться в обилии модулей и их связей. В качестве физических компонент могут выступать файлы, библиотеки, модули, исполняемые файлы, пакеты и т. п.

Composite structure diagram (диаграмма композитной/составной структуры)

Статическая структурная диаграмма, демонстрирует внутреннюю структуру классов и, по возможности, взаимодействие элементов (частей) внутренней структуры класса.

Подвидом диаграмм композитной структуры являются диаграммы кооперации (Collaboration diagram, введены в UML 2.0), которые показывают роли и взаимодействие классов в рамках кооперации. Кооперации удобны при моделировании шаблонов проектирования.

Диаграммы композитной структуры могут использоваться совместно с диаграммами классов.

Deployment diagram (диаграммы топологии)

Этот вид диаграмм (рис. 2.4) предназначен для анализа аппаратной части системы, то есть «железа», а не программ. В прямом переводе с

английского Deployment означает «развертывание», но термин «топология» точнее отражает сущность этого типа диаграмм.

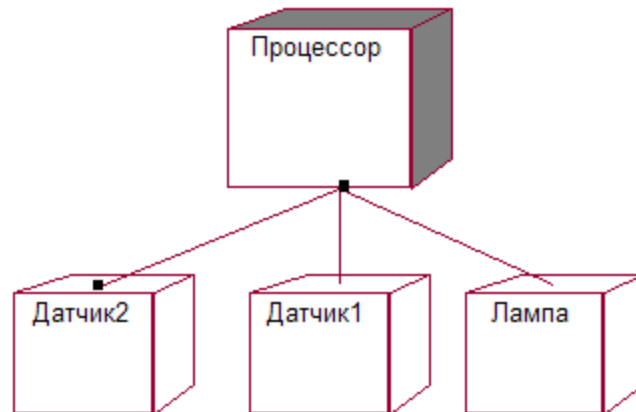


Рисунок 2.4 – Диаграмма топологии

Для каждой модели создается только одна такая диаграмма, отображающая процессоры (Processor), устройства (Device) и их соединения.

Служит для моделирования работающих узлов (аппаратных средств, англ. node) и артефактов, развёрнутых на них. В UML 2 на узлах разворачиваются артефакты (англ. artifact), в то время как в UML 1 на узлах разворачивались компоненты. Между артефактом и логическим элементом (компонентом), который он реализует, устанавливается зависимость манифестации.

Object diagram (диаграмма объектов)

Демонстрирует полный или частичный снимок моделируемой системы в заданный момент времени. На диаграмме объектов отображаются экземпляры классов (объекты) системы с указанием текущих значений их атрибутов и связей между ними.

Use case diagram (диаграммы вариантов использования/прецедентов)

Этот вид диаграмм (рис. 2.5) позволяет создать список операций, которые выполняет система. Часто этот вид диаграмм называют диаграммой функций, потому что на основе набора таких диаграмм создается список требований к системе и определяется множество выполняемых системой функций.

Каждая такая диаграмма или, как ее обычно называют, каждый Use case – это описание сценария поведения, которому следуют действующие лица (Actors).

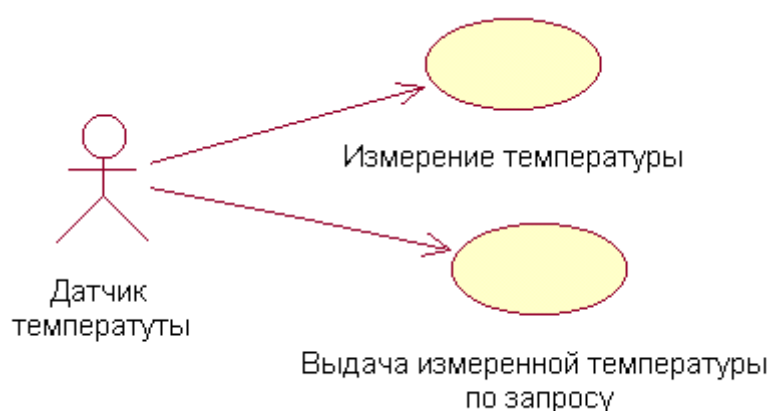


Рисунок 2.5 – Диаграмма прецедентов

State Machine diagram (диаграммы состояний)

Каждый объект системы, обладающий определенным поведением, может находиться в определенных состояниях, переходить из состояния в состояние, совершая определенные действия в процессе реализации сценария поведения объекта. Поведение большинства объектов реальных систем можно представить с точки зрения теории конечных автоматов, то есть поведение объекта отражается в его состояниях, и данный тип диаграмм позволяет отразить это графически.

Конечный автомат (англ. State machine) — спецификация последовательности состояний, через которые проходит объект или взаимодействие в ответ на события своей жизни, а также ответные действия

объекта на эти события. Конечный автомат прикреплён к исходному элементу (классу, кооперации или методу) и служит для определения поведения его экземпляров.

Statechart diagram (диаграмма состояний)

Диаграмма состояний (рис. 2.6) предназначена для отображения состояний объектов системы, имеющих сложную модель поведения. Это одна из двух диаграмм State Machine, доступ к которой осуществляется из одного пункта меню.



Рисунок 2.6 – Диаграмма состояний

Activity diagram (диаграммы активности/деятельности)

Это дальнейшее развитие диаграммы состояний. Фактически данный тип диаграмм может использоваться и для отражения состояний моделируемого объекта, однако, основное назначение Activity diagram в том, чтобы отражать бизнес-процессы объекта, технологические переходы. Этот тип диаграмм позволяет показать не только последовательность процессов, но и ветвление и даже синхронизацию процессов.

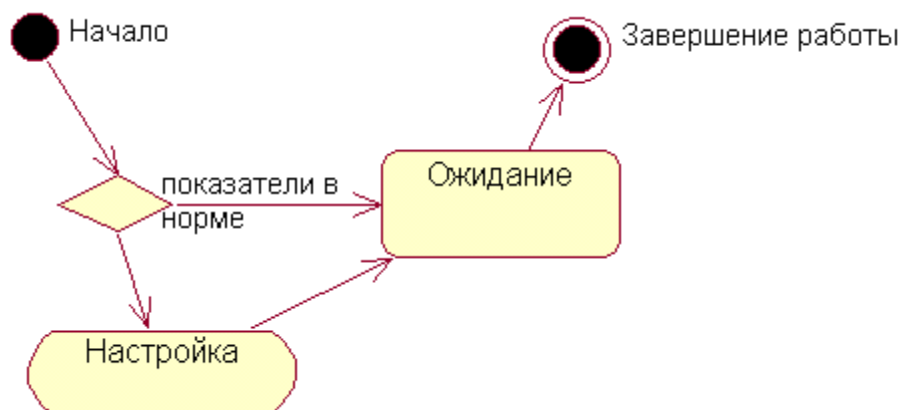


Рисунок 2.7 – Диаграмма активности

Этот тип диаграмм позволяет проектировать алгоритмы поведения объектов любой сложности, в том числе может использоваться для составления блок-схем.

Interaction diagram (диаграммы взаимодействия)

Этот тип диаграмм включает в себя диаграммы Sequence diagram (диаграммы последовательностей действий) и Collaboration diagram (диаграммы сотрудничества). Эти диаграммы позволяют с разных точек зрения рассмотреть взаимодействие объектов в создаваемой системе.

Sequence diagram (диаграммы последовательностей действий)

Взаимодействие объектов в системе происходит посредством приема и передачи сообщений объектами-клиентами и обработки этих сообщений объектами-серверами. При этом в разных ситуациях одни и те же объекты могут выступать и в качестве клиентов, и в качестве серверов.

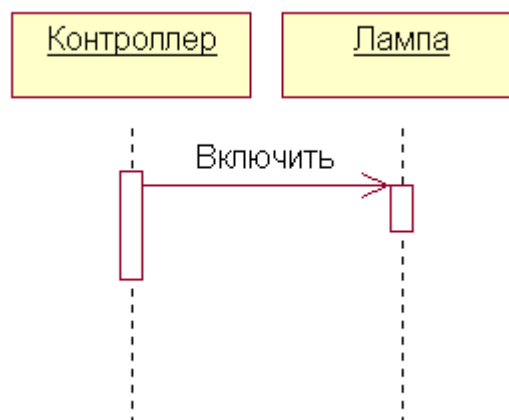


Рисунок 2.8 – Диаграмма последовательностей действий

Данный тип диаграмм позволяет отразить последовательность передачи сообщений между объектами.

Этот тип диаграммы не акцентирует внимание на конкретном взаимодействии, главный акцент уделяется последовательности

приема/передачи сообщений. Для того чтобы окинуть взглядом все взаимосвязи объектов, служит Collaboration diagram.

Collaboration diagram (диаграммы сотрудничества)

Этот тип диаграмм (рис. 2.9) позволяет описать взаимодействия объектов, абстрагируясь от последовательности передачи сообщений. На этом типе диаграмм в компактном виде отражаются все принимаемые и передаваемые сообщения конкретного объекта и типы этих сообщений.

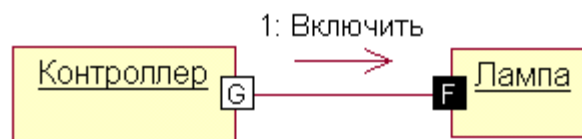


Рисунок 2.9 – Диаграмма сотрудничества

По причине того, что диаграммы Sequence и Collaboration являются разными взглядами на одни и те же процессы, Rational Rose позволяет создавать из Sequence диаграммы диаграмму Collaboration и наоборот, а также производит автоматическую синхронизацию этих диаграмм.

Interaction overview diagram (диаграмма обзора взаимодействия)

Разновидность диаграммы деятельности, включающая фрагменты диаграммы последовательности и конструкции потока управления.

Timing diagram (диаграмма синхронизации)

Альтернативное представление диаграммы последовательности, явным образом показывающее изменения состояния на линии жизни с заданной шкалой времени. Может быть полезна в приложениях реального времени.

Интерфейс Rational Rose

Рассматривается версия Rational Rose 2000 функционирующая под управлением ОС Windows. Среда поддерживает работу со всеми типами канонических диаграмм языка UML посредством меню, основной и контекстной панелей инструментов. Содержимое последней изменяется в зависимости от типа текущей диаграммы.

Кроме того, пользователю предоставляются контекстные всплывающие меню, выводимые при щелчке правой кнопкой мыши. Браузер среды позволяет быстро и легко получать доступ к диаграммам и другим элементам модели. Для вывода расширенной справки используется клавиша <F1>.

Список основных элементов интерфейса с указанием закрепленных за ними функций приведен в таблице 2.1.

Таблица 2.1 – Элементы интерфейса Rational Rose

Элемент	Назначение
1. Браузер (browser)	Быстрая навигации по модели.
2. Окно документирования (documentation window)	Документирование элементов модели.
3. Панели инструментов (toolbars)	Доступ к наиболее распространенным командам.
4. Окно диаграмм (diagram window)	Просмотр и редактирование одной или нескольких диаграмм UML.
5. Журнал (log)	Просмотр ошибок и отчетов о результатах выполнения команд.

Номера графических фрагментов рис. 2.10 соответствуют номерам элементов интерфейса, приведенных в таблице 2.1.

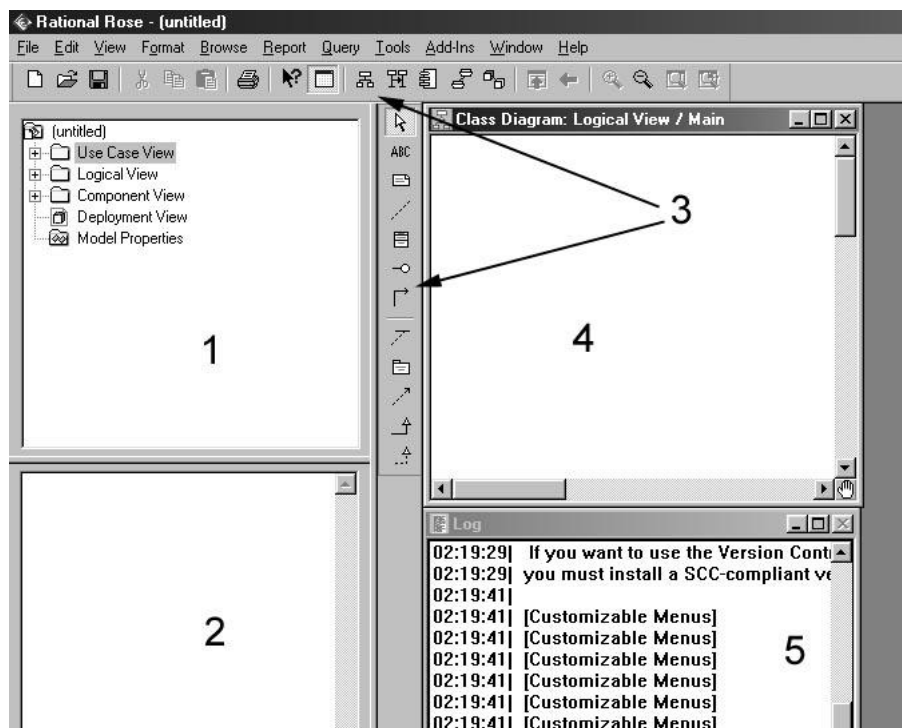


Рисунок 2.10 – Основные элементы интерфейса Rational Rose 2000

Окно браузера (рис. 2.10-1) содержит дерево проекта, благодаря которому возможна понятная и простая навигация по модели. Все, что добавляется к модели, выводится в окне браузера

Браузер поддерживает четыре группы представлений:

- представление прецедентов;
- логическое представление;
- представление компонентов;
- представление развертывания.

Это соответствует концепции визуального моделирования в UML, рассматривающей процесс моделирования как некий поуровневый спуск от наиболее общей и абстрактной концептуальной модели исходной системы к логической, а затем и к физической модели соответствующей программной системы.

Представление прецедентов предназначено для хранения концептуальной модели, логическое представление содержит логическую

модель системы и, наконец, представления компонентов и развертывания – два представления физической модели информационной системы.

С помощью окна документирования (рис. 2.10-2), например, можно сделать описание каждого актера на диаграмме прецедентов. При документировании класса все, что указано в этом окне, появится затем как комментарий в сгенерированном коде, что избавляет разработчика от необходимости впоследствии вносить комментарии вручную. Эта же документация включается в отчеты, формируемые средой Rational Rose.

Панели инструментов Rational Rose (рис. 2.10-3) обеспечивают быстрый доступ к наиболее распространенным командам. Стандартная панель видна всегда, ее кнопки соответствуют командам, которые могут использоваться для работы с различными диаграммами.

Специфичные пиктограммы стандартной панели перечислены в таблице 2.2.

Таблица 2.2 – Описание пиктограмм стандартной панели инструментов

Пиктограмма	Наименование	Назначение
	<i>Selects or deselects an item</i>	Предоставляет возможность выделять объект
	<i>Browse Class Diagram</i>	Находит и открывает диаграмму классов
	<i>Browse Interaction Diagram</i>	Находит и открывает диаграмму последовательности или диаграмму кооперации
	<i>Browse Component Diagram</i>	Находит и открывает диаграмму компонентов
	<i>Browse State Machine Diagram</i>	Находит и открывает диаграмму деятельности или состояний
	<i>Browse Deployment Diagram</i>	Находит и открывает диаграмму размещения
	<i>Browse Parent</i>	Находит и открывает родительскую диаграмму
	<i>Browse Previous Diagram</i>	Находит и открывает предыдущую диаграмму

Пиктограммы, расположенные в правой части панели инструментов (при ее настройке по умолчанию), отвечают за масштабирование диаграммы. Другие пиктограммы панели (такие как создание нового файла модели, его сохранение, печать диаграммы и проч.) присутствуют в любом развитии пакета для ОС Windows. Их назначение интуитивно понятно.

В окне диаграммы (рис. 2.10-4) выводится одна или несколько диаграмм UML создаваемой модели. Окно диаграммы и браузер связаны между собой – изменение информации об элементе на диаграмме

автоматически приводит к изменению информации в браузере и наоборот. Это позволяет поддерживать модель в непротиворечивом состоянии.

Журнал (рис. 2.10-5) содержит информацию, генерируемую средой в процессе создания и модификации модели системы. В журнал помещаются сообщения об ошибках, возникающих при генерации кода, отражаются результаты выполнения ряда операций над средой Rational Rose и моделью. Окно журнала невозможно закрыть, но можно минимизировать.

С помощью Rational Rose можно публиковать модели на Web-страницах. Таким образом, все желающие смогут изучить ваши модели, даже не будучи пользователями Rational Rose и не распечатывая большое количество соответствующей документации.

Для публикации модели в сети:

- Выберите в меню пункт *<Tools/Web Publisher>*;
- В окне Мастера публикации (рис. 2.11) укажите представления модели и пакеты.

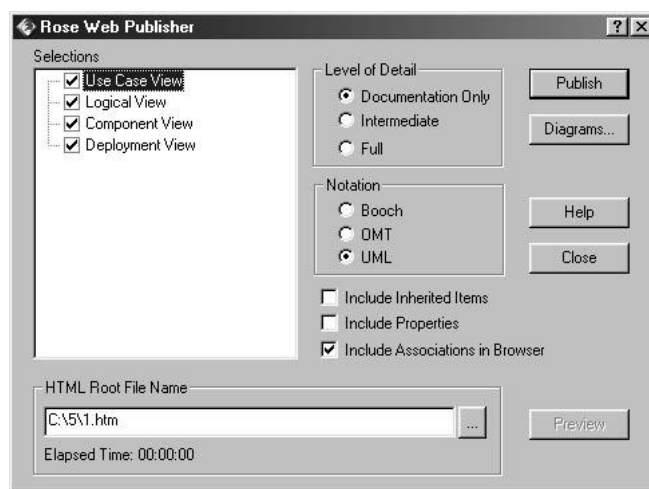


Рисунок 2.11 – Окно мастера публикаций

- Выберите требуемый уровень детализации. Уровень *Documentation Only* (Только документация) соответствует наиболее общей информации, при этом не будут показаны никакие свойства элементов модели. При выборе уровня *Intermediate* (Промежуточный) будут показаны те свойства элементов модели, которые описаны на вкладке *General* (Общие) их спецификаций. Уровень *Full* (Полный) означает

публикацию всех свойств, включая те, что содержатся на вкладке *Detail* спецификации элементов модели.

- Выберите требуемую нотацию. По умолчанию будет использоваться та нотация, что принята по умолчанию в среде Rose.
- Укажите, хотите ли вы опубликовать наследуемые элементы (inherited items).
- Укажите, надо ли публиковать свойства.
- Введите название корневого файла HTML.
- Если вы хотите использовать для диаграмм графические файлы, нажмите на кнопку *Diagrams* (Диаграммы). Появится окно *Diagram Options* (Параметры диаграммы), как показано на рис. 2.12.
- Укажите формат, в котором будут опубликованы ваши диаграммы. Можно выбрать *Windows Bitmaps* (Растровые изображения в среде Windows), *Portable Network Graphics (PNG)* (Переносимая по сети графика) или *JPEG* либо отказаться от публикации диаграмм.
- Закончив, щелкните на кнопке *Finish* (Готово). В результате будут созданы все Web-страницы, требуемые для публикации модели.
- Щелкнув мышью на кнопке *Preview* (Предварительный просмотр), посмотрите, что получилось.

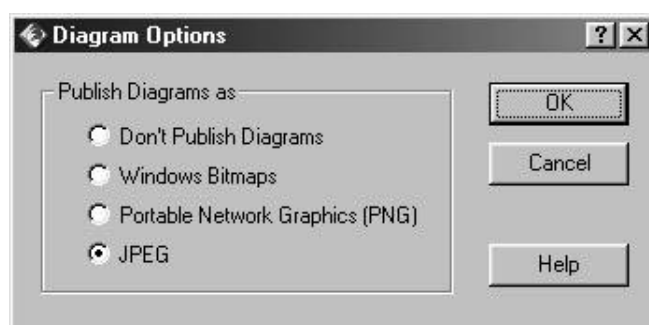


Рисунок 2.12 – Окно выбора параметров публикации диаграмм.

Интерфейс Sparx Enterprise Architect

Подобно Rational Rose, данная программа визуально состоит из различных функциональных блоков:

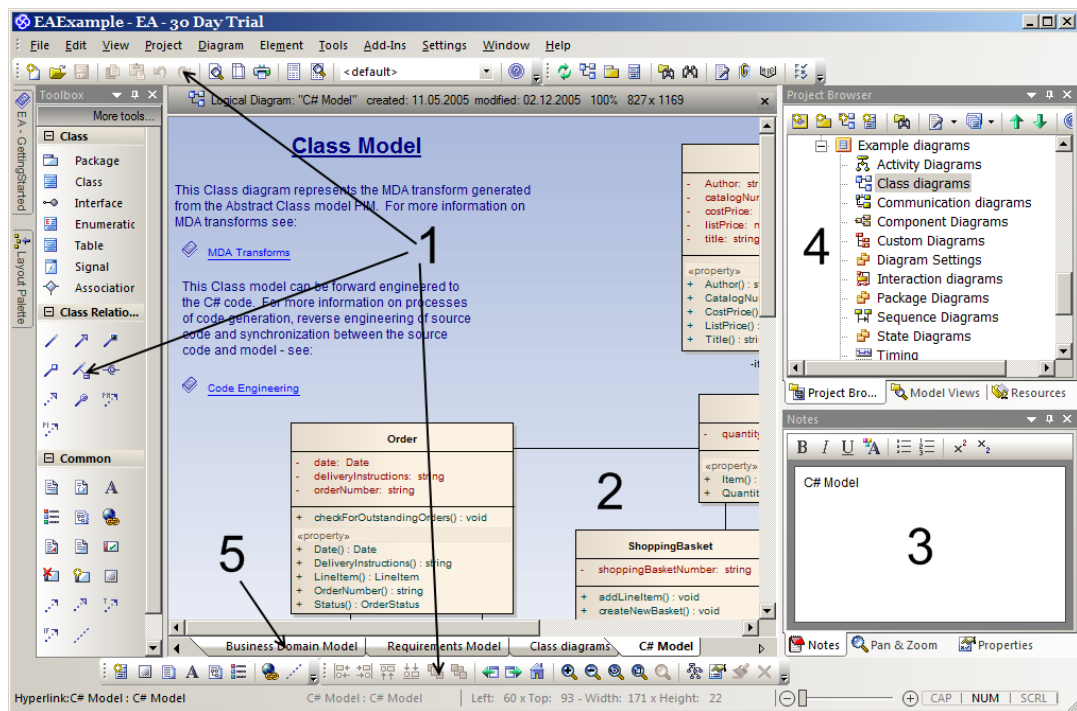


Рисунок 2.13 – Интерфейс Sparx Enterprise Architect

Визуально выделяются:

- 1) панели инструментов
- 2) окно диаграмм
- 3) окно документирования
- 4) дерево модели
- 5) вкладки диаграмм

Важным достоинством программы является поддержка стандарта UML 2.1, богатые возможности по визуализации и кодогенерации проектов разных типов.

Создание модели в Rational Rose

Первым шагом при работе с Rational Rose является создание моделей. Их можно строить либо «с нуля», либо взяв за основу существующую каркасную модель. Среда Rational Rose позволяет хранить модель в одном файле, имеющем расширение *.mdl (model). Модель создается автоматически, и при появлении окна выбора типа модели (рис. 2.14), достаточно нажать Cancel.



Рисунок 2.14 – Выбор типа создаваемой модели

В среде Rational Rose диаграммы прецедентов создаются в представлении прецедентов (Use Case View). Главная диаграмма (Main) предлагается по умолчанию. При моделировании системы существует возможность разрабатывать столько дополнительных диаграмм прецедентов, сколько этого требует проект.

Если в рамках проекта создается несколько диаграмм прецедентов, логично располагать их в рамках представления прецедентов, хотя среда позволяет расположить их в любой ветке дерева проекта. Для создания новой диаграммы прецедентов следует выполнять следующую последовательность шагов:

- щелкнуть правой кнопкой мыши на пакете представления прецедентов в браузере;
- во всплывающем меню выбрать пункт *<New/Use Case Diagram>*;
- выделив новую диаграмму, ввести ее имя.

- дважды щелкнув на названии этой диаграммы в браузере, открыть ее.

В среде Rational Rose вы можете связать с диаграммой прецедентов файл или объект, расположенный по ссылке в сети Internet, используя URL этого объекта. Таким образом, можно связать с диаграммой любой вспомогательный документ, например спецификации требований высокого уровня или документы концептуального характера (в соответствующем формате, например *.dwg или *.pdf). Все связанные файлы и ссылки будут показаны в браузере под соответствующей диаграммой прецедентов. По двойному щелчку мыши на файле или ссылке в браузере откроется соответствующее приложение и загрузится документ, находящийся в файле или по указанному адресу в сети Internet.

Для связывания файла с диаграммой прецедентов необходимо выполнять следующие шаги:




- щелкнуть правой кнопкой мыши на соответствующей диаграмме в браузере;
- в открывшемся меню выбрать пункт <New/File>;
- в диалоговом окне *Open* указать, какой файл нужно прикрепить к диаграмме.
- нажать кнопку *Open*, чтобы выполнить прикрепление.

Для связывания ссылки с диаграммой прецедентов выполняют следующие шаги.





- щелкают правой кнопкой мыши на соответствующей диаграмме в браузере;
- в открывшемся меню выбирают пункт <New/URL>;
- вводят адрес.

При открытии диаграммы прецедентов, на панели инструментов появляются соответствующие пиктограммы. Если содержимое панели не переопределяется пользователем, набор пиктограмм соответствуют приведенным в таблице 2.3.




Таблица 2.3 – Кнопки панели инструментов

	<i>Наименование</i>	<i>Назначение</i>
	<i>Selection Tool</i>	Превращает курсор в стрелку указателя, предоставляя возможность выделять объект. Основной инструмент, который позволяет выбирать элементы диаграммы, для того чтобы производить с ними дальнейшие действия. Если вы не создаете новый элемент, то этот инструмент активен. При создании нового элемента диаграммы необходимо выбрать необходимый инструмент в строке инструментов, кнопка «залипает», а после создания необходимо опять перейти в режим <i>Selection Tool</i>
	<i>Text Box</i>	Добавляет текст к диаграмме. Данный инструмент позволяет создать произвольную надпись на диаграмме, не привязанную ни к какому элементу.
	<i>Note</i>	Добавляет к диаграмме примечание. Данный инструмент создает элемент замечания, позволяющий вписать в него принятые во время анализа решения. Заметки могут содержать простой текст, фрагменты кода или ссылки на другие документы. Обычно окно <i>Note</i> соединяют с другими элементами диаграммы при помощи инструмента <i>Anchor Note</i> , для того чтобы показать к какому элементу диаграммы относится замечание. В этом отличие от элемента <i>Text Box</i> , который располагается на диаграмме без присоединения к другим элементам.

Продолжение таблицы 2.3 – Кнопки панели инструментов

	<i>Наименование</i>	<i>Назначение</i>
	<i>Anchor Note to Item</i>	Связывает примечание с объектом на диаграмме. Данный инструмент позволяет соединить элемент <i>Note</i> с любым элементом на диаграмме, в том числе и с другим элементом <i>Note</i> .
	<i>Package</i>	Помещает на диаграмму новый пакет. Данный инструмент позволяет создавать контейнеры, которые могут включать в себя группы элементов Use Case и в данной диаграмме может использоваться для определения более крупных сценариев поведения объектов с дальнейшей детализацией. Причем контейнеры могут включать в себя другие контейнеры, что позволяет создавать значительный уровень вложенности детализации.
	<i>Use case</i>	Помещает на диаграмму новый прецедент. Данный инструмент позволяет создавать простые формы сценариев поведения объектов системы. Это представление работы системы с точки зрения актеров (actors), то есть объектов, выполняющих в системе определенные функции.
	<i>Actor</i>	Помещает на диаграмму нового актера. Данный инструмент используется для создания действующих лиц в системе. На диаграмме Use Case значком <i>Actor</i> обозначают пользователей системы, для того чтобы определить задачи, выполняемые пользователями и их взаимодействие.

Окончание таблицы 2.3 – Кнопки панели инструментов

	<i>Наименование</i>	<i>Назначение</i>
		<p>Обычно значком <i>Actor</i> обозначают объект, который:</p> <ul style="list-style-type: none"> • взаимодействует с системой или использует систему; • передает или принимает информацию в систему; • является внешним по отношению к системе.
	<i>Unidirectional Association</i>	Рисует направленную ассоциацию между актером и прецедентом. Данный инструмент позволяет обозначать связи между элементами. На диаграмме Use Case эти связи могут быть определены между use case и actor. Значок <i>Unidirectional Association</i> позволяет создать однонаправленную связь класса с классом или класса с интерфейсом.
	<i>Dependency or Instantiates</i>	Рисует отношение зависимости между элементами диаграммы. Значок <i>Dependency or instantiates</i> позволяет создать связь зависимости. Установка этого типа связей показывает, что класс использует другой класс как параметр в одном из методов.
	<i>Generalization</i>	Рисует отношение обобщения между элементами диаграммы, для которых такое отношение является допустимым. Значок <i>Generalization</i> позволяет создать связь наследования, то есть создается подкласс для соединенного этой связью класса, наследуемого из родительского класса.

Для организации прецедентов их группируют в пакеты. Кроме того, прецеденты можно организовать, определив между ними отношения обобщения, включения и расширения. Эти отношения применяют, чтобы выделить некоторое общее поведение (извлекая его из других прецедентов, которые его включают) или, наоборот, вариации (поместив такое поведение в другие прецеденты, которые его расширяют).

Отношение включения (include)

Отношение включения между двумя вариантами использования указывает, что некоторое заданное поведение для одного варианта использования включается в качестве составного компонента в последовательность поведения другого варианта использования.

Семантика этого отношения определяется следующим образом. Когда экземпляр первого варианта использования в процессе своего выполнения достигает точки включения в последовательность поведения экземпляра второго варианта использования, экземпляр первого варианта использования выполняет последовательность действий, определяющую поведение экземпляра второго варианта использования, после чего продолжает выполнение действий своего поведения. При этом предполагается, что даже если экземпляр первого варианта использования может иметь несколько включаемых в себя экземпляров других вариантов, выполняемые ими действия должны закончиться к некоторому моменту, после которого должно быть продолжено выполнение прерванных действий экземпляра первого варианта использования в соответствии с заданным для него поведением.

Один вариант использования может быть включен в несколько других вариантов, а также включать в себя другие варианты. Включаемый вариант использования может быть независимым от базового варианта в том смысле, что он предоставляет ему некоторое инкапсулированное поведение, детали реализации которого скрыты и могут быть перераспределены между несколькими включаемыми вариантами использования. Более того, базовый

вариант может зависеть только от результатов выполнения включаемого в него поведения, но не от структуры включаемых в него вариантов.

Отношение включения, направленное от варианта использования А к варианту использования В, указывает, что каждый экземпляр варианта А включает в себя функциональные свойства, заданные для варианта В. Эти свойства специализируют поведение соответствующего варианта А на данной диаграмме. Графически данное отношение обозначается пунктирной линией со стрелкой, которая помечается ключевым словом «include» (включает).

Отношение расширения (extend)

Отношение расширения определяет взаимосвязь экземпляров отдельного варианта использования с более общим вариантом, свойства которого определяются на основе способа совместного объединения данных экземпляров. В метамодели отношение расширения является направленным и указывает, что применительно к отдельным примерам некоторого варианта использования должны быть выполнены конкретные условия, определенные для расширения данного варианта использования. Так, если имеет место отношение расширения от варианта использования А к варианту использования В, то это означает, что свойства экземпляра варианта использования В могут быть дополнены благодаря наличию свойств у расширенного варианта использования А.

Отношение расширения между вариантами использования обозначается пунктирной линией со стрелкой (вариант отношения зависимости), направленной от того варианта использования, который является расширением для исходного варианта использования. Данная линия со стрелкой помечается ключевым словом «extend» (расширяет).

Отношение расширения применяют для моделирования таких частей прецедента, которые пользователь воспринимает как необязательное поведение системы. Тем самым можно разделить обязательное и необязательное поведение. Отношения расширения используются также

для моделирования отдельных потоков, выполняемых лишь при определенных обстоятельствах. Наконец, их применяют для моделирования нескольких потоков, которые могут включаться в некоторой точке сценария в результате явного взаимодействия с актером.

Отношение обобщения служит для указания того факта, что некоторый вариант использования А может быть обобщен до варианта использования В. В этом случае вариант А будет являться специализацией варианта В. При этом, В называется предком или родителем по отношению А, а вариант А - потомком по отношению к варианту использования В. Потомок наследует все свойства и поведение своего родителя, а также может быть дополнен новыми свойствами и особенностями поведения. Графически данное отношение обозначается сплошной линией со стрелкой в форме незакрашенного треугольника, которая указывает на родительский вариант использования.

Отношение обобщения (generalize)

Отношение обобщения между вариантами использования применяется в том случае, когда необходимо отметить, что дочерние варианты использования обладают всеми атрибутами и особенностями поведения родительских вариантов. При этом, дочерние варианты использования участвуют во всех отношениях родительских вариантов. В свою очередь, дочерние варианты могут наделяться новыми свойствами поведения, которые отсутствуют у родительских вариантов использования, а также уточнять или модифицировать наследуемые от них свойства поведения.

Применительно к данному отношению, один вариант использования может иметь несколько родительских вариантов. В этом случае реализуется множественное наследование свойств и поведения отношения предков. С другой стороны, один вариант использования может быть предком для нескольких дочерних вариантов, что соответствует таксономическому характеру отношения обобщения.

Между отдельными актерами также может существовать отношение обобщения. Данное отношение является направленным и указывает на факт специализации одних актеров относительно других. Например, отношение обобщения от актера А к актеру В отмечает тот факт, что каждый экземпляр актера А является одновременно экземпляром актера В и обладает всеми его свойствами. В этом случае актер В является родителем по отношению к актеру А, а актер А потомком актера В. При этом актер А обладает способностью играть такое же множество ролей, что и актер В. Графически данное отношение также обозначается стрелкой обобщения.

Создание модели в Enterprise Architect

Создание модели в Enterprise Architect строится в иной последовательности. Сначала необходимо выбрать пункт меню *<File/New Project...>*. Далее следует ввести имя проекта, после чего необходимо будет определить заранее те модели, которые требуется создать в данный момент (рис 2.15) – диаграммы для моделей (и сами модели иных аспектов функционирования моделируемой системы) можно добавлять и на более поздних этапах проектирования.

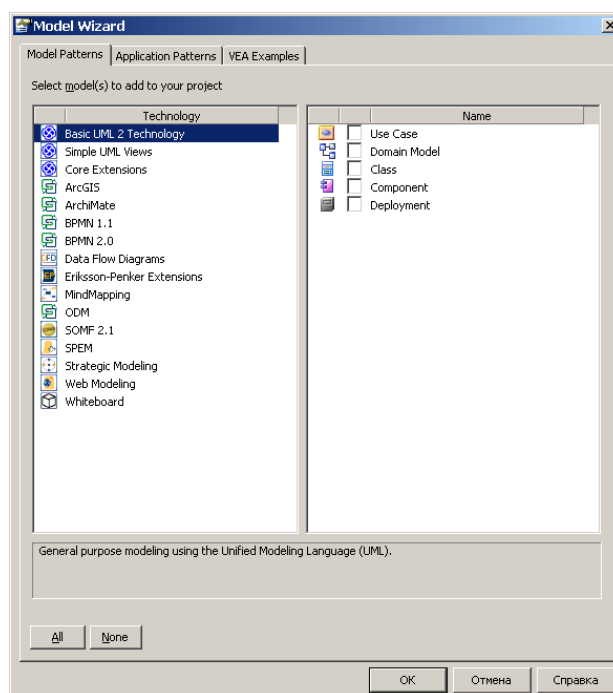


Рисунок 2.15 – Выбор диаграмм для модели в Enterprise Architect

Выбранные диаграммы станут доступны в менеджере проекта. Для перехода к нужной диаграмме необходимо выполнить двойной щелчок на нужной в дереве, после чего станет возможным перемещаться между диаграммами через выбор нужной закладки внизу экрана.

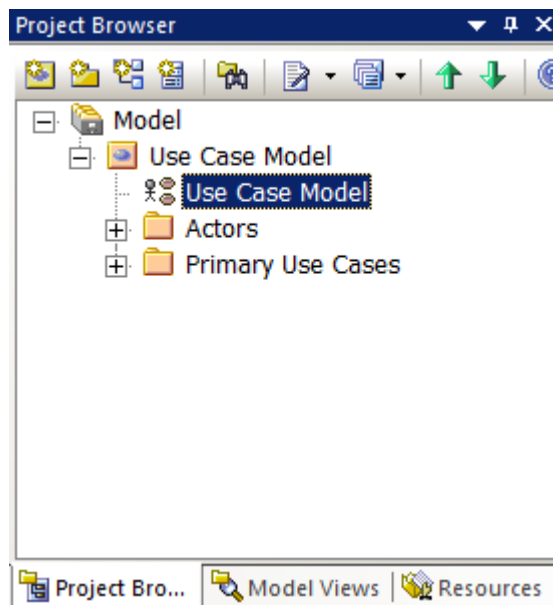


Рисунок 2.16 – Диаграммы в Project Browser

В случае, если пользователь решит добавить к модели дополнительные диаграммы, он может сделать это путём нажатия пиктограммы, изображенной на рис.2.17

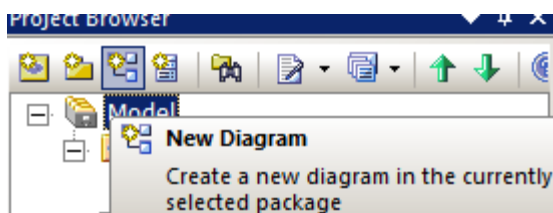


Рисунок 2.17 – Добавление диаграммы к модели

Набор кнопок в панели инструментов зависит от того, какая диаграмма применяется. Например, в случае диаграммы прецедентов, она будет иметь вид как на рис.2.18-1, в случае диаграммы классов, рис.2.18-2.

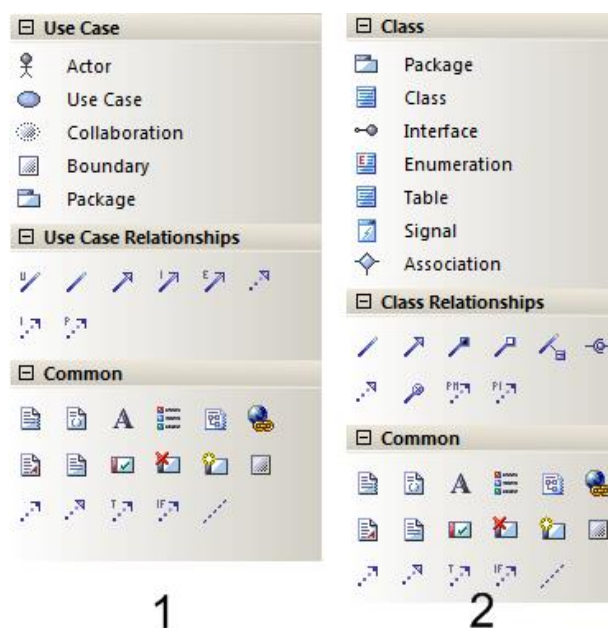






















Рисунок 2.18 – Внешний вид панели инструментов Enterprise Architect









Описание элементов интерфейса приведено в таблице 2.4. Описаны лишь те элементы, которые пригодятся для выполнения работ данного методического пособия.















Таблица 2.4 – Элементы панели инструментов Enterprise Architect













	<i>Наименование</i>	<i>Назначение</i>
	Actor	Новый актер
	Use Case	Новый прецедент
	Collaboration	Создает хранилище, логически объединяющее связанный набор ролей и связей между ними.
	Boundary	Создает визуальную структуру, объединяющую элементы. Функциональной нагрузки не несёт.
	Package	Контейнер, который может содержать в себе как другие контейнеры и диаграммы, так и

		набор элементов, объединенных какой-либо логикой
	Use	Создает отношение использования
	Associate	Создает отношение ассоциации
	Generalize	Создает отношение наследования
	Include	Создает отношение включения
	Extend	Создает отношение расширения
	Realize	Описывает отношение, при котором набор элементов приемника описывается набором элементов передатчика
	Invokes	Данные отношения определены в OML (Open Modeling Language). Описывают отношения зависимости. Invokes показывает, что прецедент А в конкретный момент влечет выполнение В, в то время, как Precedes показывает, что перед выполнением D должен полностью завершиться прецедент С.
	Precedes	
	Note	Создает заметку
	Constraint	Создает ограничение
	Text element	Создает текстовый элемент
	Diagram legend	Добавляет к диаграмме легенду
	Diagram notes	Добавляет информацию по диаграмме
	Hyperlink	Добавляет на лист диаграммы гиперссылку
	Document	Позволяет вложить в диаграмму документ
	Artifact	Экспонат – любая физическая информация, используемая или произведенная системой, представленной в диаграмме Развертывания. Экспонаты могут связывать свойства или операции, и могут иллюстрироваться

		примерами или быть связанными с другими Экспонатами. Примеры Экспонатов включают файлы моделей, исходные файлы, таблицы базы данных и т.д.
	Requirement	Описывает требование. Внешнее или внутреннее
	Issue	Описывает дефект
	Change	Описывает изменение
	Boundary	Создает визуальную структуру, объединяющую элементы.
	Dependency	Создает связь зависимости. Это означает, что элемент или набор элементов связанный отношением зависимости, требует других элементов для своего полноценного описания или выполнения
	Trace	Подвид отношения зависимости. Соединяет элементы или наборы элементов, представляющие те же принципы, но в масштабе моделей.
	Information flow	Описывает поток данных (информации) передаваемых от передатчика к приемнику
	Note link	Создает линию, описывающую связь с заметкой
	Package	Сущность, которая представляет собой группу классов и интерфейсов
	Class	Создает класс. Внешний вид класса описывается тремя секциями: - в верхней указывается имя;

		<ul style="list-style-type: none"> - в средней описываются свойства, атрибуты, и ссылки; - в нижней части описываются методы (процедуры и функции). Подчеркивание означает статическую операцию.
	Interface	Интерфейсы аналогичны классом, за исключением заголовка с < >
	Enumeration	Создает перечисление
	Table	Создает таблицу – частный вид класса. Отличается наличием свойств с описанием типа базы данных и возможностью указать информацию по колонкам, а также определить триггеры и индексы
	Signal	Определяет сигнал – передачу запроса Send. Объект получения обрабатывает экземпляры класса запроса как определено его свойствами. Данные, которые несёт Send, представлены как атрибуты сигнала. Сигнал определен независимо от классификаторов, обрабатывающих его возникновение.
	Association	n-арная ассоциация связывает три и более элемента
	Compose	Используется для демонстрации агрегации большого числа мелких частей в более крупный объект. При удалении родительского объекта удаляются и связанные
	Aggregate	Создает отношение включения
	Association class	Создает включенный класс

	Assembly	Создает связь, указывающую на необходимость описания интерфейса между связанными объектами
	Nesting	Создает связь, графически описывающую механизм включения одними элементами других
	Package merge	Создает связь, описывающую объединение элементов
	Package import	Создает связь, описывающую что один пакет импортирует другой
	Lifeline	Описывает конкретный экземпляр, участвующий во взаимодействии (множественность не подразумевается).
	Boundary	Описывает объект, который имеет некоторую ограниченность. Используется в концептуально фазе проектирования, чтобы фиксировать пользователей, взаимодействующих с системой.
	Control	Создает объект контроль – показывающий, что модель контролирует сущность или актера
	Entity	Создает сущность
	Fragment	Описывает фрагмент
	Endpoint	Описывает точку остановки
	Diagram gate	Графически обозначает точку через/в которую идёт передача информации для фрагмента
	State	Описывает некое состояние системы – статичное или динамичное
	Message	Описывает сообщение
	Self message	Описывает сообщение объекта для самого себя

	Call	Описывает вызов
	Recursion	Описывает рекурсивный вызов
	Activity	Описывает активность, возникающую как элемент функционирования системы или передачи информации
	Action	Действие описывает базовый процесс или передачу информации, которые возникают в системе
	Partition	Используется для логической организации действий
	Object	Частный случай класса в момент его выполнения
	Central buffer node	Описывает узел для управления потоками информации, поступающей из нескольких источников
	Datastore	Описывает хранилище данных
	Decision	Описывает условие – момент, в который возможно изменить поведение системы
	Merge	Узел слияния – узел контроля, который примиряет множественные потоки данных. Не используется, чтобы синхронизировать параллельные потоки: просто выбирает один из многих и идёт по нему.
	Send	Используется чтобы изобразить действие по посылке сигналов
	Receive	Используется чтобы изобразить действие по приему сигналов

	Synch	Описывает точку, изображающую ситуацию синхронизации параллельных путей функционирования модели
	Initial	Описывает точку начала
	Final	Описывает точку конца
	Flow final	Описывает конец информационного потока (преднамеренный и нет)
	Exception	Описывает ситуацию исключения
	Fork/join	Создает элемент, являющийся буферным, и позволяющий разделить информацию на несколько потоков
	Control flow	Соединяет две активности в диаграмме активности
	Object flow	Соединяет две активности или состояния и показывает факт передачи объектов между ними
	Interrupt flow	Абстрактный класс для направленных подключений между двумя узлами деятельности

Размещение объектов разных типов не вызывает никаких трудностей. Однако, следует по максимуму использовать возможности Enterprise Architect по объединению элементов в пакеты и группы. Следует учитывать, что в одной модели может быть несколько диаграмм одного типа, описывающего разные варианты поведения системы, и чем более полно разработчик стремится отразить функционирование модели, тем больше будет этих диаграмм.

