Министерство образования и науки Российской Федерации

НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

М.Г. ЗАЙЦЕВ

ОБЪЕКТНО-ОРИЕНТИРОВАННЫЙ АНАЛИЗ И ПРОГРАММИРОВАНИЕ

Утверждено Редакционно-издательским советом университета
в качестве учебного пособия

НОВОСИБИРСК
2017

УДК 004.434(075.8)
З-177

Рецензенты:

канд. техн. наук, доц. *В.Г. Кобылянский*

ст. преподаватель кафедры ТС и ВС СибГУТИ *Л.Ф. Лебеденко*

Зайцев М.Г.

З-177 Объектно-ориентированный анализ и программирование : учебное пособие / М.Г. Зайцев. – Новосибирск : Изд-во НГТУ, 2017. – 84 с.

ISBN 978-5-7782-3308-9

Предназначено для студентов, обучающихся по направлению: 38.03.05 «Бизнес-информатика», 09.03.03 «Прикладная информатика» дневной заочной и дистанционной форм обучения дисциплинам «Объектно-ориентированный анализ и программирование», «Разработка программных приложений». Излагаются вопросы объектно-ориентированного анализа с применением моделей языка UML и программирования на языке C#. Примеры упражнений выполнены в среде Visual Studio.

Работа подготовлена кафедрой теоретической
и прикладной информатики

УДК 004.434(075.8)

ISBN 978-5-7782-3308-9

ОБЪЕКТНО-ОРИЕНТИРОВАННЫЕ МЕТОДЫ АНАЛИЗА И ПРОЕКТИРОВАНИЯ ПО

МЕТОДОЛОГИЯ ОБЪЕКТНО-ОРИЕНТИРОВАННОГО ПРОГРАММИРОВАНИЯ

В конце XX века традиционные методы структурного программирования перестали справляться с растущей сложностью программ. Тогда на смену структурному программированию пришла методология объектно-ориентированного программирования. Объектно-ориентированное программирование (ООП) – совокупность принципов, технологий, а также инструментальных средств для создания программных систем на основе архитектуры взаимодействия объектов. В отличие от структурного подхода, основанного на функциональной декомпозиции программ, в объектно-ориентированном подходе в качестве отдельных структурных единиц программы рассматриваются не процедуры и функции, а классы и объекты с соответствующими свойствами и методами. В состав объектно-ориентированной методологии входят: объектно-ориентированный анализ, объектно-ориентированное проектирование, объектно-ориентированное программирование.

Цель объектно-ориентированного анализа – создание моделей с использованием объектно-ориентированного подхода. Это подход, при котором требования формируются на основе понятий классов и объектов, составляющих словарь предметной области. Под предметной областью мы будем понимать область человеческой деятельности, для которой разрабатывается программное обеспечение (ПО).

Объектно-ориентированное проектирование соединяет в себе объектную декомпозицию и представления логической физической, статической и динамической моделей проектируемой системы ПО.

Объектно-ориентированное программирование основано на представлении программы в виде совокупности объектов. Каждый из

объектов есть реализация определенного класса, а классы организованы в иерархию на принципах наследуемости. В основу объектно-ориентированного подхода (ООП) положена объектная декомпозиция. Статическая структура системы при этом описывается в терминах объектов и связей между ними. Поведение системы реализуется через обмен сообщениями между объектами. Каждый объект системы обладает своим собственным поведением, моделирующим поведение объекта реального мира.

ОБЪЕКТНАЯ МОДЕЛЬ. ОСНОВНЫЕ ПРИНЦИПЫ ПОСТРОЕНИЯ

Объектная модель – это концептуальная основа объектно-ориентированного подхода. Она построена на следующих основных принципах: абстрагирование, инкапсуляция, модульность, иерархия. Абстрагирование – это выделение существенных характеристик объекта, которые отличают его от всех других видов объектов, и отвлечение от незначительных деталей. Через абстрагирование мы можем управлять сложностью системы, концентрируясь на существенных свойствах объектов. Через абстрагирование мы выявляем внешние особенности объекта и отделяем существенные признаки его поведения от деталей их реализации. Главной задачей объектно-ориентированного проектирования является выбор правильного набора абстракций (классов) для заданной предметной области.

Инкапсуляция – физическая локализация свойств и поведения в рамках единственной абстракции (рассматриваемой как «черный ящик»), скрывающая их реализацию за общедоступным интерфейсом. Используя инкапсуляцию, мы отделяем друг от друга элементы объекта, определяющие его устройство, от элементов, определяющих его поведение. Инкапсуляция используется для того, чтобы отделить интерфейс объекта, который отражает его внешнее поведение, от механизмов реализации объекта. Объектный подход предполагает, что собственные ресурсы, которыми могут манипулировать только операции самого объекта, скрыты от внешней среды. Во многом инкапсуляция подобна сокрытию информации – это возможность скрывать многочисленные детали объекта от внешнего мира. Внешний мир объекта – это все, что находится вне его, включая остальную часть системы.

Свойство системы, связанное с возможностью ее декомпозиции на ряд внутренне сильно сцепленных, но слабо связанных между собой

подсистем (модулей), называется модульностью. Используя модульность, мы снижаем сложность системы. Она позволяет нам выполнять независимую разработку отдельных модулей. Инкапсуляция и модульность создают барьеры между абстракциями.

Ранжированная или упорядоченная система абстракций, расположение их по уровням – это иерархия. Основные виды иерархических структур применительно к системам ПО – это структуры классов (иерархия по номенклатуре) и структуры объектов (иерархия по составу). Простое и множественное наследование (один класс использует структурную или функциональную часть соответственно одного или нескольких других классов) – пример иерархии классов, а агрегация – пример иерархии объектов.

ОБЪЕКТНАЯ МОДЕЛЬ. ОСНОВНЫЕ ЭЛЕМЕНТЫ

К основным элементам объектной модели относят: объект, класс, атрибут, операцию, полиморфизм (интерфейс), компонент, связи.

Под объектом мы понимаем осязаемую сущность (предмет, явление или процесс), которая имеет четко определяемое поведение. Объект обладает состоянием, поведением и индивидуальностью.