При единократном размещении объекта в модели, его можно использовать в разных диаграммах, причём использовать в трёх разных ролях (рис.2.19):

- В качестве ссылки на объект (физически объект останется на той диаграмме, на которой был размещен первично, а пользователь изменяя свойства и имя объекта в дочерней диаграмме, будет менять свойства объекта и на родительской);
- В качестве экземпляра существующего класса (при этом размещаемый объект получит собственное имя, однако свойства его будут определяться свойствами объекта, экземпляром класса которого он является);
- В качестве потомка родительского объекта (при этом, наследуя все свойства родительского объекта, объект-потомок может обладать и своими уникальными атрибутами).

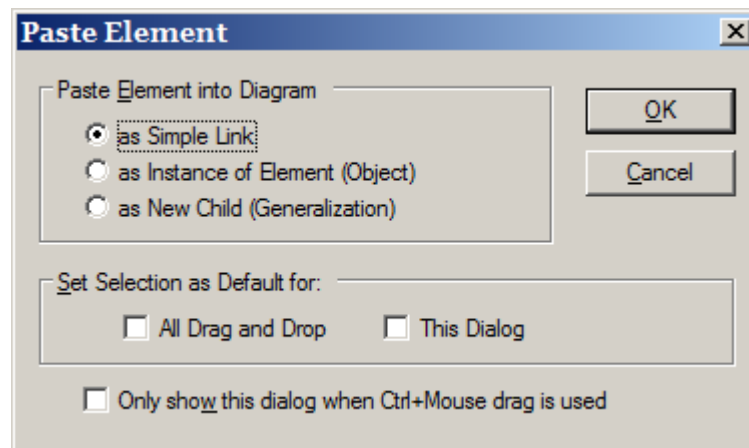


Рисунок 2.19 – Размещение элемента в диаграмме

Создание модели в Visual Paradigm

Visual Paradigm – это онлайн средство визуального проектирования. Обладает богатым функционалом и позволяет создать большинство видов диаграмм, используемых в описании бизнес-процессов организаций любого масштаба. Внешний вид приведен на рис.2.20.

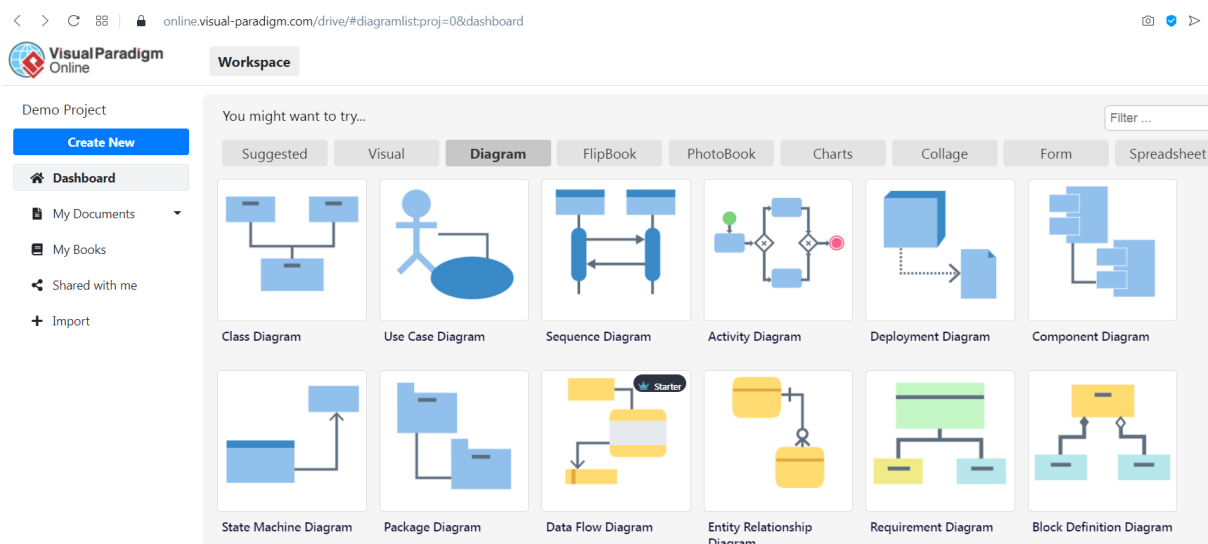















Рисунок 2.20 – Внешний вид страницы выбора типа создаваемой диаграммы

Первым этапом проектирования является выбор типа создаваемой диаграммы. В случае выбора Use Case Diagram-Create Blank, создается рабочая область с добавленными инструментами проектирования, характерными для данного типа диаграмм (см. таб. 2.5).

Таблица 2.5 – Инструменты работы с диаграммами Use Case в Visual Paradigm

	<i>Наименование</i>	<i>Назначение</i>
	Actor	Новый актер
	Use Case	Новый прецедент
	Association	Создает отношение ассоциации
	Include	Создает отношение включения
	Extend	Создает отношение расширения
	Generalize	Создает отношение наследования

	Dependency	Создает связь зависимости
	Collaboration	<p>Сотрудничество определяет набор сотрудничающих ролей и их связей. Они используются для иллюстрации конкретной функциональности при совместной реализации описываемой задачи. Сотрудничество должно указывать только роли и атрибуты, необходимые для выполнения определенной задачи или функции.</p> 
	Note	Создает заметку
	Anchor	Создает линию, описывающую связь с заметкой
	Constraint	<p>Ограничения. Ограничение – это условие или ограничение, в соответствии с которым работает вариант использования, которое включает предварительные, пост- и инвариантные условия. Предварительное условие определяет условия, которые должны быть выполнены перед активацией связанного варианта использования.</p>
	Containment	Заменяет собой вложенность. Ставится между объектом сотрудничества (или пакетом) и



Следует отметить, что вне зависимости от выбора инструментария и внешнего вида графических компонент, функционал любой диаграммы аналогичен ввиду стандартизированного подхода. Отличаются нюансы, язык интерфейса, формат хранения данных – но любая диаграмма понятна человеку, знакомому с графической нотацией.

Для Visual Paradigm следует отметить в качестве особенности работу с отношением вложенности, разбивку диаграмм по отдельным страницам (см.рис.2.21), создание ссылок на объекты других страниц через контекстное меню (см.рис.2.22).

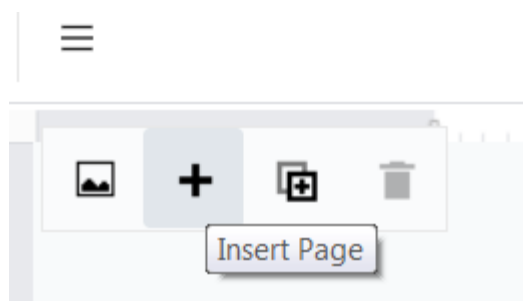


Рисунок 2.21 – Добавление новой страницы в Visual Paradigm

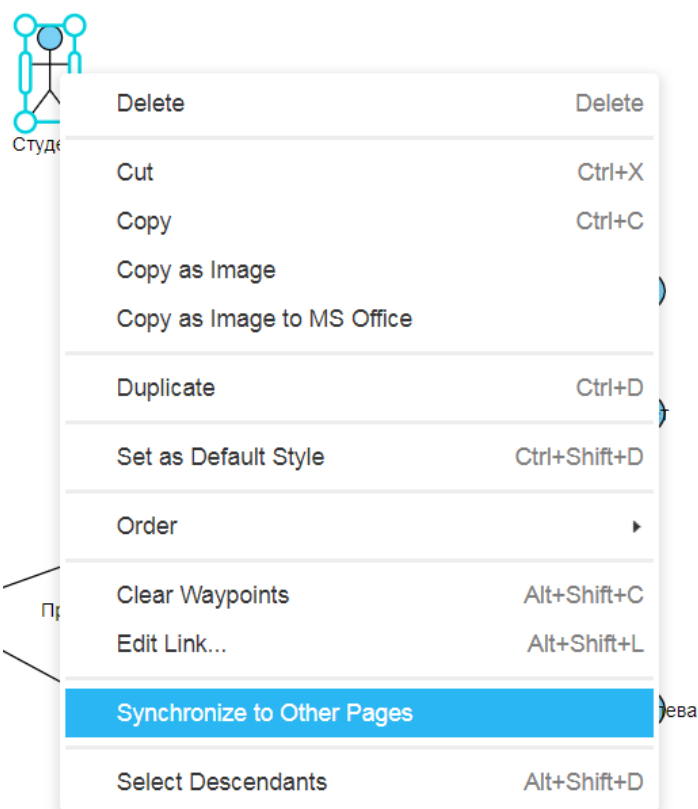


Рисунок 2.22 – Создание ссылки на объект при необходимости использовать его в другой диаграмме того же проекта

Пример выполнения задания

В качестве примера на лабораторных работах будем рассматривать предметную область «Учёба в вузе».

Постановка проблемы: проектируется программное обеспечение для управления учебным процессом. Должны быть реализованы следующие возможности: ведение (редактирование) справочников «Студенты» и связанного с ним «Рейтинг», вносить информацию об учебном плане: преподаваемых дисциплинах и преподавателях-предметниках, формирование отчетов по успеваемости и оплате. Пользователями программной системы являются деканат и администратор БД.

Администратор БД имеет возможность редактирования базы данных (создавать и удалять любые записи). Следит за работоспособностью системы в целом. Преподаватель входит в систему с правами пользователя – имеет возможность редактирования

справочника студентов и текущего рейтинга (создавать и удалять записи в этих справочниках).

Создание диаграммы вариантов использования (Use Case)

Этап 1: При создании диаграммы вариантов использования начинаем с выделения действующих лиц (или, актеров) участвующих во взаимодействии с нашей системой. В нашем случае можно выделить несколько типов действующих лиц, необходимых для отражения разных аспектов функционирования системы: *Студент, Деканат, Администратор БД, Преподаватель, Сотрудники.*

Кроме того, если мы хотим подчеркнуть общность между *Администратором БД, Деканатом* и *Преподавателем* (они являются сотрудниками одного и того же вуза), можно выделить действующее лицо *Сотрудники* и наладить связь обобщения между ними. Создание прямой связи (use) между пользователями, типа «*Пользователь А – Пользователь В*», в данном случае является ошибкой, так как не несет в себе информации о взаимодействии пользователя с системой.

Этап 2: Следующим шагом является выделение возможностей (функций, вариантов использования), которые программа должна предоставлять данным пользователям. На основании общего описания системы (см. начало главы) можно выделить следующие возможности программы: «*Регистрация оплаты в БД*», «*Регистрация учебного процесса в БД*», «*Создание записи*», «*Удаление записи*» и т.д.

Вариантами использования могут быть только те возможности, которые будут реализованы в нашей системе. Например, такие функции, как «*Администратор следит за работоспособностью системы*» или «*Преподаватель ставит оценку студенту в зачётную книжку*» происходят без использования функций проектируемой системы, это лишь последствия или внешние действия, имеющие косвенное отношение к системе. Поэтому подобные функции не будут включены в диаграмму вариантов

использования (следует оговориться, что данные функции не могут быть включены при описании программного продукта, однако вполне могут использоваться, например, при описании бизнес-процессов, важных для отражения логики функционирования нашей предметной области).

Старайтесь не вдаваться в детали методов реализации, которые будут использоваться при реализации данного программного продукта. Например, такие действия, как *«Осуществить транзакцию для записи справочника в БД»* или *«Проверить логин и пароль»* относятся уже скорее к технологии создания программы, чем к возможностям. Старайтесь опираться, прежде всего, на внешнее представление программы, поставьте себя на место конечных пользователей программы, которые «не знают», каким образом работает та или иная функция программы, однако вполне могут пользоваться всеми её возможностями.

Вариант использования – это всегда какое-то действие. Не допускайте имен, обозначающих некие объекты, например, *Отчет*, *Справочник*. Вместо этого используйте действия над этими объектами *«Формирование отчета»*, *«Редактирование справочника»*.

При определении количества вариантов использования, размещаемых на диаграмме, придерживайтесь правила «золотой середины». Каждый вариант использования должен быть не слишком обобщенной, а вполне конкретной функцией. Например, вариант использования *«Ведение БД»* есть слишком общее понятие, в нашем случае это и *«Редактирование справочников»* и *«Внести данные учебного плана»* и т.п. Однако большого количества функций также следует избегать, так как схема может потерять свою наглядность. В этом случае, некоторые варианты использования можно объединить. Например, в нашем случае *«Редактирование справочника»* и *«Заполнение поля справочника ФИО»*, *«Заполнение поля справочника Адрес»* нет смысла разделять в разные варианты использования, так как эти операции тесно взаимосвязаны, и выполняются одновременно и в совокупности друг с другом.

Этап 3: Важным шагом является создание связей между Актерами и вариантами использования. Ассоциация «Пользователь-Вариант использования» отображает связь использования. Обратная ассоциация «Вариант использования-пользователь» показывает, что «Вариант использования» при инициализации передает некоторую информацию пользователю.

Не забывайте о том, что каждый пользователь на диаграмме представляет собой «Роль», которую играет тот или иной внешний объект. Причем один и тот же человек может играть несколько ролей в системе. Например, «Преподаватель Иванов И.И.». может зайти в систему в качестве представителя деканата и исправить справочники, или посмотреть текущую успеваемость студента и т.д., а может войти в качестве администратора и удалить учётную запись студента. В связи с этим ассоциаций между ролью «Администратор БД» и вариантами использования «Редактирование справочников» – не существует.

Этап 4: Последним этапом создания диаграммы является документирование объектов диаграммы. Документация на активный объект вносится в поле *Documentation*. В табл.2.6. показан пример документации на объекты нашей диаграммы.

Таблица 2.6 – Документация на объекты диаграммы Use Case

Объект	Документация
<i>Действующие лица</i>	
Администратор БД	Управляет доступом пользователей к системе, следит за ее работоспособностью, отвечает за структуру и целостность БД.
Деканат	Работает с преподавателями, заполняя по переданным ими сведениями БД информацией по текущей успеваемости студента. Получает от

	студентов оплату за обучение и регистрирует данную информацию в БД.
Преподаватель	Вносит текущий рейтинг
Студент	Вносит плату за обучение, получает оценки в ходе учебного процесса
<i>Варианты использования</i>	
Обновить рейтинг	Преподаватель на основании данных учебного плана, подготовленных деканатом (дисциплина, рейтинговая система) по результатам оценки знаний студента в ходе учебного процесса вносит данные в справочник «Рейтинг»
Редактирование справочников	Добавление, удаление, корректировка элементов справочника «Студент» и «Рейтинг», сохранение данных в БД.
Формирование отчета	Вывод сформированного отчета на экран с целью просмотра или печати
Авторизация в системе	На основе проверки учетных данных определяется роль и предоставляются права
<i>Ассоциации</i>	
Администратор – Создание учетной записи.	Использует для добавления, удаления, изменения учетных записей пользователей системы.
Преподаватель – Редактирование справочников	Использует для добавления, удаления, изменения элементов справочника.
Студент – Оплата учёбы	Использует для передачи деканату информации о сумме и типе оплаты.

При документировании пользуйтесь следующими соображениями:

а) Пользователь – это внешний объект нашей системы. Поясните, что он собой представляет и какую роль играет в нашей системе.

б) Вариант использования – это функция системы. Поясните, какие действия выполняет данная функция, какие возможности она реализует.

в) Ассоциация – показывает, каким образом данный Пользователь использует данную Функцию. Учитывайте, что разные пользователи могут по-разному использовать один и тот же вариант использования.

Окончательный вид диаграммы прецедентов для проектируемой системы показан на рис.2.23-2.26.

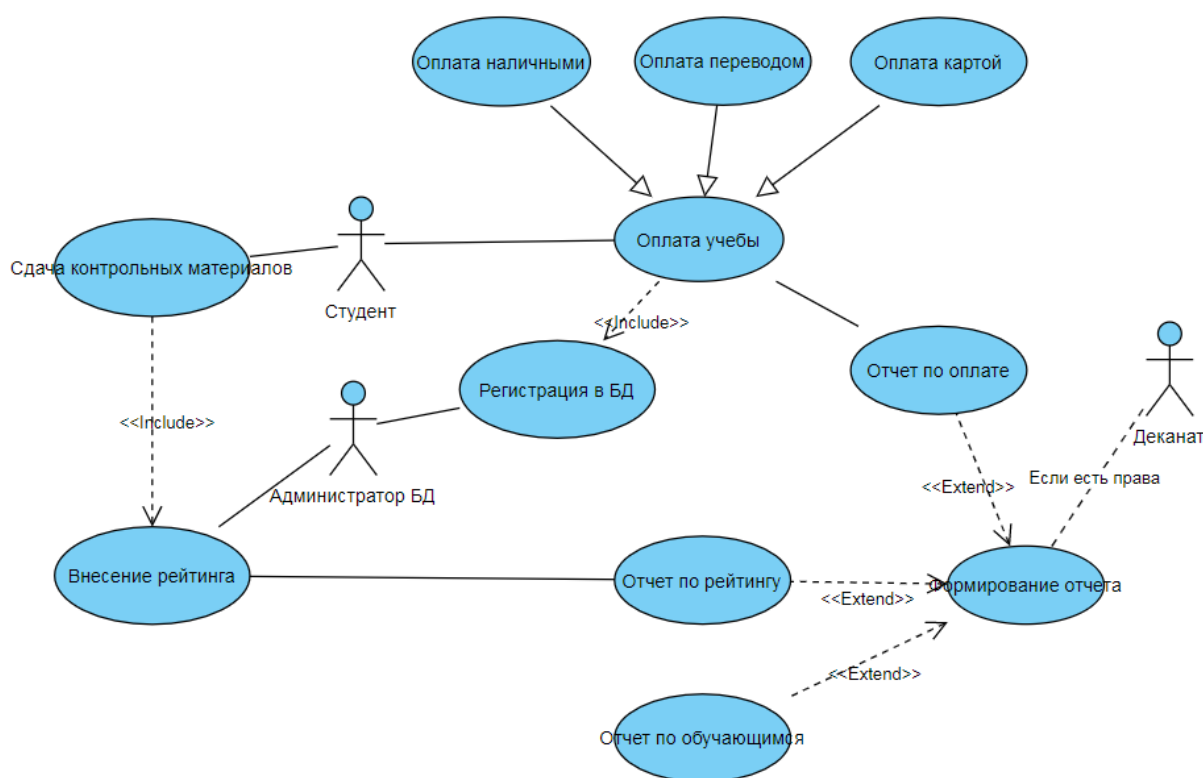


Рисунок 2.23 – Диаграмма вариантов использования «Оплата учёбы» и «Формирование отчетов»

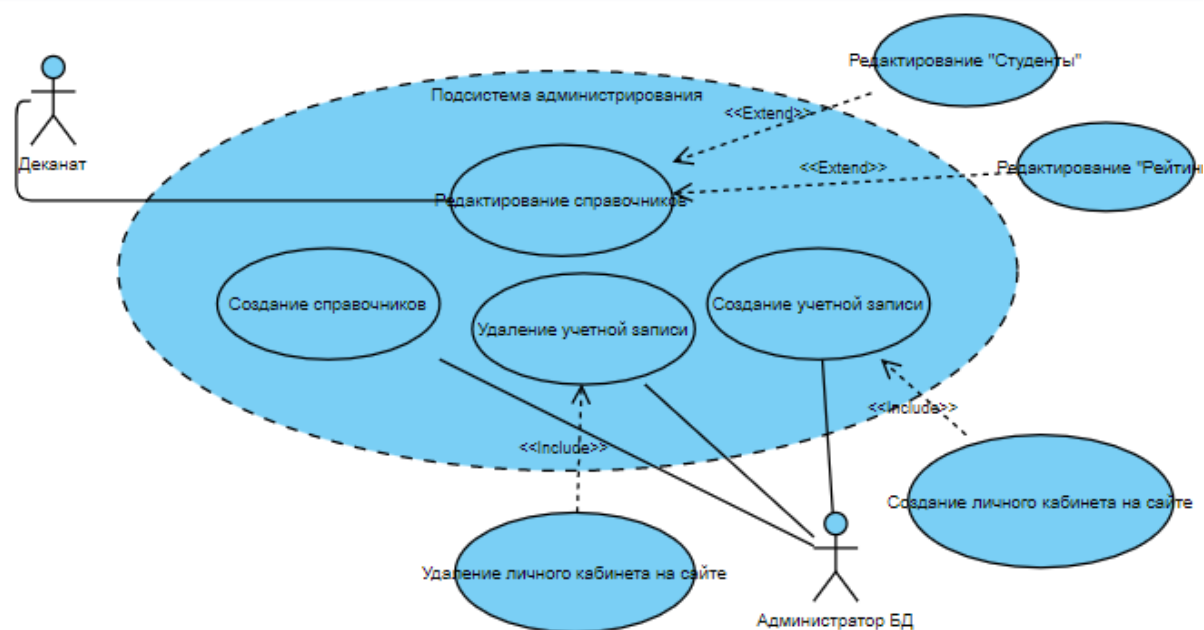


Рисунок 2.24 – Диаграмма вариантов использования «Работа со справочниками»

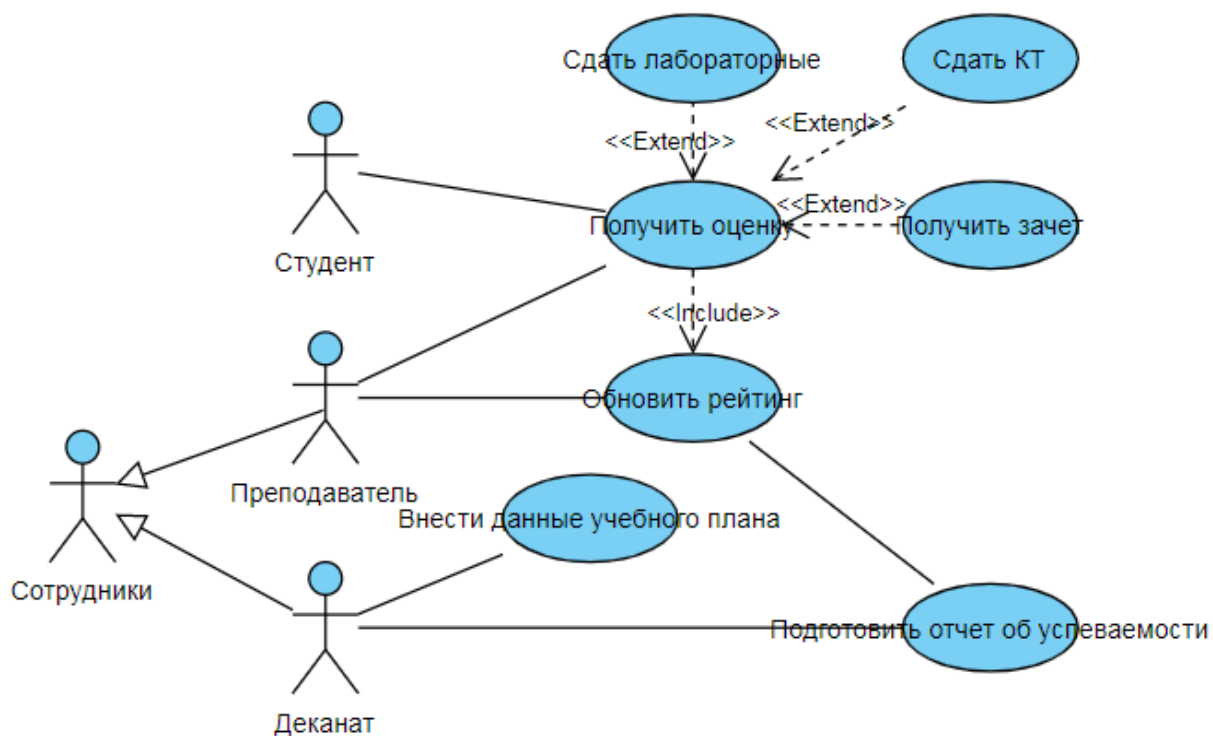


Рисунок 2.25 – Диаграмма вариантов использования «Учебный процесс»

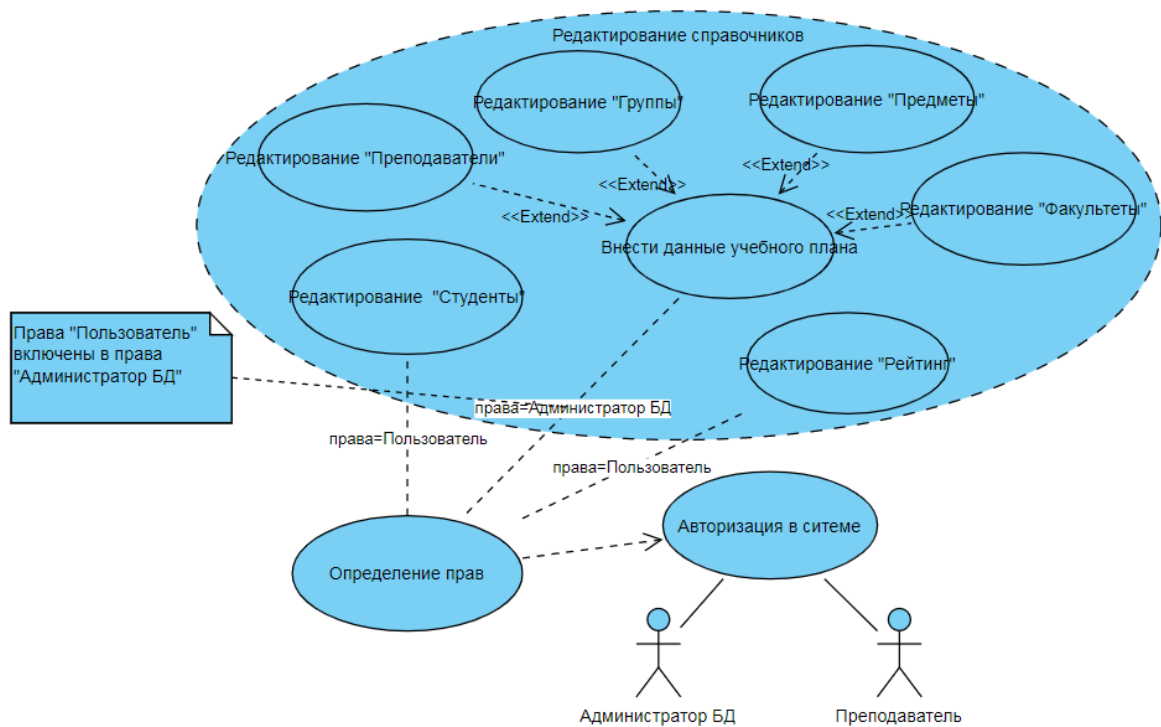


Рисунок 2.26 – Диаграмма вариантов использования «Вход в систему»

Задание

Необходимо создать диаграмму прецедентов в инструментальной среде UML редактора, следуя описанным принципам работы с системой. Объектом автоматизации является тот же процесс, что и в 1-й лабораторной работе (Приложение 1).

Документы отчетности сдаются на проверку в электронной форме и включают в себя:

- файл модели (скриншоты);
- Текстовый документ, содержащий описание потоков событий прецедентов моделируемой информационной системы.

Допускается документирование потоков событий и основных проектных решений в рамках среды UML редактора с использованием окна документирования. Помните, выходной файл данной модели будет использоваться и в остальных работах, поэтому заранее определите, какие диаграммы будут использоваться в данном проекте.

Лабораторная работа №3

Цель работы

Целью данной работы является построение диаграммы классов.

Краткое изложение теоретической части

В UML диаграмма классов является типом диаграммы статической структуры. Она описывает структуру системы, показывая её классы, их атрибуты и операторы, а также взаимосвязи этих классов

Взаимосвязь — это особый тип логических отношений между сущностями, показанных на диаграммах классов и объектов. В UML'e представлены следующие виды отношений:

1) Ассоциация — показывает, что объекты одной сущности (класса) связаны с объектами другой сущности. Существует пять различных типов ассоциации. Наиболее же распространёнными являются двунаправленная и однонаправленная. Например, классы рейс и самолёт связаны двунаправленной ассоциацией, а классы человек и кофейный автомат связаны однонаправленной. Двойные ассоциации (с двумя концами) представляются линией, соединяющей два классовых блока. Ассоциации в большей степени имеют более двух концов и представляются линиями, один конец которых идет к классовому блоку, а другой к общему ромбику. В представлении однонаправленной ассоциации добавляется стрелка, указывающая на направление ассоциации. Ассоциации могут быть именованными, и тогда на концах представляющей её линии будут подписаны роли, принадлежности, индикаторы, мультипликаторы, видимости или другие свойства.

Агрегация — это разновидность ассоциации, при отношении между целым и его частями. Как тип ассоциации, агрегация может быть именованной. Агрегация не может включать сразу несколько классов. Агрегация встречается, когда один класс является коллекцией или

контейнером других. Причём по умолчанию, агрегацией называют агрегацию по ссылке, т.е. когда время существования содержащихся классов не зависит от времени существования содержащего их класса. Если контейнер будет уничтожен, то его содержимое – нет. Графически агрегация представляется пустым ромбиком на блоке класса и линией, идущей от этого ромбика к содержащемуся классу.

Композиция – более строгий вариант агрегации. Известна так же как агрегация по значению. Композиция имеет жёсткую зависимость времени существования экземпляров класса контейнера и экземпляров содержащихся классов. Если контейнер будет уничтожен, то всё его содержимое будет также уничтожено. Графически представляется как и агрегация, но с закрашенным ромбиком.

Различие между композицией и агрегацией заключается в следующем: целое композиции должно иметь мультипликатор 0..1 или 1, что показывает, что часть является частью только одного целого, в агрегации же может быть любой мультипликатор. Приведём наглядный пример. Двигатель является частью машины, следовательно здесь подходит композиция. В то же время, когда они представлены в базе данных, данная модель двигателя может быть у разных моделей машин, поэтому следует использовать агрегацию.

2) Обобщение (Generalization) показывает, что один из двух связанных классов (подтип) является более частной формой другого (надтипа), который называется обобщением первого. На практике это означает что любой экземпляр подтипа является также экземпляром надтипа. Например: животные – супертип млекопитающих, которые в свою очередь супертип приматов и так далее. Эта взаимосвязь легче всего описывается фразой «А – это Б» (приматы — это млекопитающие, млекопитающие — это животные). Графически генерализация представляется линией с пустым треугольником у супертипа. Генерализация также известна как наследование или “is a” взаимосвязь.

3) Реализация – отношение между двумя элементами модели, в котором один элемент (клиент) реализует поведение, заданное другим (поставщиком). Графически реализация представляется также как и генерализация, но с пунктирной линией.

4) Зависимость – это отношение использования, при котором изменение в спецификации одного влечёт за собой изменение другого, причем обратное не обязательно. Графически представляется пунктирной стрелкой, идущей от зависимого элемента к тому, от которого он зависит. Существует несколько именованных вариантов. Зависимость может быть между экземплярами, классами или экземпляром и классом.

5) Уточнение отношений

Уточнение имеет отношение к уровню детализации. Один пакет уточняет другой, если в нем содержатся те же самые элементы, но в более подробном представлении. Например, при написании книги вы наверняка начнете с формулировки предложения, в котором кратко будет представлено содержание каждой главы. Предположим, что резюме к каждой главе в качестве отдельного элемента входит в пакет «Предложение». Допустим также, что «Завершённая книга» — это пакет, элементами которого являются законченные главы. В этом контексте пакет «Завершённая книга» является уточнением пакета «Предложение».

Мощность отношения (мультипликатор) означает число связей между каждым экземпляром класса (объектом) в начале линии с экземпляром класса в ее конце. Различают следующие типичные случаи:

Нотация	Название	Пример
0..1	Ноль или один	У студента может быть действительный пропуск в общагу, а может не быть
1	Только один	У студента только одна мать
0 или *	Ноль или более	В течение учёбы студент может сдать много экзаменов, а может ни одного
1..*	Один или более	Студент спит минимум в одном месте

Работа с диаграммой классов в Rational Rose и Enterprise Architect

В Rational Rose лавная диаграмма классов (*Main*) уже присутствует во вновь созданной пустой модели, но возможно создание дополнительных диаграмм при помощи уже знакомых способов посредством контекстного меню *Logical View* в окне *Browse* или при помощи пункта *<Browse>* в главном меню. В Enterprise Architect вы можете добавить диаграмму классов, стандартным способом – через Project Browser.

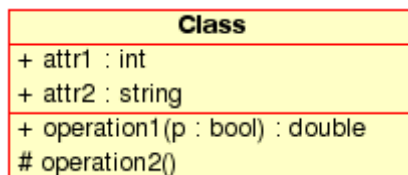


Рисунок 3.1 – Класс в UML

При активизации диаграммы строка инструментов Rational Rose меняет свой вид. Описание новых элементов управления приведено в таб. 3.1. Описание элементов управления в Enterprise Architect приведено в лабораторной работе №1. Функциональный смысл действий, которые пользователь может проделать – тот же, отличие заключается в расположении свойств по вкладкам. Для краткости будем давать пояснения для интерфейса Rational Rose, понимая, что всё сказанное справедливо и для Enterprise Architect.

Пояснение к терминологии, характерной для объектно-ориентированного программирования (ООП) и встречающейся в описании элементов управления, используемых для создания диаграммы классов, приведена в Приложении 2.

Таблица 3.1 – Элементы управления диаграммы классов



	<i>Имя</i>	<i>Назначение</i>
	<i>Class</i>	Данный инструмент позволяет создать новый класс в диаграмме и модели. Класс – это установки структуры и шаблона поведения для некоторого множества реальных объектов, которые в дальнейшем будут определены в программе на основе данного шаблона. Класс – это некоторая абстракция реального мира. Когда эта абстракция принимает конкретное воплощение, она называется объектом. Класс в UML изображается как прямоугольник, разделенный на 3 части (рис. 3.1). В верхней части записывается название класса, в середине – атрибуты, в нижней части – операции.
	<i>Interface</i>	Значок <i>Interface</i> позволяет создать интерфейсный объект, который указывает на видимые извне операции класса или компонента. Обычно интерфейс создается только для некоторых строго определенных классов или компонентов и предназначен скорее для логического отображения системы, но может присутствовать как на диаграмме классов, так и на диаграмме компонентов.

Диаграмма классов является основой проектируемой системы и является, пожалуй, наиболее важной схемой проекта. Однако создание диаграммы классов – это, прежде всего, интуитивный процесс и успех его выполнения зависит во многом от интеллекта и опыта проектировщика. Не следует также ожидать, что, пройдя все этапы создания диаграммы, вы сразу же получите диаграмму классов, учитывающую все детали функционирования системы, позволяющую реализовать все возможности и т.д.. Процесс описания классов системы является чаще всего итеративным процессом. Полученные диаграммы постоянно дорабатываются – зачастую

оказывается, что некоторые классы следует добавить, а другие следует убрать из системы, меняется набор и параметры атрибутов классов, изменяются методы классов, связи и т.п.

И Rational Rose и Enterprise Architect позволяют устанавливать значительное количество свойств класса, которые, в том числе, влияют на генерацию кода класса, поэтому, чтобы лучше ориентироваться в дальнейших действиях, разберем вкладки окна спецификаций.

Вкладка General

Это окно позволяет задать главные свойства класса, такие как его имя, тип, определить стереотип класса и доступ к нему, когда класс находится в контейнере. Так же как и во всех других диаграммах, здесь можно задать документацию к классу.

<i>Name</i>	Предназначено для задания имени класса
<i>Type</i>	Предназначено для задания типа класса. В нашем случае – это «класс», но можно выбрать значение «параметризированный класс», «инстанцированный класс» и др.
<i>Stereotype</i>	Задаёт стереотип класса
<i>Export Control (Scope/Visibility)</i>	Предназначен для определения доступа к классу, когда он расположен в контейнере. При этом <i>+Public</i> определяет, что элемент виден вне контейнера, в котором он определен и его можно импортировать в другие части создаваемой модели; <i>#Protected</i> – элемент доступен только для вложенных классов, классов с типом <i>friends</i> и собственно внутри класса; <i>-Private</i> обозначает защищенный элемент класса; <i>~Implementation</i> – элемент виден только в том контейнере, в котором определен

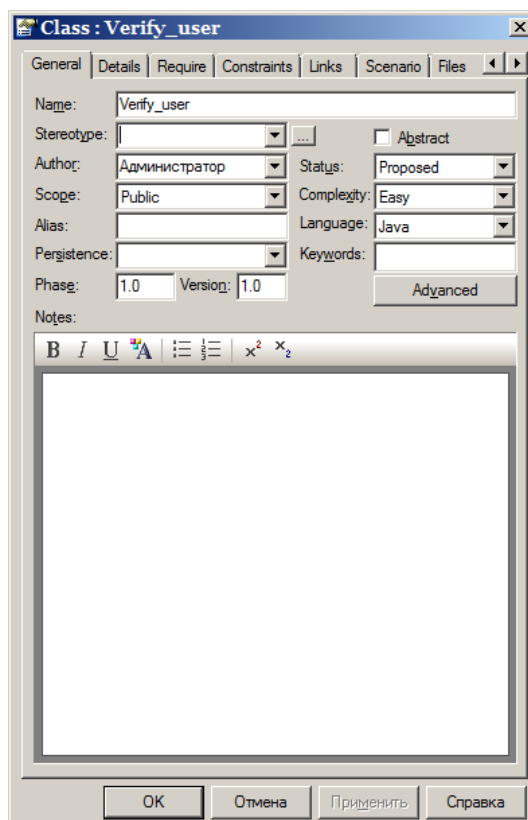


Рисунок 3.2 – Окно свойств класса в Enterprise Architect

В окне свойств Enterprise Architect дополнительно можно определить текущее состояние разработки (Status), сложность (Complexity), псевдоним (Alias), язык реализации класса (Language), время жизни (Persistence).

Вкладка Detail (детализация).

Позволяет указывать дополнительные установки класса, такие как ожидаемое количество создаваемых объектов класса, ожидаемый расход оперативной памяти и т.д.

Cardinality

(или *Multiplicity*)

Space

Persistence

Позволяет задать ожидаемое количество объектов, которые будут созданы на основе данного класса

Показывает количество оперативной памяти, необходимой для создания объекта данного класса

Определяет время жизни объекта класса. Если установлен флажок Persistent, то объект должен быть доступен в течение всей работы программы или для доступа других потоков или процессов

<i>Concurrency</i>	Обозначает поведение элемента в многопоточковой среде
<i>Abstract</i>	Обозначает, что класс является абстрактным
<i>Formal Arguments</i>	Заполняется только для параметризованных классов и утилит классов. Для обычных классов данное поле недоступно

Вкладка Attributes (атрибуты).

Данная вкладка (в Enterprise Architect подэлемент вкладки Details) позволяет добавлять, удалять, редактировать атрибуты класса.

На данной вкладке представлен список атрибутов класса, который можно редактировать при помощи контекстного меню. Флажок *Show inherited* позволяет скрыть или показать доступные атрибуты родительских классов.

Здесь пользователь может изменить название атрибута (*Name*), его тип (*Type*) и стереотип (*Stereotype*), задать начальное значение (*Initial value*) и тип доступа к атрибуту (*Export Control*).

Дополнительная вкладка *Detail* спецификаций атрибутов класса позволяет задать тип хранения атрибута в классе:

- *By Value* – по значению;
- *By Reference* – по ссылке;
- *Unspecified* – не указано.

Также пользователь может указать, что атрибут является *Static* (статическим) или *Derived* (производным).

Также можно определить видимость атрибута, а в Enterprise Architect дополнительно задать начальные/граничные значения, определить ограничения и прочие свойства, характерные для классов ООП.

Вкладка Operations (операции).

Вкладка *Operations* позволяет добавлять, удалять, редактировать операции класса.

На этой вкладке представлен список операций класса, который можно редактировать при помощи контекстного меню. Для того чтобы добавить операцию, необходимо из контекстного меню выбрать пункт *<Insert>*. По двойному нажатию мыши на операции или из контекстного меню Rational Rose предоставляет доступ к диалоговому окну спецификаций операции.

Вкладка *General* спецификаций операции аналогична вкладке *General* атрибутов.

Вкладка *Detail* спецификаций операций позволяет устанавливать дополнительные свойства операции.

<i>Arguments</i>	Позволяет устанавливать список аргументов для операции с их типами и значениями по умолчанию
<i>Exceptions</i>	Позволяет задавать список исключений, которые могут быть вызваны операцией. Здесь можно ввести имя одного или нескольких классов, обрабатывающих исключительные состояния
<i>Size</i>	Позволяет задать размер памяти, требуемой для выполнения операции
<i>Time</i>	Позволяет задать время выполнения операции
<i>Concurrency</i>	Отражает для многопоточковой программы тип выполнения операции

Вкладки *Preconditions*, *Postconditions*, *Semantics* позволяют задавать дополнительные описания процессов подготовки и завершения операции, а также описание алгоритма операции.

Вкладка Relations (связи).

На данной вкладке представлен список связей класса, который можно редактировать при помощи контекстного меню. Для добавления связи

лучше всего воспользоваться соответствующим инструментом из строки инструментов, а для удаления – контекстным меню (после удаления связей с диаграммы, полностью удалить их можно только посредством вкладки Relations).

Классы редко бывают изолированы, чаще всего они вступают в отношения друг с другом. Эти отношения показываются при помощи различного вида связей. Типы связей влияют на получаемый при генерации на основе диаграмм исходный код.

В диаграмме классов используются следующие виды связей:

- Unidirectional association (однаправленная ассоциация);
- Dependency (зависимость);
- Association class (ассоциированный класс);
- Generalization (наследование);
- Realization (реализация).

Unidirectional Association (→)

Одна из важных и сложных типов связи, которая используется в диаграмме классов. Данная связь показывает, что объекты одного класса взаимодействуют с объектами другого класса. Ассоциации имеют направление, показывая отношение одного класса к другому. Каждая сущность вовлечённая в данную связь выполняет определённую роль, роли могут быть назначены.

При нажатии правой кнопки мыши на связи активизируется контекстное меню, которое предоставляет быстрый доступ к установкам связи. Однако спецификации связи позволяют проделать то же самое при помощи вкладок диалоговых окон.

Активизируйте окно спецификаций при помощи контекстного меню или двойного нажатия мыши на стрелке ассоциации, при этом открывается вкладка *General* спецификаций (рис. 3.3).

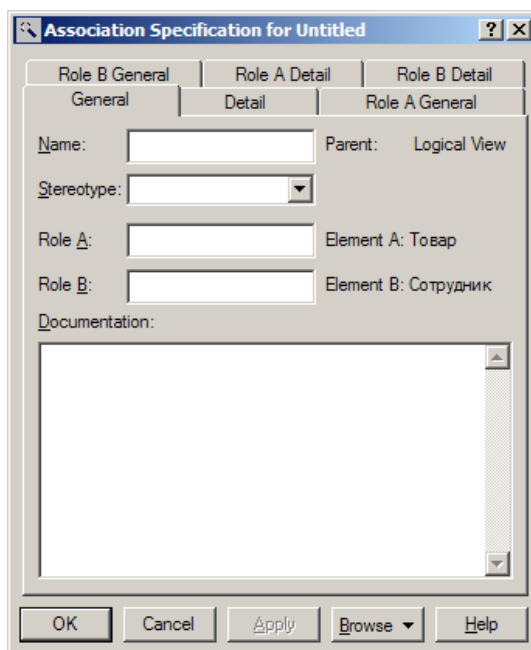


Рисунок 3.3 – Спецификация однонаправленной ассоциации

Вкладка General

Показывает информацию об имени, стереотипе, родительском классе и другую основную информацию о связи.

Name Задает имя связи. Для каждой связи может быть, хотя и не обязательно, задано имя, которое одним словом или целой фразой указывает цель или семантику связи

Parent Указывает имя контейнера, которому принадлежит связь

Stereotype Указывает стереотип

Role A/Role B Указывает имя роли, с которой один класс ассоциируется с другим

Element A/Element B Указывает имя класса, который ассоциирован с данной ролью

Вкладка Detail

Вкладка содержит информацию о дополнительных свойствах ассоциации.

<i>Name direction</i>	Указывает имя связанного класса
<i>Constraints</i>	Указывает выражение некоторого семантического условия, которое должно быть выполнено, в то время как система находится в устойчивом состоянии

Вкладка Role General

Отражает настройки переменной, которая будет включена в класс.

<i>Role</i>	Позволяет задавать имя переменной класса
<i>Element</i>	Показывает имя класса, для которого создается переменная
<i>Export Control</i>	Определяет доступ к данному элементу

Вкладка Role A/ Role B Detail

Показывает информацию об имени, стереотипе, родительском классе и другую основную информацию о связи.

<i>Multiplicity</i>	Показывает, сколько ожидается создать объектов данного класса, может быть задано числом или буквой «n». Значение «n» указывает, что количество не лимитировано. Можно задать различные варианты ожидаемого количества или диапазон. Причем, указанное число отображается рядом со стрелкой связи
<i>Navigable</i>	Показывает направление, в котором действует ассоциация. При установке этого флажка связь приобретает вид стрелки, указывающей направление связи. Это поле напрямую влияет на создаваемый код класса, так как на какой класс будет направлена стрелка связи, тот и будет

включен в другой. Для того чтобы изменить направление связи, достаточно снять флажок с одной вкладки (например, Role A Detail) и установить его во второй вкладке (Role B Detail). При этом в случае, когда сняты флажки, на обеих вкладках ни один элемент не будет включен в другой, а на диаграмме будет показана просто линия

Aggregate

Показывает, что один класс не просто использует, а содержит другой. Для того чтобы показать, что один класс входит в другой, необходимо установить этот флажок во вкладке Role Detail, соответствующей классу-агрегату. При этом стрелка связи на диаграмме приобретает ромб с обратной стороны стрелки.

Static

Обозначает, что данный реквизит – общий для всех объектов данного класса. Причем, после инициализации к нему можно обращаться, даже если еще не было создано ни одного объекта класса. Static применяется для того чтобы переменные такого типа не тиражировались при создании нового объекта класса

Friend

Определяет, что указанный класс является дружественным классом, то есть имеет доступ к защищенным методам и атрибутам класса

Key/Qualifiers

Атрибут, который идентифицирует уникальным образом единичный объект. На генерацию кода влияния не оказывает

В Enterprise Architect можно определить большую часть из перечисленных свойств (и даже больше) во вкладке *Source Role*.

Dependency or instantiates (зависимость или реализация) (↗)

Этот тип связи позволяет показать, что один класс использует объекты другого. Изменение в одной структуре может потребовать изменений в другой. Использование может осуществляться при передаче параметров или вызове операций класса. Во многих случаях другие типы зависимостей уже подразумевают какую-либо зависимость, но есть вы хотите описать зависимости более детально — вы можете использовать dependency, чтобы описать связь между элементами. Это подразумевает что зависимость слабая и не пригодна для использования в associative relation. Графически этот вид связи отражается пунктирной стрелкой.

Association Class (ассоциированный класс) (↗)

Используйте данный тип связи для отображения свойства ассоциации. Свойства сохраняются в классе и соединяются связью *Association* (рис. 3.4).

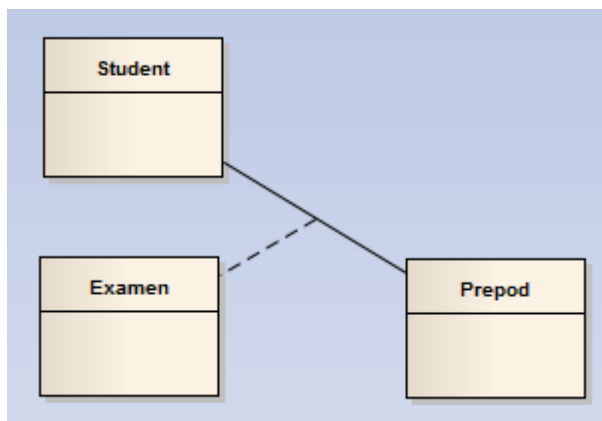


Рисунок 3.4 – Пример ассоциированной связи

Пунктирная линия в примере показывает нам что сущность **Student** относится к сущности **Prepod** и при этом существует сущность **Examen**, т.е. отношение между **Student** и **Prepod** зависит от существования **Examen**.

Generalization (наследование) (⌞)

Данный тип связи позволяет указать, что один класс является родительским по отношению к другому, при этом будет создана связь наследования класса. Пример такой связи показан на рис. 3.5.

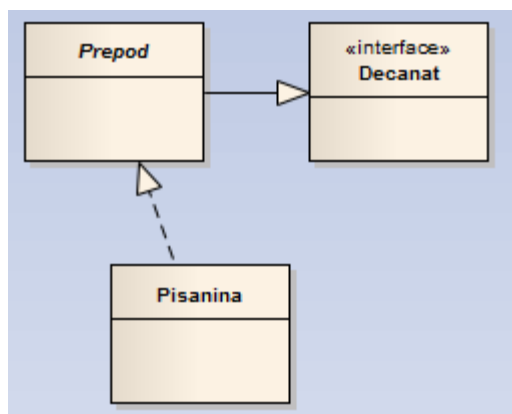


Рисунок 3.5 – Пример связи наследования

Абстрактный класс *Prepod* наследуется от интерфейса *Decanat*, *Pisanina* реализует функциональность *Prepod*.

Пример выполнения задания

Создание диаграмм классов является во многом не формализуемым процессом, однако, можно выделить несколько основных этапов:

1. Определение классов.

При идентификации классов нашей системы воспользоваться следующей методологией, которая состоит из двух шагов:

- Перечисление кандидатов в классы, найденных в постановке проблемы.
- Исключение ненужных и некорректных классов.

При выявлении кандидатов в классы выделяются все понятия (прежде всего, существительные), имеющие отношение к системе. Из общего описания проекта и диаграммы вариантов использования (см. лаб.2) выделяют список кандидатов в классы. Из списка кандидатов в классы исключают «плохие классы» согласно следующим критериям:

а) **Избыточные классы.** Если два класса отражают одну и ту же информацию, то сохраняется наиболее информативное имя. Например, хотя мы выделили отдельный класс *Дочерняя форма*, его функциональность полностью описывается родительским классом *Форма*.

б) **Нерелевантные классы.** Если класс не используется в решении проблемы, он должен быть исключен. Это определяется здравым смыслом, поскольку в другом контексте класс может быть важен. Например, в системе продажи театральных билетов должности покупателей безразличны, а должности персонала театра могут быть важны.

в) **Неясные классы.** Класс должен быть характерным, специальным. Некоторые пробные классы могут иметь плохо определенные границы или быть слишком широкими. Например, классы *Регистрация оплаты в БД*, *Регистрация процесса в БД* – являются неясными классами.

г) **Атрибуты.** Имена, которые изначально описывают другие объекты, должны быть утверждены как атрибуты. Например, имя, возраст, вес и адрес – обычно атрибуты. В нашем случае, например, *db* – атрибут класса, определённый как экземпляр класса DB.

д) **Операции.** Если имя описывает операцию, которая применяется к объектам, а не манипулирует ими с собственными правами, то это не класс. Например, телефонный звонок является последовательностью действий абонента и телефонной сети.

В нашем случае операции по взаимодействию с интерфейсом формы и методы по интерпретации процесса взаимодействия пользователя с формами ввода являются операциями класса *Main*, который содержит элементы графического интерфейса из класса *tk.Frame*, статичные без описания их функционала, реализованного в наборе соответствующих операций, при этом сценарий поведения элементов управления описан в классе верхнего уровня.

е) **Конструкции реализации.** Конструкции, являющиеся подсистемами, функциональными блоками или устройствами, используемыми в системе, должны быть исключены из модели. Они могут быть нужны позднее, на последующих стадиях проектирования, но не сейчас. Например, *Компьютер*, *Принтер*, *Информационный киоск* – конструкции реализации для большинства приложений.

2. Определение связей между классами

Любая зависимость между классами или ссылка одного класса к другому является ассоциацией. Атрибуты не должны быть ссылками на класс при моделировании, вместо них используются ассоциации.

Ассоциации часто соответствуют глаголам и глагольным фразам. Они включают физическое размещение (следующий, часть чего-либо, содержит), прямые действия (едет), коммуникацию (говорит кому-то), обладание (имеет) или удовлетворение некоторых условий (работает на, управляет).

3. Определение атрибутов классов

Атрибуты являются свойствами объектов, таких, как *размер формы, ширина бордюра* или *цвет*. При выделении атрибутов, следует учитывать следующие определения:

- 1) Атрибут должен быть характерен непосредственно для данного класса.
- 2) Атрибут должен отображать, одну и только одну характеристику класса.

Если выделенный атрибут являлся сложным объектом либо содержит некоторую совокупность характеристик, выделите атрибут в отдельный класс, либо разбейте его на несколько простых атрибутов. Например, атрибут *Элементы управления формой «Справочник «Студент»* является сложным составным атрибутом. Следует разделить данный атрибут на простые атрибуты: *Форма, Поле ввода 1, Поле ввода 2, Список значений 1* и т.д.

- 3) При описании классов, входящих в состав схемы наследования, для каждого атрибута необходимо определить будет ли он общим для классов-наследников или специфичным для определенного класса. В первом случае атрибут включается в базовый класс, во втором – в соответствующий класс-наследник.
- 4) В базе данных каждая запись имеет свой уникальный код (ключ), который может использоваться при написании кода программы для идентификации объекта. Однако при проектировании диаграммы

классов идентификаторы объектов не следует выделять в атрибуты. Например, свойство *Код студента* не будет отображаться на диаграмме классов, однако наверняка будет использоваться в коде программы, участвуя в запросах.

- 5) При определении спецификаций атрибутов (*Public*, *Protected*, *Private*) пользуйтесь следующими правилами:

Public-атрибут доступен содержащему его классу и классам, имеющим связь с данным классом. *Private*-атрибут доступен только содержащему его классу. Если атрибут является закрытым (*Private*), однако необходимо воздействовать на него извне, будет присутствовать открытая (*Public*) функция, изменяющая или высчитывающая значение этого атрибута. *Protected*-атрибут определяется в базовом классе и доступен базовому классу и его наследникам.

4. Выделение операторов (методов) классов.




- 1) Метод класса должен быть свойственным тому классу, где он определен. Например, метод *change_rate_combo_name()* будет являться оператором класса *Main*, несмотря на то, что он вызывается из метода *new_windows_rate()* и управляет заполнением списка значений на дочерней по отношению к классу *Main* форме, создаваемой внутри метода *new_windows_rate()*.
- 2) Доступ к методу извне содержащего его класса может быть осуществлен только в том случае, если метод описан с открытыми (*Public*) правами доступа. Закрытые (*Private*) методы также обычно иницируются открытыми методами. Т.е. не стоит менять значение атрибута класса *Main* напрямую, следует написать обработчик, который меняет значение атрибута, и обращаться для изменений именно к обработчику.
- 3) Операторы не должны быть слишком сложными, состоящими из множества различных функций. В этом случае необходимо разбить оператор на более простые составляющие. Например, для каждого






отдельного вида изменений (добавить, редактировать, удалить запись) пишется своя функция – *insert_pred()*, *update_pred()*, *delete_pred()* и т.д.

- 4) Исключение из правила принадлежности метода классу составляют методы по созданию и удалению объектов класса. Подобные методы не могут быть включены в класс, так как класс не может вызвать метод создания собственного объекта. Создание объекта класса может произойти лишь извне, то есть из другого класса. Так, например, метод по созданию объекта класса *DB*, может быть включен в класс *Main*, но никак не в сам класс *DB*.
- 5) Соблюдайте принцип полноты методов класса. Если, например, в классе присутствует метод *insert_rate()*, то должен быть определен метод *delete_rate()*.

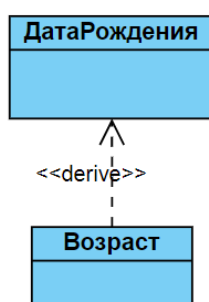
Результатом работы будет построенная в Visual Paradigm диаграмма классов, внешний вид которой приведен на рис.3.6. Элементы по работе с диаграммой приведены в таблице 3.2.

Таблица 3.2 – Элементы управления для Class Diagram в Visual Paradigm

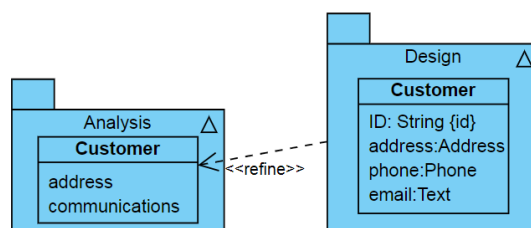
	<i>Имя</i>	<i>Назначение</i>
	<i>Class</i> (<i>Interface</i> , <i>Enumeration</i> , <i>Primitive</i> , <i>Entity Bean</i>)	Создает новый класс
	<i>Generalization</i> (<i>Realization</i>)	Создает отношение наследования
	<i>Usage</i>	Создает отношение использования. Использование – это отношение зависимости, в котором одному элементу (клиенту) требуется другой элемент (или набор элементов) (поставщик) для его полной реализации или работы. Зависимость использования не указывает, как клиент использует поставщика,

		<p>кроме того факта, что поставщик используется определением или реализацией клиента. Например, это может означать, что некоторые методы в классе (клиента) используют объекты (например, параметры) другого класса (поставщика).</p>
	<p><i>Association</i> (<i>Composition</i>, <i>Aggregation</i>)</p>	<p>Создает отношение ассоциации (композиции, агрегации)</p>
	<p><i>N-Array association</i></p>	<p>N-арная ассоциация – это связь, существующая между тремя или более классами. Связь не может быть разбита на обычную двоичную ассоциацию без потери некоторой информации.</p>
	<p><i>Association class</i></p>	<p>Создает ассоциированный класс</p>
	<p><i>Dependency</i> (<i>Binding</i> <i>Dependancy</i>, <i>Premission</i>, <i>Access</i>, <i>Import</i>, <i>Merge</i>, <i>Instantiation</i>, <i>Substitution</i>)</p>	<p>Создает связь зависимости</p>
	<p><i>Abstraction</i> (<i>Derive</i>, <i>Refine</i>, <i>Trace</i>)</p>	<p>Абстракция – это отношение зависимости, которое связывает два именованных элемента или наборы именованных элементов, представляющих одну и ту же концепцию, но на разных уровнях абстракции или с разных точек зрения. Отношение абстракции показано в виде стрелки отношения зависимости от клиента к поставщику (острие стрелки) с ключевым</p>






словом «abstraction». Derive – это стандартный стереотип абстракции, который используется для определения производного отношения между элементами модели, которые обычно, но не обязательно, относятся к одному типу. Этот вывод определяется в UML как «клиент может быть вычислен от поставщика». Причиной появления такого «вычисляемого клиента» может быть эффективность реализации.



Refine – это стандартный стереотип абстракции, который используется для определения отношения уточнения между элементами модели на разных семантических уровнях, таких как анализ, проектирование и реализация. Его можно использовать для моделирования преобразований от анализа к дизайну, от дизайна к реализации и т.д. Отображение абстракции может быть вычислимым или нет, и оно может быть однонаправленным или двунаправленным. В примере ниже класс клиента из модели Design уточняет класс клиента из модели Analysis.



Trace – это стандартный стереотип абстракции, который в основном используется для отслеживания требований и изменений в моделях для элементов или наборов элементов, которые представляют одну и ту же концепцию в разных моделях. Таким образом, Trace представляет собой «межмодельное» отношение. Вот несколько примеров использования Trace: 1) Use Case в модели вариантов использования может указывать на сотрудничество или пакет в соответствующей модели проекта; 2) Интерфейсы и классы из модели проектирования могут прослеживаться до компонентов в модели реализации; 3) Компоненты в модели реализации могут отслеживать артефакты в модели развертывания.

	<i>Collaboration</i>	Создает сотрудничество
	<i>Package</i> (<i>Model</i> , <i>Subsystem</i>)	Создается пакет (или модель, или подсистема)
	<i>Note</i>	Создает заметку
	<i>Anchor</i>	Создает линию, описывающую связь с заметкой
	<i>Constraint</i>	Создает ограничение

	<i>Containment</i>	Создает связь вложенности
--	--------------------	---------------------------

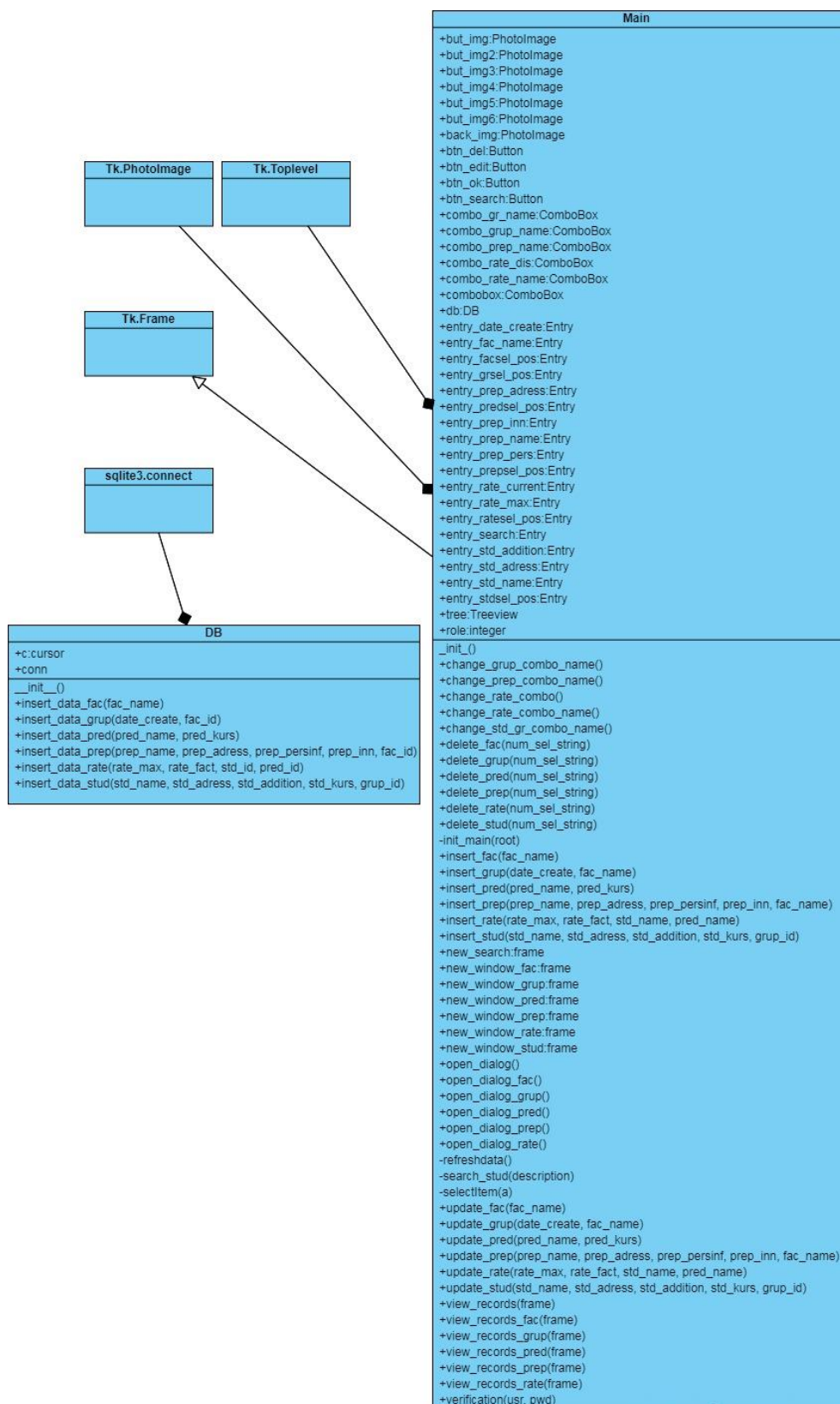


Рисунок 3.6 – Готовая диаграмма классов для проекта «Учёба в вузе».

Задание

Необходимо создать диаграмму, следуя описанным принципам работы с системой. Объектом автоматизации является тот же процесс, что и в предыдущих работах.

Лабораторная работа №4**Цель работы**

Целью данной работы является построение диаграммы последовательности.

Краткое изложение теоретической части

На диаграмме последовательности (*Sequence Diagram*) изображаются только те объекты, которые непосредственно участвуют во взаимодействии. Ключевым моментом для диаграмм последовательности является динамика взаимодействия объектов во времени.

В UML диаграмма последовательности имеет как бы два измерения. Первое слева направо в виде вертикальных линий, каждая из которых изображает линию жизни отдельного объекта, участвующего во взаимодействии. Крайним слева на диаграмме изображается объект, который является инициатором взаимодействия. Правее изображается другой объект, который непосредственно взаимодействует с первым. Таким образом, все объекты на диаграмме последовательности образуют некоторый порядок, определяемый очередностью или степенью активности объектов при взаимодействии друг с другом. Графически каждый объект изображается прямоугольником и располагается в верхней части своей линии жизни. Внутри прямоугольника записываются имя объекта и имя класса, разделенные двоеточием.

Вторым измерением диаграммы последовательности является вертикальная временная ось, направленная сверху вниз. Начальному моменту времени соответствует самая верхняя часть диаграммы. Взаимодействия объектов реализуются посредством сообщений, которые посылаются одними объектами другим в рамках одного прецедента (use case). Сообщения изображаются в виде горизонтальных стрелок с именем сообщения, а их порядок определяется временем возникновения. То есть, сообщения, расположенные на диаграмме последовательности выше, инициируются раньше тех, которые расположены ниже. Масштаб на оси времени не указывается, поскольку диаграмма последовательности моделирует лишь временную упорядоченность взаимодействий типа «раньше-позже».

Диаграммы последовательности обязательно опираются на диаграмму классов. Объекты должны принадлежать классам, описанным в диаграмме классов. Если создаваемый объект не может принадлежать ни одному из существующих классов, возможно, следует доработать диаграмму классов. Поэтому, в диаграммах последовательности нет необходимости создавать новые объекты, их достаточно просто перетащить в рабочую область текущей диаграммы из соответствующей диаграммы классов и определить имя экземпляра данного класса (рис. 4.1-4.3).

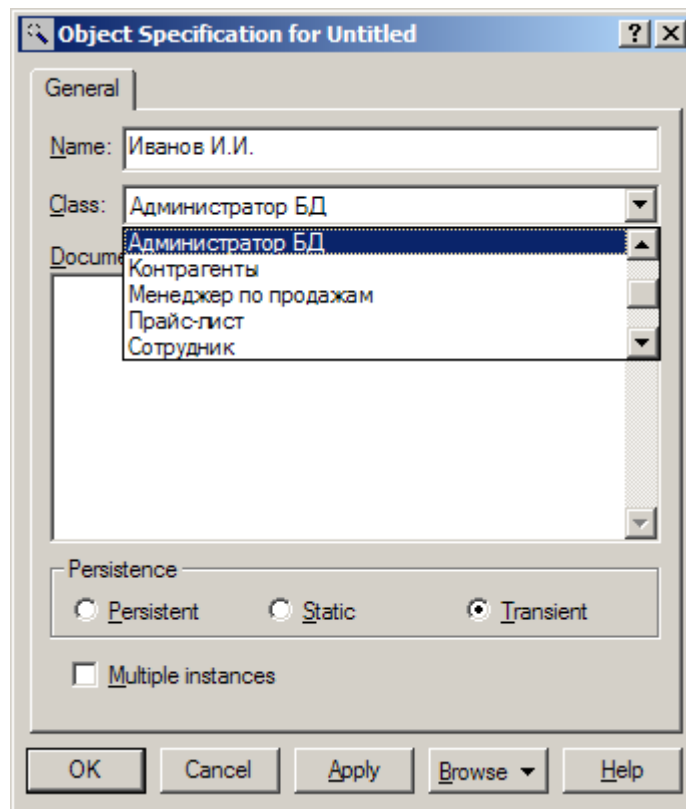


Рисунок 4.1 – Окно свойств класса диаграммы последовательности
Rational Rose

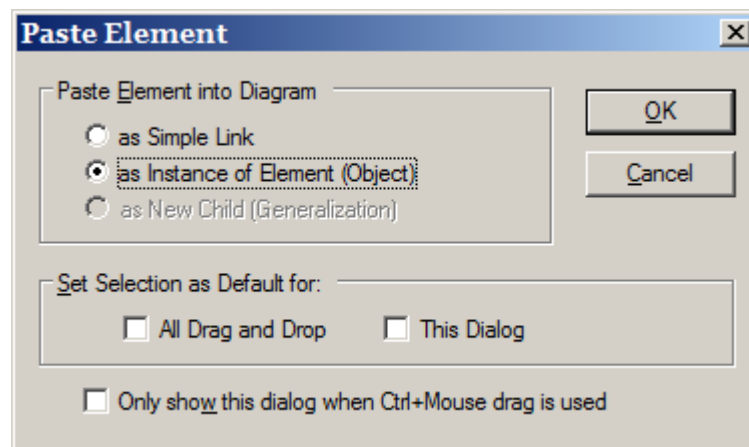


Рисунок 4.2 – Создание экземпляра класса в Enterprise Architect

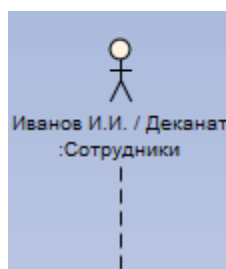


Рисунок 4.3 – Экземпляр класса Сотрудники с заданной ролью «Деканат»

Задать объекты в Rational Rose можно с помощью пиктограммы *Object*



Линия жизни объекта

Линия жизни объекта (object lifeline) изображается пунктирной вертикальной линией (рис.4.3), ассоциированной с единственным объектом на диаграмме последовательности. Линия жизни служит для обозначения периода времени, в течение которого объект существует в системе и, следовательно, может потенциально участвовать во всех ее взаимодействиях. Если объект существует в системе постоянно, то и его линия жизни должна продолжаться по всей плоскости диаграммы последовательности от самой верхней ее части до самой нижней.

Каждый объект графически изображается в форме прямоугольника и располагается в верхней части своей линии жизни. Внутри прямоугольника записываются собственное имя объекта со строчной буквы и имя класса, разделенные двоеточием. При этом вся запись подчеркивается, что является признаком объекта, который, как указывалось ранее, представляет собой экземпляр класса. Если на диаграмме последовательности отсутствует собственное имя объекта, то при этом должно быть указано имя класса. Такой объект считается анонимным. Может отсутствовать и имя класса, но при этом должно быть указано собственное имя объекта. Такой объект считается сиротой. Роль классов в именах объектов на диаграммах последовательности, как правило, не указывается.

Отдельные объекты, выполнив свою роль в системе, могут быть уничтожены, чтобы освободить занимаемые ими ресурсы. Для таких объектов линия жизни обрывается в момент его уничтожения. Для обозначения момента уничтожения объекта в языке UML используется специальный символ в форме латинской буквы «X». Не обязательно

создавать все объекты диаграммы в начальный момент времени. Отдельные объекты могут создаваться по мере необходимости, экономя ресурсы системы и повышая ее производительность. В этом случае прямоугольник объекта изображается не в верхней части диаграммы последовательности, а в той части, которая соответствует моменту создания объекта. При этом прямоугольник объекта вертикально располагается в том месте диаграммы, которое по оси времени совпадает с моментом его возникновения в системе. Объект обязательно создается со своей линией жизни и, возможно, с фокусом управления. Установка признака окончания жизни осуществляется в разных инструментальных средствах по-разному: например, в Enterprise Architect через задание признака в свойствах сообщения, а в Visual Paradigm – через установку чекбокса «Stopped» на вкладке «Style» в свойствах линии жизни.

Фокус управления

В процессе функционирования объектно-ориентированных систем одни объекты могут находиться в активном состоянии, непосредственно выполняя определенные действия, или состоянии пассивного ожидания сообщений от других объектов. Чтобы явно выделить подобную активность объектов, в языке UML применяется специальное понятие, получившее название фокуса управления (focus of control). Фокус управления изображается в форме вытянутого узкого прямоугольника, верхняя сторона которого обозначает начало получения фокуса управления объектом (начало активности), а его нижняя сторона - окончание фокуса управления (окончание активности). Прямоугольник располагается ниже обозначения соответствующего объекта и может заменять его линию жизни, если на всем ее протяжении он является активным.

Периоды активности объекта могут чередоваться с периодами его пассивности или ожидания. В этом случае у такого объекта имеются несколько фокусов управления. Важно сознавать, что получить фокус

управления может только существующий объект, у которого в этот момент имеется линия жизни. Если же некоторый объект был уничтожен, то вновь возникнуть в системе он уже не может. Вместо него лишь может быть создан другой экземпляр этого же класса, который, строго говоря, будет являться другим объектом.

В отдельных случаях инициатором взаимодействия в системе может быть актер или внешний пользователь. В этом случае актер изображается на диаграмме последовательности самым первым объектом слева со своим фокусом управления. Чаще всего актер и его фокус управления будут существовать в системе постоянно, отмечая характерную для пользователя активность в инициировании взаимодействий с системой. При этом актер может иметь собственное имя или оставаться анонимным.

Иногда некоторый объект может инициировать рекурсивное взаимодействие с самим собой. Наличие во многих языках программирования специальных средств построения рекурсивных процедур требует визуализации соответствующих понятий в форме графических примитивов. На диаграмме последовательности рекурсия обозначается небольшим прямоугольником, присоединенным к правой стороне фокуса управления того объекта, для которого изображается это рекурсивное взаимодействие.

Сообщения

В UML каждое взаимодействие описывается совокупностью сообщений, которыми участвующие в нем объекты обмениваются между собой. Сообщение (message) представляет собой законченный фрагмент информации, который отправляется одним объектом другому. Прием сообщения инициирует выполнение определенных действий, направленных на решение отдельной задачи тем объектом, которому это сообщение отправлено.

Таким образом, сообщения не только передают некоторую информацию, но и требуют или предполагают выполнения ожидаемых действий от принимающего объекта. Сообщения могут инициировать выполнение операций объектом соответствующего класса, а параметры этих операций передаются вместе с сообщением. На диаграмме последовательности все сообщения упорядочены по времени своего возникновения в моделируемой системе. В таком контексте каждое сообщение имеет направление от объекта, который инициирует и отправляет сообщение, к объекту, который его получает. Иногда отправителя сообщения называют клиентом, а получателя сервером. Тогда сообщение от клиента имеет форму запроса некоторого сервиса, а реакция сервера на запрос после получения сообщения может быть связана с выполнением определенных действий или передачи клиенту необходимой информации тоже в форме сообщения.

Сообщения изображаются горизонтальными стрелками, соединяющими линии жизни или фокусы управления двух объектов на диаграмме последовательности. В языке UML различаются несколько разновидностей сообщений, каждое из которых имеет свое графическое изображение:

- первая разновидность сообщения является наиболее распространенной и используется для вызова процедур, выполнения операций или обозначения отдельных вложенных потоков управления. Начало этой стрелки всегда соприкасается с фокусом управления или линией жизни того объекта-клиента, который инициирует это сообщение. Конец стрелки соприкасается с линией жизни того объекта, который принимает это сообщение и выполняет в ответ определенные действия. Принимающий объект, как правило, получает фокус управления, становясь активным;
- вторая разновидность сообщения используется для обозначения простого потока управления. Каждая такая стрелка указывает на

выполнение одного шага потока. Такие сообщения, обычно, являются асинхронными, то есть могут возникать в произвольные моменты времени. Передача такого сообщения, как правило, сопровождается получением фокуса управления принявшим его объектом;

- третья разновидность явно обозначает асинхронное сообщение между двумя объектами в некоторой процедурной последовательности. Примером такого сообщения может служить прерывание операции при возникновении исключительной ситуации. В этом случае информация о такой ситуации передается вызывающему объекту для продолжения процесса дальнейшего взаимодействия;
- четвертая разновидность сообщения используется для возврата из вызова процедуры. Примером может служить простое сообщение о завершении некоторых вычислений без предоставления результата расчетов объекту-клиенту. В процедурных потоках управления эта стрелка может быть опущена, поскольку ее наличие неявно предполагается в конце активизации объекта. Считается, что каждый вызов процедуры имеет свою пару - возврат вызова. Для непроцедурных потоков управления, включая параллельные и асинхронные сообщения, стрелка возврата должна указываться явным образом.

Предполагается, что время передачи сообщения достаточно мало по сравнению с процессами выполнения действий объектами, то есть, за время передачи сообщения с соответствующими объектами не может произойти никаких изменений. Если же это предположение не может быть признано справедливым, то стрелка сообщения изображается под некоторым наклоном, так чтобы конец стрелки располагался ниже ее начала.

В отдельных случаях объект может посылать сообщения самому себе, иницируя так называемые рефлексивные сообщения. Такие сообщения изображаются прямоугольником со стрелкой, начало и конец которой совпадают. Подобные ситуации возникают, например, при обработке

нажатий на клавиши клавиатуры при вводе текста в редактируемый документ, при наборе цифр номера телефона абонента.

Таким образом, в языке UML каждое сообщение ассоциируется с некоторым действием, которое должно быть выполнено принявшим его объектом. Действие может иметь некоторые аргументы или параметры, в зависимости от конкретных значений которых может быть получен различный результат. Соответствующие параметры будет иметь и вызывающее это действие сообщение. Более того, значения параметров отдельных сообщений могут содержать условные выражения, образуя ветвление или альтернативные пути основного потока управления.

В Rational Rose сообщения задаются через значок *Message* ()

Ветвление потока управления

Для изображения ветвления потока управления рисуются две или более стрелки, выходящие из одной точки фокуса управления объекта. При этом соответствующие условия должны быть явно указаны рядом с каждой из стрелок в форме сторожевого условия. Сторожевые условия должны взаимно исключать одновременную передачу альтернативных сообщений.

Стереотипы сообщений

В языке UML предусмотрены некоторые стандартные действия, выполняемые в ответ на получение соответствующего сообщения. Эти действия могут быть явно указаны на диаграмме последовательности в форме стереотипа рядом с сообщением, к которому относятся. В этом случае они записываются в кавычках. Используются следующие стереотипы сообщений:

«**call**» (**вызвать**) – сообщение, требующее вызова операции или процедуры принимающего объекта. Если сообщение с этим стереотипом рефлексивное, то оно инициирует локальный вызов операции у самого пославшего это сообщение объекта;

«return» (возвратить) – сообщение, возвращающее значение выполненной операции или процедуры вызвавшему ее объекту. Значение результата может инициировать ветвление потока управления;

«create» (создать) – сообщение, требующее создания другого объекта для выполнения определенных действий. Созданный объект может получить фокус управления, а может и не получить его;








«destroy» (уничтожить) – сообщение с явным требованием уничтожить соответствующий объект. Посылается в том случае, когда необходимо прекратить нежелательные действия со стороны существующего в системе объекта, либо когда объект больше не нужен и должен освободить задействованные им системные ресурсы;

«send» (послать) – обозначает посылку другому объекту некоторого сигнала, который асинхронно инициируется одним объектом и принимается другим. Отличие сигнала от сообщения заключается в том, что сигнал должен быть явно описан в том классе, объект которого инициирует его передачу.

Кроме стереотипов, сообщения могут иметь собственное обозначение операции, вызов которой они инициируют у принимающего объекта. В этом случае рядом со стрелкой записывается имя операции с круглыми скобками, в которых могут указываться параметры или аргументы соответствующей операции. Если параметры отсутствуют, то скобки все равно должны присутствовать после имени операции. Примерами таких операций могут служить следующие: *«Выдать клиенту наличными сумму (n)»*, *«Установить соединение между абонентами (a, b)»*, *«Сделать вводимый текст невидимым ()»*, *«Подать звуковой сигнал тревоги ()»*.

Определение свойств сообщений в Rational Rose имеет некоторую специфику. Способ синхронизации можно задать в группе свойств *Synchronization* (таб. 4.1).

Таблица 4.1 – Характеристика свойств синхронизации сообщений

Название свойства	Графическое изображение стрелки	Назначение свойства
<i>Simple</i> (Простое)		Данное сообщение выполняется в одном потоке управления. Это свойство задается добавляемому на диаграмму сообщению по умолчанию
<i>Synchronous</i> (Синхронное)		После передачи данного сообщения клиент ожидает ответа от объекта-приемника о результате выполнения соответствующей операции
<i>Balking</i> (С отказом)		После передачи данного сообщения объект-приемник отказывает клиенту в выполнении соответствующей операции, если он занят выполнением других операций
<i>Timeout</i> (С ожиданием)		После передачи данного сообщения объект-приемник может поместить данное сообщение в очередь с ограниченным временем ожидания, если он занят выполнением других операций
<i>Procedure Call</i> (Вызов процедуры)		Клиент посылает данное сообщение объекту-приемнику и, чтобы продолжить свою работу ожидает, пока вся дальнейшая вложенная последовательность сообщений не будет обработана приемником
<i>Asynchronous</i> (Асинхронное)		Клиент посылает данное сообщение и продолжает свою работу, не ожидая подтверждения от объекта-приемника о получении этого сообщения. При этом соответствующая операция может быть как выполнена, так и не выполнена
<i>Return</i> (Возврат)		Данное сообщение посылается клиенту после окончания выполнения вызова процедуры

Временные ограничения на диаграммах последовательности

В отдельных случаях выполнение тех или иных действий на диаграмме последовательности может потребовать явной спецификации временных ограничений, накладываемых на сам интервал выполнения операций или передачу сообщений. В языке UML для записи временных ограничений используются фигурные скобки. Временные ограничения

могут относиться как к выполнению определенных действий объектами, так и к самим сообщениям, явно специфицируя условия их передачи или приема. В отличие от условий ветвления, которые должны выполняться альтернативно, временные ограничения имеют обязательный или директивный характер для ассоциированных с ними объектов.

Временные ограничения могут записываться рядом с началом стрелки соответствующего сообщения. Но наиболее часто они записываются слева от стрелки на одном уровне с ней. Если временная характеристика относится к конкретному объекту, то имя этого объекта записывается перед именем характеристики и отделяется от нее точкой.

Примерами таких ограничений на диаграмме последовательности могут служить ситуации, когда необходимо явно специфицировать время, в течение которого допускается передача сообщения от клиента к серверу или обработка запроса клиента сервером:

```
{время_приема_сообщения время_отправки_сообщения < 1 сек.};
{время_ожидания_ответа < 5 сек.};
{время_передачи_пакета < 10 сек.};
{объект_1. время_подачи_сигнала_тревоги > 30 сек.}.
```

Рекомендации по построению диаграмм последовательности

Построение диаграммы последовательности целесообразно начинать с выделения из всей совокупности классов только тех, объекты которых участвуют в моделируемом взаимодействии. После этого все объекты наносятся на диаграмму, с соблюдением порядка инициализации сообщений. Здесь необходимо установить, какие объекты будут существовать постоянно, а какие временно - только на период выполнения ими требуемых действий.

Когда объекты визуализированы, можно приступать к спецификации сообщений. При этом необходимо учитывать те операции, которые имеют классы соответствующих объектов в модели системы. При необходимости уточнения этих операций следует использовать их стереотипы. Для

уничтожения объектов, которые создаются на время выполнения своих действий, нужно предусмотреть явное сообщение. Наиболее простые случаи ветвления процесса взаимодействия можно изобразить на одной диаграмме с использованием соответствующих графических примитивов. В более сложных случаях для моделирования каждой ветви управления может потребоваться отдельная диаграмма последовательности. Следует помнить, что каждый альтернативный поток управления затрудняет понимание построенной модели.

Общим правилом является визуализация особенностей реализации каждого варианта использования на отдельной диаграмме последовательности. В этой ситуации отдельные диаграммы должны рассматриваться совместно как одна модель взаимодействия. Необходимость синхронизации сложных потоков управления, как правило, требуют введение в модель дополнительных ограничений. При этом общая запись таких ограничений должна следовать семантике языка объектных ограничений OCL.

Пример выполнения задания

Прежде всего, при создании диаграмм взаимодействия необходимо определить сценарии, которые будут отображены на диаграммах. Для каждого сценария составляется описание, отражающее ситуацию, которую будет описывать диаграмма. При составлении описания сценария, следует придерживаться следующих рекомендаций:

- Сценарий должен отображать вполне конкретную, реальную ситуацию работы с системой. Следует по возможности исключать абстрактные понятия и альтернативные варианты развития событий. Например, сценарий *«Администратор входит в систему»* будет изначально ошибочным, так как оперирует абстрактным понятием *«Администратор»* и неконкретным развитием событий *«Входит в систему»* (вход может быть успешным и неуспешным). Вместо

абстрактного понятия *Администратор* следует взять конкретного пользователя системы (например, *Преподаватель Иванов И.И.*) и рассмотреть конкретную последовательность действий (например, *«Преподаватель Иванов И.И. вводит логин и пароль, получает права Администратор БД и успешно входит в систему»*).

- Так как диаграмма последовательности отображает конкретную последовательность событий, то для отображения других вариантов развития событий следует построить несколько диаграмм. Например, в дополнение сценарию *«Преподаватель Иванов И.И. вводит логин и пароль, получает права Администратор БД и успешно входит в систему»* можно рассмотреть последовательность *«Преподаватель Иванов И.И. вводит логин и пароль, но ошибается и доступ в систему не получает»*.
- При выделении сценариев пользуйтесь диаграммой вариантов использования. Диаграммы взаимодействия обычно соответствуют одному или нескольким вариантам использования. Например, сценарий *«Преподаватель Иванов И.И. вводит логин и пароль, получает права Администратор БД и успешно входит в систему»* привязан к варианту использования *«Авторизация в системе»*.
- Старайтесь избегать избыточности диаграмм взаимодействия. Например, если в диаграмме *«Преподаватель Иванов И.И. вводит логин и пароль, получает права Администратор БД и успешно входит в систему»* вы описали последовательности входа в систему администратора, то в диаграмме *«Администратор БД Иванов И.И. вносит новую запись в справочник «Студент»»* последовательность входа можно уже не описывать. Старайтесь, чтобы диаграммы взаимодействия отражали наиболее сложные и неочевидные участки работы системы.

Рассмотрим простейший сценарий *«Преподаватель Иванов И.И. вводит логин и пароль, получает права Пользователь и успешно входит в систему»*.

Первым этапом создания диаграммы последовательности является выделение объектов-участников сценария. В данной диаграмме выделим два объекта:

- *Иванов И.И.: Пользователь;*
- *Main: форма ввода приложения.*

Следующим этапом создания диаграммы последовательности является определение событий, которыми обмениваются выделенные объекты. При создании событий помните, что подавляющее большинство событий будет являться либо методами классов-получателей, доступными классу-отправителю, либо воздействием на атрибуты класса, либо результатом выполнения какого-либо метода. Если при создании события вы считаете, что оно должно являться методом соответствующего класса и обнаруживаете, что этот метод отсутствует, следует добавить его для соответствующего класса в диаграмме классов.

На рис.4.3 отображена диаграмма последовательности для сценария *«Преподаватель Иванов И.И. вводит логин и пароль, получает права Пользователь и успешно входит в систему»*. В случае успешной проверки данных, заданных в соответствующих полях, внутри процедуры *verification*, элементы управления, отвечающие за процедуру аутентификации скрываются с формы, но выводится блок управления справочниками.

Также на рис.4.3. можно заметить блок альтернатив с пометкой «alt» в левом углу – таким образом мы на одной диаграмме отобразили альтернативный сценарий *«Преподаватель Иванов И.И. вводит логин и пароль, но ошибается и входа в систему не получает»*. В случае ошибки

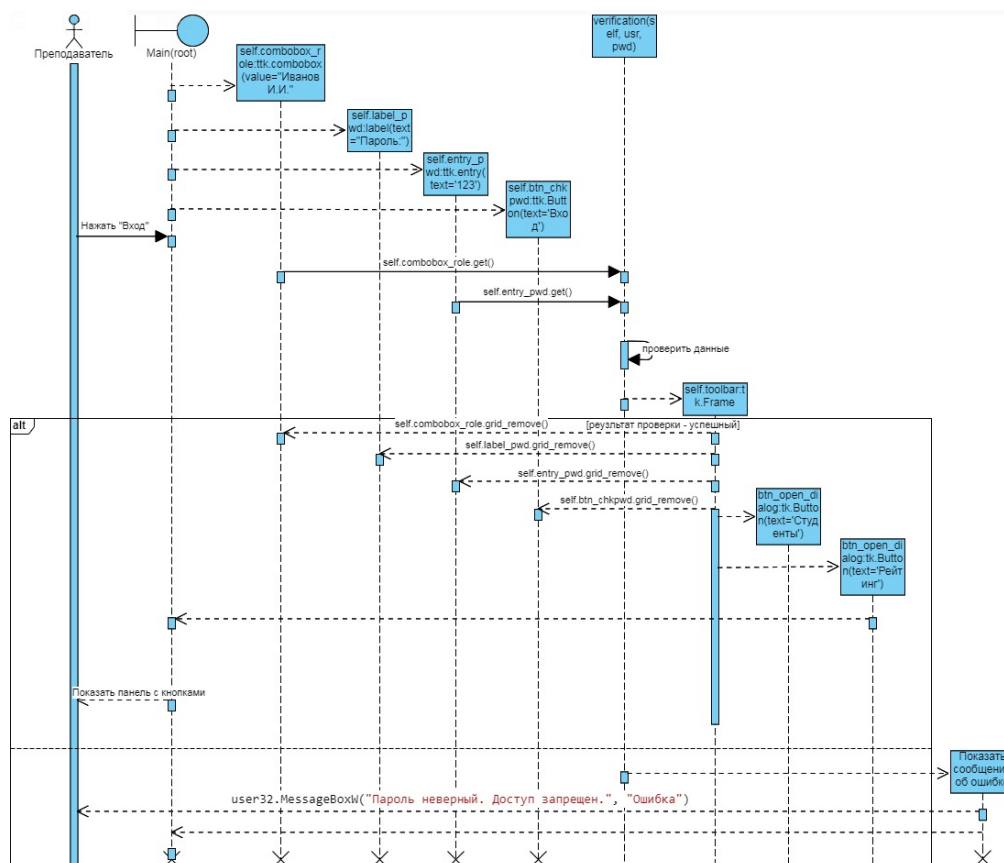


Рисунок 4.3 – Диаграмма последовательности «Процедура верификации»

Вводит логин и пароль – это изменение значений атрибутов *combobox_role* и *entry_pwd* класса *Main* на *Администратор БД* и *123* соответственно.

verification(Администратор БД, 123) – это вызов открытого (Public) метода класса *Main*.