Одно из возможных условий, в которых объект может существовать, называется состоянием объекта. Состояние может изменяться со временем. Состояние объекта характеризуется перечнем (статических) свойств и текущими значениями (динамическими) каждого из этих свойств. Значения свойств (атрибутов) и связи с другими объектами определяют состояние объекта.

Поведение объекта определяют его действия и реакция на запросы от других объектов. Поведение характеризуется воздействием объекта на другие объекты, изменяющим их состояние. Поведение определяется также набором сообщений, которые объект может воспринимать (операций, которые может выполнять объект).

Индивидуальность – это свойства объекта, отличающие его от всех других объектов. Каждый объект обладает индивидуальностью. Объекты, сходные по структуре и поведению, объединяют в один класс. Термины «экземпляр класса» и «объект» используются как синонимы.

В языке моделирования UML объекты графически представляют, как показано на рис. 1.

Класс – описывает множество объектов, которые обладают одинаковыми свойствами, поведением, связями и семантикой. Класс объединяет

в себе описание данных (атрибутов) и поведения (операций). Класс является определением объекта и служит шаблоном для создания объектов. На рис. 2 показано графическое представление класса в языке UML. Его изображают в виде прямоугольника, состоящего из трех частей. Имя класса заносится в первую часть, его атрибуты – во вторую. В последнюю часть заносят операции класса. Именно они отражают поведение объектов класса. Объект класса является экземпляром класса. Одна из основных задач объектно-ориентированного проектирования – выявление объектов и описание их классов.

Рис. 1. Графическое представление объектов в UML

Атрибут – поименованное свойство класса, определяющее диапазон допустимых значений, которые могут принимать экземпляры данного свойства. Атрибут – это элемент информации, связанный с классом. Например, у класса Компания могут быть атрибуты Название, Адрес и Число служащих.

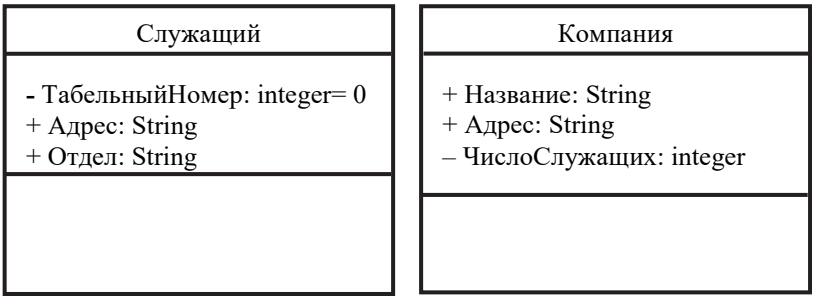


Рис. 2. Графическое представление класса

Поскольку атрибуты содержатся внутри класса, они скрыты от других классов. В связи с этим может понадобиться указать, какие классы имеют право читать и изменять атрибуты. Это свойство называется видимостью атрибута. У атрибута можно определить три возможных значения этого параметра. Рассмотрим каждый из них на примере (см. рис. 2). Пусть имеется класс Служащий и класс Компания. Видимость атрибута Public (общий, открытый) предполагает, что он будет виден всеми остальными классами. Любой класс может просмотреть или изменить значение атрибута. Поэтому класс Компания может изменить значение атрибута Адрес класса Служащий. В языке UML общему атрибуту предшествует знак «+». Видимость атрибута Private – закрытый, секретный. Такой атрибут не виден никаким другим классам. Класс Служащий будет знать значение атрибута Адрес и сможет изменять его, но класс Компания не сможет его ни увидеть, ни изменить. Если это понадобится, он должен попросить класс Служащий просмотреть или изменить значение этого атрибута, что обычно делается с помощью операций с уровнем доступа Public. Закрытый атрибут обозначается знаком «–» в соответствии с нотацией UML. Видимость атрибута Protected – защищенный. Такой атрибут доступен только самому классу и его потомкам в иерархии наследования. В языке UML для обозначения защищенного атрибута используют знак «#». Атрибуты рекомендуется делать закрытыми или защищенными. При этом удается избежать ситуации, когда значение атрибута изменяется всеми классами системы. Вместо этого логика изменения атрибута будет заключена в том же классе, что и сам этот атрибут.

Определенное воздействие одного объекта на другой с целью вызвать соответствующую реакцию называется операцией. Операция – это реализация услуги, которую можно запросить у любого объекта данного класса. Операции отражают поведение объекта. Операция-запрос не изменяет состояния объекта. Операция-команда может изменить состояние объекта. Результат операции зависит от текущего состояния объекта. Как правило, в объектных и объектно-ориентированных языках программирования операции, выполняемые над данным объектом, называются методами и являются составной частью определения класса. Операции реализуют связанное с классом поведение (иначе говоря, реализуют обязанности класса). Операция включает три части: имя, параметры и тип возвращаемого значения. Параметры – это аргументы, получаемые операцией «на входе». Тип возвращаемого значения относится к результату действия операции.

Понятие полиморфизма может быть интерпретировано как способность класса принадлежать более чем одному типу. Полиморфизм – это способность скрывать множество различных реализаций под единственным общим интерфейсом. Интерфейс – совокупность операций, определяющих набор услуг класса или компонента. Интерфейс не определяет внутреннюю структуру, все его операции имеют открытую видимость.

Компонент – относительно независимая и замещаемая часть системы, выполняющая четко определенную функцию в контексте заданной архитектуры. Компонент представляет собой физическую реализацию проектной абстракции и может быть: компонентом исходного кода; компонентом времени выполнения (run time); исполняемым компонентом. Компонент обеспечивает физическую реализацию набора интерфейсов.

Между элементами объектной модели существуют различные виды связей. К основным типам связей относятся связи ассоциации, зависимости и обобщения. *Ассоциация* – это семантическая связь между классами. Ее изображают на диаграмме классов в виде обыкновенной линии (рис. 3). Ассоциация отражает структурные связи между объектами различных классов.