Далее идёт проверка логина и пароля с помощью методов функции *verification()*, причём от результата выполнения функции зависит либо отображение панели с кнопками по работе со справочниками, либо вывод сообщения об ошибке.

В случае успешного завершения проверки передается сообщение *Показать панель с кнопками* – действие, отражающее тот факт, что Иванов И.И. зашел в систему с правами Пользователя и имеет возможность редактировать справочники «Студенты» и «Рейтинг».

Обратите внимание, что также, как и при создании диаграммы вариантов использования и диаграммы классов, в данном случае рассматривается только то, что входит в рамки проектируемой системы. События, происходящие вне нее (например, *Иванов И.И. записывает в бумажную тетрадь время входа в систему*, или *Электрик дядя Вася меняет пробки для того, чтобы восстановить подачу электроэнергии в корпус, обесточенный 5 минут назад*) не рассматриваются.

Задание

Создайте диаграмму последовательности на основании диаграммы классов, опираясь на логику, заложенную в диаграмму прецедентов использования.

Лабораторная работа №5

Цель работы

Целью данной работы является построение диаграмм активности (деятельности).

Краткое изложение теоретической части

– диаграмма, которая представляет конечный автомат.

Диаграмма состояний (statechart diagram) Семантика понятия состояния довольно сложна. Дело в том, что характеристика состояний системы явным образом не зависит от логической структуры, зафиксированной на диаграмме классов. При рассмотрении состояний системы приходится отвлекаться от особенностей ее объектной структуры и мыслить категориями, описывающими динамический контекст поведения моделируемой системы. При построении диаграмм состояний необходимо

использовать специальные понятия, о которых и пойдет речь в данной лекции.

Ранее отмечалось, что любая прикладная система характеризуется не только структурой составляющих ее элементов, но и некоторым поведением или функциональностью. Для общего представления функциональности моделируемой системы предназначены диаграммы вариантов использования, которые на концептуальном уровне описывают поведение системы в целом. Для того чтобы представить наиболее общее поведение на логическом уровне, следует ответить на вопрос: «В процессе какого поведения система реализует необходимую пользователям функциональность?».

Главное назначение диаграммы состояний - описать возможные последовательности состояний и переходов, которые в совокупности характеризуют поведение моделируемой системы в течение всего ее жизненного цикла. Диаграмма состояний представляет динамическое поведение сущностей, на основе спецификации их реакции на восприятие некоторых конкретных событий. Системы, которые реагируют на внешние действия от других систем или от пользователей, иногда называют **реактивными**. Если такие действия инициируются в произвольные случайные моменты времени, то говорят об асинхронном поведении модели.

Диаграммы состояний чаще всего используются для описания поведения отдельных систем и подсистем. Они также могут быть применены для спецификации функциональности экземпляров отдельных классов, т.е. для моделирования всех возможных изменений состояний конкретных объектов. Диаграмма состояний по существу является графом специального вида, который служит для представления конечного автомата.

Диаграммы состояний могут быть вложены друг в друга, образуя вложенные диаграммы для более детального представления состояний отдельных элементов модели. Для понимания семантики конкретной

диаграммы состояний необходимо представлять особенности поведения моделируемой сущности, а также иметь общие сведения из теории конечных автоматов.

Конечный автомат (state machine) – модель для спецификации поведения объекта в форме последовательности его состояний, которые описывают реакцию объекта на внешние события, выполнение объектом действий, а также изменение его отдельных свойств.

В контексте языка UML понятие конечного автомата обладает дополнительной семантикой. Вершинами графа конечного автомата являются состояния и другие типы элементов модели, которые изображаются соответствующими графическими символами. Дуги графа служат для обозначения переходов из состояния в состояние. Конечный автомат описывает поведение отдельного объекта в форме последовательности состояний, охватывающих все этапы его жизненного цикла, начиная от создания объекта и заканчивая его уничтожением. Каждая диаграмма состояний представляет собой конечный автомат.

Основными понятиями, характеризующими конечный автомат, являются состояние и переход. Ключевое различие между ними заключается в том, что длительность нахождения системы в отдельном состоянии существенно превышает время, которое затрачивается на переход из одного состояния в другое. Предполагается, что в пределе время перехода из одного состояния в другое равно нулю (если дополнительно ничего не сказано). Другими словами, переход объекта из состояния в состояние происходит мгновенно.

В общем случае конечный автомат представляет динамические аспекты моделируемой системы в виде ориентированного графа, вершины которого соответствуют состояниям, а дуги - переходам. При этом поведение моделируется как последовательное перемещение по графу состояний от вершины к вершине по связывающим их дугам с учетом их

ориентации. Для графа состояний системы можно ввести в рассмотрение специальные свойства.

Среди таких свойств - выделение из всей совокупности состояний двух специальных: начального и конечного. Ни в графе состояний, ни на диаграмме состояний время нахождения системы в том или ином состоянии явно не учитывается, однако предполагается, что последовательность изменения состояний упорядочена во времени. Другими словами, каждое последующее состояние может наступить позже предшествующего ему состояния.

Диаграммы деятельности

При моделировании поведения проектируемой или анализируемой системы возникает необходимость не только представить процесс изменения ее состояний, но и детализировать особенности алгоритмической и логической реализации выполняемых системой операций.

Для моделирования процесса выполнения операций в языке UML используются диаграммы деятельности (Activity diagram). Применяемая в них графическая нотация во многом похожа на нотацию диаграммы состояний (Statechart diagram), поскольку на этих диаграммах также присутствуют обозначения состояний и переходов. Каждое состояние на диаграмме деятельности соответствует выполнению некоторой элементарной операции, а переход в следующее состояние выполняется только при завершении этой операции.

Таким образом, диаграммы деятельности можно считать частным случаем диаграмм состояний. Они позволяют реализовать в языке UML особенности процедурного и синхронного управления, обусловленного завершением внутренних деятельностей и действий. Основным направлением использования диаграмм деятельности является визуализация особенностей реализации операций классов, когда необходимо представить алгоритмы их выполнения.

В контексте языка UML деятельность (activity) представляет собой совокупность отдельных вычислений, выполняемых автоматом, приводящих к некоторому результату или действию (action). Иными словами, activity – это процесс, а action – единичное действие. На диаграмме деятельности отображается логика и последовательность переходов от одной деятельности к другой, а внимание аналитика фокусируется на результатах. Результат деятельности может привести к изменению состояния системы или возвращению некоторого значения.

Диаграмма деятельности является графом специального вида, который представляет некоторый автомат. Вершинами графа являются возможные состояния автомата, изображаемые соответствующими графическими символами, а дуги обозначают его переходы из состояния в состояние. Диаграммы деятельности могут быть вложены друг в друга для более детального представления отдельных элементов модели.

Поведение автомата моделируется как последовательное перемещение по графу от вершины к вершине с учетом ориентации связывающих их дуг.

Для автомата должны выполняться следующие обязательные условия:

- состояние, в которое может перейти объект, определяется только его текущим состоянием и не зависит от предыстории;
- в каждый момент времени автомат может находиться только в одном из своих состояний. При этом, автомат может находиться в отдельном состоянии как угодно долго, если не происходит никаких событий;
- время нахождения автомата в том или ином состоянии, а также время достижения того или иного состояния никак не специфицируются;
- количество состояний автомата должно быть конечным и все они должны быть специфицированы явным образом. Отдельные псевдосостояния могут не иметь спецификаций (начальное и конечное состояния). В этом случае их назначение и семантика

полностью определяются из контекста модели и рассматриваемой диаграммы состояний;

- граф автомата не должен содержать изолированных состояний и переходов. Для каждого состояния, кроме начального, должно быть определено предшествующее состояние, а каждый переход должен соединять два состояния автомата;
- автомат не должен содержать конфликтующих переходов, когда объект одновременно может перейти в два и более последующих состояния (кроме случая параллельных подавтоматов рис.5.4). В языке UML исключение конфликтов возможно на основе введения сторожевых условий.

Понятие состояния (state) является фундаментальным не только в метамодели языка UML, но и в прикладном системном анализе. Вся концепция динамической системы основывается на понятии состояния.

Состояние (state) – условие или ситуация в ходе жизненного цикла объекта, в течение которого он удовлетворяет логическому условию, выполняет определенную деятельность или ожидает события.

Состояние на диаграмме изображается прямоугольником со скругленными вершинами. Под действием в языке UML понимают некоторую атомарную операцию, выполнение которой приводит к изменению состояния или возврату некоторого значения (например, «истина» или «ложь»).

Имя состояния представляет собой строку текста, которая раскрывает его содержательный смысл. Имя всегда записывается с заглавной буквы. Поскольку состояние системы является составной частью процесса ее функционирования, рекомендуется в качестве имени использовать глаголы в настоящем времени (звонит, печатает, ожидает) или соответствующие причастия (занят, свободен, передано, получено). Имя у состояния может отсутствовать и этом случае состояние является анонимным. Если на

диаграмме анонимных состояний несколько, то они должны различаться между собой.

Действие (action) – спецификация выполнимого утверждения, которая образует абстракцию вычислительной процедуры.

Действие обычно приводит к изменению состояния системы, и может быть реализовано посредством передачи сообщения объекту, модификацией связи или значения атрибута. Для ряда состояний может потребоваться дополнительно указать действия, которые должны быть выполнены моделируемым элементом. Для этой цели служит добавочная секция в обозначении состояния, содержащая перечень внутренних действий или деятельность, которые производятся в процессе нахождения моделируемого элемента в данном состоянии. Каждое действие записывается в виде отдельной строки и имеет следующий формат:

<метка действия '/' выражение действия>

Метка действия указывает на обстоятельства или условия, при которых будет выполняться деятельность, определенная выражением действия. При этом выражение действия может использовать любые атрибуты и связи, принадлежащие области имен или контексту моделируемого объекта. Если список выражений действия пустой, то метка действия с разделителем в виде наклонной черты '/' не указывается. Перечень меток действий в языке UML фиксирован, причем эти метки не могут быть использованы в качестве имен событий:

Входное действие (entry action) – действие, которое выполняется в момент перехода в данное состояние.

Обозначается с помощью ключевого слова – метки действия entry, которое указывает на то, что следующее за ней выражение действия должно быть выполнено в момент входа в данное состояние.

Действие выхода (exit action) – действие, производимое при выходе из данного состояния.

Обозначается с помощью ключевого слова – метки действия `exit`, которое указывает на то, что следующее за ней выражение действия должно быть выполнено в момент выхода из данного состояния.

Внутренняя деятельность (`do activity`) – выполнение объектом операций или процедур, которые требуют определенного времени.

Обозначается с помощью ключевого слова - метки деятельности `do`, которое специфицирует так называемую «ду-деятельность», выполняемую в течение всего времени, пока объект находится в данном состоянии, или до тех пор, пока не будет прервано внешним событием. При нормальном завершении внутренней деятельности генерируется соответствующее событие.

Во всех остальных случаях метка действия идентифицирует событие, которое запускает соответствующее выражение действия. Эти события называются внутренними переходами. Семантически они эквивалентны переходам в само это состояние, за исключением той особенности, что выход из этого состояния или повторный вход в него не происходит. Это означает, что действия входа и выхода не производятся. При этом выполнение внутренних действий в состоянии не может быть прервано никакими внешними событиями, в отличие от внутренней деятельности, выполнение которой требует определенного времени.

Действие может быть записано на естественном языке, некотором псевдокоде или языке программирования. Никаких дополнительных или неявных ограничений при записи действий не накладывается. Рекомендуется в качестве имени простого действия использовать глагол с пояснительными словами. Если же действие может быть представлено в некотором формальном виде, то целесообразно записать его на том языке программирования, на котором предполагается реализовывать конкретный проект.

Иногда возникает необходимость представить на диаграмме деятельности некоторое сложное действие, которое, в свою очередь, состоит

из нескольких более простых действий. В этом случае можно использовать специальное обозначение **состояния под-деятельности** (subactivity state). Такое состояние является самостоятельным графом деятельности и обозначается специальной пиктограммой в правом нижнем углу символа состояния действия (рис. 5.1). Составное состояние называют также состоянием-композитом. Эта конструкция может применяться к любому элементу языка UML, который поддерживает вложенность своей структуры. При этом пиктограмма может быть дополнительно помечена типом вложенной структуры.

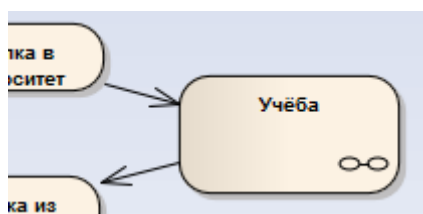


Рисунок 5.1 – Состояние под-деятельности «Учёба»

Составное состояние может содержать или несколько последовательных подсостояний, или несколько параллельных конечных подавтоматов. Каждое состояние-композиит может уточняться только одним из указанных способов. При этом любое из подсостояний, в свою очередь, может быть состоянием-композиитом и содержать внутри себя другие вложенные подсостояния. Количество уровней вложенности составных состояний в языке UML не фиксировано.

Начальное состояние представляет собой частный случай состояния, которое не содержит никаких внутренних действий (псевдосостояние). В этом состоянии находится объект по умолчанию в начальный момент времени. Оно служит для указания на диаграмме графической области, от которой начинается процесс изменения состояний. Графически начальное состояние в языке UML обозначается в виде закрашенного кружка, из которого может только выходить стрелка, соответствующая переходу.

На самом верхнем уровне представления объекта переход из начального состояния может быть помечен событием создания

(инициализации) данного объекта. В противном случае переход никак не помечается. Если этот переход не помечен, то он является первым переходом в следующее за ним состояние.

Конечное состояние представляет собой частный случай состояния, которое также не содержит никаких внутренних действий (псевдосостояние). В этом состоянии будет находиться объект по умолчанию после завершения работы автомата в конечный момент времени. Оно служит для указания на диаграмме графической области, в которой завершается процесс изменения состояний или жизненный цикл данного объекта. Графически конечное состояние в языке UML обозначается в виде закрашенного кружка, помещенного в окружность, которую может только входить стрелка, соответствующая переходу.

Каждая диаграмма деятельности должна иметь единственное начальное и единственное конечное состояния. Они имеют такие же обозначения, как и на диаграмме состояний. При этом каждая деятельность начинается в начальном состоянии и заканчивается в конечном состоянии (рис.5.2).

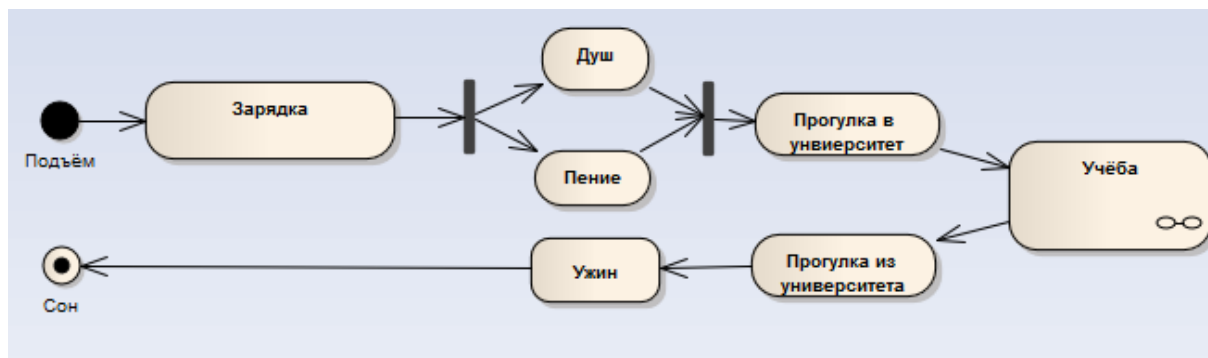


Рисунок 5.2 – Диаграмма деятельности «День студента»

Простой переход (simple transition) представляет собой отношение между двумя последовательными состояниями, которое указывает на факт смены одного состояния объекта другим. Если пребывание моделируемого объекта в первом состоянии сопровождается выполнением некоторых действий, то переход во второе состояние будет возможен только после

завершения этих действий и, возможно, после выполнения некоторых дополнительных условий, называемых сторожевыми условиями.

На диаграмме состояний переход изображается сплошной линией со стрелкой, которая направлена в целевое состояние. Каждый переход может быть помечен строкой текста, которая имеет следующий общий формат:

<сигнатура события> '['<сторожевое условие>']' <выражение действия>.

Сторожевое условие (guard condition), если оно есть, всегда записывается в прямых скобках после события-триггера и представляет собой некоторое булевское выражение. Из контекста диаграммы состояний должна явно следовать семантика этого выражения, а для записи выражения может использоваться синтаксис языка объектных ограничений.

Введение для перехода сторожевого условия позволяет явно специфицировать семантику его срабатывания. Однако вычисление истинности сторожевого условия происходит только после возникновения, ассоциированного с ним события-триггера (рис.5.3), инициирующего соответствующий переход.

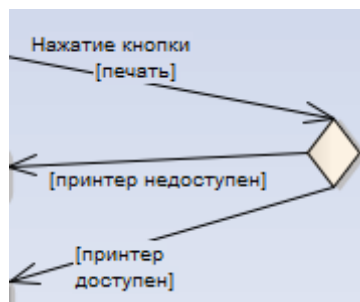


Рисунок 5.3 – Событие-триггер «Доступен ли принтер?»

В общем случае из одного состояния может быть несколько переходов с одним и тем же событием-триггером. При этом никакие два сторожевых условия не должны одновременно принимать значение «истина». Каждое из сторожевых условий необходимо вычислять всякий раз при наступлении соответствующего события-триггера.

Переход называется триггерным, если его специфицирует событие-триггер, связанное с внешними условиями по отношению к рассматриваемому состоянию.

В этом случае рядом со стрелкой триггерного перехода обязательно указывается имя события в форме строки текста, начинающейся со строчной буквы. Наиболее часто в качестве имен триггерных переходов задают имена операций, вызываемых у тех или иных объектов системы. После имени такого события следуют круглые скобки для явного задания параметров соответствующей операции. Если таких параметров нет, то список параметров со скобками может отсутствовать. Например, переход на рис. 5.3, является триггерным, поскольку с ним связано конкретное событие-триггер.

Переход называется нетриггерным, если он происходит по завершении выполнения деятельности в данном состоянии.

Примером события-триггера может служить, например, автоматическое отключение компьютера по окончании загрузки фильма (если это задано в настройках закачки). В этом случае сторожевое условие есть не что иное, как ответ на вопрос: «Закончилось ли скачивание?». В случае положительного ответа – «истина», следует выключить компьютер. В случае отрицательного ответа – «ложь», следует оставаться в состоянии загрузки и ждать, пока не скачается 100%.

Выражение действия (action expression) выполняется только при срабатывании перехода. Оно представляет собой атомарную операцию, выполняемую сразу после срабатывания соответствующего перехода до начала каких бы то ни было действий в целевом состоянии. Атомарность действия означает, что оно не может быть прервано никаким другим действием до тех пор, пока не закончится его выполнение. Данное действие может влиять как на сам объект, так и на его окружение, если это с очевидностью следует из контекста модели.

В общем случае выражение действия может содержать целый список отдельных действий, разделенных символом «;». Все действия списка должны различаться между собой и следовать в порядке записи. На синтаксис записи выражений действия не накладывается никаких ограничений. Основное условие, чтобы запись была понятна разработчикам модели и программистам. Как правило, выражение действия записывают на одном из языков программирования, который предполагается использовать для реализации модели.

Один из недостатков обычных блок-схем алгоритмов связан с проблемой изображения параллельных ветвей отдельных вычислений. Поскольку распараллеливание вычислений существенно повышает общее быстродействие программных систем, необходимы графические примитивы для представления параллельных процессов. В языке UML для этой цели используется специальный символ для разделения и слияния параллельных вычислений или потоков управления. Таким символом является прямая черточка (рис.5.4). Как правило, такая черточка изображается отрезком горизонтальной/вертикальной линии, толщина которой несколько шире основных сплошных линий диаграммы деятельности. При этом **разделение** (concurrent fork) имеет один входящий переход и несколько выходящих, а **слияние** (concurrent join) имеет несколько входящих переходов и один выходящий.

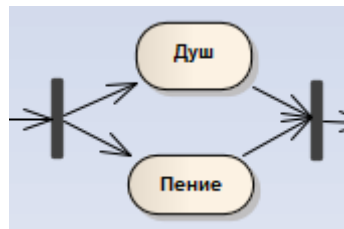


Рисунок 5.4 – Пример случая параллельных процессов

В случае, если переход из состояния в состояние сопровождается возникновением некоторой **исключительной ситуации** (exception), уместно использовать специальный блок обработки, который графически

обозначается также как и обычный блок состояния, но со специально обозначенной точкой входа (рис.5.5). Для обозначения факта прерывания потока данных используется специальная связь разрыв потока (interrupt flow).

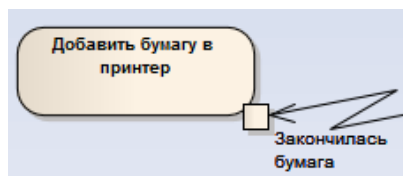


Рисунок 5.5 – Исключительная ситуация «Закончилась бумага»

Пример выполнения задания

Разработка диаграмм деятельности не представляет особого труда, поскольку отражает логику выполнения процесса. Ближайшим аналогом для диаграмм деятельности являются блок-схемы, логику их функционирования и построения регламентирует ГОСТ 19.701-9.

Для примера рассмотрим диаграмму «Удаление записи из справочника «Предметы» (рис.5.6). Необходимо предусмотреть функционал по защите от удаления записей, используемых в других справочниках – в данном случае, конкретный предмет может использоваться в таблице рейтинга.

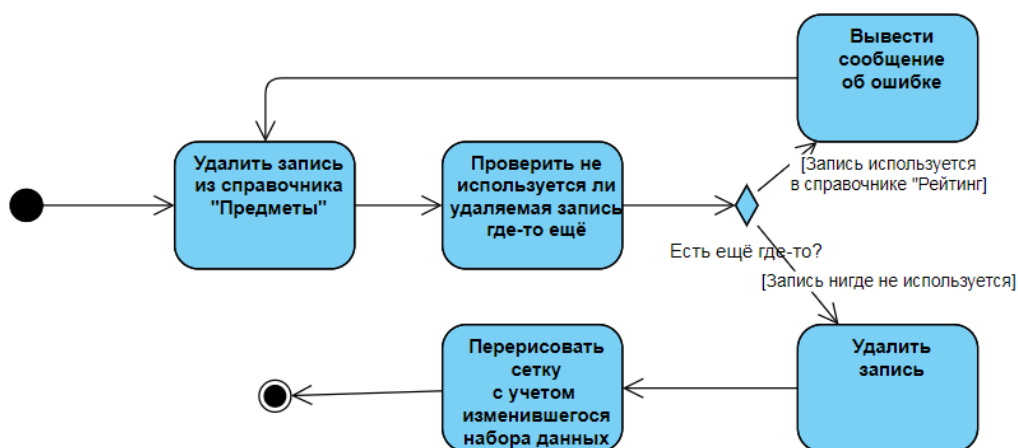


Рисунок 5.6 – Диаграмма деятельности «Удаление записи из справочника «Предметы»

Также приведем диаграмму «Вход в систему» (рис.5.7), описывающую следующую логику: пользователь выбирает учетную запись, вводит пароль и пытается получить доступ в систему – в зависимости от

результата он либо получает доступ к таблицам (вариант отображения зависит от категории пользователя), либо получает сообщение об ошибке и приложение закрывается.

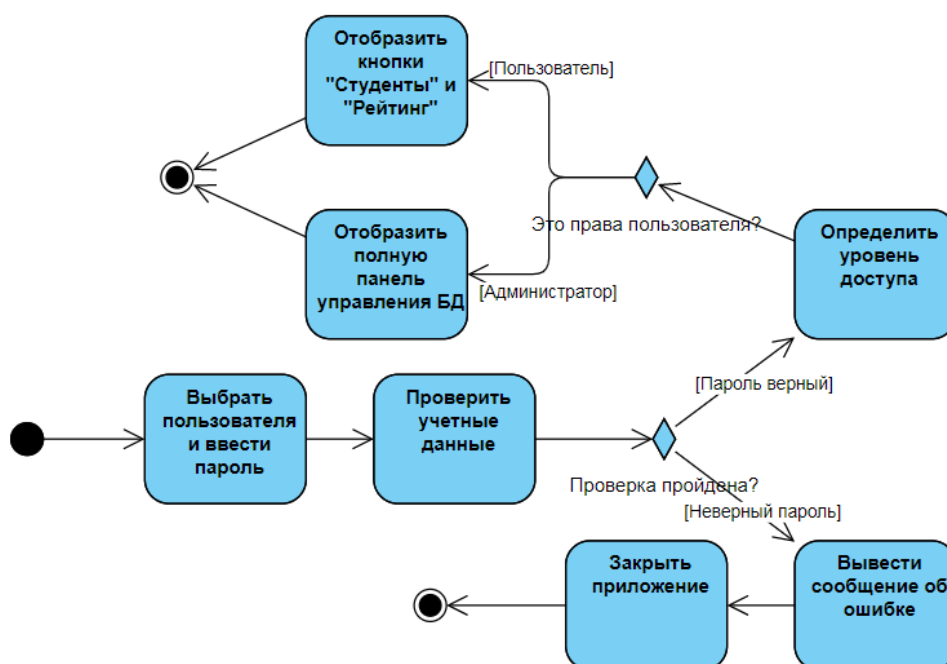


Рисунок 5.7 – Диаграмма деятельности «Вход в систему»

Задание

Разработать диаграммы деятельности для выбранной предметной области. Общее количество блоков «Activity» во всех диаграммах не менее 15.

Лабораторная работа №6

Цель работы

Целью данной работы является построение структурной схемы базы данных. Требуется создать полную атрибутивную модель, т.е. модель, содержащую все сущности в третьей нормальной форме (3НФ) со всеми атрибутами и связями.

Краткое изложение теоретической части

Бурное развитие информационных технологий в начале XXI века и переход основных сфер деятельности из безусловного «офлайн-режима» в той или иной степени «онлайн», подстегнуло развитие интернет-торговли: а где есть потребность в торговле, есть и потребность в накоплении информации. Это обусловило появление большого количества средств визуального проектирования баз данных, которых условно можно разделить на автономные и онлайн-средства. К первым относится, например, ERwin Data Modeler, ко вторым: Lucidchart, Creately, Visual Paradigm и др.

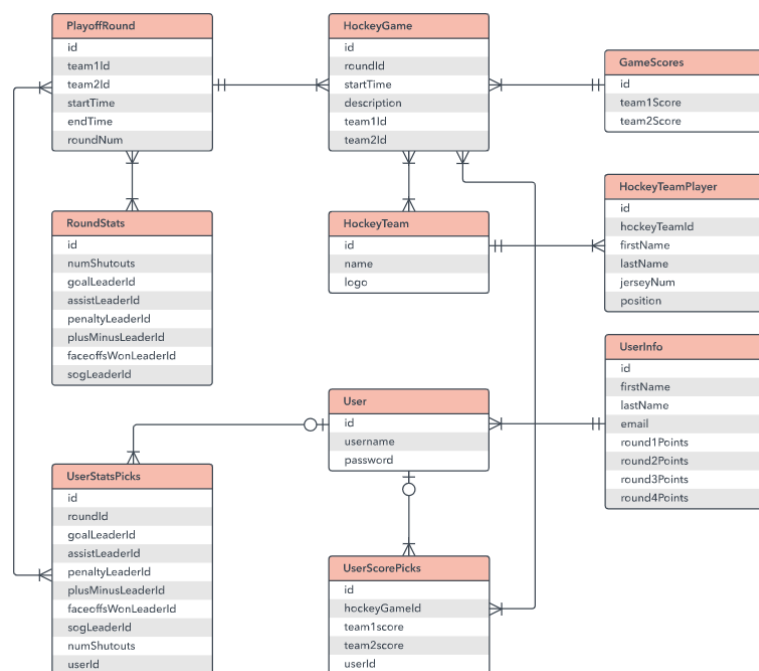


Рисунок 6.1 – Пример ER-диаграммы, созданной в Lucidchart

Вне зависимости от года появления на рынке и механизма функционирования, все программные средства визуального проектирования баз данных обладают одним и тем же функционалом: они позволяют размещать на рабочей области сущности и соединять их между собой отношениями (рис.6.1). Прочий функционал, такой как импорт данных из существующих баз данных, из CSV, встроенный редактор SQL, возможность онлайн коллаборации при работе над диаграммой, обычно является платным и не рассматривается в рамках задачи обучения основам визуального проектирования баз данных значимым.

Что же до сущностей и отношений, тут необходимо остановиться поподробнее. Тому как быстро и эффективно строить диаграммы, посвящено большое количество интернет-ресурсов, подкастов, роликов в интернете — поэтому рассмотрим только самые основы (на примере информации, размещенной Lucidchart).

Схема «сущность-связь» (также ERD или ER-диаграмма) — это разновидность блок-схемы, где показано, как разные «сущности» (люди, объекты, концепции и так далее) связаны между собой внутри системы. ER-диаграммы чаще всего применяются для проектирования и отладки реляционных баз данных в сфере образования, исследования и разработки программного обеспечения и информационных систем для бизнеса. ER-диаграммы (или ER-модели) полагаются на стандартный набор символов, включая прямоугольники, ромбы, овалы и соединительные линии, для отображения сущностей, их атрибутов и связей. Эти диаграммы устроены по тому же принципу, что и грамматические структуры: сущности выполняют роль существительных, а связи — глаголов.

ER-диаграммы — «родственники» схем структуры данных (DSD), где вместо связей между самими сущностями отображаются отношения между элементами внутри них. ER-диаграммы часто используются в сочетании с диаграммами DFD, которые схематично показывают движение потоков информации в рамках процесса или системы.

В ER-моделях и моделях данных обычно выделяют до трех уровней детализации:

- 1) Концептуальная модель данных — схема наивысшего уровня с минимальным количеством подробностей. Достоинство этого подхода заключается в возможности отобразить общую структуру модели и всю архитектуру системы. Менее масштабные системы могут обойтись и без этой модели. В этом случае можно сразу переходить к логической модели.
- 2) Логическая модель данных содержит более подробную информацию, нежели концептуальная модель. На этом уровне определяются более подробные операционные и транзакционные сущности. Логическая модель не зависит от технологии, в которой она будет применяться.
- 3) Физическая модель данных: на основе каждой логической модели данных можно составить одну или две физических модели. В последних должно присутствовать достаточно технических подробностей для составления и внедрения самой базы данных.

Обращаем ваше внимание на тот факт, что похожие уровни масштаба и детализации встречаются и в других видах схем (например, в диаграммах DFD), однако данная классификация отличается от трехсхемного подхода в разработке ПО, где деление информации осуществляется по несколько иному принципу. Правда, иногда разработчики применяют ER-диаграммы с дополнительными иерархиями, если дизайн базы данных требует больше информационных уровней. К примеру, разработчик может добавить новые группы по принципу расширения вверх (суперклассы) и вниз (подклассы).

- **Только реляционные данные.** Следует четко понимать, что цель ER-диаграмм — показать связи и отношения между элементами, поэтому они отображают только реляционную структуру.
- **Только для структурированных данных.** Данные должны быть четко разбиты на поля, столбцы и строки, иначе пользы от ER-

диаграммы будет мало. Это касается и частично структурированных данных, так как только некоторые из них будут пригодны для работы.

- **Сложность интеграции с существующей базой данных.**

Применение ER-моделей для интеграции с существующей базой данных — непростая задача по причине различия в архитектурах.

ER-диаграммы применяются для моделирования и проектирования реляционных баз данных, причем как в плане логических и бизнес-правил (логические модели данных), так и в плане внедрения конкретных технологий (физические модели данных). В сфере разработки программного обеспечения ER-диаграмма, как правило, служит первым шагом в определении требований проекта по созданию информационных систем. На дальнейших этапах работы ER-диаграммы также применяются для моделирования конкретных баз данных. Реляционная база данных сопровождается соответствующей реляционной таблицей и при необходимости может быть представлена в этом формате.

База данных – представленная в объективной форме совокупность самостоятельных материалов (статей, расчетов, нормативных актов, судебных решений и иных подобных материалов), систематизированных таким образом, чтобы эти материалы могли быть найдены и обработаны с помощью электронной вычислительной машины (ЭВМ).

ER-диаграммы применяются для анализа уже имеющихся баз данных с целью выявить и устранить ошибки в логике или развертывании. Диаграмма позволяет выявить, где именно закрались ошибки.

ER-схемы используются для проектирования и анализа реляционных баз данных, применяемых в бизнес-процессах. Реляционные базы данных могут пригодиться в любом бизнес-процессе, где задействованы данные, разбитые на поля, включая сущности, действия и взаимосвязи. Базы данных помогают оптимизировать процессы, извлекать данные и повышать качество результатов.

Диаграммы «сущность-связь» (или ERD) — неотъемлемая составляющая процесса моделирования любых систем, включая простые и сложные базы данных, однако применяемые в них фигуры и способы нотации могут запросто ввести в заблуждение любого. Это руководство поможет вам стать настоящим экспертом по нотации ER-диаграмм и уверенно взяться за моделирование собственных баз данных!

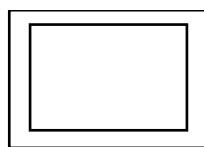
Концептуальные модели данных дают общее представление о том, что должно входить в состав модели. Концептуальные ER-диаграммы можно брать за основу логических моделей данных. Их также можно использовать для создания отношений общности между разными ER-моделями, положив их в основу интеграции. В случае использования ПО Lucidchart все приведенные ниже символы можно найти в библиотеках «Сущность-связь» для UML» и «Фигуры по модели «сущность-связь».

Под понятием «сущности» подразумеваются объекты или понятия, несущие важную информацию. С точки зрения грамматики, они, как правило, обозначаются существительными, например, «товар», «клиент», «заведение» или «промоакция». Ниже представлены три наиболее распространенных типа сущностей, используемых в ER-диаграммах.



Независимая
(сильная)
сущность

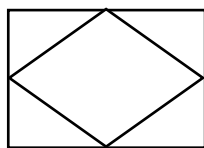
Не зависит от других сущностей и часто называется «родительской», так как у нее в подчинении обычно находятся слабые сущности. Независимые сущности сопровождаются первичным ключом, который позволяет идентифицировать каждый экземпляр сущности.



Зависимая
(слабая)
сущность

Сущность, которая зависит от сущности другого типа. Не сопровождается первичным ключом и не имеет значения

в схеме без своей родительской сущности.

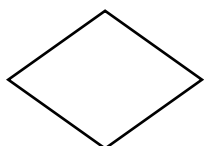


Ассоциативная
сущность

Соединяет экземпляры сущностей разных типов. Также содержит атрибуты, характерные для связей между этими сущностями.

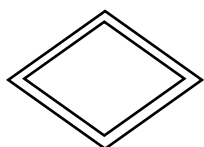
Символы ERD-связей

Связи используются в схемах «сущность-связь» для обозначения взаимодействия между двумя сущностями. Грамматически связи, как правило, выражаются глаголами, например, «назначить», «закрепить», «отследить», и несут полезную информацию, которую невозможно получить, опираясь только на типы сущностей.



Связь

Отношение между сущностями.

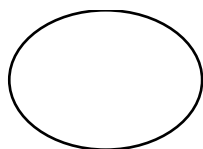


Слабая связь

Связь между зависимой сущностью и ее «хозяином».

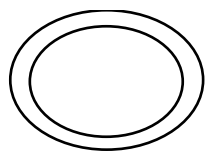
Символы ERD-атрибутов

ERD-атрибуты характеризуют сущности, позволяя пользователям лучше разобраться в устройстве базы данных. Атрибуты содержат информацию о сущностях, выделенных в концептуальной ER-диаграмме.

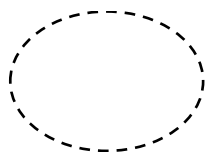


Атрибут

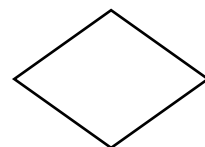
Характеризует сущность, а также отношения между двумя или более элементами.



Многозначный Атрибут, которому может быть
атрибут присвоено несколько значений.



Производный Атрибут, чье значение можно
атрибут вычислить, опираясь на значения
связанных с ним атрибутов.



Связь Отношение между сущностями.

Символы физических ER-диаграмм

Физическая модель данных — самый детальный уровень ER-схем, где представлен процесс добавления информации в базу данных. Физические модели «сущность-связь» отображают всю структуру таблицы, включая названия столбцов, типы данных, ограничения столбцов, первичные и внешние ключи, а также отношения между таблицами.

Как показано ниже, таблицы представляют собой еще один способ отображения сущностей. Вот ключевые составляющие таблиц «сущность-связь»:

Поля

Поля — это участки таблицы, где задаются атрибуты сущностей. Под атрибутами обычно подразумеваются столбцы базы данных, которая моделируется по принципу «сущность-связь».

Банк	
	Процентная_ставка Сумма_займа

Рисунок 6.2 – Таблица «Банк»

В примере выше Процентная_ставка и Сумма_займа оба являются атрибутами сущности и оба находятся внутри полей.

Ключи

Ключи — один из способов категоризации атрибутов. Напоминаем, что ER-диаграммы помогают пользователям моделировать базы данных посредством таблиц, которые обеспечивают им упорядоченность, эффективность и высокую скорость работы. Ну а ключи применяются с целью максимально эффективно связать между собой разные таблицы в базе данных.

Первичный ключ — это атрибут или сочетание атрибутов, идентифицирующих один конкретный экземпляр сущности.

Внешний ключ — понятие теории реляционных баз данных, относящееся к ограничениям целостности базы данных. Внешний ключ создается каждый раз, когда атрибут привязывается к сущности посредством единичной или множественной связи.

Потенциальный ключ — в реляционной модели данных — подмножество атрибутов отношения, удовлетворяющее требованиям уникальности и минимальности (несократимости).

Первым шагом при построении модели уровня ключей является разрешение неопределенных связей (связей многие-ко-многим). Для этого в модели все неопределенные связи заменяются связующими сущностями и двумя определенными связями.

После разрешения неопределенных связей необходимо определить атрибуты, которые являются первичными и внешними ключами, а также их домены. Чтобы указать, что атрибут является первичным ключом отношения, необходимо установить флажок Primary Key, для атрибутов, являющихся внешними ключами – Foreign Key.

Типы (домены)

Под типом подразумевается тип данных в соответствующем поле таблицы. Однако это также может быть и тип сущности, то есть описание ее составляющих. Например, у сущности «книга» будут следующие типы: «автор», «название» и «дата публикации».

Книга	
автор	text
название	text
дата_публикации	date

Рисунок 6.3 – Таблица «Книга»

Следует отметить, что в некоторых средствах визуального проектирования ER-диаграмм используется понятие «домен», который близок по своему понятию к типу данных. Домены определяются после определения ключевых атрибутов. Домены будут использоваться при определении типа колонки на уровне физической модели. Необходимо отметить, что на одном домене может быть задано сразу несколько атрибутов.

Для домена необходимо сформулировать и задать в соответствующем месте программного средства четыре параметра:

- имя домена,
- родительский домен,
- базовый тип данных домена,
- проверку на значения атрибутов домена.

Домены, определенные в ERwin Data Modeler приведены в таблице 6.1. Для типов данных других программных средств ограничений нет – разработчик сам указывает в желаемом формате необходимый тип атрибута.

Таблица 6.1 – Преднастроенные домены в Erwin Data Modeler

BLOB	информация в двоичных кодах
DATETIME	множество всех дат и времен
NUMBER	множество всех чисел
STRING	множество всех строк

Нормальные формы

Требование первой нормальной формы (1NF) очень простое и оно заключается в том, чтобы таблицы соответствовали реляционной модели данных и соблюдали определённые реляционные принципы.

Чтобы база данных находилась в 1 нормальной форме, необходимо чтобы ее таблицы соблюдали следующие реляционные принципы:

- В таблице не должно быть дублирующих строк
- В каждой ячейке таблицы хранится атомарное значение (одно не составное значение)
- В столбце хранятся данные одного типа
- Отсутствуют массивы и списки в любом виде

Процесс преобразования отношений базы данных к виду, отвечающему нормальным формам, называется **нормализацией**.

Чтобы база данных находилась во второй нормальной форме (2NF), необходимо чтобы ее таблицы удовлетворяли следующим требованиям:

- Таблица должна находиться в первой нормальной форме
- Таблица должна иметь ключ
- Все неключевые столбцы таблицы должны зависеть от полного ключа (в случае если он составной)

Требование третьей нормальной формы (3NF) заключается в том, чтобы в таблицах отсутствовала транзитивная зависимость.

Транзитивная зависимость – это когда неключевые столбцы зависят от значений других неключевых столбцов.

Между 3 и 4 нормальной формой есть еще и промежуточная нормальная форма, она называется – Нормальная форма Бойса-Кодда (BCNF). Иногда ее еще называют «Усиленная третья нормальная форма». Промежуточной или усиленной третьей нормальной формой ее называют потому, что ситуации, в которых могут предъявляться требования нормальной формы Бойса-Кодда, возникают не всегда, т.е. это некий частный случай, именно поэтому данная форма не включена в основную

градацию. Однако во всех источниках эта форма рассматривается, поэтому и мы ее тоже рассмотрим.

Перед тем как переходить к процессу приведения таблиц базы данных до нормальной формы Бойса-Кодда, необходимо, чтобы эти таблицы уже находились в третьей нормальной форме, подробно процесс приведения таблиц базы данных до третьей нормальной формы, а также все требования, предъявляемые к третьей нормальной форме мы рассматривали в предыдущей статье – третья нормальная форма (3NF).

После того как таблицы базы данных находятся в третьей нормальной форме, мы можем начинать приводить базу данных к нормальной форме Бойса-Кодда и рассматривать соответствующие требования.

Требования нормальной формы Бойса-Кодда следующие:

- Таблица должна находиться в третьей нормальной форме. Здесь все как обычно, т.е. как и у всех остальных нормальных форм, первое требование заключается в том, чтобы таблица находилась в предыдущей нормальной форме, в данном случае в третьей нормальной форме;
- Ключевые атрибуты составного ключа не должны зависеть от неключевых атрибутов.

Отсюда следует, что требования нормальной формы Бойса-Кодда предъявляются только к таблицам, у которых первичный ключ составной. Таблицы, у которых первичный ключ простой, и они находятся в третьей нормальной форме, автоматически находятся и в нормальной форме Бойса-Кодда.

Главное правило нормальной формы Бойса-Кодда (BCNF) звучит следующим образом: «Часть составного первичного ключа не должна зависеть от неключевого столбца».

Требование четвертой нормальной формы (4NF) заключается в том, чтобы в таблицах отсутствовали нетривиальные многозначные зависимости.

Переменная отношения находится в пятой нормальной форме (иначе – в проекционно-соединительной нормальной форме) тогда и только тогда, когда каждая нетривиальная зависимость соединения в ней определяется потенциальным ключом (ключами) этого отношения.

Требование пятой нормальной формы (5NF) заключается в том, чтобы в таблице каждая нетривиальная зависимость соединения определялась потенциальным ключом этой таблицы.

Требование шестой нормальной формы заключается в том, что таблица должна удовлетворять всем нетривиальным зависимостям соединения.

Нотация ER-диаграмм

Хотя нотация Crow's Foot («вороньи лапки») часто признается наиболее интуитивной, некоторые пользователи отдают предпочтение нотации Бахмана, ОМТ, IDEF или UML. Выбор конкретной нотации остается за разработчиком.

Кардинальность и ординальность

Под кардинальностью подразумевается максимальное число связей, которое может быть установлено между экземплярами разных сущностей. Ординальность, в свою очередь, указывает минимальное количество связей между экземплярами двух сущностей.