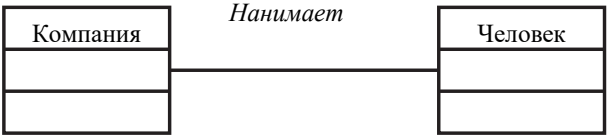


Рис. 3. Ассоциация

Агрегация представляет собой форму ассоциации – более сильный тип связи между целым (составным) объектом и его частями (компонентными объектами). Язык UML обеспечивает ограниченную поддержку агрегации. Сильная форма агрегации является в UML композицией. В композиции составной объект может физически содержать компонентные объекты. Компонентный объект может принадлежать только одному составному объекту. Слабая форма агрегации в UML называется просто агрегацией. При этом составной объект физически не содержит компонентного объекта. Один компонентный объект может обладать несколькими связями ассоциации или агрегации. Агрегация изображается линией между классами с ромбом на стороне целого объекта (рис. 4).

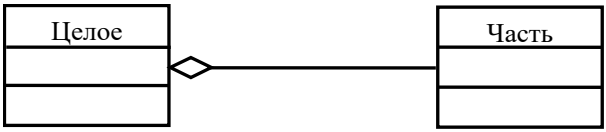


Рис. 4. Агрегация

Сплошной ромб представляет композицию (рис. 5).

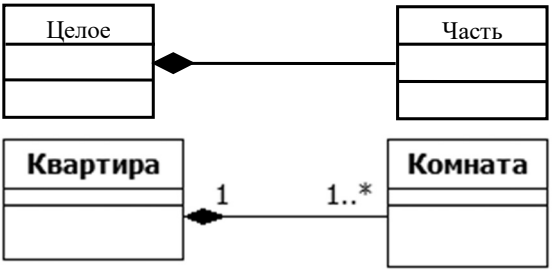


Рис. 5. Композиция

Мощность показывает, как много объектов участвует в связи. Мощность – это число объектов одного класса, связанных с одним объектом другого класса. Понятие мощности связи в объектной модели аналогично понятиям мощности и класса принадлежности связи в модели «сущность–связь» (с точностью до расположения показателя мощности на диаграмме). Для каждой связи можно обозначить два показателя мощности – по одному на каждом конце связи. В языке UML приняты следующие нотации для обозначения мощности.

Значение мощности	
Мощность	**Значение**
*	Много
0	Нуль
1	Один
0..*	Нуль или больше
1..*	Один или больше
0..1	Нуль или один
1..1	Ровно один

Если будет разрабатываться система информатизации университета, в ней можно определить классы Группа и Студент. Между ними

установлена связь Агрегация. Каждый студент может входить в состав только одной студенческой группы, а группа, например, может содержать от 15 до 30 студентов. Диаграмма классов для этого примера будет выглядеть следующим образом (рис. 6).

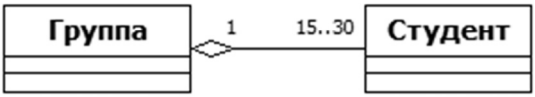


Рис 6. Мощность связи

Зависимость – связь между двумя элементами модели, при которой изменения в спецификации одного элемента могут повлечь за собой изменения в другом элементе. Зависимость – слабая форма связи между клиентом и сервером (клиент зависит от сервера и не имеет знаний о нем). Зависимость изображается пунктирной линией, направленной от клиента к серверу (рис. 7).

Рис 7. Зависимость

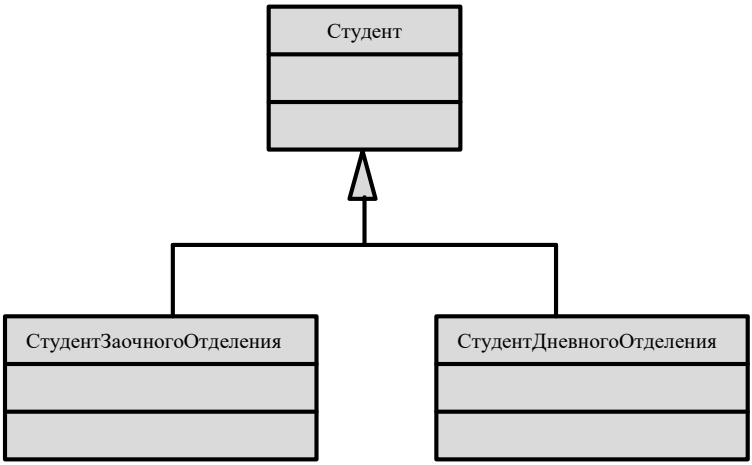


Рис. 8. Обобщение

Зависимость между двумя элементами имеет место в том случае, если изменения в определении одного элемента могут повлечь за собой изменения в другом. Зависимость может быть вызвана разными обстоятельствами: один класс посылает сообщение другому; один класс включает часть данных другого класса; один класс использует другой в качестве параметра операции.

Обобщение – связь «тип–подтип» реализует механизм наследования. Большинство объектно-ориентированных языков непосредственно поддерживает концепцию наследования. Она позволяет одному классу наследовать все атрибуты, операции и связи другого. В языке UML связи наследования называют обобщениями и изображают в виде стрелок от класса-потомка к классу-предку (рис. 8).

Общие атрибуты, операции и/или связи отображаются на верхнем уровне иерархии. Помимо наследуемых, каждый подкласс имеет свои собственные уникальные атрибуты, операции и связи.

КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Что такое объектно-ориентированное программирование?
2. Что такое предметная область?
3. В чем особенность объектно-ориентированного анализа?
4. Что такое абстрагирование?
5. Что такое инкапсуляция?
6. В чем состоит принцип модульности применительно к программному обеспечению?
7. Какие бывают виды иерархии?
8. Назовите и охарактеризуйте основные элементы объектной модели.
9. Назовите и охарактеризуйте виды связей между элементами объектной модели.

УНИФИЦИРОВАННЫЙ ЯЗЫК МОДЕЛИРОВАНИЯ UML

Унифицированный язык моделирования UML (Unified Modeling Language) представляет собой язык для определения, представления, проектирования и документирования программных систем, организационно-экономических систем, технических систем и других систем различной природы. UML содержит стандартный набор диаграмм и нотаций самых разнообразных видов.

UML применяется в процессе объектно-ориентированного анализа и проектирования, для моделирования систем ПО и предметных областей. UML начали создавать в середине 90-х годов XX века. Полное описание UML можно найти на сайте http://www.omg.org.