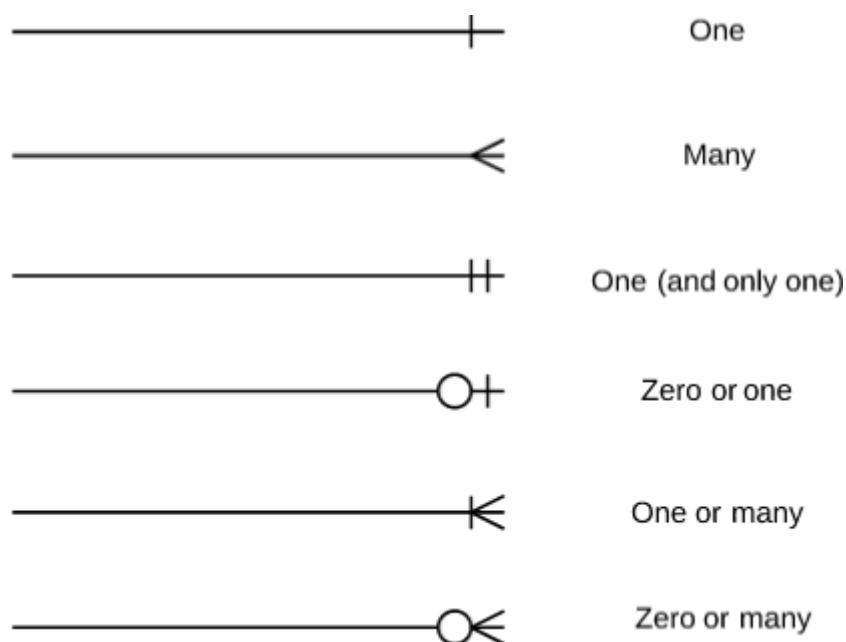


Рисунок 6.4 – Типы кардинальности в нотации «Вороньи лапки»

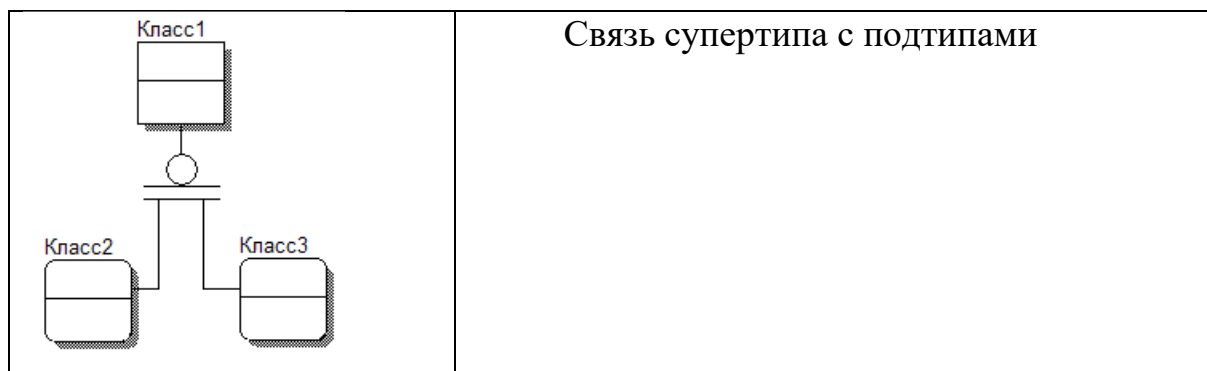
Кардинальность и ординальность отображаются на соединительных линиях согласно выбранному формату нотации. Различие в визуальном отображении идентифицирующей и неидентифицирующей связи заключается в типе линии – в случае неидентифицирующей, она пунктирная. Для неидентифицирующей связи имеется возможность задания обязательности связи (область Nulls):

- Не обязательная (Null Allowed)
- Обязательная (No Nulls)

Типы связей, применяемые в ERwin Data Modeler приведены в таблице 6.2.

Таблица 6.2 – Типы связей и их графическое отображение в ErWin

	Идентифицирующая связь
	Неидентифицирующая связь
	Связь типа «многие-ко-многим»



Пример выполнения задания

Полноатрибутная модель, приведенная на рис.6.5 и 6.6 построена на основе данных, описанных в таблицах 6.3-6.8.

Таблица 6.3 – Справочник students

Название	Описание	Тип данных
std_id	Поле – счетчик	INTEGER
std_name	Имя студента	TEXT
std_adress	Адрес студента	TEXT
std_addition	Поле для примечаний	TEXT
std_kurs	Курс студента	TEXT

Таблица 6.4 – Справочник groups

Название	Описание	Тип данных
grup_id	Поле – счетчик	INTEGER
date_create	Дата создания группы	TEXT

Таблица 6.5 – Справочник faculty

Название	Описание	Тип данных
fac_id	Поле – счетчик (РК)	INTEGER
fac_name	Название факультета	TEXT

Таблица 6.6 – Справочник prepros

Название	Описание	Тип данных
prep_id	Поле – счетчик (PK)	INTEGER
prep_name	Имя преподавателя	TEXT
prep_adress	Адрес преподавателя	TEXT
prep_persinf	Персональные данные преподавателя (паспорт)	TEXT
prep_inn	ИНН преподавателя	TEXT

Таблица 6.7 – Справочник predmets

Название	Описание	Тип данных
pred_id	Поле – счетчик (PK)	INTEGER
pred_name	Название предмета	TEXT
pred_kurs	Курс, на котором ведётся предмет	TEXT

Таблица 6.8 – Справочник rate

Название	Описание	Тип данных
rate_id	Поле – счетчик (PK)	INTEGER
rate_max	Максимальный рейтинг	INTEGER
rate_fact	Фактический рейтинг	INTEGER

Чтобы оценить всю пользу от метода «сущность-связь», попробуйте создать собственную ER-схему с помощью простой инструкции.

1. Задайте сущности (они, как правило, обозначаются существительными, например, «автомобиль», «банк», «студент», «товар» и так далее)

Сущности — важнейшая составляющая ER-диаграммы. В нашем примере мы будем создавать концептуальную ER-диаграмму простой системы, где студент обучается в вузе.

2. Определитесь со связями (они показывают, как сущности взаимодействуют между собой)

Связи обычно обозначаются глаголами, например, «покупает», «содержит», «выполняет» и так далее. В нашем примере взаимоотношения между сущностями раскрываются посредством связок «выполняется», «преподает», «работает» и др.

3. Добавьте атрибуты (они иллюстрируют конкретные характеристики сущности, фокусируя внимание на важной информации в контексте модели)

Атрибуты необходимы для моделирования характеристик, которые будут сопровождать каждую сущность ER-диаграммы. На практике в схемах часто встречаются такие атрибуты, как «ID», «наименование» и «артикул».

4. Внесите последние штрихи

Чтобы ER-диаграмма была понятной, очень важно расположить все ее элементы в логичном порядке. Ведь главная цель схемы «сущность-связь» состоит в том, чтобы смоделировать сложную базу данных, и поэтому ваша задача номер один — научиться выстраивать ER-схемы просто и логично.

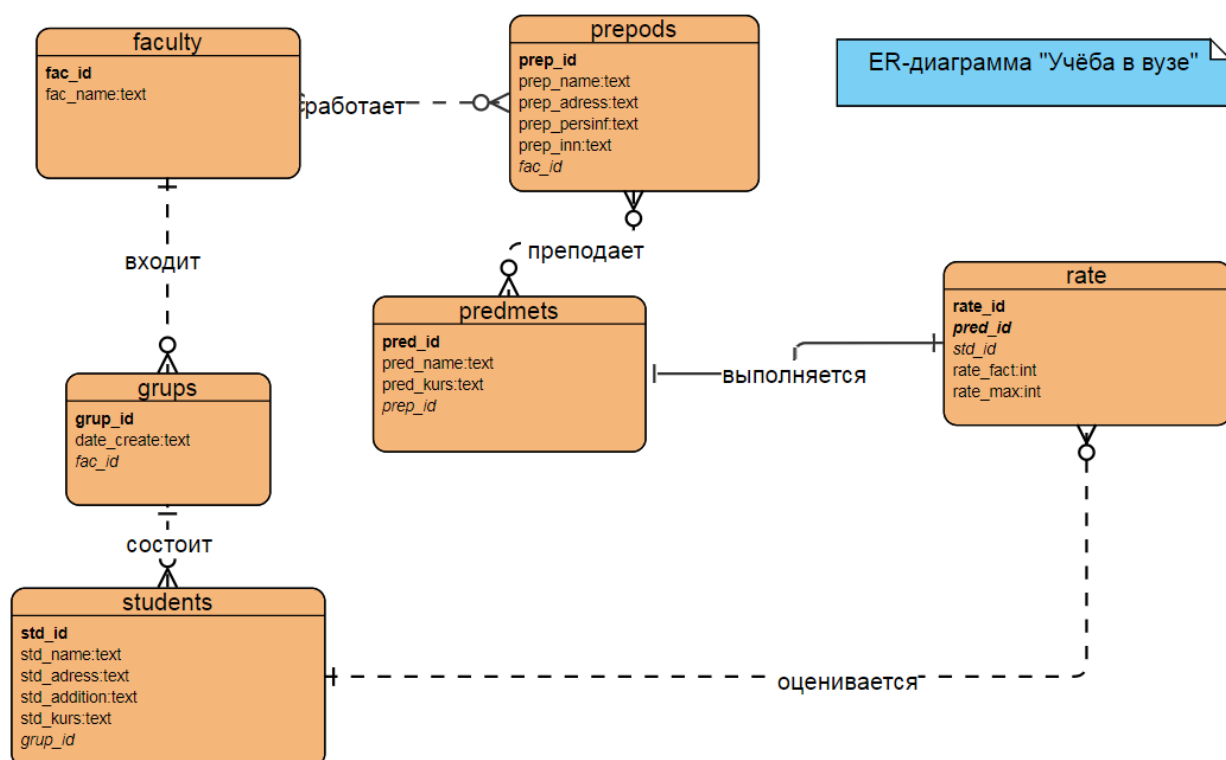


Рисунок 6.5 – ER-диаграмма, выполненная в Visual Paradigm

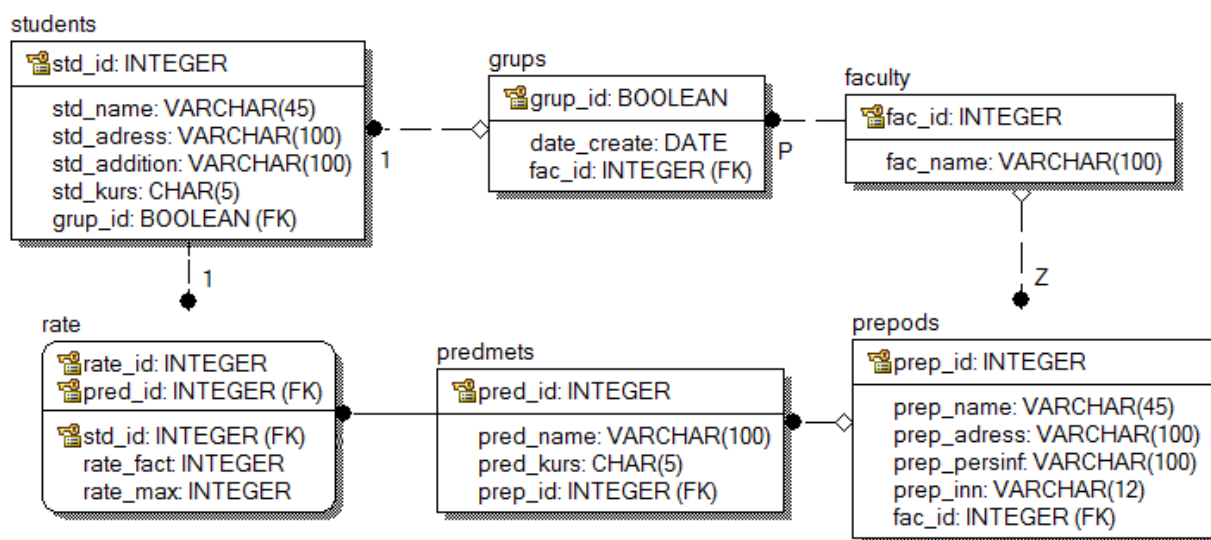


Рисунок 6.6 – ER-диаграмма, выполненная в ERwin Data Modeler

Задание

Создайте полноатрибутную модель базы данных для выбранной предметной области.

Лабораторная работа №7

Цель работы

Целью данной работы является изучение основ языка SQL и построение базы данных.

Краткое изложение теоретической части

Структурированный язык запросов (Structured Query Language) – стандарт коммуникации с базой данных, который поддержан ANSI. В общих терминах, «SQL база данных» является общим названием для реляционной системы управления базами данных (РСУБД). Для некоторых систем, «база данных» также относится к группе таблиц, данных, конфигурационной информации, которые являются неотъемлемо отдельной частью от других, подобных конструкций. В этом случае, каждая инсталляция SQL базы данных может состоять из нескольких баз данных. В других системах, они упомянуты как таблицы.

Таблица – конструкция базы данных, которая состоит из столбцов, содержащих строки данных. Обычно таблицы созданы для того, чтобы содержать связанную информацию. В пределах той же самой базы данных могут быть созданы несколько таблиц.

Каждый столбец представляет собой атрибут или совокупность атрибутов объектов, например идентификационные номера служащих, рост, цвет машин и т.п. Часто в отношении столбца используется термин поле с указанием имени, например «в поле Name». Поле строки является минимальным элементом таблицы. Каждый столбец в таблице имеет определенное имя, тип данных и размер. Имена столбцов должны быть уникальны в пределах таблицы.

Каждая строка (или запись) представляет собой совокупность атрибутов конкретного объекта, например, в строке может содержаться идентификационный номер служащего, размер его зарплаты, год его рождения и т.д. Строки таблиц не имеют названий. Чтобы обратиться к

конкретной строке, пользователю необходимо указать какой-то атрибут (или набор атрибутов), уникально ее идентифицирующий.

Одной из важнейших операций, которые выполняются при работе с данными, является выборка хранящейся в базе данных информации. Для этого пользователь должен выполнить запрос (query).

Рассмотрим основные типы запросов к базе данных, которые сосредоточены на манипуляции данными в пределах базы.

Таблица 7.1 – Управление данными (Data Manipulation Language, DML)

DELETE	Удаление записи или нескольких записей
INSERT	Добавление записи или нескольких записей
LOAD DATA INFILE	Добавление в таблицу записей из файла
REPLACE	Добавление записи или записей с заменой
SELECT	Выборка записей
SELET JOIN	Варианты выборки записей из нескольких таблиц
TRUNCATE	Удаление таблицы и создание аналогичной новой
UPDATE	Изменение записей

Таблица 7.2 – Определение данных (Data Definition Language, DDL)

ALTER TABE	Изменение структуры таблицы
CREATE DATABASE	Создание базы данных
CREATE INDEX	Создание индекса
CREATE TABLE	Создание таблицы
DROP DATABASE	Удаление базы данных
DROP INDEX	Удаление индекса
DROP TABLE	Удаление таблицы
RENAME TABLE	Переименование таблицы

Таблица 7.3 – Служебные команды (Service)

DESCRIBE	Отображает информацию о полях таблицы
SHOW	Команда отображает информацию о базах данных, таблицах, полях или о состоянии сервера
USE	Назначает текущую базу данных

Остановимся более подробно на разборе синтаксиса, приведенных выше команд.

CREATE DATABASE

CREATE DATABASE [IF EXISTS] db_name

Создает базу с указанным именем. Если указан параметр `IF EXISTS`, то MySQL не будет сообщать об ошибке («Существует база данных с таким именем»). База данных в MySQL физически представляет собой папку, в которой хранятся файлы описания, индексов и данных таблиц. Так как таблиц еще нет, то данная команда просто создает пустую директорию.

```
CREATE DATABASE u4eba;
```

DROP DATABASE

DROP DATABASE [IF EXISTS] db_name

Удаляет все таблицы в базе и саму базу данных. Если команда выполняется над ссылкой на базу данных, то удаляются и ссылка, и база.

Будьте ОЧЕНЬ осторожны с этой командой!

Возвращает количество удаленных файлов в папке базы данных. Обычно это утроенное количество таблиц, так как для хранения таблиц нужно обычно три файла ("table_name.frm", "table_name.myd", "table_name.myi"). Команда удаляет все файлы в папке базы данных следующих типов: *.BAK, *.DAT, *.db, *.frm, *.HSH, *.ISD, *.ISM, *.MRG, *.MYD, *.MYI.

Удаляются также все вложенные папки с именем из 2 символов (папки RAID). Начиная с версии 3.22 можно использовать ключевое слово `IF`

EXISTS для того, чтобы MySQL не сообщал об ошибке удаления базы данных, которой не существует.

```
DROP DATABASE u4eba;
```

CREATE TABLE

CREATE [TEMPORARY]

TABLE [IF NOT EXISTS] tbl_name

[(create_definition,...)] [table_options]

Команда создает таблицу с именем `tbl_name` в текущей БД. Если не выбрана БД или таблица с выбранным именем существует, то происходит ошибка. Начиная с MySQL версии 3.22 указать имя таблицы используя синтаксис `имя_БД.имя_таблицы`, причем при таком синтаксисе можно не выбирать текущую БД.

Каждая таблица хранится в нескольких файлах. Например, для таблиц типа MyISAM эти файлы следующие:

*.frm – Файл определения таблицы;

*.MYD – Файл данных таблицы;

*.MYI – Файл индексов таблицы.

```
CREATE TABLE u4eba.student (
    `std_id` INTEGER UNSIGNED NOT NULL AUTO_INCREMENT,
    `std_name` VARCHAR(45) NOT NULL,
    `std_adress` VARCHAR(100),
    `std_formofstudy` CHAR(3) NOT NULL,
    `std_kurs` INTEGER UNSIGNED NOT NULL,
    `std_group` VARCHAR(8) NOT NULL,
    PRIMARY KEY (`std_id`)
);
```

DROP TABLE

DROP TABLE [IF EXISTS] tbl_name [, tbl_name,...] [RESTRICT /

CASCADE]

Удаляет таблицу или несколько таблиц. Все их данные и определения будут уничтожены, так что будьте осторожны с этой командой. Начиная с версии 3.22 можно указать ключевое слово `IF EXISTS`, для того, чтобы СУБД не выдала ошибку, если таблицы с таким именем нет. `RESTRICT` и `CASCADE` существуют для совместимости. Пока они ничего не делают. Команда небезопасна для транзакций, поэтому она автоматически зафиксирует все активные транзакции.

```
DROP TABLE student;
```

CREATE INDEX

CREATE [UNIQUE/FULLTEXT]

INDEX index_name

ON tbl_name (col_name[(length)],...)

Индексирует поле `col_name` в таблице `tbl_name`. Индексировать можно только поля, которые могут принимать значение `NULL`, или поля типов `BLOB/TEXT`, если версия MySQL 3.23.2 или новее и таблица типа `MyISAM`. Команда ничего не делает вплоть до версии MySQL 3.22. В версии 3.22 и более поздних, команда `CREATE INDEX` привязана к команде `ALTER TABLE`.

Обычно все индексы создаются вместе с таблицей (см. команду `CREATE TABLE`). С помощью `CREATE INDEX` индексы добавляются к существующей таблице.

```
CREATE INDEX nameInd ON u4eba.student (std_name);
```

INSERT

INSERT INTO table_name (col1,[col2,...]) VALUES (val1,[val2,...])

Команда вставляет новую запись в существующую таблицу. Вид команды `INSERT ... VALUES` вставляет значения, заданные явным образом, а синтаксис `INSERT ... SELECT` вставляет записи из другой таблицы (или из других таблиц).

Синтаксис `INSERT ... VALUES` с возможностью вставить сразу несколько записей поддерживается в MySQL, начиная с версии 3.22.5, а синтаксис `SET col_name=expression, col_name=expression, ...` - начиная с версии 3.22.10.

```
INSERT INTO u4eba.student VALUES (
1,
"Sidorov K.",
"Tomsk, Lytkina str., 12-200",
0,
1,
"510-1"
);
```

ALTER TABLE

ALTER [IGNORE] TABLE tbl_name alter_spec [, alter_spec ...]

Команда позволяет изменять структуру существующей таблицы, например, добавлять и удалять поля, создавать и уничтожать индексы, менять тип существующего поля, переименовать поля и саму таблицу. Для использования команды необходимо иметь привилегии `ALTER`, `INSERT` и `CREATE` для таблицы.

При изменении таблицы создается временная копия таблицы, над которой и производятся все изменения, после чего оригинал таблицы удаляется, а временная таблица переименовывается. Все операции реализованы таким образом, что все изменения записей происходят в новой таблице без потерянных изменений. Пока происходит выполнение `ALTER TABLE`, старая таблица доступна для чтения другим клиентам. Любые изменения с записями таблицы откладываются, чтобы выполняться над измененной таблицей.

```
ALTER TABLE u4eba.student ADD std_primechanie VARCHAR(150),
ADD std_inn INTEGER, DROP std_formofstudy;
```

SELECT

SELECT select_expression,...

[FROM table_references [WHERE where_definition] [ORDER BY {unsigned_integer | col_name | formula} [ASC | DESC] ,...]]

Команда используется для того, чтобы извлечь записи из одной или нескольких таблиц. Все операторы должны встречаться в такой же последовательности, как показано выше. Например, секция HAVING должна идти после секции GROUP BY и перед секцией ORDER BY.

```
SELECT * FROM u4eba.student ORDER BY std_name;
```

RENAME TABLE

RENAME TABLE tbl_name TO new_table_name[, tbl_name2 TO new_table_name2,...]

Переименовывает таблицу. Переименование автоматическое, это означает, что больше ни один процесс не будет иметь доступ ни к одной из таблиц, имена которых изменяются, поэтому возможно, например, заменить таблицу с записями пустой таблицей:

```
CREATE TABLE new_table (...)
```

```
RENAME TABLE old_table TO backup_table, new_table TO old_table  
или поменять местами имена двух таблиц (выполнение команды ведется  
слева направо):
```

```
RENAME TABLE old_table TO backup_table, new_table TO old_table,  
backup_table TO new_table
```

Если две базы данных находятся на одном диске:

```
RENAME          TABLE          current_database.table_name      TO  
other_database.table_name
```

Для выполнения команды не должно быть заблокированных таблиц или активных транзакций. У вас также должны быть привелегии ALTER и DROP на переименовываемую таблицу и привелегии CREATE и INSERT для создания новой таблицы. Если во время переименования нескольких таблиц произойдет ошибка, то ни одна таблица не будет переименована.

```
RENAME TABLE u4eba.student TO studenty;
```


DROP INDEX

DROP INDEX index_name ON tbl_name

Удаляет индекс с именем `index_name` в таблице `tbl_name`. Команда ничего не делает до версии 3.22. В более поздних версиях команда аналогична команде `ALTER TABLE` с синтаксисом `DROP INDEX`.

```
DROP INDEX nameInd ON u4eba.student;
```

LOAD DATA INFILE

*LOAD DATA [LOW_PRIORITY | CONCURRENT] [LOCAL] INFILE
'file_name.txt' [REPLACE | IGNORE] INTO TABLE tbl_name [FIELDS
[TERMINATED BY '\t'] [[OPTIONALLY] ENCLOSED BY '"'] [ESCAPED BY
'\']]] [LINES TERMINATED BY '\n'] [IGNORE number LINES]
[(col_name,...)]*

Команда считывает записи из текстового файла и добавляет их в таблицу с большой скоростью. Записи в файле должны быть разделены табулятором и не содержать пустых строк.

```
LOAD DATA LOCAL INFILE "c:\\list.txt" INTO TABLE student;
```

DELETE

DELETE FROM table_name [WHERE where_definition]

Команда удаляет записи из таблицы `table_name`, которые подходят под условия, указанные в выражении `where_definition` и возвращает количество удаленных записей.

Если условие `WHERE` пропущено, удаляются все записи. Если записи удаляются в режиме `AUTOCOMMIT`, команда работает как `TRUNCATE`. В этом случае команда `DELETE` возвращает ноль как количество удаленных записей, но эта ошибка будет исправлена в версии 4.0

Если необходимо узнать число записей, удаленных при очистке таблицы, можно использовать следующий синтаксис:

```
DELETE FROM tbl_name WHERE 1>0
```

Обратите внимание, что подобный способ работает гораздо медленнее, чем запрос без условий WHERE, так как в последнем случае записи уничтожаются за один запрос.

```
DELETE FROM u4eba.student WHERE std_kurs=1;
```

SELECT JOIN

SELECT table_reference, table_reference

table_reference [CROSS] JOIN table_reference

table_reference INNER JOIN table_reference join_condition

table_reference STRAIGHT_JOIN table_reference

table_reference LEFT [OUTER] JOIN table_reference join_condition

table_reference LEFT [OUTER] JOIN table_reference

table_reference NATURAL [LEFT [OUTER]] JOIN table_reference

{ oj table_reference LEFT OUTER JOIN table_reference ON conditional_expr }

table_reference RIGHT [OUTER] JOIN table_reference join_condition

table_reference RIGHT [OUTER] JOIN table_reference

table_reference NATURAL [RIGHT [OUTER]] JOIN table_reference

Осуществляет выборку из нескольких таблиц

```
SELECT * FROM student LEFT JOIN predmet ON student.std_kurs =  
predmet.pred_kurs;
```

USE

USE db_name

Назначает базу данных db_name текущей для последующих запросов. База данных будет текущей до завершения соединения либо до назначения текущей другой базы данных.

Если вы сделали какую-то базу данных текущей, то это не означает, что вы не можете обращаться к другим базам данных.

```
USE other_base;
SELECT count(*) FROM other_table;
USE u4eba;
SELECT count(*) FROM student;
```

TRUNCATE

TRUNCATE TABLE table_name

Существует в версиях 3.23. Команда подобна запросу `DELETE FROM table_name`. Различия между ними:

`TRUNCATE` выполнена как удаление таблицы и создание ее заново, поэтому выполняется гораздо быстрее, чем `DELETE`, когда в таблице много записей; `TRUNCATE` автоматически завершит текущую транзакцию, точно так же, как если бы выполнялась команда `COMMIT`;

`TRUNCATE` не завершает количество удаленных записей;

Пока существует неповрежденный файл `'table_name.frm'`, таблица `table_name` может быть создана заново, даже если повреждены файл данных или индексный файл.

```
TRUNCATE TABLE u4eba.predmet;
```

REPLACE

REPLACE [LOW_PRIORITY | DELAYED] [INTO] tbl_name SET col_name=expression, col_name=expression,...

Команда работает точно так же как `INSERT`, за исключением того, что если в уникальном индексе есть то же значение, что и во новой записи, то перед тем, как вставить новую запись, старая будет удалена.

Другими словами, Вы не можете получить значение старой записи с помощью этой команды. В некоторых старых версиях было похоже, что это можно сделать, но то была ошибка, которая уже исправлена.

```
REPLACE INTO u4eba.student VALUES (
6,
"Vasilkov V.",
```

```
"Tomsk, Lytkina str., 12-201",
1,
"510-1",
"",
7012);
```

UPDATE

UPDATE table_name

SET col_name1=expr1, [col_name2=expr2, ...]

[WHERE where_definition]

Заменяет значения в полях таблицы новыми значениями.

```
UPDATE u4eba.student SET
u4eba.std_kurs=3 WHERE u4eba.std_kurs=2;
```

DESCRIBE

{DESCRIBE / DESC} tbl_name {col_name / wild}

Это короткая запись команды `SHOW COLUMNS FROM`

Команда отображает информацию о полях таблицы. Col_name может быть именем поля или строкой, содержащей специальные символы SQL «%» и «_».

```
DESCRIBE u4eba.student;
```

SHOW

SHOW DATABASES [LIKE wild]

SHOW [OPEN] TABLES [FROM db_name] [LIKE wild]

SHOW [FULL] COLUMNS FROM tbl_name [FROM db_name] [LIKE wild]

SHOW INDEX FROM tbl_name [FROM db_name]

SHOW TABLE STATUS [FROM db_name] [LIKE wild]

SHOW STATUS [LIKE wild]

SHOW VARIABLES [LIKE wild]

SHOW LOGS

SHOW [FULL] PROCESSLIST

SHOW GRANTS FOR user

SHOW CREATE TABLE table_name

SHOW MASTER STATUS

SHOW MASTER LOGS

SHOW SLAVE STATUS

Команда отображает информацию о базах данных, таблицах, полях или о состоянии сервера. Если используется необязательный параметр `LIKE wild` должен содержать строку, в которой используются специальные символы SQL `'%'` и `'_'`.

`SHOW DATABASES` показывает список всех баз данных на сервере MySQL. Также этот список можно получить, запустить в командной строке системы `mysqlshow`.

`SHOW TABLES` отображает список всех таблиц в заданной базе данных. Также можно в командной строке набрать `mysqlshow имя_таблицы`. В этих списках не будут указаны таблицы, на которые у вас нет никаких привелегий.

`SHOW OPEN TABLES` возвращает список таблиц, которые в данный момент открыты в табличном кеше. Поле `comment` указывает, сколько раз таблица кешировалась и использовалась.

`SHOW COLUMNS` демонстрирует список полей в таблице. Если задан параметр `FULL`, то также будет доступна информация о привелегиях пользователя для каждого поля. Типы полей могут отличаться от тех, что были заданы при создании или изменении таблицы.

`SHOW FIELDS` - синоним команды `SHOW COLUMNS`, а `SHOW KEYS` - синоним `SHOW INDEX`. Список полей и индексов можно также получить из командной строки, запустив утилиту `mysqlshow db_name tbl_name` или `mysqlshow -k db_name tbl_name`.

`SHOW INDEX` возвращает информацию о ключах (индексах) таблицы в формате, который близок к ответу команды `SQLStatistics` в ODBC. В ответе будут такие колонки:

`SHOW TABLE STATUS` Доступен, начиная с MySQL 3.23. Работает так же, как и `SHOW STATUS`, но показывает много информации о каждой таблице. Также можно использовать системную команду `mysqlshow --status db_name`.

`SHOW STATUS` Команда сообщает о состоянии сервера (как и `mysqladmin extended-status`).

`SHOW VARIABLES`

Команда показывает значение некоторых системных переменных MySQL. Ту же информацию можно получить утилитой `mysqladmin variables`. Если значение некоторых из них вас не устраивает, можно их изменить из командной строки при старте `mysqld`.

`SHOW LOGS`

Информация о существующих файлах протоколов.

`SHOW PROCESSLIST`

Показывает активные потоки. Также информацию можно получить, запустив утилиту `mysqladmin processlist`. Если у вас есть привелегии `process`, то вы увидите все потоки; в противном случае вы увидите только свои потоки. При отсутствии параметра `FULL` вы увидите только первые 100 символов запроса

`SHOW GRANTS`

Показывает список разрешенных пользователю команд.

`SHOW CREATE TABLE`

Показывает команду SQL, выполнение которой создаст данную таблицу.

`SHOW DATABASES;`

`SHOW TABLES FROM u4eba;`

Пример выполнения задания

Рассмотрим в качестве примера запрос, результатом выполнения которого будет создание таблицы *Студент*.

```
CREATE TABLE `students` (
    `std_id` INTEGER UNSIGNED NOT NULL,
    `std_name` VARCHAR(45) NOT NULL,
    `std_adress` VARCHAR(100),
    `std_addition` VARCHAR(100),
    `std_kurs` CHAR(5) NOT NULL,
    PRIMARY KEY (`std_id`)
);
```

CREATE TABLE — команда создания, в кавычках название таблицы, дальше в скобках указываются параметры таблицы, потом скобка закрывается, устанавливается кодировка данных, в конце запроса обязательно ставится точка с запятой.

NOT NULL — означает что поле не может содержать пустых значений - если пользователь попытается сохранить значение для NULL, то произойдет ошибка на уровне СУБД, а обрабатывать вам эту ошибку в своей программе или нет, это уже ваше дело.

Поля `std_name`, `std_adress` и `std_addition` имеют текстовое значение, в SQL оно обозначается как `VARCHAR` и в скобках указывается максимальное количество возможных символов.

Поле `std_id` назначается в качестве первичного ключа таблицы.

Т.к. синтаксис запросов зависит от диалекта SQL, то, в зависимости от выбранного средства реализации запрос может выглядеть иначе. Например, если в качестве конструктора базы использовать онлайн-средство <https://sqliteonline.com>, то запрос будет выглядеть несколько иначе:

```
CREATE TABLE IF NOT EXISTS students (std_id integer primary key, std_name
text, std_adress text, std_addition text, std_kurs text)
```

Задание

Изучить синтаксис основных команд языка SQL. Используя полученные знания, построить базу данных своей предметной области. В базе данных реализовать не менее 4-5 таблиц. Каждую таблицу заполнить не менее, чем 10 осмысленными записями.

Лабораторная работа №8

Цель работы

Целью работы является разработка СУБД для базы данных, построенной в ходе предыдущей лабораторной работы.

Краткое изложение теоретической части

Система управления базами данных (СУБД) – совокупность программных и лингвистических средств общего или специального назначения, обеспечивающих управление созданием и использованием баз данных.

Разработать структуру базы данных и реализовать ее – это важный этап проектирования, но не последний. Для пользователя наиболее важным представляется визуальная реализация этой базы данных в виде приложения или веб-формы, в которой осуществляется управление данными.

Безусловно, можно использовать механизм заполнения таблиц, работая с визуальными редакторами SQL таблиц, например, с phpMyAdmin, но если стоит задача разработки удобного пользовательского интерфейса, скрывающего от пользователя всю «внутреннюю кухню» приложения, то решение должно быть принципиально иным. Таковым решением является разработка приложения на одном из популярных языков программирования с использованием всего богатого функционала, заложенного в них.

Более точно, к числу функций СУБД принято относить следующие:

Непосредственное управление данными во внешней памяти

Эта функция включает обеспечение необходимых структур внешней памяти как для хранения данных, непосредственно входящих в БД, так и для служебных целей, например, для убыстрения доступа к данным в некоторых случаях (обычно для этого используются индексы). В некоторых реализациях СУБД активно используются возможности существующих файловых систем, в других работа производится вплоть до уровня устройств

внешней памяти. Но подчеркнем, что в развитых СУБД пользователи в любом случае не обязаны знать, использует ли СУБД файловую систему, и если использует, то как организованы файлы. В частности, СУБД поддерживает собственную систему именования объектов БД.

Управление буферами оперативной памяти

СУБД обычно работают с БД значительного размера; по крайней мере этот размер обычно существенно больше доступного объема оперативной памяти. Понятно, что если при обращении к любому элементу данных будет производиться обмен с внешней памятью, то вся система будет работать со скоростью устройства внешней памяти. Практически единственным способом реального увеличения этой скорости является буферизация данных в оперативной памяти. При этом, даже если операционная система производит общесистемную буферизацию (как в случае ОС UNIX), этого недостаточно для целей СУБД, которая располагает гораздо большей информацией о полезности буферизации той или иной части БД. Поэтому в развитых СУБД поддерживается собственный набор буферов оперативной памяти с собственной дисциплиной замены буферов.

Заметим, что существует отдельное направление СУБД, которое ориентировано на постоянное присутствие в оперативной памяти всей БД. Это направление основывается на предположении, что в будущем объем оперативной памяти компьютеров будет настолько велик, что позволит не беспокоиться о буферизации. Пока эти работы находятся в стадии исследований.

Управление транзакциями

Транзакция — это последовательность операций над БД, рассматриваемых СУБД как единое целое. Либо транзакция успешно выполняется, и СУБД фиксирует (COMMIT) изменения БД, произведенные этой транзакцией, во внешней памяти, либо ни одно из этих изменений никак не отражается на состоянии БД. Понятие транзакции необходимо для поддержания логической целостности БД. Если вспомнить наш пример

информационной системы с файлами СОТРУДНИКИ и ОТДЕЛЫ, то единственным способом не нарушить целостность БД при выполнении операции приема на работу нового сотрудника является объединение элементарных операций над файлами СОТРУДНИКИ и ОТДЕЛЫ в одну транзакцию. Таким образом, поддержание механизма транзакций является обязательным условием даже однопользовательских СУБД (если, конечно, такая система заслуживает названия СУБД). Но понятие транзакции гораздо более важно в многопользовательских СУБД.

То свойство, что каждая транзакция начинается при целостном состоянии БД и оставляет это состояние целостным после своего завершения, делает очень удобным использование понятия транзакции как единицы активности пользователя по отношению к БД. При соответствующем управлении параллельно выполняющимися транзакциями со стороны СУБД каждый из пользователей может в принципе ощущать себя единственным пользователем СУБД (на самом деле, это несколько идеализированное представление, поскольку в некоторых случаях пользователи многопользовательских СУБД могут ощутить присутствие своих коллег).

С управлением транзакциями в многопользовательской СУБД связаны важные понятия сериализации транзакций и сериального плана выполнения смеси транзакций. Под сериализацией параллельно выполняющихся транзакций понимается такой порядок планирования их работы, при котором суммарный эффект смеси транзакций эквивалентен эффекту их некоторого последовательного выполнения. Сериальный план выполнения смеси транзакций – это такой план, который приводит к сериализации транзакций. Понятно, что если удастся добиться действительно сериального выполнения смеси транзакций, то для каждого пользователя, по инициативе которого образована транзакция, присутствие других транзакций будет незаметно (если не считать некоторого замедления работы по сравнению с однопользовательским режимом).

Существует несколько базовых алгоритмов сериализации транзакций. В централизованных СУБД наиболее распространены алгоритмы, основанные на синхронизационных захватах объектов БД. При использовании любого алгоритма сериализации возможны ситуации конфликтов между двумя или более транзакциями по доступу к объектам БД. В этом случае для поддержания сериализации необходимо выполнить откат (ликвидировать все изменения, произведенные в БД) одной или более транзакций. Это один из случаев, когда пользователь многопользовательской СУБД может реально (и достаточно неприятно) ощутить присутствие в системе транзакций других пользователей.

Журнализация

Одним из основных требований к СУБД является надежность хранения данных во внешней памяти. Под надежностью хранения понимается то, что СУБД должна быть в состоянии восстановить последнее согласованное состояние БД после любого аппаратного или программного сбоя. Обычно рассматриваются два возможных вида аппаратных сбоев: так называемые мягкие сбои, которые можно трактовать как внезапную остановку работы компьютера (например, аварийное выключение питания), и жесткие сбои, характеризующиеся потерей информации на носителях внешней памяти. Примерами программных сбоев могут быть: аварийное завершение работы СУБД (по причине ошибки в программе или в результате некоторого аппаратного сбоя) или аварийное завершение пользовательской программы, в результате чего некоторая транзакция остается незавершенной. Первую ситуацию можно рассматривать как особый вид мягкого аппаратного сбоя; при возникновении последней требуется ликвидировать последствия только одной транзакции.

Понятно, что в любом случае для восстановления БД нужно располагать некоторой дополнительной информацией. Другими словами, поддержание надежности хранения данных в БД требует избыточности хранения данных, причем та часть данных, которая используется для

восстановления, должна храниться особо надежно. Наиболее распространенным методом поддержания такой избыточной информации является ведение журнала изменений БД.

Журнал – это особая часть БД, недоступная пользователям СУБД и поддерживаемая с особой тщательностью (иногда поддерживаются две копии журнала, располагаемые на разных физических дисках), в которую поступают записи обо всех изменениях основной части БД. В разных СУБД изменения БД журналируются на разных уровнях: иногда запись в журнале соответствует некоторой логической операции изменения БД (например, операции удаления строки из таблицы реляционной БД), иногда – минимальной внутренней операции модификации страницы внешней памяти; в некоторых системах одновременно используются оба подхода.

Во всех случаях придерживаются стратегии "упреждающей" записи в журнал (так называемого протокола Write Ahead Log – WAL). Грубо говоря, эта стратегия заключается в том, что запись об изменении любого объекта БД должна попасть во внешнюю память журнала раньше, чем измененный объект попадет во внешнюю память основной части БД. Известно, что если в СУБД корректно соблюдается протокол WAL, то с помощью журнала можно решить все проблемы восстановления БД после любого сбоя.

Самая простая ситуация восстановления – индивидуальный откат транзакции. Строго говоря, для этого не требуется общесистемный журнал изменений БД. Достаточно для каждой транзакции поддерживать локальный журнал операций модификации БД, выполненных в этой транзакции, и производить откат транзакции путем выполнения обратных операций, следуя от конца локального журнала. В некоторых СУБД так и делают, но в большинстве систем локальные журналы не поддерживают, а индивидуальный откат транзакции выполняют по общесистемному журналу, для чего все записи от одной транзакции связывают обратным списком (от конца к началу).

При мягком сбое во внешней памяти основной части БД могут находиться объекты, модифицированные транзакциями, не закончившимися к моменту сбоя, и могут отсутствовать объекты, модифицированные транзакциями, которые к моменту сбоя успешно завершились (по причине использования буферов оперативной памяти, содержимое которых при мягком сбое пропадает). При соблюдении протокола WAL во внешней памяти журнала должны гарантированно находиться записи, относящиеся к операциям модификации обоих видов объектов. Целью процесса восстановления после мягкого сбоя является состояние внешней памяти основной части БД, которое возникло бы при фиксации во внешней памяти изменений всех завершившихся транзакций и которое не содержало бы никаких следов незаконченных транзакций. Для того, чтобы этого добиться, сначала производят откат незавершенных транзакций (undo), а потом повторно воспроизводят (redo) те операции завершенных транзакций, результаты которых не отображены во внешней памяти. Этот процесс содержит много тонкостей, связанных с общей организацией управления буферами и журналом. Более подробно мы рассмотрим это в соответствующей лекции.

Для восстановления БД после жесткого сбоя используют журнал и архивную копию БД. Грубо говоря, архивная копия – это полная копия БД к моменту начала заполнения журнала (имеется много вариантов более гибкой трактовки смысла архивной копии). Конечно, для нормального восстановления БД после жесткого сбоя необходимо, чтобы журнал не пропал. Как уже отмечалось, к сохранности журнала во внешней памяти в СУБД предъявляются особо повышенные требования. Тогда восстановление БД состоит в том, что исходя из архивной копии по журналу воспроизводится работа всех транзакций, которые закончились к моменту сбоя. В принципе, можно даже воспроизвести работу незавершенных транзакций и продолжить их работу после завершения восстановления.

Однако в реальных системах это обычно не делается, поскольку процесс восстановления после жесткого сбоя является достаточно длительным.

Поддержка языков БД

Для работы с базами данных используются специальные языки, в целом называемые языками баз данных. В ранних СУБД поддерживалось несколько специализированных по своим функциям языков. Чаще всего выделялись два языка – язык определения схемы БД (SDL – Schema Definition Language) и язык манипулирования данными (DML – Data Manipulation Language). SDL служил главным образом для определения логической структуры БД, т.е. той структуры БД, какой она представляется пользователям. DML содержал набор операторов манипулирования данными, т.е. операторов, позволяющих заносить данные в БД, удалять, модифицировать или выбирать существующие данные. Мы рассмотрим более подробно языки ранних СУБД в следующей лекции.

В современных СУБД обычно поддерживается единый интегрированный язык, содержащий все необходимые средства для работы с БД, начиная от ее создания, и обеспечивающий базовый пользовательский интерфейс с базами данных. Стандартным языком наиболее распространенных в настоящее время реляционных СУБД является язык SQL (Structured Query Language). В нескольких лекциях этого курса язык SQL будет рассматриваться достаточно подробно, а пока мы перечислим основные функции реляционной СУБД, поддерживаемые на "языковом" уровне (т.е. функции, поддерживаемые при реализации интерфейса SQL).

Прежде всего, язык SQL сочетает средства SDL и DML, т.е. позволяет определять схему реляционной БД и манипулировать данными. При этом именование объектов БД (для реляционной БД – именование таблиц и их столбцов) поддерживается на языковом уровне в том смысле, что компилятор языка SQL производит преобразование имен объектов в их внутренние идентификаторы на основании специально поддерживаемых

служебных таблиц-каталогов. Внутренняя часть СУБД (ядро) вообще не работает с именами таблиц и их столбцов.

Язык SQL содержит специальные средства определения ограничений целостности БД. Опять же, ограничения целостности хранятся в специальных таблицах-каталогах, и обеспечение контроля целостности БД производится на языковом уровне, т.е. при компиляции операторов модификации БД компилятор SQL на основании имеющихся в БД ограничений целостности генерирует соответствующий программный код.

Специальные операторы языка SQL позволяют определять так называемые представления БД, фактически являющиеся хранимыми в БД запросами (результатом любого запроса к реляционной БД является таблица) с именованными столбцами. Для пользователя представление является такой же таблицей, как любая базовая таблица, хранимая в БД, но с помощью представлений можно ограничить или наоборот расширить видимость БД для конкретного пользователя. Поддержание представлений производится также на языковом уровне.

Наконец, авторизация доступа к объектам БД производится также на основе специального набора операторов SQL. Идея состоит в том, что для выполнения операторов SQL разного вида пользователь должен обладать различными полномочиями. Пользователь, создавший таблицу БД, обладает полным набором полномочий для работы с этой таблицей. В число этих полномочий входит полномочие на передачу всех или части полномочий другим пользователям, включая полномочие на передачу полномочий. Полномочия пользователей описываются в специальных таблицах-каталогах, контроль полномочий поддерживается на языковом уровне.