Мы рассмотрим диаграммы UML трех видов: диаграмму классов для моделирования статической структуры классов системы и связей между ними; диаграмму вариантов использования (прецедентов) для моделирования бизнес-процессов и функциональных требований к создаваемой системе; диаграмму последовательности для моделирования процесса обмена сообщениями между объектами.

ДИАГРАММЫ ВАРИАНТОВ ИСПОЛЬЗОВАНИЯ (ПРЕЦЕДЕНТОВ)

Модель вариантов использования (use case) была предложена Иваром Якобсоном в 1986 г. для описания функциональных требований. Вариант использования – это последовательность действий (транзакций), которые система выполняет в ответ на событие, инициируемое некоторым внешним объектом (действующим лицом). Вариант использования (прецедент) отражает представление о поведении системы с точки зрения пользователя. Он описывает типичное взаимодействие между пользователем и системой. Вариант использования выявляется в процессе обсуждения с пользователем тех услуг (сервисов), которые

он хотел бы получить от системы ПО. Действующее лицо (эктор) – это роль, которую пользователь играет по отношению к системе, а не конкретный человек. На диаграммах вариантов использования экторы изображаются в виде стилизованных человеческих фигурок. Однако действующим лицом может быть и внешняя система, которая получает информацию от данной системы. Действующие лица могут относиться к одной из трех категорий: пользователи системы, другие системы, взаимодействующие с данной, время. Если в системе периодически возникает некоторое событие, то время становится действующим лицом.

Для графического представления прецедентов в языке UML предназначены диаграммы вариантов использования. На рис. 9 показан пример такой диаграммы для приложения Конвертер_p1_p2, которое преобразует действительные числа из системы счисления с основанием p1 в систему счисления с основанием p2. Основания систем счисления p1, p2 принадлежат диапазону значений от 2 до 16.

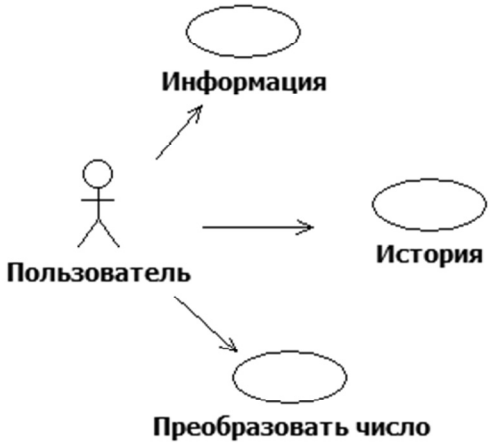


Рис. 9. Диаграмма вариантов использования
для приложения Конвертер_p1_p2

На диаграмме представлены: одно действующее лицо («Пользователь»), изображенное в форме человеческой фигурки; прецеденты, изображенные эллипсами; связи между действующим лицом и прецедентами, а также между самим прецедентами, изображенные линиями. На диаграмме показаны три базовых прецедента: «Информация», «История», «Преобразовать». Эти прецеденты будет выполнять моделиру-

емое приложение. На диаграмме прецедентов показано взаимодействие между действующим лицом и вариантами использования. Прецеденты, которые непосредственно запускает Пользователь, называются базовыми. Прецеденты, которые входят в состав базового, могут быть прецедентами включения или расширения. Направленная от варианта использования к действующему лицу стрелка показывает, что вариант использования предоставляет некоторую информацию для действующего лица. Действующие лица (экторы) могут играть различные роли по отношению к варианту использования. Они могут сами непосредственно в нем участвовать или пользоваться его результатами. Если стрелка направлена от эктора к прецеденту, то действующее лицо участвует в прецеденте; если в противоположном направлении, то эктор пользуется результатом этого прецедента. Цель построения диаграмм вариантов использования – документирование функциональных требований к системе в самом общем виде, поэтому они должны быть предельно простыми. Диаграмма вариантов использования представляет функциональные требования к системе в самом общем виде. На этапе проектирования системы требуется более детальное представление прецедента. Поэтому диаграмму вариантов использования дополняют документом, который носит название «сценарий варианта использования» или «поток событий». Цель потока событий – подробное документирование взаимодействия действующего лица с системой, которое реализуется в ходе прецедента. В сценарии прецедента необходимо описать все, что служит удовлетворению запроса эктора. Цель сценария прецедента – описание того, что будет делать система, а не того, как она будет это делать. Один из шаблонов сценария прецедента включает такие разделы: краткое описание, предусловия, основной поток событий, альтернативные потоки событий, постусловия, расширения.

Рассмотрим содержание этих разделов.

Краткое описание. Начинается сценарий прецедента с того, что мы кратко описываем происходящие в нем события. Прецедент «Преобразование числа», например, может быть кратко описан так.

Он позволяет пользователю перевести действительное число, представленное в системе счисления с основанием p1, в систему счисления с основанием p2.

Предусловия. Предусловия прецедента – это такие условия, которые должны быть выполнены перед его началом. Для одного преце-

дента предусловием может быть выполнение в это самое время другого прецедента. У прецедента может не быть предусловий.

Потоки событий основной и альтернативный. Подробное описание прецедента выполняется в основном и в альтернативных потоках событий. Поток событий описывается с точки зрения пользователя. В нем мы указываем, что будет делать система, а не как она будет делать это.

В основном потоке событий описывается типичное выполнение прецедента без ошибок со стороны действующего лица или программы. Основной поток описывает нормальный ход событий (при отсутствии ошибок). Основной поток событий может разветвляться на несколько потоков. Если в ходе его выполнения возникают ошибки, их обработку мы выполняем в альтернативных потоках. Потоки событий прецедента «Преобразовать число», например, могут выглядеть так, как показано ниже.

ОСНОВНОЙ ПОТОК СОБЫТИЙ

1. Пользователь выбирает основание системы счисления p1 исходного числа.
2. Пользователь выбирает основание системы счисления p2 результата.
3. Пользователь вводит действительное число, представленное в системе счисления с основанием p1.
4. Пользователь вводит команду «Преобразовать число».
5. Система выводит введенное пользователем число, представленное в системе счисления с основанием p2.
6. Система сохраняет исходные данные и результат преобразования в «Историю».

Альтернативный поток событий 1. Введенное пользователем число выходит за границы допустимого диапазона
(*к пункту 3 основного потока событий*)

3.1. Пользователь получает окно с сообщением.
3.2. Приложение переходит в режим «Ввод и редактирование».

Альтернативный поток событий 2. Количество разрядов в результате превышает размер поля вывода визуального компонента (*к пункту 4 основного потока событий*)

4.1. Пользователь получает окно с сообщением.

4.2. Приложение переходит в режим «Ввод и редактирование».

При составлении описания потока событий следует стремиться к тому, чтобы оно легко читалось и состояло из простых предложений, которые написаны в единой грамматической форме. В описании основного потока событий в каждом пункте необходимо явно указывать, кто выполняет действие – эктор или система. Не надо описывать детали работы пользователя с интерфейсом.

В альтернативных потоках событий описывают ошибочные ситуации: ошибка в действиях пользователя, бездействие действующего лица, ошибка в поведении разрабатываемой системы, низкая производительность системы.

Постусловия. Условия, которые должны быть выполнены после завершения прецедента, называются постусловиями. С помощью постусловий можно указать порядок выполнения прецедентов. Когда за одним прецедентом должен выполняться другой, это можно описать как постусловие. Постусловия у прецедента могут отсутствовать.

Расширения. Если в основном потоке событий можно выделить логически связанную последовательность событий, которая выполняется не каждый раз (иногда), то ее выносят в расширение.

Связи, которые присутствуют на диаграмме прецедентов, бывают следующих видов: связь коммуникации, связь включения (include), связь расширения (extend) и связь обобщения. **Связь коммуникации** – это связь между действующим лицом и прецедентом. Ее изображают с помощью линии со стрелкой (это однонаправленная ассоциация). Направление стрелки позволяет понять, кто инициирует коммуникацию. Если в потоке событий базового прецедента можно выделить логически связанную часть, которая выполняется каждый раз и может входить в состав нескольких прецедентов, мы ее выделяем в отдельный прецедент. Выделенный прецедент становится для базового прецедента прецедентом включения и обозначается на диаграмме **связью включения.** Так, если в приложении «Конвертер» мы проанализируем поток событий базового прецедента «Преобразование числа», то можем выявить несколько фрагментов в потоке событий, которые можно выделить в отдельные

прецеденты и оформить их как прецеденты включения. Это следующие прецеденты: «Ввести число», «Перевести в десятичное», «Перевести в систему счисления с основанием p2» (рис. 10). Эти прецеденты выполняются всегда. Но мы можем выявить и несколько прецедентов, которые будут выполняться не всегда в составе базового прецедента, и оформить их как прецеденты расширения. Это такие прецеденты, как «Выбрать основание систему счисления p1», «Выбрать основание систему счисления p1». Связи включения и расширения изображаются в виде зависимостей, как показано на рис. 10.

Рис. 10. Связи включения и расширения

Выявить прецеденты на стадии формирования требований к системе ПО нам необходимо для того, чтобы учесть все запросы пользователей. Если мы требование не выявили, то мы его не сможем и реализовать. Модель прецедентов получила широкое распространение в силу следующих причин. Ее можно использовать в любых технологиях программирования. Она служит удобным средством для обсуждения с заказчиками функций будущей системы ПО. Она позволяет отделить разрабатываемую систему ПО, представленную прецедентами, от внешнего мира, представленного действующими лицами. Прецеденты определяют интерфейсы системы ПО. Прецеденты служат хорошей основой для написания тестов и пользовательской документации.

ДИАГРАММЫ ВЗАИМОДЕЙСТВИЯ

Диаграммы взаимодействия описывают поведение взаимодействующих групп объектов (в рамках варианта использования или некоторой операции класса). Не следует перегружать диаграммы информацией, чтобы они были легко понимаемы. Поэтому диаграмму взаимодействия рекомендуют строить отдельно для основного и каждого из альтернативных потоков событий прецедента. На такой диаграмме отображаются ряд объектов и те сообщения, которыми они обмениваются между собой. **Сообщение** – средство, с помощью которого объект-отправитель запрашивает у объекта-получателя выполнение одной из его операций.

Существуют два вида диаграмм взаимодействия: диаграммы последовательности и кооперативные диаграммы. Диаграммы последовательности отражают временную последовательность событий, происходящих в рамках варианта использования. Основной поток событий варианта использования «Перевести число» показан на рис. 11.

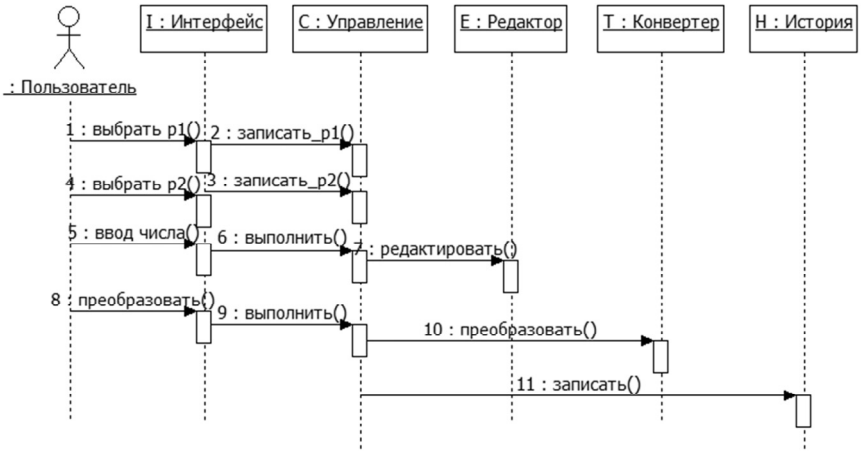


Рис. 11. Диаграмма последовательности

Действующее лицо прецедента «Пользователь» показано в верхней левой части диаграммы. Объекты, реализующие выполнение прецедента «Перевести число», изображены в верхней части диаграммы. Сообщения, которыми обмениваются объекты между собой и действующее лицо с объектом, изображены стрелками; над ними написаны номер и имя сообщения. Время существования объекта в приложении показано пунктирной вертикальной линией, называемой линией жизни объекта.