Более точное описание возможных реализаций этих функций на основе языка SQL будет приведено в лекциях, посвященных языку SQL и его реализации.

Типовая организация современной СУБД

Естественно, организация типичной СУБД и состав ее компонентов соответствует рассмотренному нами набору функций. Напомним, что мы выделили следующие основные функции СУБД:

- управление данными во внешней памяти;
- управление буферами оперативной памяти;
- управление транзакциями;
- журнализация и восстановление БД после сбоев;
- поддержание языков БД.

Логически в современной реляционной СУБД можно выделить наиболее внутреннюю часть – ядро СУБД (часто его называют Data Base Engine), компилятор языка БД (обычно SQL), подсистему поддержки времени выполнения, набор утилит. В некоторых системах эти части выделяются явно, в других – нет, но логически такое разделение можно провести во всех СУБД.

Ядро СУБД отвечает за управление данными во внешней памяти, управление буферами оперативной памяти, управление транзакциями и журнализацию. Соответственно, можно выделить такие компоненты ядра (по крайней мере, логически, хотя в некоторых системах эти компоненты выделяются явно), как менеджер данных, менеджер буферов, менеджер транзакций и менеджер журнала. Как можно было понять из первой части этой лекции, функции этих компонентов взаимосвязаны, и для обеспечения корректной работы СУБД все эти компоненты должны взаимодействовать по тщательно продуманным и проверенным протоколам. Ядро СУБД обладает собственным интерфейсом, не доступным пользователям напрямую и используемым в программах, производимых компилятором SQL (или в подсистеме поддержки выполнения таких программ) и утилитах БД. Ядро СУБД является основной резидентной частью СУБД. При использовании архитектуры «клиент-сервер» ядро является основной составляющей серверной части системы.

Основной функцией компилятора языка БД является компиляция операторов языка БД в некоторую выполняемую программу. Основной проблемой реляционных СУБД является то, что языки этих систем (а это, как правило, SQL) являются непроцедурными, т.е. в операторе такого языка специфицируется некоторое действие над БД, но эта спецификация не является процедурой, а лишь описывает в некоторой форме условия совершения желаемого действия (вспомните примеры из первой лекции). Поэтому компилятор должен решить, каким образом выполнять оператор языка прежде, чем произвести программу. Применяются достаточно сложные методы оптимизации операторов, которые мы подробно рассмотрим в следующих лекциях. Результатом компиляции является выполняемая программа, представляемая в некоторых системах в машинных кодах, но более часто в выполняемом внутреннем машинно-независимом коде. В последнем случае реальное выполнение оператора производится с привлечением подсистемы поддержки времени выполнения, представляющей собой, по сути дела, интерпретатор этого внутреннего языка.

Наконец, в отдельные утилиты БД обычно выделяют такие процедуры, которые слишком накладно выполнять с использованием языка БД, например, загрузка и выгрузка БД, сбор статистики, глобальная проверка целостности БД и т.д. Утилиты программируются с использованием интерфейса ядра СУБД, а иногда даже с проникновением внутрь ядра.

Синтаксис языка Python

Python – высокоуровневый язык программирования общего назначения с динамической строгой типизацией и автоматическим управлением памятью, ориентированный на повышение производительности разработчика, читаемости кода и его качества, а также на обеспечение переносимости написанных на нём программ. Язык является полностью объектно-ориентированным – всё является объектами.

Необычной особенностью языка является выделение блоков кода пробельными отступами. Синтаксис ядра языка минималистичен, за счёт чего на практике редко возникает необходимость обращаться к документации. Сам же язык известен как интерпретируемый и используется в том числе для написания скриптов. Недостатками языка являются зачастую более низкая скорость работы и более высокое потребление памяти написанных на нём программ по сравнению с аналогичным кодом, написанным на компилируемых языках, таких как Си или C++.

Овладение данным языком не представляет существенной сложности для человека, ранее изучившего любой другой язык программирования и знакомого с аббревиатурой ООП. Остановимся на основных конструкциях (более полно с языком и его особенностями можно ознакомиться в сети Интернет – на сайте проекта www.python.org, в Википедии, при просмотре мастер-классов на Youtube и в TikTok).

Python поддерживает динамическую типизацию, то есть тип переменной определяется только во время исполнения. Поэтому вместо «присваивания значения переменной» лучше говорить о «связывании значения с некоторым именем». К примитивным типам в Python относятся булевый, целое число произвольной точности, число с плавающей запятой и комплексное число. Из контейнерных типов в Python встроены: строка, список, кортеж, словарь и множество. Все значения являются объектами, в том числе функции, методы, модули, классы.

Добавить новый тип можно либо написав класс (class), либо определив новый тип в модуле расширения (например, написанном на языке C). Система классов поддерживает наследование (одиночное и множественное) и метапрограммирование. Возможно наследование от большинства встроенных типов и типов расширений.

Имя (идентификатор) может начинаться с буквы любого алфавита в Юникоде любого регистра или подчёркивания, после чего в имени можно использовать и цифры. В качестве имени нельзя использовать ключевые

слова (их список можно узнать по `import keyword; print(keyword.kwlist)`) и нежелательно переопределять встроенные имена. Имена, начинающиеся с символа подчёркивания, имеют специальное значение.

В каждой точке программы интерпретатор имеет доступ к трём пространствам имён (то есть отображениям имён в объекты): локальному, глобальному и встроенному.

Области видимости имён могут быть вложенными друг в друга (внутри определяемой функции видны имена из окружающего блока кода). На практике с областями видимости и связыванием имён связано несколько правил «хорошего тона», о которых можно подробнее узнать из документации.

Одной из интересных синтаксических особенностей языка является выделение блоков кода с помощью отступов (пробелов или табуляций), поэтому в Python отсутствуют операторные скобки `begin/end`, как в языке Паскаль, или фигурные скобки, как в C. В качестве аналога `begin` или `{` выступает символ двоеточие «:».

В качестве символа, определяющего комментарии (необязательный фрагмент кода), используется символ решётка «#».

Набор операторов достаточно традиционен.

Условный оператор if (если).

При наличии нескольких условий и альтернатив применяется необязательный блок `elif` (сокр. от `else if`) который может повторяться в коде неограниченное число раз. Если ни одно из условий не было соблюдено, то выполняется необязательный блок `else` (иначе).

Пример:

```
gre_score = int(input("Введите текущий лимит: "))
per_grad = int(input("Введите кредитный рейтинг: "))

if per_grad > 70:
    if gre_score > 150:
        print("Поздравляем, вам выдан кредит")
```

```

else:
    print("У вас низкий кредитный лимит")
else:
    print("Извините, вы не имеете права на кредит")

```

Оператор цикла while.

Оператор while предназначен для организации циклического процесса в программе. В языке программирования Python оператор while используется в случаях, если количество повторений цикла заранее неизвестно (в отличие от оператора for). В операторе while следующий шаг итерации цикла определяется на основе истинности некоторого условия.

Примеры:

```

# сформировать список и инициализировать значениями
переменные
A=[1,3,5,8,-3,10]
i=0
summ=0 # искомая сумма

# цикл вычисления суммы
while i<len(A):
    summ=summ+A[i]
    i=i+1
print("summ = ", summ) # summ = 24

# Подсчет количества вхождений заданной строки в
# Входные данные
LIST = [ 'abc', 'bcd', 'xvm', 'abc', 'abd', 'bcd', 'abc' ]
ITEM = str(input('Введите строку: '))

# вычисление
i=0
k=0
while i<len(LIST):
    if LIST[i]==ITEM:

```

```

        k=k+1

    i=i+1
print ("k = ", k)

```

Оператор цикла for.

Цикл `for` присваивает итерируемой переменной каждое значение из предоставленного списка, массива или строки и повторяет код в теле цикла `for` для каждого установленного таким образом значения переменной-итератора. Функция `range()` возвращает последовательность целых чисел на основе переданных ей аргументов. Аргумент `start` — это первое значение в диапазоне. Если функция `range(start, stop[, step])` вызывается только с одним аргументом, то Python считает, что `start = 0`. Аргумент `stop` — это верхняя граница диапазона. Важно понимать, что само граничное значение не включается в последовательность.

Примеры:

```

# Простой пример цикла for
for i in [0, 1, 2, 3, 4, 5]:
    print(i, end="; ") # выведет: 0; 1; 2; 3; 4; 5;

# Использование range() с единственным аргументом
for i in range(6):
    print(i, end="; ") # выведет: 0; 1; 2; 3; 4; 5;

# Здесь используются все аргументы range()
for i in range(-2, 6, 2):
    print(i, end="; ") # выведет: -2; 0; 2; 4;

```

Операторы обработки исключений `try — except — else — finally`.

В программировании на Python обработка исключений позволяет программисту включить управление потоком. Использование `try-except` является наиболее распространенным и естественным способом обработки непредвиденных ошибок наряду со многими другими конструкциями

обработки исключений. Обработка ошибок или исключений в Python может быть осуществлена путем настройки исключений. Используя блок `try`, вы можете реализовать исключение и обработать ошибку внутри блока исключений. Всякий раз, когда код прерывается внутри блока `try`, обычный поток кода останавливается, и элемент управления переключается на блок `except` для обработки ошибки. Вызов исключений также допустимо в Python. Это означает, что вы можете бросить или вызвать исключение, когда это необходимо. Вы можете сделать это, просто вызвав в вашем коде `raise Exception('Test error!')`. Возникнувшее исключение прекратит текущее выполнение как обычно и пойдет дальше в стек вызовов, пока не будет обработано.

Примеры:

```
#Обработка ошибок, возникающих при работе с файлами
try:
```

```
    file = open('input-file', 'open mode')
```

```
except EOFError as ex:
```

```
    print("Ошибка конец файла.")
```

```
    raise ex
```

```
except IOError as e:
```

```
    print("Ошибка ввода-вывода")
```

```
    raise ex
```

#Использовать исключение без упоминания какого-либо атрибута исключения может быть полезно, если вы не имеете ни малейшего представления об исключении, которое может выдать ваша программа

```
try:
```

```
    file = open('input-file', 'open mode')
```

```
except:
```

```
    raise
```

Оператор определения класса `class`.

Оператор `class` создает новое определение класса. Имя класса сразу следует за ключевым словом `class`, после которого ставится двоеточие «:».

```
class ClassName:
    """Необязательная строка документации класса"""
    class_suite
```

У класса есть строка документации, к которой можно получить доступ через `ClassName.__doc__`.

`class_suite` состоит из частей класса, атрибутов данных и функции.

Конструктор класса вызывается через метод `__init__`.

Доступ к атрибутам класса осуществляется через использование оператора точка «.» после объекта класса. Доступ к классу можно получить используя имя переменной класса. Чтобы создать экземпляры классов, нужно вызвать класс с использованием его имени и передать аргументы, которые принимает метод `__init__`.

Пример:

```
class Employee:
    """Базовый класс для всех сотрудников"""
    emp_count = 0

    def __init__(self, name, salary):
        self.name = name
        self.salary = salary
        Employee.emp_count += 1

    def display_count(self):
        print('Всего сотрудников: %d' %
Employee.emp_count)

    def display_employee(self):
        print('Имя: {}. Зарплата: {}'.format(self.name,
self.salary))
```



```
# Это создаст первый объект класса Employee
emp1 = Employee("Андрей", 2000)
# Это создаст второй объект класса Employee
emp2 = Employee("Мария", 5000)
emp1.display_employee()
emp2.display_employee()
print("Всего сотрудников: %d" % Employee.emp_count)
```

Переменная `emp_count` – переменная класса, значение которой разделяется между экземплярами этого класса. Получить доступ к этой переменной можно через `Employee.emp_count` из класса или за его пределами.

Первый метод `__init__()` – специальный метод, который называют конструктором класса или методом инициализации. Его вызывает Python при создании нового экземпляра этого класса.

Объявляйте другие методы класса, как обычные функции, за исключением того, что первый аргумент для каждого метода `self`. Python добавляет аргумент `self` в список для вас; и тогда вам не нужно включать его при вызове этих методов.

Python автоматически удаляет ненужные объекты (встроенные типы или экземпляры классов), чтобы освободить пространство памяти. С помощью процесса ‘Garbage Collection’ Python периодически восстанавливает блоки памяти, которые больше не используются.

Сборщик мусора Python запускается во время выполнения программы и тогда, когда количество ссылок на объект достигает нуля. С изменением количества обращений к нему, меняется количество ссылок.

Когда объект присваивают новой переменной или добавляют в контейнер (список, кортеж, словарь), количество ссылок объекта увеличивается. Количество ссылок на объект уменьшается, когда он удаляется с помощью `del`, или его ссылка выходит за пределы видимости. Когда количество ссылок достигает нуля, Python автоматически собирает его.

Обычно вы не заметите, когда сборщик мусора уничтожает экземпляр и очищает свое пространство. Но классом можно реализовать специальный метод `__del__()`, называемый деструктором. Он вызывается, перед уничтожением экземпляра. Этот метод может использоваться для очистки любых ресурсов памяти.

Классы наследники объявляются так, как и родительские классы. Только, список наследуемых классов, указан после имени класса.

Пример:

```
class Parent: # объявляем родительский класс
    parent_attr = 100

    def __init__(self):
        print('Вызов родительского конструктора')

    def parent_method(self):
        print('Вызов родительского метода')

    def set_attr(self, attr):
        Parent.parent_attr = attr

    def get_attr(self):
        print('Атрибут родителя:
{}'.format(Parent.parent_attr))

class Child(Parent): # объявляем класс наследник
    def __init__(self):
        print('Вызов конструктора класса наследника')

    def child_method(self):
        print('Вызов метода класса наследника')

c = Child() # экземпляр класса Child
```

```

c.child_method() # вызов метода child_method
c.parent_method() # вызов родительского метода
parent_method
c.set_attr(200) # еще раз вызов родительского метода
c.get_attr() # снова вызов родительского метода

```

Вы можете использовать функции `issubclass()` или `isinstance()` для проверки отношений двух классов и экземпляров.

Логическая функция `issubclass(sub, sup)` возвращает значение `True`, если данный подкласс `sub` действительно является подклассом `sup`.

Логическая функция `isinstance(obj, Class)` возвращает `True`, если `obj` является экземпляром класса `Class` или является экземпляром подкласса класса.

Атрибуты класса могут быть не видимыми вне определения класса. Вам нужно указать атрибуты с `__` вначале, и эти атрибуты не будут вызваны вне класса.

Пример:

```

class JustCounter:
    __secret_count = 0

    def count(self):
        self.__secret_count += 1
        print(self.__secret_count)

counter = JustCounter()
counter.count()
counter.count()
print(counter.__secret_count)

```

Оператор определения функции, метода или генератора `def`.

Внутри возможно применение `return` (возврат) для возврата из функции или метода, а в случае генератора – `yield` (давать). Функция в

`python` – объект, принимающий аргументы и возвращающий значение. Генераторы – это функции, сохраняющие внутреннее состояние: значения локальных переменных и текущую. Генераторы могут использоваться как итераторы для структур данных и для ленивых вычислений.

При вызове генератора функция немедленно возвращает объект-итератор, который хранит текущую точку исполнения и состояние локальных переменных функции. При запросе следующего значения (посредством метода `next()`, неявно вызываемого в цикле `for`) генератор продолжает исполнение функции от предыдущей точки остановки до следующего оператора `yield` или `return`.

Примеры:

```
#пример функции с одним аргументом
def newfunc(n):
    def myfunc(x):
        return x + n
    return myfunc

#пример генератора
>>> print(sum(i for i in range(1, 100) if i % 2 != 0))
2500
```

Функция может принимать произвольное количество аргументов или не принимать их вовсе. Также распространены функции с произвольным числом аргументов, функции с позиционными и именованными аргументами, обязательными и необязательными.

Пример:

```
def func(a, b, c=2): # c - необязательный аргумент
    return a + b + c
```

Функция также может принимать переменное количество позиционных аргументов, тогда перед именем ставится «*». Функция может

принимать и произвольное число именованных аргументов, тогда перед именем ставится «**».

Примеры:

```
>>> def func(*args):
...     return args
...
>>> func(1, 2, 3, 'abc')
(1, 2, 3, 'abc')
>>> func()
()
>>> func(1)
(1,)

>>> def func(**kwargs):
...     return kwargs
...
>>> func(a=1, b=2, c=3)
{'a': 1, 'c': 3, 'b': 2}
>>> func()
{}
>>> func(a='python')
{'a': 'python'}
```

Анонимные функции могут содержать лишь одно выражение, но и выполняются они быстрее. Анонимные функции создаются с помощью инструкции `lambda`. Кроме этого, их не обязательно присваивать переменной, как делали мы инструкцией `def func()`:

Пример:

```
>>> func = lambda x, y: x + y
>>> func(1, 2)
3
>>> func('a', 'b')
'ab'
>>> (lambda x, y: x + y)(1, 2)
```

```
3
>>> (lambda x, y: x + y)('a', 'b')
'ab'
```

lambda функции, в отличие от обычной, не требуется инструкция return, а в остальном, она ведет себя точно так же.

Пример:

```
>>> func = lambda *args: args
>>> func(1, 2, 3, 4)
(1, 2, 3, 4)
```

Оператор pass

Оператор pass ничего не делает. Используется для пустых блоков кода.

Синтаксис операций

Состав, синтаксис, ассоциативность и приоритет операций достаточно привычны для языков программирования и призваны минимизировать употребление скобок. Если сравнивать с математикой, то приоритеты операторов зеркалируют соответствующие в математике, при этом оператор присвоения значения = соответствует типографскому \leftarrow . Хотя приоритеты операций позволяют не использовать скобки во многих случаях, на анализ больших выражений может тратиться лишнее время, в результате чего в таких случаях выгоднее явно расставлять скобки.

Отдельно стоит упомянуть операцию форматирования для строк (работает по аналогии с функцией printf() из Си), которая использует тот же символ, что и взятие остатка от деления. Кроме того, логические операции (or и and) являются ленивыми: если для вычисления значения операции достаточно первого операнда, этот операнд и является результатом, в противном случае вычисляется второй операнд логической операции. Это основывается на свойствах алгебры логики: например, если один аргумент

операции «ИЛИ» (or) является истиной, то и результат этой операции всегда является истиной. В случае, если второй операнд является сложным выражением, это позволяет сократить издержки на его вычисление. Этот факт широко использовался до версии 2.5 вместо условной конструкции.

Примеры:

```
>>> str_var = "world"
>>> int_var = 2021
>>> print("Hello, %s " % str_var + "%d" % int_var) #в
заданном %s месте выведется строка, к которой добавится значение
из %d
```

```
Hello, world 2021
```

```
>>> a = 200
>>> b = 200
>>> print(a < b and "меньше" or "больше или равно")
больше или равно
```

```
>>> a = 100
>>> b = 200
>>> print(a < b and "меньше" or "больше или равно")
меньше
```

О Python

Ознакомившись с приведенной выше информацией об основах программирования на языке Python и обладая начальными навыками программирования, можно достаточно эффективно читать чужой код и писать свой. Важно понимать, что компьютер оперирует числами и совершает над ними различные операции, а всё, что мы видим на экране ЭВМ – это не магия и не волшебство, а всего лишь результат интерпретации инструкций, заданных программистом. Так, любой интерфейс состоит из простых элементов-кирпичиков, которые описаны в соответствующих библиотеках языка программирования. Помимо стандартной библиотеки

существует множество библиотек, предоставляющих интерфейс ко всем системным вызовам на разных платформах; в частности, на платформе Win32 поддерживаются все вызовы Win32 API, а также COM в объёме не меньшем, чем у Visual Basic или Delphi. Количество прикладных библиотек для Python в самых разных областях без преувеличения огромно (веб, базы данных, обработка изображений, обработка текста, численные методы, приложения операционной системы и т. д.). То, какие библиотеки будут подключены для реализации своей идеи в виде приложения, определяет сам программист.

Язык Python в равной степени успешно может строить свое взаимодействие с пользователем через консоль (что мы и наблюдали в изложенных выше примерах), либо через элементы графического интерфейса.

С Python поставляется библиотека tkinter на основе Tcl/Tk для создания кроссплатформенных программ с графическим интерфейсом.

Существуют расширения, позволяющие использовать все основные библиотеки графических интерфейсов – wxPython, основанное на библиотеке wxWidgets, PyGObject для GTK, PyQt и PySide для Qt и другие. Некоторые из них также предоставляют широкие возможности по работе с базами данных, графикой и сетями, используя все возможности библиотеки, на которой основаны.

Для создания игр и приложений, требующих нестандартного интерфейса, можно использовать библиотеку Pygame. Она также предоставляет обширные средства работы с мультимедиа: с её помощью можно управлять звуком и изображениями, воспроизводить видео. Предоставляемое pygame аппаратное ускорение графики OpenGL имеет более высокоуровневый интерфейс по сравнению с PyOpenGL, копирующей семантику C-библиотеки для OpenGL. Есть также PyOgre, обеспечивающая привязку к Ogre – высокоуровневой объектно-ориентированной библиотеке 3D-графики. Кроме того, существует

библиотека `pythonOCC`, обеспечивающая привязку к среде 3D-моделирования и симуляции `OpenCascade`.

Подключить модуль можно с помощью инструкции `import`. После ключевого слова `import` указывается название модуля. Одной инструкцией можно подключить несколько модулей, хотя этого не рекомендуется делать, так как это снижает читаемость кода.

Пример:

```
>>> import time, random
>>> time.time()
1376047104.056417
>>> random.random()
0.9874550833306869
```

После импортирования модуля его название становится переменной, через которую можно получить доступ к атрибутам модуля. Например, можно обратиться к константе `e`, расположенной в модуле `math`:

```
>>> import math
>>> math.e
2.718281828459045
```

Если название модуля слишком длинное, или оно вам не нравится по каким-то другим причинам, то для него можно создать псевдоним, с помощью ключевого слова `as`.

Пример:

```
>>> import math as m
>>> m.e
2.718281828459045
```

Подключить определенные атрибуты модуля можно с помощью инструкции `from`. Она имеет несколько форматов:

```
from <Название модуля> import <Атрибут 1> [ as <Псевдоним
1> ], [<Атрибут 2> [ as <Псевдоним 2> ] ...]
from <Название модуля> import *
```

Первый формат позволяет подключить из модуля только указанные вами атрибуты. Для длинных имен также можно назначить псевдоним, указав его после ключевого слова `as`.

Пример:

```
>>> from math import e, ceil as c
>>> e
2.718281828459045
>>> c(4.6)
5
```

Второй формат инструкции `from` позволяет подключить все (точнее, почти все) переменные из модуля. Для примера импортируем все атрибуты из модуля `sys`:

```
>>> from sys import *
>>> version
'3.3.2 (v3.3.2:d047928ae3f6, May 16 2013, 00:03:43) [MSC
v.1600 32 bit (Intel)]'
>>> version_info
sys.version_info(major=3,          minor=3,          micro=2,
releaselevel='final', serial=0)
```

Следует заметить, что не все атрибуты будут импортированы. Если в модуле определена переменная `__all__` (список атрибутов, которые могут быть подключены), то будут подключены только атрибуты из этого списка. Если переменная `__all__` не определена, то будут подключены все атрибуты, не начинающиеся с нижнего подчёркивания. Кроме того, необходимо учитывать, что импортирование всех атрибутов из модуля может нарушить пространство имен главной программы, так как переменные, имеющие одинаковые имена, будут перезаписаны.

Средства программирования

IDE (или интегрированная среда разработки) – это программа, предназначенная для разработки программного обеспечения. Как следует из названия, IDE объединяет несколько инструментов, специально

предназначенных для разработки. Эти инструменты обычно включают редактор, предназначенный для работы с кодом (например, подсветка синтаксиса и автодополнение); инструменты сборки, выполнения и отладки; и определённую форму системы управления версиями.

Большинство IDE поддерживают множество языков программирования и имеют много функций, из-за чего могут быть большими, занимать много времени для загрузки и установки и требуют глубоких знаний для правильного использования.

С другой стороны, есть редакторы кода, которые представляют собой текстовый редактор с подсветкой синтаксиса и возможностями форматирования кода. Большинство хороших редакторов кода могут выполнять код и использовать отладчик, а лучшие даже могут взаимодействовать с системами управления версиями. По сравнению с IDE, хороший редактор кода, как правило, легче и быстрее, но зачастую ценой меньшей функциональности.

Какие требования предъявляются для среды разработки? Набор функций разных сред может отличаться, но есть набор базовых вещей, упрощающих программирование:

- Сохранение файлов. Если IDE или редактор не дают вам возможности сохранить работу и позже всё открыть в том же состоянии, в котором оно было во время закрытия, то не такая уж это и IDE;
- Запуск кода из среды. То же самое, если вам нужно выйти из среды для запуска кода, то это не более, чем простой текстовый редактор;
- Поддержка отладки. Возможность пошагово выполнить код является базовой функцией всех IDE и большинства хороших редакторов кода;
- Подсветка синтаксиса. Возможность быстро найти ключевые слова, переменные и прочее делает чтение и понимание кода на порядок проще;
- Автоматическое форматирование кода. Любой редактор или IDE, который действительно таковым является, распознает двоеточие

после `while` или `for` выражения и автоматически сделает отступ на следующей строке.

Перечисленным выше требованиям отвечают такие IDE, как: связка Eclipse + PyDev, PyCharm, Visual Studio Code и другие. Выбор среды программирования – это выбор, базирующийся на индивидуальных предпочтениях, эстетическом чувстве и условиях среды. Какую бы IDE не выбрал разработчик, конечным результатом будет скомпилированный файл расширения *.py.

Пример выполнения задания

Первое, что необходимо сделать для программирования на Python, это скачать сам язык программирования с официального сайта <https://www.python.org>. Поскольку для начинающего программиста писать весь код в текстовом редакторе и выполнять его из командной строки разработчика не очень удобно, вторым этапом следует установить понравившуюся IDE.

В рассматриваемом случае разработка будет вестись в среде Visual Studio Code. Для начала необходимо скачать (<https://code.visualstudio.com>) и установить среду разработки (рис.8.1).

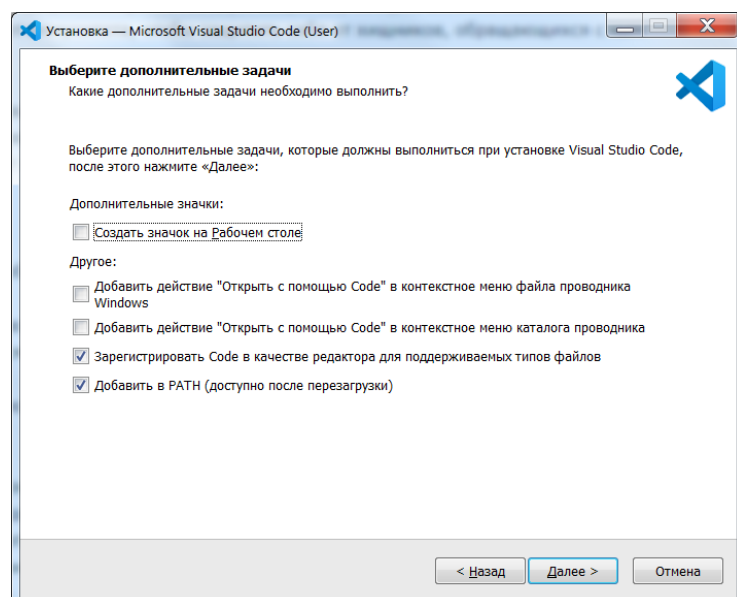


Рисунок 8.1 – Установка Visual Studio Code

После установки и успешного запуска необходимо установить соответствующие расширения IDE для Python, необходимые и облегчающие программирование – такие как Python, Python for VSCode, Python Preview. Установка осуществляется по нажатию на Install (Установить).

Следует подготовить место для сохранения проекта и поместить в выбранную папку все необходимые графические материалы (Python может обращаться к ресурсам и вне выбранной папки, но размещение ресурсов в одном месте с проектом позволяет сократить и упростить код). В рассматриваемом примере используются пять графических ресурсов формата *.gif, в которых хранятся изображения для кнопок на панели приложения. Для разработки интерфейса этих кнопок можно использовать встроенные в операционную систему графические редакторы.

После выполнения всех указанных выше шагов, в редакторе кода необходимо набрать код программы, приведенной ниже, и запустить ее на отладку через соответствующий пункт меню.

Внешний вид приложения приведён на рис. 8.2-8.6. Код приведен в приложении 3. Следует учитывать, что внешние файлы в примере хранятся в той же директории, что и файл ru. Внешние файлы – это изображения формата gif (6 файлов с иконками кнопок и 1 с фоном), а также база данных.

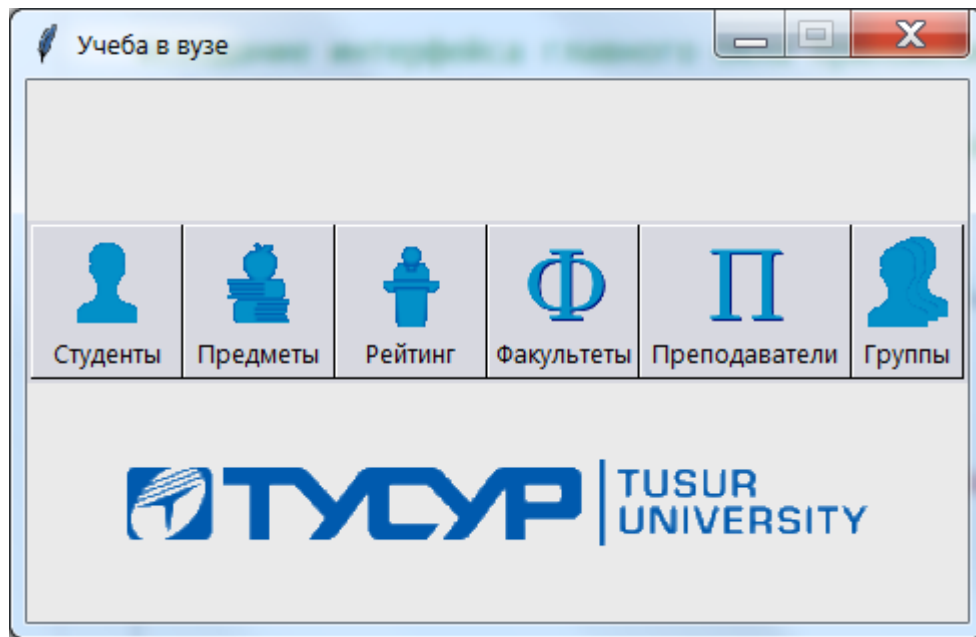


Рисунок 8.2. – Внешний вид приложения с полным функционалом

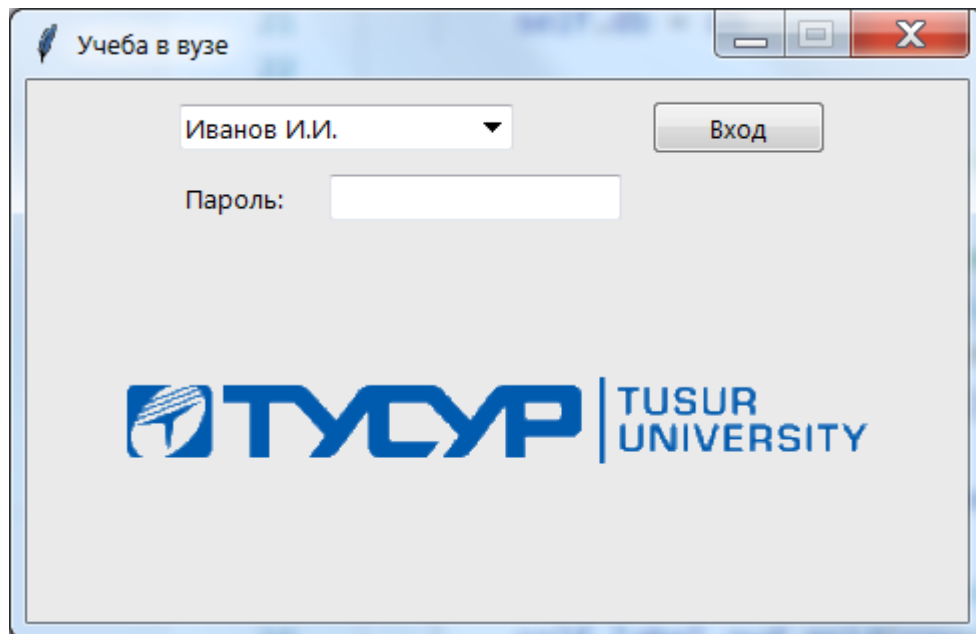


Рисунок 8.3 – Внешний вид приложения в режиме запроса учетных данных

Таблица rate

Позиция: 1

ФИО: Боширов А.А.

Дисциплина: Физика

Макс.рейтинг: 100

Текущий рейтинг: 34

Редактировать Добавить Удалить Заккрыть

ID	Макс.рейтинг	Факт.рейтинг	ФИО	Дисциплина
1	100	34	Боширов А.А.	Физика
2	100	61	Петров А.А.	Физика
4	100	100	Яблоков А.С.	Алгебра
11	100	45	Боширов А.А.	Физика
12	100	22	Боширов А.А.	Начертательная
13	100	22	Боширов А.А.	Химия
14	100	11	Тыквина М.Н.	Алгебра
15	100	10	Орлова Ю.И.	Физика
16	100	12	Васенев С.А.	История
17	100	13	Петров А.А.	Начертательная

Рисунок 8.4 – Внешний вид приложения после открытия справочника
«Рейтинг»

Таблица students

Позиция: 5

Обновить

ФИО: Яблоков А.С.

Курс: 2 курс

Адрес: Лыткина, 1-201

Примечание: заочно

Группа: 6

Редактировать Добавить Поиск Удалить Заккрыть

ID	ФИО	Адрес	Примечание	Курс	Группа
1	Петров А.А.	Кирова, 2	очно	3 курс	1
2	Боширов А.А.	Кирова, 2	заочно	1 курс	5
4	Орлов Ю.М.	Лыткина, 18Б, 202	очно	1 курс магистр	6
5	Яблоков А.С.	Лыткина, 1-201	заочно	2 курс	6
6	Яблоков А.В.	Лыткина, 18Б, 202	очно	1 курс магистр	6
7	Васенев С.А.	Лыткина, 18Б, 203	очно	4 курс	6
8	Жиров О.В.	Лыткина, 18Б, 204	очно	1 курс магистр	4
9	Орлова Ю.И.	Лыткина, 18Б, 202	очно	1 курс магистр	6
10	Тыквина М.Н.	Лыткина, 18Б, 205	очно	1 курс магистр	6
11	Хренин О.М.	Лыткина, 18Б, 206	очно	1 курс магистр	4

Рисунок 8.5 – Внешний вид приложения после открытия справочника
«Студенты»

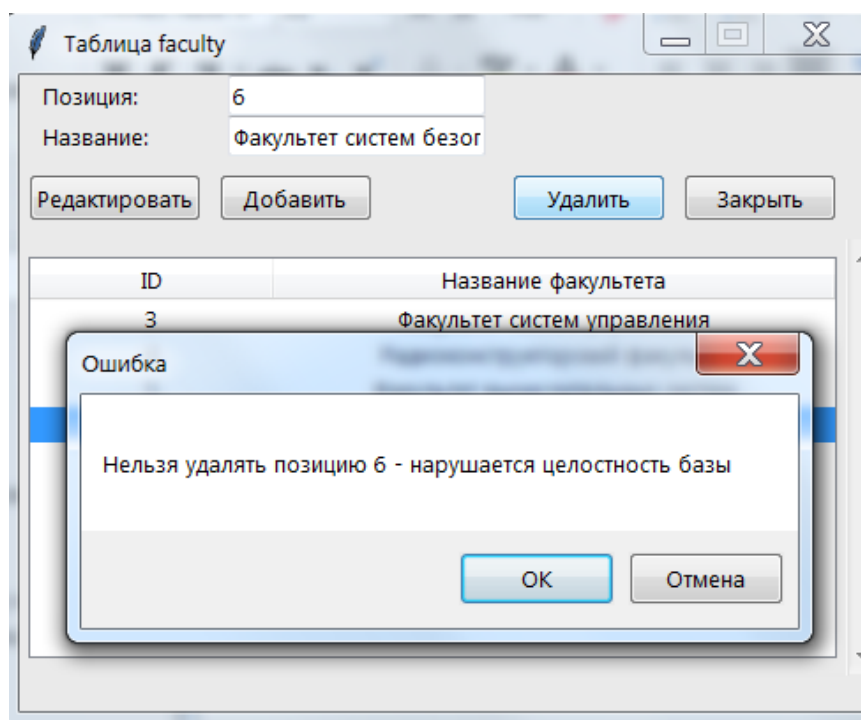


Рисунок 8.6 – Окно предотвращения нарушения ссылочной целостности

Задание

Разработать СУБД для базы данных, построенной в 7-й лабораторной работе. Программные средства для проектирования СУБД подобрать самостоятельно.

9. Организация самостоятельной работы

9.1 Рабочая программа и план обучения

Рабочая программа дисциплины – программа освоения учебного материала, соответствующая требованиям государственного образовательного стандарта высшего образования и учитывающая специфику подготовки студентов по выбранному направлению.

В рабочей программе дана общая информация по дисциплине: цели и задачи дисциплины, место в образовательной программе, требование к результатам освоения дисциплины (на формирование каких компетенций

направлена), объем и виды учебной работы, формы аттестации, основная и дополнительная литература, примерные оценочные материалы.

Рабочая программа позволяет представить объем предстоящей работы по дисциплине и составить план самостоятельного обучения. План должен включать следующие важные составляющие:

- 1) изучение теоретического материала дисциплины, освоение терминологии, ознакомление с классификациями и др. (для этого необходимо оценить объем материалов электронного курса и дополнительной литературы);
- 2) подготовка и выполнение контрольных работ;
- 3) подготовка и прохождение итоговой аттестации по дисциплине.

Рабочая программа дисциплины доступна из рабочего учебного плана соответствующей основной профессиональной образовательной программы (<https://edu.tusur.ru/opops>).

9.2 Теоретический материал

Учебное пособие (теоретический материал электронного курса) содержит основную теоретическую информацию по дисциплине. Деление на главы (темы) позволяет проследить логику изложения материала и равномерно распределить его в своем плане обучения. Составление конспектов по каждой теме поможет закрепить материал. Особое внимание следует обращать на блоки теоретического материала, выделенные пиктограммами. Данный материал обязателен для усвоения и запоминания. Рекомендуется уделить внимание содержанию таблиц и рисункам, являющимся источниками важной наглядной и структурированной информации.

Самостоятельная работа предполагает обязательное изучение рекомендованной литературы, выполнение практических заданий и закрепление усвоенного материала. Для углубленного изучения тем рекомендуется использовать дополнительные литературные источники.

Студенты ТУСУР имеют доступ к полным текстам изданий электронных библиотечных систем «ЮРАЙТ» и «ЛАНЬ».

9.3 Рекомендации по работе с учебной и научной литературой

Самостоятельная работа по освоению теоретического материала учебного пособия и другой учебной и научной литературы (а также самостоятельное теоретическое исследование изучаемых проблем) – это важнейшее условие формирования научного способа познания. Таким образом, чтение научного текста является частью познавательной деятельности, главная цель которой – извлечение из текста необходимой информации. Осознание читающим внутренней установки при обращении к печатному слову (выделение нужных сведений, усвоение информации полностью или частично, критический анализ материала и т. п.) определяет эффективность осуществляемого действия.

Грамотная работа с книгой, особенно если речь идет о научной литературе, предполагает *соблюдение ряда правил*, для овладения которыми необходимо учиться. Прежде всего, при такой работе невозможен формальный, поверхностный подход. Главное правило при работе над книгой – не механическое заучивание, не простое накопление цитат, выдержек, а сознательное усвоение прочитанного, осмысление его, стремление дойти до сути.

Изучение книги должно происходить в определенной последовательности. Вначале следует ознакомиться с оглавлением (содержанием), предисловием, введением. Это позволяет получить общее представление о структуре и вопросах, которые рассматриваются в книге. Затем можно переходить непосредственно к чтению. Чтобы получить цельное представление о книге, первый раз стоит прочитать ее от начала до конца. При повторном чтении происходит постепенное глубокое осмысление каждой главы, критического материала и позитивного изложения; выделение основных идей, системы аргументов, наиболее ярких

примеров и т. д. Непременным правилом чтения должно быть выяснение незнакомых слов, терминов, выражений, неизвестных имен, названий. Важная роль в связи с этим принадлежит библиографической подготовке студентов. Она включает в себя умение активно, быстро пользоваться научным аппаратом книги, справочными изданиями, каталогами; вести поиск необходимой информации, обрабатывать и систематизировать ее.

Рекомендуется использовать четыре основные установки в чтении научного текста:

- информационно-поисковая (задача – найти, выделить искомую информацию);
- усваивающая (усилия читателя направлены на то, чтобы как можно полнее осознать и запомнить как сами сведения, излагаемые автором, так и всю логику его рассуждений);
- аналитико-критическая (читатель стремится критически осмыслить материал, проанализировав его, определив свое отношение к нему);
- творческая (создает у читателя готовность в том или ином виде – как отправной пункт для своих рассуждений, как образ для действия по аналогии и т. п. – использовать суждения автора, ход его мыслей, результат наблюдения, разработанную методику, дополнить их, подвергнуть новой проверке).

Научная методика работы с литературой предусматривает фиксирование прочитанной информации в систематизированных записях разного рода. Это позволяет привести в систему знания, полученные при чтении, сосредоточить внимание на главных положениях, зафиксировать, закрепить их в памяти, а при необходимости вновь обратиться к ним.

Основные виды систематизированной записи прочитанного:

- тезирование – лаконичное воспроизведение основных утверждений автора без привлечения фактического материала;
- цитирование – дословное выписывание из текста выдержек, извлечений, наиболее существенно отражающих ту или иную мысль автора;

– конспектирование – краткое и последовательное изложение содержания прочитанного.

Методические рекомендации по составлению конспекта

Внимательно прочитайте текст. Уточните в глоссарии или справочной литературе непонятные слова. Выделите главное, составьте план, представляющий собой перечень заголовков, подзаголовков, вопросов, последовательно раскрываемых затем в конспекте. Это первый элемент конспекта.

Вторым элементом конспекта являются тезисы. Тезис – это кратко сформулированное положение. Для лучшего усвоения и запоминания материала следует записывать тезисы своими словами. Тезисы, выдвигаемые в конспекте, нужно доказывать.

Третий элемент конспекта – основные доводы, доказывающие истинность рассматриваемого тезиса. В конспекте могут быть положения и примеры.

Законспектируйте материал, четко следуя пунктам плана. При конспектировании старайтесь выразить мысль своими словами. Записи следует вести четко, ясно. Грамотно записывайте цитаты. Цитируя, учитывайте лаконичность, значимость мысли.

При оформлении конспекта необходимо стремиться к емкости каждого предложения. Мысли автора книги следует излагать кратко, заботясь о стиле и выразительности написанного.

Число дополнительных элементов конспекта должно быть логически обоснованным, записи должны распределяться в определенной последовательности, отвечающей логической структуре произведения. Для уточнений и дополнений необходимо оставлять поля.

Овладение навыками конспектирования требует от студента целеустремленности, повседневной самостоятельной работы. Конспект ускоряет повторение материала, экономит время при повторном обращении к работе.

9.4 Консультации

В связи с тем, что не всегда возможно самостоятельно найти ответы на возникшие в ходе изучения дисциплины вопросы, существует возможность задать вопрос преподавателю, записавшись к нему на консультацию. Вопросы можно задавать как по теоретическому материалу, так и по подготовке, оформлению отчетов любого вида работ.

Предусмотрено несколько ресурсов, где можно задать вопрос:

- на учебном форуме из страницы электронного курса;
- написав письмо с любого другого почтового сервера на адрес преподавателя.

Также существует возможность обсудить любой учебный вопрос на форуме с другими студентами (внутри электронного курса), коллективно найти решение. Однако следует помнить, что мнения других студентов могут быть ошибочными, и критически относиться к высказанным идеям.

9.5 Практические и лабораторные работы

Лабораторные и практические работы являются основными видами деятельности, направленными на экспериментальное подтверждение теоретических положений и формирование учебных и профессиональных практических умений.

Выполнение студентами лабораторных и практических работ необходимо:

- для формирования практических умений в соответствии с требованиями к уровню подготовки студентов, установленными рабочей программой дисциплины по конкретным разделам (темам);
- обобщения, систематизации, углубления, закрепления полученных теоретических знаний;
- совершенствования умений применять полученные знания на практике, реализации единства интеллектуальной и практической деятельности;

- развития интеллектуальных умений;
- выработки при решении поставленных задач таких профессионально значимых качеств, как самостоятельность, ответственность, точность, творческая инициатива.

Практическая работа

В курсе может предусмотрено выполнение самостоятельной практической работы. Ее целью является развитие умений и закрепление навыков в решении учебных и профессиональных задач, подготовке к промежуточной и итоговой аттестации. Перед тем как приступить к выполнению работы, следует ознакомиться с образцом решения, предоставленным преподавателем.