Порядок следования сообщений во времени показан на странице сверху вниз. Определяя сообщение, можно указывать кроме имени такие атрибуты, как аргументы, тип возвращаемого значения. Если объект посылает сообщение сам себе, это называется самоделегированием. Самоделегирование изображают стрелкой, которая начинается и кончается на линии жизни одного и того же объекта.

Один из вопросов, возникающих при построении диаграммы прецедентов: откуда появляются объекты, которые мы размещаем на ней? Часть этих объектов мы находим при анализе потока событий сценария прецедента (объект для редактирования, объект для преобразования, объект для хранения истории). Другую часть объектов (объект для взаимодействия пользователя с приложением, объект для управления обменом сообщениями между другими объектами) мы добавляем, исходя из необходимости обеспечить возможность взаимодействия пользователя с приложением и управления обменом сообщениями между объектами.

ДИАГРАММЫ КЛАССОВ

Диаграммы классов встречаются чаще других при объектно-ориентированном моделировании систем ПО. Диаграммы классов полезны при описании классов, объекты которых участвуют в реализации отдельного прецедента или сложной операции класса. На них отображают связи между классами. Связь между классами появляется, если объекты классов обмениваются между собой сообщениями. На диаграммах классов изображаются также атрибуты классов, операции классов и ограничения, которые накладываются на связи между классами. Диаграмма классов для варианта использования «Перевести число» показана на рис. 12. На этой диаграмме присутствуют четыре класса: Интерфейс (класс, обеспечивающий взаимодействие пользователя с приложением), Конвертер (преобразователь чисел из системы счисления с основанием p1 в систему счисления с основанием p2), Редактор (обеспечивает ввод, хранение и редактирование чисел, представленных в системе счисления с основанием p1) и История (журнал сеанса взаимодействий пользователя с системой).

Линии, которые связывают классы на диаграмме, отражают характер взаимодействия между классами. В изображении классов присутствуют также стереотипы. Так, стереотип «boundary» означает, что это граничный класс, отвечающий за взаимодействие действующего лица

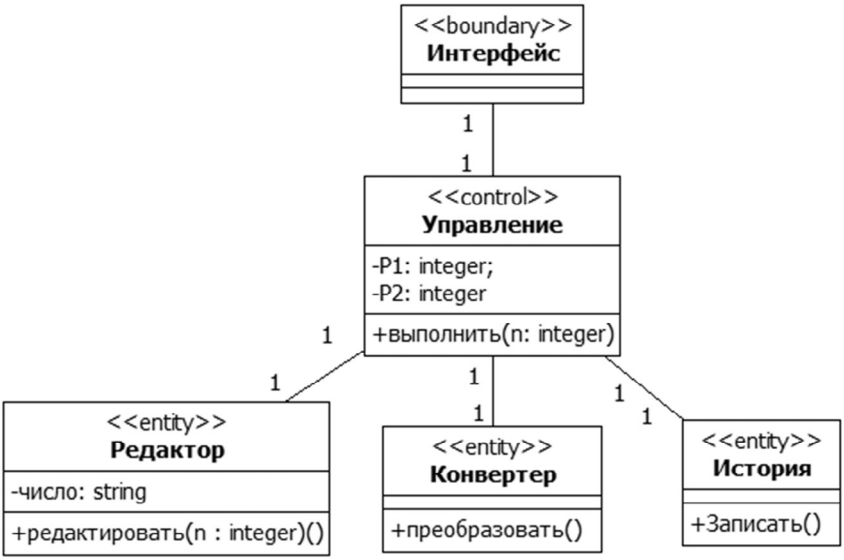


Рис. 12. Диаграмма классов для прецедента «Перевести число»

и приложения. Класс со стереотипом «entity» отвечает за хранение и обработку данных. Класс со стереотипом «control» отвечает за управление обменом сообщениями между объектами.

КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Что описывает диаграмма прецедентов?
2. Что описывает сценарий прецедента?
3. Что описывают диаграмма последовательности?
4. Что такое сообщение?
5. Что определяет диаграмма классов?

КЛАССЫ C#

Классы являются средствами объектно-ориентированного программирования языков высокого уровня.

ОПИСАНИЕ КЛАССА C#

В языке C# класс является типом данных, определяемым пользователем. Класс описывает свойства и поведение множества объектов. С помощью объектов мы можем моделировать явления, предметы и процессы реального мира. В описание класса можно помещать такие элементы, как *данные* и *функции*, предназначенные для обработки данных. Обязательными в описании класса являются ключевое слово class, имя и тело класса. Тело класса размещают в фигурных скобках. Таким образом, простейшее описание класса может выглядеть так:

```
class Simple {}
```

Такой класс называется пустым, он не содержит ни одного элемента. Тело класса может содержать элементы его описания. Класс может наследовать от одного класса и/или от одного или нескольких интерфейсов. Их мы указываем в описании как предков. В описании класса могут присутствовать необязательные атрибуты и спецификаторы, определяющие различные характеристики класса:

```
[атрибуты] [спецификаторы] class имя_класса [: предки]
тело_класса
```

Тело класса – это *список* описаний его элементов, заключенный в *фигурные скобки*. *Список* может быть пустым, если класс не содержит ни одного элемента. *Спецификаторы* определяют свойства класса, а также доступность его для других элементов программы. Некоторые значения спецификаторов перечислены в табл. 1. Класс можно описывать непосредственно внутри пространства имен или внутри другого класса. В последнем случае *класс* называется *вложенным*.

Таблица 1

Спецификаторы класса

№ п/п	Спецификатор	Описание
1	Public	Доступ не ограничен
2	Private	Используется для вложенных классов. Доступ только из элементов класса, внутри которого описан данный класс
3	Static	Статический класс

К спецификаторам доступа относятся спецификаторы 1, 2. Они показывают, откуда можно непосредственно обращаться к данному классу. Класс определяет характеристики и поведение некоторого *множества* объектов этого класса, называемых *экземплярами* или *объектами класса*. Объекты, или экземпляры, класса создаются с помощью *операции* new, например:

```
Simple a = new Simple();// создание экземпляра класса
                        // Simple

Simple b = new Simple();// создание другого экземпляра
                        // класса Simple
```

Память под объекты выделяется в куче (область памяти под ссылочные значения). Для каждого объекта при его создании в памяти выделяется отдельная область, в которой хранятся его данные. Кроме того, в классе могут присутствовать *статические элементы*, которые существуют в единственном экземпляре для всех объектов класса. Часто статические данные называют *данными класса*, а остальные – *данными экземпляра*. Функциональные элементы класса всегда хранятся в единственном экземпляре. Для работы с данными класса используются *методы класса* (*статические методы*), для работы с данными экземпляра – *методы экземпляра*, или просто *методы*. Основные элементы класса – это поля и методы. Кроме того, в классе можно задавать другие элементы, такие как свойства, события, индексаторы, операции, конструкторы, деструкторы, а также типы (рис. 13).

Приведенные на рис. 13 элементы класса имеют следующее назначение. Неизменяемые значения хранятся в константах класса. Методы, описанные в классе, предназначены для выполнения вычислений или реализуют другие действия, которые выполняет класс или экземпляр

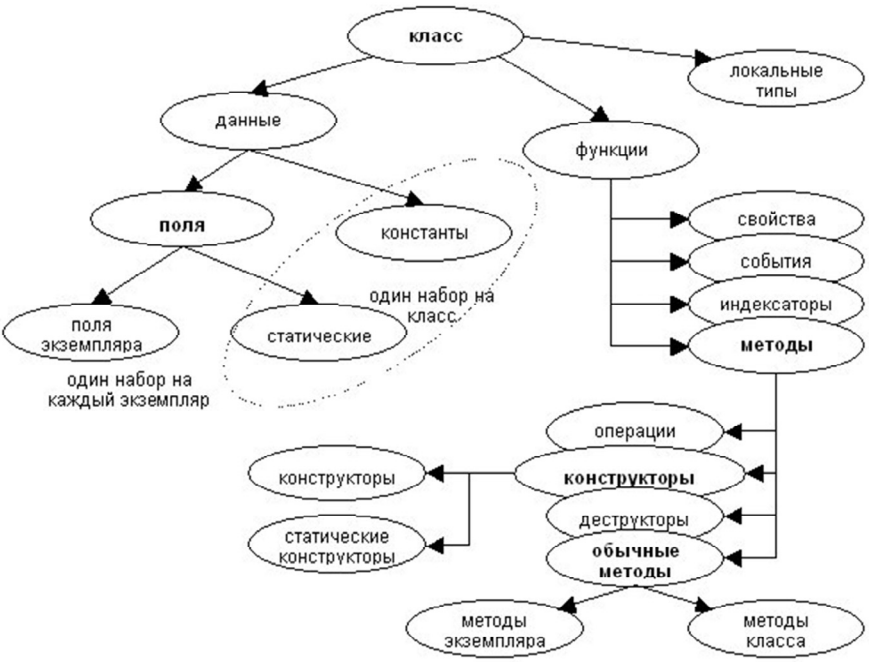


Рис. 13. Состав класса

класса. С помощью свойств определяют характеристики объектов класса, причем в свойстве объединяются методы для чтения и записи. В конструкторе описывают действия по инициализации полей объектов или класса в целом начальными значениями. В деструкторах описывают действия, необходимые при выполнении перед уничтожением объекта. С помощью индексаторов можно обеспечить доступ к элементам объектов класса по их порядковым номерам (индексам). Операции – это специального вида методы, с помощью которых можно переопределить действия, показываемые знаками стандартных операций языка над объектами класса. События определяют уведомления, которые может генерировать класс. Локальные типы – это описания новых типов данных, которые по отношению к классу являются внутренними. Прежде чем начать изучение элементов класса, необходимо пояснить присваивание и сравнение объектов. Механизм выполнения присваивания один и тот же для величин любого типа как ссылочного, так и значимого, однако результаты различаются. При присваивании

значения копируется *значение*, а при присваивании ссылки – *ссылка*, поэтому после присваивания одного объекта другому мы получим две ссылки, указывающие на одну и ту же область памяти (рис. 14).

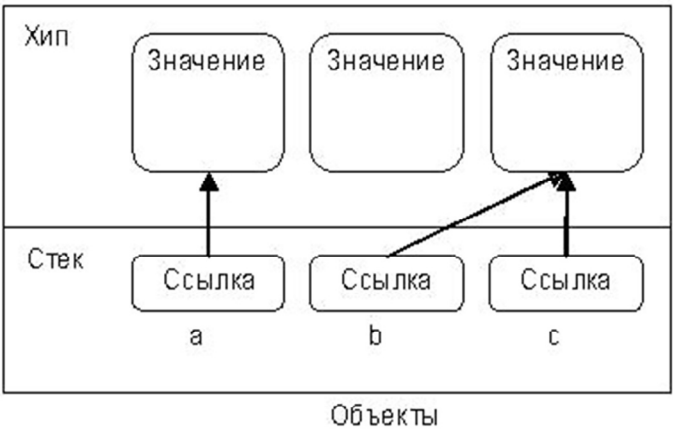


Рис. 14. Присваивание объектов

Аналогичная ситуация с операцией проверки на *равенство*. Величины *значимого типа* равны, если равны их значения. Величины ссылочного типа равны, если они ссылаются на одни и те же данные (на рисунке объекты b и c равны, но a не равно b даже при равенстве их значенийl).

ДАННЫЕ: ПОЛЯ И КОНСТАНТЫ

Для того чтобы включить в состав класса данные, можно воспользоваться *переменными* или *константами* и задать их в соответствии с правилами. Данные, описанные в классе как переменные, называются *полями* класса. При описании элементов данных класса следует непременно указать тип и имя. Ключевое слово const отличает константу от поля. Для задания различных характеристик элементов можно использовать атрибуты и спецификаторы. Переменным, как и константам, можно присваивать начальные значения. *Синтаксическое правило* описания элемента данных приведено ниже:

```
[атрибуты] [спецификаторы] [const] тип имя
[= начальное_значение]
```

Поля и константы могут иметь *спецификаторы* (табл. 2). Для констант можно использовать только спецификаторы 1–4.