Лабораторная работа

Задание к текстовой лабораторной работе располагается в учебных материалах. Для выполнения работы необходимо повторить уже усвоенные разделы учебного пособия по теме, изучить дополнительный материал, который приведен в методических указаниях и дополнительной литературе. Для успешного выполнения лабораторных работ рекомендуется:

- подробно изучить текст задания и определить свой вариант;
- обязательно оформить отчет в соответствии с представленными требованиями;
- отправить его на проверку преподавателю.

В отличие от самостоятельной практической работы лабораторные работы оцениваются, получение зачета является обязательным условием для аттестации по дисциплине.

9.6 Контрольные мероприятия

Самоконтроль

Самоконтроль – один из важнейших факторов, обеспечивающих самостоятельную деятельность обучающихся. Самоконтроль необходим для проверки остаточных знаний по итогам изучения каждой темы. Большинство курсов снабжены двумя уровнями самоконтроля. В текстовом материале после каждой главы представлен перечень вопросов и заданий, позволяющих проконтролировать усвоение ключевых вопросов темы. Если ответы на вопросы вызывают затруднение, следует еще раз перечитать материал.

Второй уровень самоконтроля – тестовые задания в электронном курсе. Тестирование является наиболее эффективной формой контроля и имеет ряд преимуществ перед другими формами контроля:

- затрата небольшого количества времени на выполнение тестовых заданий;
- возможность оперативного получения информации о степени усвоения знаний;
- возможность оперативной корректировки знаний и умений и др.

Формат электронных тестов максимально приближен к тем заданиям, которые необходимо будет выполнить на итоговой аттестации по дисциплине.

Контрольная работа

Цель выполнения контрольной работы – закрепить знания, полученные студентами при изучении теоретического материала, а также, в ряде случаев, отработать навыки решения практических задач. К выполнению контрольной работы следует серьезно подготовиться, повторив весь теоретический материал и потренировавшись в решении задач.

Учебный план дисциплины «Спецкурс» предполагает выполнение двух компьютерных тестовых контрольных работ, задания на которые размещены в электронном курсе.

В отличие от самоконтроля, контрольные работы оцениваются, получение зачета является обязательным условием для аттестации по дисциплине.

9.7 Итоговая аттестация

Для получения зачета достаточно успешно выполнить все оцениваемые работы по дисциплине «Спецкурс» и выполнить итоговое тестирование.

Наиболее ответственным этапом в обучении студентов является экзаменационная сессия, во время нее студенты отчитываются о выполнении учебной программы, об уровне и объеме полученных знаний. Это государственная отчетность студентов за изучение учебной дисциплины, поэтому ответственность за успешную сдачу экзаменационной сессии велика. На сессии студенты сдают экзамены и дифференцированные зачеты.

10. Рекомендуемая литература

Настоящее методическое пособие содержит достаточную информацию для выполнения лабораторных работ. Для более подробного рассмотрения излагаемых вопросов, предлагается обратиться к литературе, список которой приводится ниже.

1. Орлов С.А. Технологии разработки программного обеспечения. Разработка сложных программных систем : Учебное пособие для вузов / Сергей Александрович Орлов. - СПб. : Питер, 2002.

2. Буч Г. Введение в UML от создателей языка: руководство пользователя / Г. Буч, Д. Рамбо, И. Якобсон. - 2-е изд. - М. : ДМК Пресс, 2012. - 494 с.
3. Черемных С.В. Моделирование и анализ систем. IDEF-технологии : Практикум / С. В. Черемных, И. О. Семенов, В. С. Ручкин. - М. : Финансы и статистика, 2005. - 188[4] с.
4. Маклаков С.В. BPwin и ERwin. CASE-средства разработки информационных систем. – М.: Диалог-МИФИ, 2001, 256 с.
5. Марка Д.А., МакГоуэн К. Методология структурного анализа и проектирования SADT. – М.: Метатехнология, 1993

Также возможно воспользоваться каталогом электронных библиотек и поискать информацию по UML, SADT и системному анализу и выбрать стилистически подходящие обучающемуся издания.

Приложение 1 – Список тем для лабораторных работ

- 1) Кафе
- 2) Железнодорожная касса
- 3) Автосервис
- 4) Больница
- 5) Ветеринарная клиника
- 6) Спартакиада
- 7) Конкурс красоты
- 8) Библиотека
- 9) Авиакасса
- 10) Магазин промышленных товаров
- 11) Банк
- 12) Прачечная
- 13) Речной порт
- 14) Гостиница
- 15) Фотосалон
- 16) Морг
- 17) Политическая партия
- 18) Отдел кадров
- 19) Аэропорт
- 20) Компьютерный магазин

Приложение 2 – Краткий глоссарий терминов ООП

CRC-карточки, CRC cards. CRC - Class/Responsibilities/Collaborators, Класс/Ответственности/Сотрудники; простое, но достаточно эффективное средство мозгового штурма при выявлении ключевых абстракций и механизмов.

абстрактная операция, abstract operation. Объявленная, но не реализованная операция в абстрактном классе. В C++ абстрактные операции объявляются как чисто виртуальные функции-члены.

абстрактный класс, abstract class. Класс, который не может иметь экземпляров. Абстрактный класс пишется в предположении, что его конкретные подклассы дополняют его структуру и поведение, скорее всего, реализовав абстрактные операции.

абстракция, abstraction. Существенные характеристики объекта, которые отличают его от всех других объектов и четко определяют его концептуальные границы для наблюдателя. Абстрагирование – процесс выявления абстракций. Один из основных элементов объектной модели.

агент, agent. Объект, который подвергается воздействию со стороны и сам воздействует на другие объекты. Обычно агенты создаются для выполнения некоторой работы по поручению актеров или других агентов.

актер, actor. Объект, воздействующий на другие объекты, но сам не подвергающийся воздействию с их стороны. В некоторых контекстах то же самое, что активный объект.

активный объект, active object. Объект, которому выделен свой поток управления.

алгоритмическая декомпозиция, algorithmic decomposition. Процесс разделения системы на части, каждая из которых отражает этап общего процесса. Применение структурного подхода к проектированию приводит к алгоритмической декомпозиции, которая фокусируется на потоке управления в системе.

архитектура модулей, module architecture. Граф, вершины которого соответствуют модулям, а ребра - отношениям модулей между собой. Архитектура модулей системы представляется совокупностью диаграмм модулей.

архитектура процессов, process architecture. Граф, вершины которого соответствуют процессорам и устройствам, а ребра – соединениям между

ними. Для описания архитектуры процессов системы используются диаграммы процессов.

архитектура, architecture. Логическая и физическая структура системы, сформированная всеми стратегическими и тактическими проектными решениями.

ассоциация, association. Отношение, означающее некоторую смысловую связь между классами.

атрибут, attribute. Часть составного объекта (агрегата).

базовый класс, base class. Наиболее общий класс в какой-либо структуре классов. В большинстве приложений есть несколько таких корневых классов. В некоторых языках программирования определяется всеобщий базовый класс, который является суперклассом для всех остальных классов.

блокирующий объект, blocking object. Пассивный объект, способный работать в многопоточном окружении. Вызов операции блокирующего объекта блокирует клиента на все время операции.

видимость, visibility. Способность одной абстракции видеть другую и, таким образом, ссылаться на ее ресурсы извне. Абстракции видимы друг другу, только если они находятся в одном пространстве имен. Контроль экспорта может еще более ограничить доступ к видимым абстракциям.

виртуальная функция, virtual function. Какая-либо операция над объектом. Виртуальная функция может быть переопределена в подклассах, следовательно, ее реализация определяется всем множеством методов, объявленных во всех классах дерева наследования. Термины «обобщенная функция» и «виртуальная функция» взаимозаменяемы.

временная сложность, time complexity. Относительное или абсолютное время, за которое выполняется операция.

действие, action. Некое происшествие в системе, требующее, с практической точки зрения, нулевого времени для своего завершения. Действием может быть вызов операции, запуск другого события, начало или остановка деятельности.

делегирование, delegation. При делегировании один объект, ответственный за операцию, передает выполнение этой операции другому объекту.

деструктор, destructor. Операция класса, которая освобождает состояние объекта и/или уничтожает сам объект.

деятельность, activity. Операция, выполнение которой требует некоторого времени.

диаграмма взаимодействий, interaction diagram. Часть системы обозначений объектно-ориентированного проектирования; используется для демонстрации выполнения какого-либо сценария в контексте диаграммы объектов.

диаграмма классов, class diagram. Часть системы обозначений объектно-ориентированного проектирования; используется, чтобы наглядно показать классы и их взаимоотношения в логическом проекте системы. Может представлять всю структуру классов или ее часть.

диаграмма модулей, module diagram. Часть системы обозначений объектно-ориентированного проектирования; используется для демонстрации разбиения классов и объектов по модулям в физическом проекте системы. Диаграмма модулей отображает архитектуру модулей системы.

диаграмма объектов, object diagram. Часть системы обозначений объектно-ориентированного проектирования; используется, чтобы наглядно показать объекты и отношения между ними в логическом проекте системы. Может отражать всю объектную структуру или часть ее; обычно иллюстрирует смысл механизмов в логическом проекте. Отдельная диаграмма объектов – моментальный снимок из жизни системы.

диаграмма переходов и состояний, state transition diagram. Часть обозначений объектно-ориентированного проектирования; используется для отображения пространства состояний данного класса, событий, которые вызывают переход из одного состояния в другое, и действий, возникающих в результате смены состояния.

диаграмма процессов, process diagram. Часть системы обозначений объектно-ориентированного проектирования; используется, чтобы наглядно показать, как процессы размещены по процессорам в физическом проекте системы. Диаграмма процессов отражает архитектуру процессов.

динамическое связывание, dynamic binding. Связывание означает установление соответствия имени (например, объявленной переменной) с классом. Динамическое связывание происходит при выполнении программы в тот момент, когда создается объект, обозначенный именем.

друг, friend. Класс или операция, имеющие доступ к закрытым операциям или данным некоторого класса. Только сам класс может называть своих друзей.

закрытая часть, private. Часть интерфейса какого-либо класса, объекта или модуля, закрытая (невидимая) для других классов, объектов и модулей.

защищенная часть, protected. Часть интерфейса какого-либо класса, объекта или модуля, невидимая для всех других классов, объектов и модулей за исключением подклассов.

идентичность, identity. Природа объекта; то, что отличает его от других объектов.

идиома, idiom. Выражение, общепринятое в каком-либо языке программирования или культуре какого-либо приложения, отражающее общепринятый способ использования данного языка.

иерархия, hierarchy. Подчинение или упорядочение абстракций. Две типичных иерархии в сложной системе - структура классов (включая иерархию «общее/частное») и структура объектов (включая иерархию «целое/часть»); иерархии можно также обнаружить в архитектурах модулей и процессов.

инвариант, invariant. Логическое выражение некоторого условия, истинность которого необходимо соблюдать.

инкапсуляция, encapsulation. Процесс разделения элементов абстракции, которые образуют ее структуру и поведение. Служит для отделения внешних обязательств объекта от его реализации.

инстанцирование, instantiation. Подстановка параметров шаблона обобщенного или параметризованного класса; в результате создается конкретный класс, который может иметь экземпляры.

интерфейс, interface. Внешний вид класса, объекта или модуля, выделяющий его существенные черты и не показывающий внутреннего устройства и секретов поведения.

исключение, exception. Возбуждение исключения показывает, что некоторый логический инвариант не соблюдается. В C++ мы возбуждаем исключение, чтобы избежать неправомерное исполнение операций и дать знать о возникшей проблеме другим объектам, которые могут перехватить исключение и принять меры.

использовать, use. Ссылаться на абстракцию извне.

итератор, iterator. Операция, позволяющая навещать части некоторого объекта.

категория классов, class category. Логически полный набор классов, одни из которых видимы для других категорий классов, а другие - нет. Классы в категории сотрудничают для предоставления некоторого набора услуг.

класс, class. Множество объектов с общей структурой и поведением. Термины «класс» и «тип» в большинстве случаев (но не всегда) взаимозаменяемы. Понятие класса отличается от понятия типа тем, что концентрируется на классификации по структуре и поведению.

класс-контейнер, container class. Класс, экземпляры которого представляют собой коллекции других объектов. Контейнер может быть однородным (коллекции включают экземпляры только одного класса) либо неоднородным (коллекции включают экземпляры разных классов, имеющих обычно общий суперкласс). В C++ контейнеры обычно определяются как параметризованные классы с параметром, обозначающим класс объектов коллекции.

клиент, client. Объект, который пользуется услугами другого объекта либо выполняя операции над последним, либо через доступ к его состоянию.

ключ, key. Атрибут, значение которого однозначно идентифицирует объект.

ключевая абстракция, key abstraction. Класс или объект, являющийся частью словаря предметной области.

конкретный класс, concrete class. Класс, реализация которого завершена и который, поэтому, может иметь экземпляры.

конструктор, constructor. Операция, создающая объект и/или инициализирующая его состояние.

метакласс, metaclass. Класс класса; класс, экземпляры которого сами являются классами.

метод, method. Операция над объектом, определенная как часть описания класса. Не любая операция является методом, но все методы – операции. Термины «метод», «сообщение» и «операция» обычно взаимозаменяемы. В некоторых языках методы существуют сами по себе и могут переопределяться подклассами; в других языках метод не может быть переопределен, – он служит как часть реализации обобщенных или виртуальных функций, которые можно переопределять в подклассах.

механизм, mechanism. Структура, посредством которой объекты сотрудничают друг с другом, осуществляя поведение, которое соответствует требованиям системы.

модификатор, modifier. Операция, изменяющая состояние объекта.

модуль, module. Единица кода, служащая строительным блоком физической структуры системы; программный блок, который содержит объявления, выраженные в соответствии с требованиями языка и образующие физическую реализацию части или всех классов и объектов логического проекта системы. Как правило, модуль состоит из интерфейсной части и реализации.

модульность, modularity. Свойство системы, которая была разделена на связанные и слабо зацепленные между собой модули.

мономорфизм, monomorphism. Положение теории типов, согласно которому имена (например, переменных) могут обозначать только объекты одного и того же класса.

мощность, cardinality. Число экземпляров класса: число экземпляров, участвующих в связи классов.

наследование, inheritance. Отношение между классами, при котором класс использует структуру или поведение другого (одиночное наследование) или других (множественное наследование) классов. Наследование вводит иерархию «общее/частное» в которой подкласс наследует от одного или нескольких более общих суперклассов. Подклассы обычно дополняют или переопределяют унаследованную структуру и поведение.

обобщенная функция, generic function. Какая-либо операция над объектом. Обобщенная функция класса может быть переопределена в подклассах; следовательно, ее реализация определяется всем множеством методов, объявленных во всех классах дерева наследования. Термины «обобщенная функция» и «виртуальная функция» взаимозаменяемы.

обобщенный класс, generic class. Класс, служащий шаблоном для создания других классов: шаблон параметризуется другими классами, объектами и/или операциями. Обобщенный класс до создания объектов должен быть инстанцирован. Обобщенные классы используются как контейнерные классы. Термины «обобщенный класс» и «параметризованный класс» взаимозаменяемы.

обратный инжиниринг, reverse-engineering. Восстановление логической или физической модели системы по коду. Противопоставляется прямому инжинирингу.

объект, object. Нечто, чем можно оперировать. Объект имеет состояние, поведение и идентичность. Структура и поведение сходных объектов

определены в общем для них классе. Термины «экземпляр» и «объект» взаимозаменяемы.

объектная модель, object model. Совокупность основополагающих принципов, лежащих в основе объектно-ориентированного проектирования; парадигма программирования, основанная на принципах абстрагирования, инкапсуляции, модульности, иерархичности, типизации, параллелизма и устойчивости.

объектное программирование, object-based programming. Метод программирования, основанный на представлении программы как совокупности объектов, каждый из которых является экземпляром некоторого типа. Типы образуют иерархию, но не наследственную. В таких программах типы рассматриваются как статические, а объекты имеют более динамическую природу, которую ограничивают статическое связывание и мономорфизм.

объектно-ориентированная декомпозиция, object-oriented decomposition. Процесс разбиения системы на части, соответствующие классам и объектам предметной области. Практическое применение методов объектно-ориентированного проектирования приводит к объектно-ориентированной декомпозиции, при которой мы рассматриваем мир как совокупность объектов, согласованно действующих для обеспечения требуемого поведения.

объектно-ориентированное программирование, object-oriented programming (OOP). Методология реализации, при которой программа организуется, как совокупность сотрудничающих объектов, каждый из которых является экземпляром какого-либо класса, а классы образуют иерархию наследования. При этом классы обычно статичны, а объекты очень динамичны, что поощряется динамическим связыванием и полиморфизмом.

объектно-ориентированное проектирование, object-oriented design (OOD). Методология проектирования, соединяющая процесс объектно-ориентированной декомпозиции и систему обозначений для представления логической и физической, статической и динамической моделей проектируемой системы. Система обозначений состоит из диаграмм классов, объектов, модулей и процессов.

объектно-ориентированный анализ, object-oriented analysis. Метод анализа, согласно которому требования рассматриваются с точки зрения классов и объектов, составляющих словарь предметной области.

объект-член, member object. Часть состояния объекта. В совокупности объекты-члены полностью определяют структуру объекта. Термины

«переменная экземпляра», «поле», «объект-член» и «слот» взаимозаменяемы.

ограничение, constraint. Выражение некоторого смыслового условия, которое должно выполняться.

операция класса, class operation. Операция, например, конструктор или деструктор, общая для всего класса и не принадлежащая конкретному объекту.

операция, operation. Нечто, проделываемое одним объектом над другим, чтобы вызвать реакцию. Все операции, которые можно выполнить над каким-либо объектом, сосредоточены в свободных подпрограммах и функциях-членах (методах). Термины «операция», «метод» и «сообщение» взаимозаменяемы.

ответственность, responsibility. Поведение, за которое ответственен объект.

открытая часть, public. Часть интерфейса какого-либо класса, объекта или модуля, открытая (видимая) для всех классов, объектов и модулей.

параллелизм, concurrency. Свойство, отличающее активные объекты от неактивных.

параллельный объект, concurrent object. Активный объект, способный работать в многопоточной среде.

параметризованный класс, parameterized class. Класс, служащий шаблоном для других классов; шаблон параметризуется другими классами, объектами и/или операциями. Параметризованный класс должен быть инстанцирован до создания объектов. Параметризованные классы используются как контейнеры. Термины «обобщенный класс» и «параметризованный класс» взаимозаменяемы.

пассивный объект, passive object. Объект, не имеющий собственного потока управления.

переменная класса, class variable. Часть состояния класса. Совокупность всех переменных класса образует его структуру. Переменные класса совместно используются всеми его экземплярами. В C++ переменная класса объявляется как статический член.

переменная экземпляра, instance variable. Часть состояния объекта. В совокупности переменные экземпляра полностью определяют структуру

объекта. Термины «переменная экземпляра», «поле», «объект-член» и «слот» взаимозаменяемы.

переход, transition. Переход из одного состояния в другое.

поведение, behavior. Действия и реакции объекта, выраженные в терминах передачи сообщений и изменения состояния; видимая извне и воспроизводимая активность объекта.

подкласс, subclass. Класс, наследующий от одного или нескольких классов (которые называются его непосредственными суперклассами).

подсистема, subsystem. Совокупность модулей, часть которых видима для других подсистем, а часть - скрыта.

поле, field. Часть состояния объекта; совокупность полей объекта образуют его структуру. Термины «поле», «переменная экземпляра», «объект-член» и «слот» означают одно и то же.

полиморфизм, polymorphism. Положение теории типов, согласно которому имена (например, переменных) могут обозначать объекты разных (но имеющих общего родителя) классов. Следовательно, любой объект, обозначаемый полиморфным именем, может по-своему реагировать на некий общий набор операций.

последовательное проектирование, round-trip gestalt design. Стил проектирования, который подчеркивает последовательность и итеративность в развитии системы: посредством уточнения различных, хотя и согласованных логических и физических представлений системы в целом; объектно-ориентированное проектирование основывается на последовательном проектировании, что является выражением взаимозависимости общей картины проекта и его деталей.

последовательный объект, sequential object. Пассивный объект, рассчитанный на работу в однопоточном окружении.

постусловие, postcondition. Инвариант, соблюдаемый на выходе из операции.

поток управления, thread of control. Отдельный процесс. Запуск потока управления приводит к возникновению независимой динамической деятельности в системе; данная система может иметь несколько одновременно выполняемых потоков, некоторые из которых могут динамически возникать и уничтожаться. Многопроцессорные системы допускают истинную многопоточность. в то время как на однопроцессорных компьютерах возможна только иллюзия

многопоточности. (Термин «thread of control» переводится также «нить управления». В некоторых случаях используется термин «flow of control», который также переведен)

предусловие, precondition. Инвариант, предполагаемый на входе в операцию.

примесь, mixin. Класс, реализующий какое-либо четко выделенное поведение; используется для уточнения поведения других классов посредством наследования; поведение примеси обычно ортогонально поведению класса, с которым она смешивается.

пространственная сложность, space complexity. Относительный или абсолютный объем памяти, занимаемый объектом.

пространство состояний, state space. Перечислимое множество всех возможных состояний объекта. Пространство состояний программы содержит неопределенное, но конечное число состояний (не обязательно желаемых или ожидаемых).

протокол, protocol. Способы, которыми объекты могут действовать и реагировать; полное статическое и динамическое представление объекта; протокол объекта определяет допустимое поведение объекта.

процесс, process. Запуск одного потока управления.

процессор, processor. Часть аппаратного обеспечения, имеющая вычислительные ресурсы.

прямой инжиниринг, forward-engineering. Создание исполнимого кода по логической или физической модели. Противопоставляется обратному инжинирингу.

раздел, partition. Категории классов или подсистемы, составляющие часть данного уровня абстракции.

реактивная система, reactive system. Система, движимая событиями. Поведение такой системы не определяется простым отображением «вход-выход».

реализация, implementation. Внутреннее представление класса, объекта или модуля, включая секреты его поведения.

роль, role. Способность или цель, с которой класс или объект участвует в отношениях с другими; некоторая четко выделяемая черта поведения

объекта в определенный момент времени; роль – это лицо, которое объект являет миру в данный момент.

свободная подпрограмма, free subprogram. Процедура или функция, которая выполняется как непримитивная операция над объектом или объектами одного и того же или различных классов. Свободная подпрограмма – это любая подпрограмма, которая не является методом какого-либо класса.

связь, link. Связь между объектами, экземпляр ассоциации.

селектор, selector. Операция, имеющая доступ к состоянию объекта, но не изменяющая его.

сервер, server. Объект, который никогда не воздействует на другие объекты, но используется ими; объект, предоставляющий некоторые услуги.

сигнатура, signature. Полная спецификация операции с указанием типов аргументов и возвращаемого значения.

сильно типизированный, strongly typed. Свойство языка программирования, в соответствии с которым во всех выражениях гарантируется согласованность типов.

синхронизация, synchronization. Семантика параллельности операции. Операция может быть простой (присутствует только один поток управления); синхронной (рандеву двух потоков); односторонней (рандеву, при котором одному из потоков приходится ждать); по истечении времени (рандеву, в котором один процесс ждет другого определенное время); асинхронной (два процесса независимы друг от друга).

система реального времени, real-time system. Система, в которой некоторые существенные процессы должны укладываться в отведенное время. Система «жесткого» реального времени должна быть детерминированной; запаздывание с реакцией грозит катастрофой.

скрытие информации, information hiding. Процесс скрытия всех секретов объекта, которые ничего не добавляют к его существенным характеристикам; обычно скрывают структуру объекта и реализацию его методов.

словарь данных, data dictionary. Полный перечень всех классов в системе.

слой, layer. Совокупность категорий классов или подсистем одного уровня абстракции.

слот, slot. Часть состояния объекта; совокупность слотов образуют структуру объекта. Термины «поле», «переменная экземпляра», «объект-член» и «слот» означают одно и то же.

событие, event. Что-то, что может изменить состояние системы.

сообщение, message. Операция, которую один объект может выполнять над другим. Термины «сообщение», «метод» и «операция» обычно взаимозаменяемы.

составной объект (агрегат), aggregate object. Объект, состоящий из других объектов (его частей).

состояние, state. Совокупный результат поведения объекта: одно из стабильных условий, в которых объект может существовать, охарактеризованных количественно; в любой конкретный момент времени состояние объекта включает в себя перечень (обычно, статический) свойств объекта и текущие значения (обычно, динамические) этих свойств.

сотрудничество, collaboration. Процесс, в котором несколько объектов сотрудничают для обеспечения требуемого поведения верхнего уровня.

сохраняемость, persistence. Способность объекта существовать во времени, переживая породивший его процесс, и (или) в пространстве, перемещаясь из одного адресного пространства в другое.

среда разработки, framework. Набор классов, предоставляющих некоторые базовые услуги в определенной области. Таким образом, среда разработки экспортирует классы и механизмы, которые клиенты могут использовать или адаптировать.

статическое связывание, static binding. Связывание означает установление соответствия имени (например, объявленной переменной) классу. Статическое связывание происходит при объявлении имени (во время компиляции), до того, как объект будет создан.

страж, guard. Логическое выражение, применяемое к событию; если выражение истинно, то событие происходит и система изменяет состояние.

стратегическое проектное решение, strategic design decision. Проектные решения, которые имеют решающее влияние на архитектуру.

структура классов, class structure. Граф, вершины которого соответствуют классам, а ребра – отношениям классов. Структура классов для конкретной системы представляется в виде совокупности диаграмм классов.

структура объектов, object structure. Граф, вершины которого соответствуют объектам, а ребра – отношениям объектов. Для отражения структуры объектов или ее части используются диаграммы объектов.

структура, structure. Конкретное представление состояния объекта. Каждый объект имеет собственное состояние, независимое от других объектов, хотя все объекты одного класса имеют одинаковое представление состояния.

структурное проектирование, structured design. Метод проектирования, основанный на алгоритмической декомпозиции.

суперкласс, superclass. Класс, которому наследуют другие классы (называемые непосредственными подклассами).

сценарий, scenario. Последовательность событий, выражающая некий аспект поведения системы.

тактическое проектное решение, tactical design decision. Проектное решение, имеющее ограниченное значение для архитектуры.

тип, type. Определение области допустимых значений, которые может принимать объект, и множества операций, которые могут выполняться над объектом. Термины «класс» и «тип» обычно (но не всегда) взаимозаменяемы; тип отличается от класса тем, что фокусируется на поддержке общего протокола.

типизация, typing. Механизмы, препятствующие замене объектов одного типа на другой или, в крайнем случае, жестко ограничивающие такую замену.

трансформационная система, transformational system. Система, поведение которой определяется в терминах отображения «вход-выход».

управление доступом, access control. Механизм доступа к данным и операциям класса. В С-подобных языках открытые элементы доступны всем, защищенные элементы доступны подклассам, так называемым друзьям класса и файлам реализации, закрытые элементы доступны реализации и друзьям класса. Наконец, элементы с доступом на уровне реализации доступны только в файле реализации класса.

уровень абстракции, level of abstraction. Относительное упорядочение абстракций по структурам классов, объектов, модулей или процессов. В терминах иерархии «часть/целое» объект находится на более высоком уровне абстракции, чем другие, если он строится на основе этих объектов: в

терминах иерархии «общее/частное», высокоуровневые абстракции носят более обобщенный характер, чем низкоуровневые.

услуга, service. Поведение, обеспечиваемое некоторой частью системы.

устройство, device. Часть аппаратуры, не имеющая собственных вычислительных ресурсов.

утверждение, assertion. Логическое выражение некоторого условия, истинность которого необходимо обеспечить.

утилиты класса, class utility. Совокупность свободных подпрограмм. В С-подобных языках – класс, который состоит только из статических членов и/или функций-членов.

функциональная точка, function point. В контексте анализа требований к системе – отдельное поведение, видимое извне и поддающееся проверке.

функция, function. Некоторое преобразование «вход-выход», вытекающее из поведения объекта.

функция-член, member function. Операция над объектом, определенная как часть описания класса. Все функции-члены – операции, но не все операции – функции-члены. Термины «функции-члены» и «методы» взаимозаменяемы. В некоторых языках функции-члены существуют сами по себе и могут переопределяться подклассами; в других языках функция-член не может быть переопределена, – она служит как часть реализации обобщенных или виртуальных функций, которые можно переопределять в подклассах.

экземпляр, instance. Нечто, чем можно оперировать. Экземпляр имеет состояние, поведение и идентичность. Структура и поведение всех экземпляров класса определяются этим классом. Термины «объект» и «экземпляр» взаимозаменяемы.

Приложение 3 – Листинг СУБД

```

#импорт библиотеки графического интерфейса по синониму
import tkinter as tk
#импорт всех функций
from tkinter import *
#импорт из библиотеки графического интерфейса отдельной функции
from tkinter import ttk
#подключение библиотеки для работы с базами данных
import sqlite3
#импорт библиотеки ctypes, которая
#предоставляет C-совместимые типы данных и позволяет вызывать функции из DLL или
разделяемых библиотек
import ctypes

#создание класса основного окна приложения
class Main(tk.Frame):
    #определяем атрибуты класса
    role=0
    #конструктор класса
    def __init__(self, root):
        super().__init__(root)
        self.init_main()
        self.db = db

    #создание интерфейса главного окна приложения
    def init_main(self):
        #создание списка с предопределенными значениями
        self.combobox_role = ttk.Combobox(self, values=[u'Иванов И.И.', u'Администратор
БД', u'Коцубинский В.П.', u'Черкашин М.В.'])
        self.combobox_role.current(0)
        self.combobox_role.grid(row=5, column=0,
                                columnspan=3,
                                sticky=W+E, padx=10)

        self.label_pwd = tk.Label(self, text='Пароль:')
        self.label_pwd.grid(row=6, column=0,
                             sticky=W, padx=10)

        self.entry_pwd = ttk.Entry(self)
        self.entry_pwd.grid(row=6, column=1,
                             columnspan=3,
                             sticky=W, padx=10)

        self.btn_chkpwd = ttk.Button(self, text='Вход')
        self.btn_chkpwd.grid(row=5, column=5,
                              sticky=E, padx=5, pady=10)
        self.btn_chkpwd.bind('<Button-1>', lambda event: Main.verification(self,
self.combobox_role.get(), self.entry_pwd.get()))

```

```

#содание фонового изображения с размещением внизу

self.back_img = tk.PhotoImage(file='logo.gif')
panel = Label(root, image = self.back_img)
panel.pack(side = "bottom", fill = "both", expand = "yes")

def verification(self, usr, pwd):
    #role == 2 - Администратор БД, role == 1 - Пользователь
    if (usr=='Иванов И.И.' and pwd=='1') or (usr=='Черкашин М.В.' and pwd=='2') or
(usr=='Коцубинский В.П.' and pwd=='3'):
        Main.role=1
    if (usr=='Администратор БД' and pwd=='123'):
        Main.role=2
    if (Main.role==1) or (Main.role==2):
        #задание панели управления с определенным цветом и шириной бордюра
        #панель создается как дочерний элемент родительской формы Frame
        toolbar = tk.Frame(bg='#d7d8e0', bd=2)
        #определение места расположения созданной панели - верх, с заполнением всей
области
        toolbar.pack(side=tk.TOP, fill=tk.X)
        ctypes.windll.user32.MessageBoxW(0, "Доступ предоставлен.", "Успех", 1)
        #убираем элементы управления процедуры авторизации с формы - т.к. доступ
предоставлен
        self.combobox_role.grid_remove()
        self.label_pwd.grid_remove()
        self.entry_pwd.grid_remove()
        self.btn_chkpwd.grid_remove()
    if (Main.role==1):
        self.but_img = tk.PhotoImage(file='add.gif')
        #создание кнопки "Таблица студентов" на родительской панели, с подписью,
выполняющей команду open_dialog
        #цветом #d7d8e0' с бордюром 1 выровненной по верхней границе панели и с
изображением, загруженным в but_img
        btn_open_dialog = tk.Button(toolbar, text='Студенты',
command=self.open_dialog, bg='#d7d8e0', bd=1,
                                compound=tk.TOP, image=self.but_img, width=70)
        btn_open_dialog.pack(side=tk.LEFT)

        self.but_img3 = tk.PhotoImage(file='rate.gif')
        btn_open_dialog2 = tk.Button(toolbar, text='Рейтинг',
command=self.open_dialog_rate, bg='#d7d8e0', bd=1,
                                compound=tk.TOP, image=self.but_img3, width=70)
        btn_open_dialog2.pack(side=tk.LEFT)
    if (Main.role==2):
        self.but_img = tk.PhotoImage(file='add.gif')
        #создание кнопки "Таблица студентов" на родительской панели, с подписью,
выполняющей команду open_dialog
        #цветом #d7d8e0' с бордюром 1 выровненной по верхней границе панели и с
изображением, загруженным в but_img

```

```

        btn_open_dialog = tk.Button(toolbar, text='Студенты', command=self.open_dialog,
        bg='#d7d8e0', bd=1,
                                compound=tk.TOP, image=self.but_img, width=70)
        btn_open_dialog.pack(side=tk.LEFT)

        self.but_img2 = tk.PhotoImage(file='pred.gif')
        btn_open_dialog2 = tk.Button(toolbar, text='Предметы',
        command=self.open_dialog_pred, bg='#d7d8e0', bd=1,
                                compound=tk.TOP, image=self.but_img2, width=70)
        btn_open_dialog2.pack(side=tk.LEFT)

        self.but_img3 = tk.PhotoImage(file='rate.gif')
        btn_open_dialog2 = tk.Button(toolbar, text='Рейтинг',
        command=self.open_dialog_rate, bg='#d7d8e0', bd=1,
                                compound=tk.TOP, image=self.but_img3, width=70)
        btn_open_dialog2.pack(side=tk.LEFT)

        self.but_img4 = tk.PhotoImage(file='fac.gif')
        btn_open_dialog2 = tk.Button(toolbar, text='Факультеты',
        command=self.open_dialog_fac, bg='#d7d8e0', bd=1,
                                compound=tk.TOP, image=self.but_img4, width=70)
        btn_open_dialog2.pack(side=tk.LEFT)

        self.but_img5 = tk.PhotoImage(file='prep.gif')
        btn_open_dialog2 = tk.Button(toolbar, text='Преподаватели',
        command=self.open_dialog_prep, bg='#d7d8e0', bd=1,
                                compound=tk.TOP, image=self.but_img5, width=100)
        btn_open_dialog2.pack(side=tk.LEFT)

        self.but_img6 = tk.PhotoImage(file='gr.gif')
        btn_open_dialog2 = tk.Button(toolbar, text='Группы',
        command=self.open_dialog_grup, bg='#d7d8e0', bd=1,
                                compound=tk.TOP, image=self.but_img6, width=50)
        btn_open_dialog2.pack(side=tk.LEFT)
        if Main.role==0:
            ctypes.windll.user32.MessageBoxW(0, "Пароль неверный. Доступ запрещен.",
            "Ошибка", 1)
            raise SystemExit

#функция добавления в таблицу студента и перерисовки таблицы с результатом
def insert_stud(frame, std_name, std_address, std_addition, std_kurs, grup_id):
    db.insert_data_stud(std_name, std_address, std_addition, std_kurs, grup_id)
    Main.view_records(frame)

#функция изменения студента и перерисовки таблицы с результатом
def update_stud(frame, std_name, std_address, std_addition, std_kurs, grup_id):
    db.c.execute('''UPDATE students SET std_name=?, std_address=?, std_addition=?,
std_kurs=?, grup_id=? WHERE std_id=?''',
                (std_name, std_address, std_addition, std_kurs, grup_id,
frame.tree.set(frame.tree.selection(), '#1'))))

```

```

db.conn.commit()
Main.view_records(frame)

#функция удаления студента и перерисовки таблицы с результатом
def delete_stud(frame, num_sel_string):
    db.c.execute(''DELETE FROM students WHERE std_id=?'',
                (num_sel_string,))
    db.conn.commit()
    Main.view_records(frame)

#функция перерисовки таблицы с результатом для таблицы students
def view_records(frame):
    db.c.execute(''SELECT * FROM students'')
    [frame.tree.delete(i) for i in frame.tree.get_children()]
    [frame.tree.insert('', 'end', values=row) for row in db.c.fetchall()]

#функция перерисовки таблицы с результатом для таблицы predmets
def view_records_pred(frame):
    db.c.execute(''SELECT * FROM predmets'')
    [frame.tree.delete(i) for i in frame.tree.get_children()]
    [frame.tree.insert('', 'end', values=row) for row in db.c.fetchall()]

#функция добавления предмета
def insert_pred(frame, pred_name, pred_kurs):
    db.insert_data_pred(pred_name, pred_kurs)
    Main.view_records_pred(frame)

#функция изменения предмета
def update_pred(frame, pred_name, pred_kurs):
    db.c.execute(''UPDATE predmets SET pred_name=?, pred_kurs=? WHERE pred_id=?'',
                (pred_name, pred_kurs, frame.tree.set(frame.tree.selection(),
'#1'))))
    db.conn.commit()
    Main.view_records_pred(frame)

#функция удаления предмета
def delete_pred(frame, num_sel_string):
    #перед удалением проверим целостность, уточнив, не используется ли позиция в других
таблицах
    db.c.execute(''SELECT * FROM rate WHERE pred_id LIKE ?'', (num_sel_string,))
    result = []
    for row in db.c.fetchall():
        result.append(row[0])

    if not result:
        db.c.execute(''DELETE FROM predmets WHERE pred_id=?'',
                    (num_sel_string,))
        db.conn.commit()
    else:
        ctypes.windll.user32.MessageBoxW(0, "Нельзя удалять позицию {name} -
нарушается целостность базы".format(name=num_sel_string), "Ошибка", 1)

```

```

Main.view_records_pred(frame)

#---
#функция перерисовки таблицы с результатом для таблицы faculty
def view_records_fac(frame):
    db.c.execute('''SELECT * FROM faculty''')
    [frame.tree.delete(i) for i in frame.tree.get_children()]
    [frame.tree.insert('', 'end', values=row) for row in db.c.fetchall()]

#функция добавления факультета
def insert_fac(frame, fac_name):
    db.insert_data_fac(str(fac_name))
    Main.view_records_fac(frame)

#функция изменения факультета
def update_fac(frame, fac_name):
    db.c.execute('''UPDATE faculty SET fac_name=? WHERE fac_id=?''',
                  (fac_name, frame.tree.set(frame.tree.selection(), '#1')))
    db.conn.commit()
    Main.view_records_fac(frame)

#функция удаления факультета
def delete_fac(frame, num_sel_string):
    #перед удалением проверим целостность, уточнив, не используется ли позиция в других
таблицах
    db.c.execute('''SELECT * FROM prepods WHERE fac_id LIKE ?''', (num_sel_string,))
    result = []
    for row in db.c.fetchall():
        result.append(row[0])

    db.c.execute('''SELECT * FROM groups WHERE fac_id LIKE ?''', (num_sel_string,))
    result2 = []
    for row in db.c.fetchall():
        result2.append(row[0])
    if not (result or result2):
        db.c.execute('''DELETE FROM faculty WHERE fac_id=?''',
                      (num_sel_string,))
        db.conn.commit()
    else:
        ctypes.windll.user32.MessageBoxW(0, "Нельзя удалять позицию {name} -
нарушается целостность базы".format(name=num_sel_string), "Ошибка", 1)

    Main.view_records_fac(frame)

#---
#---
#функция перерисовки таблицы с результатом для таблицы groups
def view_records_group(frame):
    db.c.execute('''SELECT grup_id, date_create, t2.fac_name AS fac_name FROM groups,
faculty t2 WHERE groups.fac_id = t2.fac_id''')
    [frame.tree.delete(i) for i in frame.tree.get_children()]

```

```

[frame.tree.insert('', 'end', values=row) for row in db.c.fetchall()]

#функция добавления группы
def insert_grup(frame, date_create, fac_name):
    #сначала надо получить fac_id соответствующий выбранному в списке значению
    db.c.execute(''SELECT fac_id FROM faculty WHERE fac_name=?'', (fac_name,))
    #т.к. результат выполнения запроса - таблица значений, получим это значение
    result_prep_id = []
    for row in db.c.fetchall():
        result_prep_id.append(row[0])
    #конвертируем тип значения в string, чтобы можно было передать в запрос
    result_grup_id1=''.join(str(e) for e in result_prep_id)
    db.insert_data_grup(date_create, result_grup_id1)
    Main.view_records_grup(frame)

#функция изменения группы
def update_grup(frame, date_create, fac_name):
    #сначала надо получить fac_id соответствующий выбранному в списке значению
    db.c.execute(''SELECT fac_id FROM faculty WHERE fac_name=?'', (fac_name,))
    #т.к. результат выполнения запроса - таблица значений, получим это значение
    result_prep_id = []
    for row in db.c.fetchall():
        result_prep_id.append(row[0])
    #конвертируем тип значения в string, чтобы можно было передать в запрос
    result_grup_id1=''.join(str(e) for e in result_prep_id)
    db.c.execute(''UPDATE groups SET date_create=?, fac_id=? WHERE grup_id=?'',
                (date_create, result_grup_id1,
frame.tree.set(frame.tree.selection(), '#1'))
    db.conn.commit()
    Main.view_records_grup(frame)

#функция удаления группы
def delete_grup(frame, num_sel_string):
    #перед удалением проверим целостность, уточнив, не используется ли позиция в других
таблицах
    db.c.execute(''SELECT * FROM students WHERE grup_id LIKE ?'', (num_sel_string,))
    result = []
    for row in db.c.fetchall():
        result.append(row[0])

    if not result:
        db.c.execute(''DELETE FROM groups WHERE grup_id=?'',
                    (num_sel_string,))
        db.conn.commit()
    else:
        ctypes.windll.user32.MessageBoxW(0, "Нельзя удалять позицию {name} -
нарушается целостность базы".format(name=num_sel_string), "Ошибка", 1)

    Main.view_records_grup(frame)

#---
#функция перерисовки таблицы с результатом для таблицы rperods

```

```

def view_records_prep(frame):
    #db.c.execute('''SELECT * FROM prepods''')
    db.c.execute('''SELECT prep_id, prep_name, prep_address, prep_persinf, prep_inn,
t2.fac_name AS fac_name FROM prepods, faculty t2 WHERE prepods.fac_id = t2.fac_id''')
    [frame.tree.delete(i) for i in frame.tree.get_children()]
    [frame.tree.insert('', 'end', values=row) for row in db.c.fetchall()]

#функция добавления преподавателя
def insert_prep(frame, prep_name, prep_address, prep_persinf, prep_inn, fac_name):
    #сначала надо получить fac_id соответствующий выбранному в списке значению
    db.c.execute('''SELECT fac_id FROM faculty WHERE fac_name=?''',(fac_name,))
    #т.к. результат выполнения запроса - таблица значений, получим это значение
    result_prep_id = []
    for row in db.c.fetchall():
        result_prep_id.append(row[0])
    #конвертируем тип значения в string, чтобы можно было передать в запрос
    result_prep_id1=''.join(str(e) for e in result_prep_id)
    db.insert_data_prep(prep_name, prep_address, prep_persinf, prep_inn,
result_prep_id1)
    Main.view_records_prep(frame)

#функция изменения преподавателя
def update_prep(frame, prep_name, prep_address, prep_persinf, prep_inn, fac_name):
    #сначала надо получить fac_id соответствующий выбранному в списке значению
    db.c.execute('''SELECT fac_id FROM faculty WHERE fac_name=?''',(fac_name,))
    #т.к. результат выполнения запроса - таблица значений, получим это значение
    result_prep_id = []
    for row in db.c.fetchall():
        result_prep_id.append(row[0])
    #конвертируем тип значения в string, чтобы можно было передать в запрос
    result_prep_id1=''.join(str(e) for e in result_prep_id)
    db.c.execute('''UPDATE prepods SET prep_name=?, prep_address=?, prep_persinf=?,
prep_inn=?, fac_id=? WHERE prep_id=?''',
                    (prep_name, prep_address, prep_persinf, prep_inn, result_prep_id1,
frame.tree.set(frame.tree.selection(), '#1'))))
    db.conn.commit()
    Main.view_records_prep(frame)

#функция удаления преподавателя
def delete_prep(frame, num_sel_string):
    db.c.execute('''DELETE FROM prepods WHERE prep_id=?''',
                    (num_sel_string,))
    db.conn.commit()
    Main.view_records_prep(frame)

#---
#функция перерисовки таблицы с результатом для таблицы rate
def view_records_rate(frame):
    db.c.execute('''SELECT rate_id, rate_max, rate_fact, t2.std_name AS std_name,
t3.pred_name AS pred_name FROM rate, students t2, predmets t3 WHERE rate.std_id = t2.std_id
AND rate.pred_id = t3.pred_id''')
    [frame.tree.delete(i) for i in frame.tree.get_children()]

```

```

[frame.tree.insert('', 'end', values=row) for row in db.c.fetchall()]
pass

#функция поиска для таблицы students
def search_stud(frame, description):
    description = ('%' + description + '%',)
    db.c.execute('''SELECT * FROM students WHERE std_name LIKE ?''', description)
    [frame.tree.delete(i) for i in frame.tree.get_children()]
    [frame.tree.insert('', 'end', values=row) for row in db.c.fetchall()]
    pass

#функция добавления записи в таблицу rate
def insert_rate(frame, rate_max, rate_fact, std_name, pred_name):
    #сначала надо получить std_id соответствующий выбранному в списке значению
    db.c.execute('''SELECT std_id FROM students WHERE std_name=?''', (std_name,))
    #т.к. результат выполнения запроса - таблица значений, получим это значение
    result_std_id = []
    for row in db.c.fetchall():
        result_std_id.append(row[0])
    #конвертируем тип значения в string, чтобы можно было передать в запрос
    result_std_id1=''.join(str(e) for e in result_std_id)

    db.c.execute('''SELECT pred_id FROM predmets WHERE pred_name=?''', (pred_name,))
    result_pred_id = []
    for row in db.c.fetchall():
        result_pred_id.append(row[0])
    result_pred_id1=''.join(str(e) for e in result_pred_id)
    #передача подготовленных значений в функцию добавления записи
    db.insert_data_rate(rate_max, rate_fact, result_std_id1, result_pred_id1)
    Main.view_records_rate(frame)

#функция изменения записи в таблице rate
def update_rate(frame, rate_max, rate_fact, std_name, pred_name):
    db.c.execute('''SELECT std_id FROM students WHERE std_name=?''', (std_name,))
    result_std_id = []
    for row in db.c.fetchall():
        result_std_id.append(row[0])
    result_std_id1=''.join(str(e) for e in result_std_id)

    db.c.execute('''SELECT pred_id FROM predmets WHERE pred_name=?''', (pred_name,))
    result_pred_id = []
    for row in db.c.fetchall():
        result_pred_id.append(row[0])
    result_pred_id1=''.join(str(e) for e in result_pred_id)
    db.c.execute('''UPDATE rate SET rate_max=?, rate_fact=?, std_id=?, pred_id=? WHERE
rate_id=?''',
                (rate_max, rate_fact, str(result_std_id1), str(result_pred_id1),
frame.tree.set(frame.tree.selection(), '#1'))
    db.conn.commit()
    Main.view_records_rate(frame)

```



```

#функция удаления записи из таблицы rate
def delete_rate(frame, num_sel_string):
    db.c.execute(''DELETE FROM rate WHERE rate_id=?'',
                (num_sel_string,))
    db.conn.commit()
    Main.view_records_rate(frame)

#функция подготовки набора значений и заполнения combobox с названиями предметов
def change_rate_combo(self):
    db.c.execute(''SELECT pred_name FROM predmets'')
    result = []
    for row in db.c.fetchall():
        result.append(row[0])
    self.combo_rate_dis['values'] =result

#функция подготовки набора значений и заполнения combobox с названиями предметов
def change_rate_combo_name(self):
    db.c.execute(''SELECT std_name FROM students'')
    result = []
    for row in db.c.fetchall():
        result.append(row[0])
    self.combo_rate_name['values'] =result

#функция подготовки набора значений и заполнения combobox с названиями факультетов
def change_prep_combo_name(self):
    db.c.execute(''SELECT fac_name FROM faculty'')
    result = []
    for row in db.c.fetchall():
        result.append(row[0])
    self.combo_prep_name['values'] =result

#функция подготовки набора значений и заполнения combobox с названиями факультетов
def change_std_gr_combo_name(self):
    db.c.execute(''SELECT grup_id FROM grups'')
    result = []
    for row in db.c.fetchall():
        result.append(row[0])
    self.combo_gr_name['values'] =result

#функция подготовки набора значений и заполнения combobox с названиями факультетов
def change_grup_combo_name(self):
    db.c.execute(''SELECT fac_name FROM faculty'')
    result = []
    for row in db.c.fetchall():
        result.append(row[0])
    self.combo_grup_name['values'] =result

#функция создания окна для поиска значений в таблице students
def new_search(frame):
    #создание формы
    self = tk.Toplevel(root)

```

```

#задание заголовка
self.title('Поиск')
#определение размера окна 300x100
# и смещения при выводе формы относительно левого верхнего угла экрана
self.geometry('300x100+200+100')
#запрет изменения геометрии окна
self.resizable(False, False)
#создание метки
label_search = tk.Label(self, text='Поиск')
#размещение метки относительно левого верхнего угла
#Place – это менеджер геометрии, который размещает элементы GUI, используя
абсолютное позиционирование.
#Pack – это менеджер геометрии, который размещает элементы GUI по горизонтали и
вертикали.
#Grid – это менеджер геометрии, который размещает элементы GUI в двухмерной сетке
label_search.place(x=50, y=20)
#создание поля ввода
self.entry_search = ttk.Entry(self)
self.entry_search.place(x=105, y=20, width=150)
#создание кнопки Закрытия
btn_cancel = ttk.Button(self, text='Закреть', command=self.destroy)
btn_cancel.place(x=185, y=50)
#создание кнопки Поиска
btn_search = ttk.Button(self, text='Поиск')
btn_search.place(x=105, y=50)
#в tkinter с помощью метода bind между собой связываются виджет, событие и
действие.
btn_search.bind('<Button-1>', lambda event: Main.search_stud(frame,
self.entry_search.get()))
btn_search.bind('<Button-1>', lambda event: self.destroy(), add='+')

#создание формы работы с таблицей students
def new_window_stud():
    self = tk.Toplevel(root)
    self.title('Таблица students')
    self.geometry('590x470+200+100')
    self.resizable(False, False)

#внутренняя функция перерисовки таблицы
def refreshdata():
    db.c.execute('''SELECT * FROM students''')
    [self.tree.delete(i) for i in self.tree.get_children()]
    [self.tree.insert('', 'end', values=row) for row in db.c.fetchall()]

#создание метки
label_id_pos = tk.Label(self, text='Позиция:')
#размещение метки в сетке
#При размещении виджетов методом grid родительский контейнер (обычно это окно)
# условно разделяется на ячейки подобно таблице. Адрес каждой ячейки состоит из
номера
# строки и номера столбца. Нумерация начинается с нуля. Ячейки можно объединять

```

```

# как по вертикали, так и по горизонтали.
label_id_pos.grid(row=0, column=0,
                  sticky=W, padx=10)

label_description = tk.Label(self, text='ФИО:')
label_description.grid(row=1, column=0,
                      sticky=W, padx=10)

label_select = tk.Label(self, text='Курс:')
label_select.grid(row=2, column=0,
                 sticky=W, padx=10)

label_adres = tk.Label(self, text='Адрес:')
label_adres.grid(row=3, column=0,
                sticky=W, padx=10)

label_descr = tk.Label(self, text='Примечание:')
label_descr.grid(row=4, column=0,
                sticky=W, padx=10)

label_gr = tk.Label(self, text='Группа:')
label_gr.grid(row=5, column=0,
             sticky=W, padx=10)

self.entry_stdsel_pos = ttk.Entry(self)
self.entry_stdsel_pos.grid(row=0, column=1,
                          colspan=4,
                          sticky=W, padx=10)

self.entry_std_name = ttk.Entry(self)
self.entry_std_name.grid(row=1, column=1,
                        colspan=4,
                        sticky=W, padx=10)

self.entry_std_adres = ttk.Entry(self)
self.entry_std_adres.grid(row=3, column=1,
                         sticky=W, padx=10)

self.entry_std_addition = ttk.Entry(self)
self.entry_std_addition.grid(row=4, column=1,
                            colspan=4,
                            sticky=W, padx=10)

#создание списка с predefined значениями
self.combobox = ttk.Combobox(self, values=[u'1 курс', u'2 курс', u'3 курс', u'4
курс', u'5 курс', u'1 курс магистр', u'2 курс магистр', u'3 курс магистр'])
self.combobox.current(0)
self.combobox.grid(row=2, column=1,
                  colspan=4,
                  sticky=W, padx=10)

```

```

#создание списка с предопределенными значениями
self.combo_gr_name = ttk.Combobox(self)
Main.change_std_gr_combo_name(self)
self.combo_gr_name.grid(row=5, column=1,
                        columnspan=4,
                        sticky=W, padx=10)

btn_cancel = ttk.Button(self, text='Заккрыть', command=self.destroy)
btn_cancel.grid(row=6, column=5,
                sticky=W, padx=5, pady=10)

#создание кнопки "Добавить"
self.btn_ok = ttk.Button(self, text='Добавить')
self.btn_ok.grid(row=6, column=1,
                 sticky=W, padx=5, pady=10)
#назначение обработчика для кнопки в виде лямбда события
self.btn_ok.bind('<Button-1>', lambda event: Main.insert_std(self,
self.entry_std_name.get(),
                                                    self.entry_std_adres
s.get(),
                                                    self.entry_std_addit
ion.get(),
                                                    self.combobox.get(),
                                                    self.combo_gr_name.g
et()))

self.btn_edit = ttk.Button(self, text='Редактировать')
self.btn_edit.grid(row=6, column=0,
                  sticky=W, padx=5, pady=10)
self.btn_edit.bind('<Button-1>', lambda event: Main.update_std(self,
self.entry_std_name.get(),
                                                    self.entry_std_adr
ess.get(),
                                                    self.entry_std_add
ition.get(),
                                                    self.combobox.get(
),
                                                    self.combo_gr_name
.get()))

self.btn_search = ttk.Button(self, text='Поиск')
self.btn_search.grid(row=6, column=2,
                    sticky=W, padx=5, pady=10)
self.btn_search.bind('<Button-1>', lambda event: Main.new_search(self))

self.btn_search = ttk.Button(self, text='Обновить')
self.btn_search.grid(row=0, column=5,
                    sticky=W, padx=5, pady=10)
self.btn_search.bind('<Button-1>', lambda event: Main.view_records(self))

```

```

#вывод сетки grid с содержимым таблицы
self.tree = ttk.Treeview(self, columns=('std_id', 'std_name', 'std_adress',
'std_addition', 'std_kurs', 'grup_id'), show='headings')
#задание размеров и выравнивания данных внутри сетки
self.tree.column('std_id', width=20, anchor=tk.CENTER)
self.tree.column('std_name', width=135, anchor=tk.CENTER)
self.tree.column('std_adress', width=100, anchor=tk.CENTER)
self.tree.column('std_addition', width=100, anchor=tk.CENTER)
self.tree.column('std_kurs', width=100, anchor=tk.CENTER)
self.tree.column('grup_id', width=20, anchor=tk.CENTER)
#задание заголовков для сетки
self.tree.heading('std_id', text='ID')
self.tree.heading('std_name', text='ФИО')
self.tree.heading('std_adress', text='Адрес')
self.tree.heading('std_addition', text='Примечание')
self.tree.heading('std_kurs', text='Курс')
self.tree.heading('grup_id', text='Группа')
#выравнивание сетки
self.tree.grid(row=7, column=0, columnspan=6,
               sticky='news', padx=5, pady=10)
#задание полосы прокрутки
vsb = tk.Scrollbar(self, orient="vertical", command=self.tree.yview)
vsb.grid(row=7, column=6, sticky='ns')
self.tree.configure(yscrollcommand=vsb.set)

#вызов внутренней функции перерисовки таблицы
refreshdata()

#функция выводит значения из таблицы students в соответствующие поля ввода для
выбранной позиции в сетке grid
# предварительно значения очищаются
def selectItem(a):
    curItem = self.tree.focus()
    self.entry_std_name.delete(0, END)
    self.entry_std_name.insert(0, self.tree.set(curItem, 'std_name'))
    self.entry_std_adress.delete(0, END)
    self.entry_std_adress.insert(0, self.tree.set(curItem, 'std_adress'))
    self.entry_std_addition.delete(0, END)
    self.entry_std_addition.insert(0, self.tree.set(curItem, 'std_addition'))
    self.combobox.delete(0, END)
    self.combobox.insert(0, self.tree.set(curItem, 'std_kurs'))
    self.combo_gr_name.delete(0, END)
    self.combo_gr_name.insert(0, self.tree.set(curItem, 'grup_id'))
    self.entry_stdsel_pos.delete(0, END)
    #для того, чтобы корректно записать значение str_id делаем попытку привести его
к int
    # в случае неудачи выводим системное сообщение (ctypes был нужен для этого)
    # и ставим значение по умолчанию "1"
    try:
        valtochk = self.tree.set(curItem, 'std_id')
        int(valtochk)

```

```

        self.entry_stdsel_pos.insert(0, self.tree.set(curItem, 'std_id'))
    except ValueError:
        ctypes.windll.user32.MessageBoxW(0, "Значение номера позиции не
является числом", "Ошибка", 1)
        self.entry_stdsel_pos.insert(0, 1)

self.btn_del = ttk.Button(self, text='Удалить')
self.btn_del.grid(row=6, column=4,
                    sticky=W, padx=5, pady=10)
self.btn_del.bind('<Button-1>', lambda event: Main.delete_stud(self,
self.entry_stdsel_pos.get()))
#устанавливаем курсор в сетке на первую позицию
try:
    child_id = self.tree.get_children()[0] # для ссылки на последний элемент в
наборе

    self.tree.focus(child_id)
    self.tree.selection_set(child_id)
    selectItem(0)
except:
    ctypes.windll.user32.MessageBoxW(0, "Нет строк в таблице", "Ошибка", 1)
    #"подвешиваем" функцию определения текущей позиции selectItem на ожидание
отпускания кнопки над сеткой grid
    self.tree.bind('<ButtonRelease-1>', selectItem)
#---prepods---
#создание формы работы с таблицей students
def new_window_prep():
    self = tk.Toplevel(root)
    self.title('Таблица prepods')
    self.geometry('590x430+200+100')
    self.resizable(False, False)

#внутренняя функция перерисовки таблицы
def refreshdata():
    #db.c.execute('''SELECT * FROM prepods''')
    db.c.execute('''SELECT prep_id, prep_name, prep_address, prep_persinf, prep_inn,
t2.fac_name AS fac_name FROM prepods, faculty t2 WHERE prepods.fac_id = t2.fac_id''')
    [self.tree.delete(i) for i in self.tree.get_children()]
    [self.tree.insert('', 'end', values=row) for row in db.c.fetchall()]

#создание метки
label_id_pos = tk.Label(self, text='Позиция:')
#размещение метки в сетке
#При размещении виджетов методом grid родительский контейнер (обычно это окно)
# условно разделяется на ячейки подобно таблице. Адрес каждой ячейки состоит из
номера

# строки и номера столбца. Нумерация начинается с нуля. Ячейки можно объединять
# как по вертикали, так и по горизонтали.
label_id_pos.grid(row=0, column=0,
                    sticky=W, padx=10)

```

```

label_description = tk.Label(self, text='ФИО:')
label_description.grid(row=1, column=0,
                        sticky=W, padx=10)

label_adres = tk.Label(self, text='Адрес:')
label_adres.grid(row=2, column=0,
                 sticky=W, padx=10)

label_persinf = tk.Label(self, text='Паспорт:')
label_persinf.grid(row=3, column=0,
                   sticky=W, padx=10)

label_inn = tk.Label(self, text='ИНН:')
label_inn.grid(row=4, column=0,
               sticky=W, padx=10)

label_inn = tk.Label(self, text='Факультет:')
label_inn.grid(row=5, column=0,
               sticky=W, padx=10)

self.entry_prep_sel_pos = ttk.Entry(self)
self.entry_prep_sel_pos.grid(row=0, column=1,
                              colspan=4,
                              sticky=W, padx=10)

self.entry_prep_name = ttk.Entry(self)
self.entry_prep_name.grid(row=1, column=1,
                           colspan=4,
                           sticky=W, padx=10)

self.entry_prep_address = ttk.Entry(self)
self.entry_prep_address.grid(row=2, column=1,
                              sticky=W, padx=10)

self.entry_prep_pers = ttk.Entry(self)
self.entry_prep_pers.grid(row=3, column=1,
                           colspan=4,
                           sticky=W, padx=10)

self.entry_prep_inn = ttk.Entry(self)
self.entry_prep_inn.grid(row=4, column=1,
                          colspan=4,
                          sticky=W, padx=10)

#создание списка с предопределенными значениями
self.combo_prep_name = ttk.Combobox(self)
Main.change_prep_combo_name(self)
self.combo_prep_name.grid(row=5, column=1,
                           colspan=4,
                           sticky=W, padx=10)

```

```

btn_cancel = ttk.Button(self, text='Закреть', command=self.destroy)
btn_cancel.grid(row=6, column=5,
                sticky=W, padx=5, pady=10)

#создание кнопки "Добавить"
self.btn_ok = ttk.Button(self, text='Добавить')
self.btn_ok.grid(row=6, column=1,
                sticky=W, padx=5, pady=10)
#назначение обработчика для кнопки в виде лямбда события
self.btn_ok.bind('<Button-1>', lambda event: Main.insert_prep(self,
self.entry_prep_name.get(),
                                                    self.entry_prep_adre
ss.get(),
                                                    self.entry_prep_pers
.get(),
                                                    self.entry_prep_inn.
get(),
                                                    self.combo_prep_name
.get()))

self.btn_edit = ttk.Button(self, text='Редактировать')
self.btn_edit.grid(row=6, column=0,
                sticky=W, padx=5, pady=10)
self.btn_edit.bind('<Button-1>', lambda event: Main.update_prep(self,
self.entry_prep_name.get(),
                                                    self.entry_prep_ad
ress.get(),
                                                    self.entry_prep_pe
rs.get(),
                                                    self.entry_prep_in
n.get(),
                                                    self.combo_prep_na
me.get()))

#вывод сетки grid с содержимым таблицы
self.tree = ttk.Treeview(self, columns=('prep_id', 'prep_name', 'prep_adress',
'prep_persinf', 'prep_inn', 'fac_id'), show='headings')
#задание размеров и выравнивания данных внутри сетки
self.tree.column('prep_id', width=20, anchor=tk.CENTER)
self.tree.column('prep_name', width=135, anchor=tk.CENTER)
self.tree.column('prep_adress', width=100, anchor=tk.CENTER)
self.tree.column('prep_persinf', width=100, anchor=tk.CENTER)
self.tree.column('prep_inn', width=100, anchor=tk.CENTER)
self.tree.column('fac_id', width=100, anchor=tk.CENTER)
#задание заголовков для сетки
self.tree.heading('prep_id', text='ID')
self.tree.heading('prep_name', text='ФИО')
self.tree.heading('prep_adress', text='Адрес')

```



```

self.tree.heading('prep_persinf', text='Паспорт')
self.tree.heading('prep_inn', text='ИНН')
self.tree.heading('fac_id', text='Факультет')
#выравнивание сетки
self.tree.grid(row=7, column=0, columnspan=6,
               sticky='news', padx=5, pady=10)
#задание полосы прокрутки
vsb = tk.Scrollbar(self, orient="vertical", command=self.tree.yview)
vsb.grid(row=7, column=6, sticky='ns')
self.tree.configure(yscrollcommand=vsb.set)

#вызов внутренней функции перерисовки таблицы
refreshdata()

#функция выводит значения из таблицы students в соответствующие поля ввода для
выбранной позиции в сетке grid
# предварительно значения очищаются
def selectItem(a):
    curItem = self.tree.focus()
    self.entry_prep_name.delete(0, END)
    self.entry_prep_name.insert(0, self.tree.set(curItem, 'prep_name'))
    self.entry_prep_adress.delete(0, END)
    self.entry_prep_adress.insert(0, self.tree.set(curItem, 'prep_adress'))
    self.entry_prep_pers.delete(0, END)
    self.entry_prep_pers.insert(0, self.tree.set(curItem, 'prep_persinf'))
    self.entry_prep_inn.delete(0, END)
    self.entry_prep_inn.insert(0, self.tree.set(curItem, 'prep_inn'))
    self.combo_prep_name.delete(0, END)
    self.combo_prep_name.insert(0, self.tree.set(curItem, 'fac_id'))
    self.entry_prepsel_pos.delete(0, END)
    #для того, чтобы корректно записать значение prep_id делаем попытку привести
его к int
    # в случае неудачи выводим системное сообщение (ctypes был нужен для этого)
    # и ставим значение по умолчанию "1"
    try:
        valtochk = self.tree.set(curItem, 'prep_id')
        int(valtochk)
        self.entry_prepsel_pos.insert(0, self.tree.set(curItem, 'prep_id'))
    except ValueError:
        ctypes.windll.user32.MessageBoxW(0, "Значение номера позиции не
является числом", "Ошибка", 1)
        self.entry_prepsel_pos.insert(0, 1)

#устанавливаем курсор в сетке на первую позицию
try:
    child_id = self.tree.get_children()[0] # для ссылки на последний элемент в
наборе
    self.tree.focus(child_id)
    self.tree.selection_set(child_id)
    selectItem(0)
except:

```

```

ctypes.windll.user32.MessageBoxW(0, "Нет строк в таблице", "Ошибка", 1)

self.btn_del = ttk.Button(self, text='Удалить')
self.btn_del.grid(row=6, column=4,
                  sticky=W, padx=5, pady=10)
self.btn_del.bind('<Button-1>', lambda event: Main.delete_prep(self,
self.entry_prepssel_pos.get()))

#"подвешиваем" функцию определения текущей позиции selectItem на ожидание
отпускания кнопки над сеткой grid
self.tree.bind('<ButtonRelease-1>', selectItem)

self.focus_set()

# -----
#создание формы для работы таблицей predmets
def new_window_pred():
    self = tk.Toplevel(root)
    self.title('Таблица predmets')
    self.geometry('425x370+200+100')
    self.resizable(False, False)

    def refreshdata():
        db.c.execute('SELECT * FROM predmets')
        [self.tree.delete(i) for i in self.tree.get_children()]
        [self.tree.insert('', 'end', values=row) for row in db.c.fetchall()]

    label_id_pos = tk.Label(self, text='Позиция:')
    label_id_pos.grid(row=0, column=0,
                    sticky=W, padx=10)

    label_description = tk.Label(self, text='Предмет:')
    label_description.grid(row=1, column=0,
                        sticky=W, padx=10)

    label_select = tk.Label(self, text='Курс:')
    label_select.grid(row=2, column=0,
                    sticky=W, padx=10)

    self.entry_predssel_pos = ttk.Entry(self)
    self.entry_predssel_pos.grid(row=0, column=1,
                                colspan=4,
                                sticky=W, padx=10)

    self.entry_pred_name = ttk.Entry(self)
    self.entry_pred_name.grid(row=1, column=1,
                              colspan=4,
                              sticky=W, padx=10)

    self.combobox = ttk.Combobox(self, values=[u'1 курс', u'2 курс', u'3 курс', u'4
курс', u'5 курс', u'1 курс магистр', u'2 курс магистр', u'3 курс магистр'])

```

```

self.combobox.current(0)
self.combobox.grid(row=2, column=1,
                    columnspan=4,
                    sticky=W, padx=10)

btn_cancel = ttk.Button(self, text='Заккрыть', command=self.destroy)
btn_cancel.grid(row=5, column=5,
                sticky=W, padx=5, pady=10)

self.btn_ok = ttk.Button(self, text='Добавить')
self.btn_ok.grid(row=5, column=1,
                  sticky=W, padx=5, pady=10)
self.btn_ok.bind('<Button-1>', lambda event: Main.insert_pred(self,
self.entry_pred_name.get(),
                                                    self.combobox.get())
)

self.btn_edit = ttk.Button(self, text='Редактировать')
self.btn_edit.grid(row=5, column=0,
                    sticky=W, padx=5, pady=10)
self.btn_edit.bind('<Button-1>', lambda event: Main.update_pred(self,
self.entry_pred_name.get(),
                                                    self.combobox.get(
)))

self.tree = ttk.Treeview(self, columns=('pred_id', 'pred_name', 'pred_kurs'),
show='headings')
self.tree.column('pred_id', width=20, anchor=tk.CENTER)
self.tree.column('pred_name', width=235, anchor=tk.CENTER)
self.tree.column('pred_kurs', width=100, anchor=tk.CENTER)

self.tree.heading('pred_id', text='ID')
self.tree.heading('pred_name', text='ФИО')
self.tree.heading('pred_kurs', text='Курс')

self.tree.grid(row=6, column=0, columnspan=6,
                sticky='news', padx=5, pady=10)

vsb = tk.Scrollbar(self, orient="vertical", command=self.tree.yview)
vsb.grid(row=6, column=6, sticky='ns')
self.tree.configure(yscrollcommand=vsb.set)

refreshdata()

def selectItem(a):
    curItem = self.tree.focus()
    self.entry_pred_name.delete(0, END)
    self.entry_pred_name.insert(0, self.tree.set(curItem, 'pred_name'))
    self.combobox.delete(0, END)

```

```

self.combobox.insert(0, self.tree.set(curItem, 'pred_kurs'))
self.entry_predsel_pos.delete(0, END)

try:
    valtochk = self.tree.set(curItem, 'pred_id')
    int(valtochk)
    self.entry_predsel_pos.insert(0, self.tree.set(curItem, 'pred_id'))
except ValueError:
    ctypes.windll.user32.MessageBoxW(0, "Значение номера позиции не
является числом", "Ошибка", 1)
    self.entry_predsel_pos.insert(0, 1)

#устанавливаем курсор в сетке на первую позицию
try:
    child_id = self.tree.get_children()[0] # для ссылки на последний элемент в
наборе
    self.tree.focus(child_id)
    self.tree.selection_set(child_id)
    selectItem(0)
except:
    ctypes.windll.user32.MessageBoxW(0, "Нет строк в таблице", "Ошибка", 1)

self.btn_del = ttk.Button(self, text='Удалить')
self.btn_del.grid(row=5, column=4,
                    sticky=W, padx=5, pady=10)

self.btn_del.bind('<Button-1>', lambda event: Main.delete_pred(self,
self.entry_predsel_pos.get()))

self.tree.bind('<ButtonRelease-1>', selectItem)

self.focus_set()

#создание формы для работы таблицей rate
def new_window_rate():
    self = tk.Toplevel(root)
    self.title('Таблица rate')
    self.geometry('510x410+200+100')
    self.resizable(False, False)

    def refreshdata():
        db.c.execute('''SELECT rate_id, rate_max, rate_fact, t2.std_name AS std_name,
t3.pred_name AS pred_name FROM rate, students t2, predmets t3 WHERE rate.std_id = t2.std_id
AND rate.pred_id = t3.pred_id''')
        [self.tree.delete(i) for i in self.tree.get_children()]
        [self.tree.insert('', 'end', values=row) for row in db.c.fetchall()]

    label_id_pos = tk.Label(self, text='Позиция:')
    label_id_pos.grid(row=0, column=0,
                      sticky=W, padx=10)

```

```

label_description = tk.Label(self, text='ФИО:')
label_description.grid(row=1, column=0,
                        sticky=W, padx=10)

label_select = tk.Label(self, text='Дисциплина:')
label_select.grid(row=2, column=0,
                  sticky=W, padx=10)

label_adres = tk.Label(self, text='Макс.рейтинг:')
label_adres.grid(row=3, column=0,
                  sticky=W, padx=10)

label_descr = tk.Label(self, text='Текущий рейтинг:')
label_descr.grid(row=4, column=0,
                  sticky=W, padx=10)

self.entry_ratesel_pos = ttk.Entry(self)
self.entry_ratesel_pos.grid(row=0, column=1,
                             colspan=4,
                             sticky=W, padx=10)

self.combo_rate_name = ttk.Combobox(self)
Main.change_rate_combo_name(self)
self.combo_rate_name.grid(row=1, column=1,
                           colspan=4,
                           sticky=W, padx=10)

self.entry_rate_max = ttk.Entry(self)
self.entry_rate_max.grid(row=3, column=1,
                          sticky=W, padx=10)

self.entry_rate_current = ttk.Entry(self)
self.entry_rate_current.grid(row=4, column=1,
                              sticky=W, padx=10)

self.combo_rate_dis = ttk.Combobox(self)
Main.change_rate_combo(self)
self.combo_rate_dis.grid(row=2, column=1,
                          colspan=4,
                          sticky=W, padx=10)

btn_cancel = ttk.Button(self, text='Закреть', command=self.destroy)
btn_cancel.grid(row=5, column=5,
                 sticky=W, padx=5, pady=10)

self.btn_ok = ttk.Button(self, text='Добавить')
self.btn_ok.grid(row=5, column=1,
                  sticky=W, padx=5, pady=10)
self.btn_ok.bind('<Button-1>', lambda event: Main.insert_rate(self,
self.entry_rate_max.get()),

```

```

ent.get(),

.get(),

get()))

self.btn_edit = ttk.Button(self, text='Редактировать')
self.btn_edit.grid(row=5, column=0,
                    sticky=W, padx=5, pady=10)
self.btn_edit.bind('<Button-1>', lambda event: Main.update_rate(self,
self.entry_rate_max.get(),

self.entry_rate_curr

ent.get(),

self.combo_rate_name

.get(),

self.combo_rate_dis.

get()))

self.tree = ttk.Treeview(self, columns=('rate_id', 'rate_max', 'rate_fact',
'std_id', 'pred_id'), show='headings')
self.tree.column('rate_id', width=20, anchor=tk.CENTER)
self.tree.column('rate_max', width=100, anchor=tk.CENTER)
self.tree.column('rate_fact', width=100, anchor=tk.CENTER)
self.tree.column('std_id', width=160, anchor=tk.CENTER)
self.tree.column('pred_id', width=100, anchor=tk.CENTER)

self.tree.heading('rate_id', text='ID')
self.tree.heading('rate_max', text='Макс.рейтинг')
self.tree.heading('rate_fact', text='Факт.рейтинг')
self.tree.heading('std_id', text='ФНО')
self.tree.heading('pred_id', text='Дисциплина')

self.tree.grid(row=6, column=0, columnspan=6,
               sticky='news', padx=5, pady=10)

vsb = tk.Scrollbar(self, orient="vertical", command=self.tree.yview)
vsb.grid(row=6, column=6, sticky='ns')
self.tree.configure(yscrollcommand=vsb.set)

refreshdata()

def selectItem(a):
    curItem = self.tree.focus()
    self.entry_rate_max.delete(0, END)
    self.entry_rate_max.insert(0, self.tree.set(curItem, 'rate_max'))
    self.entry_rate_current.delete(0, END)
    self.entry_rate_current.insert(0, self.tree.set(curItem, 'rate_fact'))

```

```

self.combo_rate_name.delete(0, END)
self.combo_rate_name.insert(0, self.tree.set(curItem, 'std_id'))
self.combo_rate_dis.delete(0, END)
self.combo_rate_dis.insert(0, self.tree.set(curItem, 'pred_id'))
self.entry_ratesel_pos.delete(0, END)

try:
    valtochk = self.tree.set(curItem, 'rate_id')
    int(valtochk)
    self.entry_ratesel_pos.insert(0, self.tree.set(curItem, 'rate_id'))
except ValueError:
    ctypes.windll.user32.MessageBoxW(0, "Значение номера позиции не
является числом", "Ошибка", 1)
    self.entry_ratesel_pos.insert(0, 1)

#устанавливаем курсор в сетке на первую позицию
try:
    child_id = self.tree.get_children()[0] # для ссылки на последний элемент в
наборе
    self.tree.focus(child_id)
    self.tree.selection_set(child_id)
    selectItem(0)
except:
    ctypes.windll.user32.MessageBoxW(0, "Нет строк в таблице", "Ошибка", 1)

self.btn_del = ttk.Button(self, text='Удалить')
self.btn_del.grid(row=5, column=4,
                    sticky=W, padx=5, pady=10)

self.btn_del.bind('<Button-1>', lambda event: Main.delete_rate(self,
self.entry_ratesel_pos.get()))

self.tree.bind('<ButtonRelease-1>', selectItem)

self.focus_set()

#создание формы для работы таблицей faculty
def new_window_fac():
    self = tk.Toplevel(root)
    self.title('Таблица faculty')
    self.geometry('480x360+200+100')
    self.resizable(False, False)

    def refreshdata():
        db.c.execute(''SELECT * FROM faculty'')
        [self.tree.delete(i) for i in self.tree.get_children()]
        [self.tree.insert('', 'end', values=row) for row in db.c.fetchall()]

    label_id_pos = tk.Label(self, text='Позиция:')
    label_id_pos.grid(row=0, column=0,

```

```

        sticky=W, padx=10)

label_description = tk.Label(self, text='Название:')
label_description.grid(row=1, column=0,
                        sticky=W, padx=10)

self.entry_facsel_pos = ttk.Entry(self)
self.entry_facsel_pos.grid(row=0, column=1,
                           colspan=4,
                           sticky=W, padx=10)

self.entry_fac_name = ttk.Entry(self)
self.entry_fac_name.grid(row=1, column=1,
                          sticky=W, padx=10)

btn_cancel = ttk.Button(self, text='Закреть', command=self.destroy)
btn_cancel.grid(row=5, column=5,
                sticky=W, padx=5, pady=10)

self.btn_ok = ttk.Button(self, text='Добавить')
self.btn_ok.grid(row=5, column=1,
                  sticky=W, padx=5, pady=10)
self.btn_ok.bind('<Button-1>', lambda event: Main.insert_fac(self,
self.entry_fac_name.get()))

self.btn_edit = ttk.Button(self, text='Редактировать')
self.btn_edit.grid(row=5, column=0,
                    sticky=W, padx=5, pady=10)
self.btn_edit.bind('<Button-1>', lambda event: Main.update_fac(self,
self.entry_fac_name.get()))

self.tree = ttk.Treeview(self, columns=('fac_id', 'fac_name'), show='headings')
self.tree.column('fac_id', width=20, anchor=tk.CENTER)
self.tree.column('fac_name', width=200, anchor=tk.CENTER)

self.tree.heading('fac_id', text='ID')
self.tree.heading('fac_name', text='Название факультета')

self.tree.grid(row=6, column=0, colspan=6,
               sticky='news', padx=5, pady=10)

vsb = tk.Scrollbar(self, orient="vertical", command=self.tree.yview)
vsb.grid(row=6, column=6, sticky='ns')
self.tree.configure(yscrollcommand=vsb.set)

refreshdata()

def selectItem(a):
    curItem = self.tree.focus()

```



```

self.entry_fac_name.delete(0, END)
self.entry_fac_name.insert(0, self.tree.set(curItem, 'fac_name'))
self.entry_facsel_pos.delete(0, END)

try:
    valtochk = self.tree.set(curItem, 'fac_id')
    int(valtochk)
    self.entry_facsel_pos.insert(0, self.tree.set(curItem, 'fac_id'))
except ValueError:
    ctypes.windll.user32.MessageBoxW(0, "Значение номера позиции не
является числом", "Ошибка", 1)
    self.entry_facsel_pos.insert(0, 1)

#устанавливаем курсор в сетке на первую позицию
try:
    child_id = self.tree.get_children()[0] # для ссылки на последний элемент в
наборе

    self.tree.focus(child_id)
    self.tree.selection_set(child_id)
    selectItem(0)
except:
    ctypes.windll.user32.MessageBoxW(0, "Нет строк в таблице", "Ошибка", 1)

self.btn_del = ttk.Button(self, text='Удалить')
self.btn_del.grid(row=5, column=4,
                    sticky=W, padx=5, pady=10)

self.btn_del.bind('<Button-1>', lambda event: Main.delete_fac(self,
self.entry_facsel_pos.get()))

self.tree.bind('<ButtonRelease-1>', selectItem)

self.focus_set()

#-----
#создание формы для работы таблицей groups
def new_window_grup():
    self = tk.Toplevel(root)
    self.title('Таблица groups')
    self.geometry('510x410+200+100')
    self.resizable(False, False)

    def refreshdata():
        db.c.execute('''SELECT grup_id, date_create, t2.fac_name AS fac_name FROM groups,
faculty t2 WHERE groups.fac_id = t2.fac_id''')
        [self.tree.delete(i) for i in self.tree.get_children()]
        [self.tree.insert('', 'end', values=row) for row in db.c.fetchall()]

    label_id_pos = tk.Label(self, text='Позиция:')
    label_id_pos.grid(row=0, column=0,
                      sticky=W, padx=10)

```

```

label_description = tk.Label(self, text='Дата создания:')
label_description.grid(row=1, column=0,
                        sticky=W, padx=10)

label_fac = tk.Label(self, text='Факультет:')
label_fac.grid(row=2, column=0,
               sticky=W, padx=10)

self.entry_grsel_pos = ttk.Entry(self)
self.entry_grsel_pos.grid(row=0, column=1,
                           colspan=4,
                           sticky=W, padx=10)

self.entry_date_create = ttk.Entry(self)
self.entry_date_create.grid(row=1, column=1,
                             sticky=W, padx=10)

#создание списка с предопределенными значениями
self.combo_grup_name = ttk.Combobox(self)
Main.change_grup_combo_name(self)
self.combo_grup_name.grid(row=2, column=1,
                           colspan=4,
                           sticky=W, padx=10)

btn_cancel = ttk.Button(self, text='Закрыть', command=self.destroy)
btn_cancel.grid(row=5, column=5,
                 sticky=W, padx=5, pady=10)

self.btn_ok = ttk.Button(self, text='Добавить')
self.btn_ok.grid(row=5, column=1,
                  sticky=W, padx=5, pady=10)
self.btn_ok.bind('<Button-1>', lambda event: Main.insert_grup(self,
self.entry_date_create.get(),
                                                                self.combo_grup_name
e.get()))

self.btn_edit = ttk.Button(self, text='Редактировать')
self.btn_edit.grid(row=5, column=0,
                    sticky=W, padx=5, pady=10)
self.btn_edit.bind('<Button-1>', lambda event: Main.update_grup(self,
self.entry_date_create.get(),
                                                                self.combo_grup_name
e.get()))

self.tree = ttk.Treeview(self, columns=('grup_id', 'date_create', 'fac_id'),
show='headings')
self.tree.column('grup_id', width=20, anchor=tk.CENTER)
self.tree.column('date_create', width=200, anchor=tk.CENTER)
self.tree.column('fac_id', width=100, anchor=tk.CENTER)

```

```

self.tree.heading('grup_id', text='ID')
self.tree.heading('date_create', text='Дата создания')
self.tree.heading('fac_id', text='Факультет')

self.tree.grid(row=6, column=0, columnspan=6,
               sticky='news', padx=5, pady=10)

vsb = tk.Scrollbar(self, orient="vertical", command=self.tree.yview)
vsb.grid(row=6, column=6, sticky='ns')
self.tree.configure(yscrollcommand=vsb.set)

refreshdata()

def selectItem(a):
    curItem = self.tree.focus()
    self.entry_date_create.delete(0, END)
    self.entry_date_create.insert(0, self.tree.set(curItem, 'date_create'))
    self.combo_grup_name.delete(0, END)
    self.combo_grup_name.insert(0, self.tree.set(curItem, 'fac_id'))
    self.entry_grsel_pos.delete(0, END)

    try:
        valtochk = self.tree.set(curItem, 'grup_id')
        int(valtochk)
        self.entry_grsel_pos.insert(0, self.tree.set(curItem, 'grup_id'))
    except ValueError:
        ctypes.windll.user32.MessageBoxW(0, "Значение номера позиции не
является числом", "Ошибка", 1)
        self.entry_grsel_pos.insert(0, 1)

#устанавливаем курсор в сетке на первую позицию
try:
    child_id = self.tree.get_children()[0] # для ссылки на последний элемент в
наборе
    self.tree.focus(child_id)
    self.tree.selection_set(child_id)
    selectItem(0)
except:
    ctypes.windll.user32.MessageBoxW(0, "Нет строк в таблице", "Ошибка", 1)

self.btn_del = ttk.Button(self, text='Удалить')
self.btn_del.grid(row=5, column=4,
                  sticky=W, padx=5, pady=10)

self.btn_del.bind('<Button-1>', lambda event: Main.delete_grup(self,
self.entry_grsel_pos.get()))

self.tree.bind('<ButtonRelease-1>', selectItem)
#-----
#вызов функции создания формы для работы с таблицей students

```

```

def open_dialog(self):
    Main.new_window_stud()
#вызов функции создания формы для работы с таблицей predmets
def open_dialog_pred(self):
    Main.new_window_pred()
#вызов функции создания формы для работы с таблицей rate
def open_dialog_rate(self):
    Main.new_window_rate()
#вызов функции создания формы для работы с таблицей faculty
def open_dialog_fac(self):
    Main.new_window_fac()
#вызов функции создания формы для работы с таблицей prepods
def open_dialog_prep(self):
    Main.new_window_prep()
#вызов функции создания формы для работы с таблицей groups
def open_dialog_grup(self):
    Main.new_window_grup()
#задание класса базы данных
class DB:
    #конструктор
    def __init__(self):
        #подключаем базу
        self.conn = sqlite3.connect('u4eba.db')
        self.c = self.conn.cursor()
        #если еще не существует, создаем таблицу students
        self.c.execute(
            '''CREATE TABLE IF NOT EXISTS students (std_id integer primary key, std_name
text, std_adress text, std_addition text, std_kurs text, grup_id integer)'''
        )
        #выполняем транзакцию
        self.conn.commit()
        #если еще не существует, создаем таблицу predmets
        self.c.execute(
            '''CREATE TABLE IF NOT EXISTS predmets (pred_id integer primary key, pred_name
text, pred_kurs text)'''
        )
        self.conn.commit()
        #если еще не существует, создаем таблицу rate
        self.c.execute(
            '''CREATE TABLE IF NOT EXISTS rate (rate_id integer primary key, rate_max text,
rate_fact text, std_id integer, pred_id integer)'''
        )
        self.conn.commit()
        #если еще не существует, создаем таблицу faculty
        self.c.execute(
            '''CREATE TABLE IF NOT EXISTS faculty (fac_id integer primary key, fac_name
text)'''
        )
        self.conn.commit()
        #если еще не существует, создаем таблицу prepods
        self.c.execute(
            '''CREATE TABLE IF NOT EXISTS prepods (prep_id integer primary key, prep_name
text, prep_adress text, prep_persinf text, prep_inn text, fac_id integer)'''
        )
        self.conn.commit()
        #если еще не существует, создаем таблицу groups

```

```

self.c.execute(
    '''CREATE TABLE IF NOT EXISTS groups (grup_id integer primary key, date_create
text, fac_id integer)'''
)
self.conn.commit()

#функция по вставке значений в таблицу students
def insert_data_stud(self, std_name, std_address, std_addition, std_kurs, grup_id):
    self.c.execute('''INSERT INTO students(std_name, std_address, std_addition,
std_kurs, grup_id) VALUES (?, ?, ?, ?, ?)''',
                    (std_name, std_address, std_addition, std_kurs, grup_id))
    self.conn.commit()

#функция по вставке значений в таблицу predmets
def insert_data_pred(self, pred_name, pred_kurs):
    self.c.execute('''INSERT INTO predmets(pred_name, pred_kurs) VALUES (?, ?)''',
                    (pred_name, pred_kurs))
    self.conn.commit()

#функция по вставке значений в таблицу rate
def insert_data_rate(self, rate_max, rate_fact, std_id, pred_id):
    self.c.execute('''INSERT INTO rate(rate_max, rate_fact, std_id, pred_id) VALUES (?,
?, ?, ?)''',
                    (rate_max, rate_fact, std_id, pred_id))
    self.conn.commit()

#функция по вставке значений в таблицу groups
def insert_data_grup(self, date_create, fac_id):
    self.c.execute('''INSERT INTO groups(date_create, fac_id) VALUES (?, ?)''',
                    (date_create, fac_id))
    self.conn.commit()

#функция по вставке значений в таблицу faculty
def insert_data_fac(self, fac_name):
    self.c.execute('''INSERT INTO faculty(fac_name) VALUES (?)''',
                    (fac_name,))
    self.conn.commit()

#функция по вставке значений в таблицу preposts
def insert_data_prep(self, prep_name, prep_address, prep_persinf, prep_inn, fac_id):
    self.c.execute('''INSERT INTO preposts(prepost_name, prepost_address, prepost_persinf,
prep_inn, fac_id) VALUES (?, ?, ?, ?, ?)''',
                    (prep_name, prep_address, prep_persinf, prep_inn, fac_id))
    self.conn.commit()

```

#Когда интерпретатор Python читает исходный файл, он исполняет весь найденный в нем код.
Перед тем, как начать выполнять команды, он определяет несколько специальных переменных.
Например, если интерпретатор запускает некоторый модуль (исходный файл) как основную программу,
он присваивает специальной переменной `__name__` значение `"__main__"`.
Одна из причин делать именно так – тот факт, что иногда вы пишете модуль (файл с расширением `.py`),
предназначенный для непосредственного исполнения. Кроме того, он также может быть импортирован и
использован из другого модуля. Производя подобную проверку, вы можете сделать так, что код будет
исполняться только при условии, что данный модуль запущен как программа, и запретить исполнять его,

```
# если его хотят импортировать и использовать функции модуля отдельно.  
if __name__ == "__main__":  
    #создание основных классов  
    root = tk.Tk()  
    db = DB()  
    #создание основной формы приложения  
    app = Main(root)  
    app.pack()  
    root.title("Учеба в вузе")  
    root.geometry("470x270+300+200")  
    root.resizable(False, False)  
    #зацикливаем выполнение программы  
    root.mainloop()
```