Таблица 2

Спецификаторы полей и констант класса

№ п/п	Спецификатор	Описание
1	New	Новое описание поля, скрывающее унаследованный элемент класса
2	Public	Доступ к элементу не ограничен
3	Protected	Доступ только из данного класса и производных классов
4	Private	Доступ только из данного класса
5	Static	Одно поле для всех экземпляров класса
6	Readonly	Поле доступно только для чтения

Если в описании элемента класса не указан спецификатор доступа, элемент считается закрытым (private) по умолчанию. Технология ООП рекомендует описывать поля класса как закрытые. Методы, описанные в классе, имеют доступ ко всем его полям, включая закрытые. Поля, описанные со спецификатором static, и константы существуют в единственном экземпляре для всех объектов класса. Именно поэтому к ним необходимо обращаться через имя класса, а не через имя объекта. Для класса, содержащего только статические элементы, создавать объекты класса не требуется.

Операция доступа (точка) позволяет ссылаться на поле класса. Имя поля задается справа от точки, имя объекта для обычных полей или имя класса для статических полей задается слева. В примере 1 класса Simple показаны оба описанных выше способа обращения к его полям.

```
{
        class Simple
        {
            public int i = 1; // Поле данных.
            public const double d = 2.5; // Константа.
            public static string s = " Simple ";// Статиче-
ское поле класса.
            double z = 3; // Закрытое поле данных.
        }
```

```
    class Program
    {
        static void Main(string[] args)
        {
            // Создание экземпляра класса Simple.
            Simple x = new Simple();
            // x.i - обращение к полю класса.
            Console.WriteLine(x.i);
            // Simple.d - обращение к константе.
            Console.WriteLine(Simple.d);
            // Обращение к статическому полю.
            Console.WriteLine(Simple.s);
        }
    }
}
```

Пример 1. Класс **Simple**, содержащий поля и константу

МЕТОДЫ

Метод представляет собой функциональный элемент класса. С его помощью описывают вычисления или другие действия, которые выполняет объект класса или класс. Именно методы определяют поведение класса. Метод – это законченный фрагмент кода. К нему можно обратиться по имени. Его описывают один раз, а вызывать его можно столько раз, сколько необходимо. Один метод может обрабатывать различные данные. Данные ему передают через параметры в качестве аргументов. Синтаксис метода представлен ниже:

```
[ атрибуты ] [ спецификаторы ] тип имя_метода ( [ параметры ] )
    тело_метода
```

Начинается описание метода с заголовка (сигнатуры). Выполняемые методом действия задаются в теле метода, который представляет собой блок. При описании методов можно использовать спецификаторы 1–5 из табл. 2, имеющие тот же смысл, что и для полей. Методы со спецификатором доступа public составляют интерфейс класса – то, с чем работает пользователь объектов класса. Пример простейшего метода:

```
public double Get_z() { return z; }
// Метод для получения значения z.
```

С помощью «типа» можно задать тип значения, *которое вычисляет метод*. Для методов, которые не возвращают никакого значения, задается тип void, для них оператор return отсутствует. С помощью *параметров* метод обменивается данными с вызывающим его методом. *Параметр* представляет собой локальную переменную, которая при вызове метода принимает *значение* соответствующего фактического параметра (аргумента, переданного при вызове). *Область действия* параметра – весь метод. Например, чтобы выделить целую часть вещественной величины *a*, мы передаем ее в качестве аргумента в метод `Truncate` класса Math, а чтобы вывести *значение* переменной *r* на экран, мы передаем ее в метод WriteLine класса Console:

```
double a = 2.56;
double r = Math.Truncate(a);
Console.WriteLine(r);
```

При этом метод `Truncate` возвращает в точку своего вызова вещественное *значение* – целую часть *a*, которое присваивается переменной r, а метод WriteLine ничего не возвращает. Вызов метода иллюстрирует рис. 15.

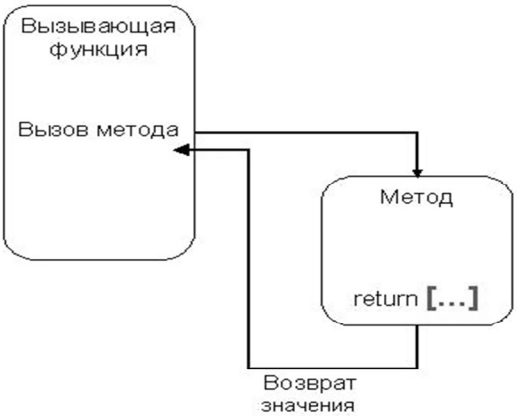


Рис. 16. Вызов метода

Вызов метода, не возвращающего значение, выполняется отдельным оператором. Метод, возвращающий значение, вызывается в правой части оператора присваивания в составе выражения. Описываемые в заголовке метода параметры определяют множество значений аргументов, которые могут быть переданы в метод. Фактические параметры

(аргументы), передаваемые в метод при вызове, как бы накладываются на список параметров. Поэтому они должны попарно соответствовать друг другу. Тип и имя необходимо задавать для каждого параметра. Например, заголовок метода **Truncate** выглядит следующим образом:

```
public static double Truncate(double d)
```

Имя метода вместе с количеством, типами и спецификаторами его параметров представляет собой *сигнатуру метода*. Методов с одинаковыми сигнатурами в классе быть не должно. В класс Simple в примере 2 добавлены методы для записи и чтения значения поля d.

```
namespace ConsoleClasses3
    class Simple
    {
        public static string g = " Simple ";// Статическое
поле класса.
        public const double p = 2.5; // Константа.
        public int j = 1; // Поле данных.
        double d = 3; // Закрытое поле данных.
        // Метод для получения значения d.
        public double Get_d() { return d; }
        // Метод установки поля d.
        public void Set_d(double d_) { d = d_; }
    }
    class Program
    {
        static void Main(string[] args)
        {
            Simple b = new Simple();// Создание экземпляра
класса Simple.
            // Вызов метода установки поля d объекта b значе-
нием 10.
            b.Set_d(10);
            // Вызов метода получения поля d объекта b.
            Console.WriteLine(b.Get_d());
        }
    }
}
```

Пример 2. Методы получения и установки значения.

Параметры методов

При вызове метода выполняются следующие действия: вычисляются выражения, стоящие на месте аргументов; выделяется память в стеке вызова под параметры метода в соответствии с их типом; каждому из параметров ставится в соответствие соответствующий аргумент (аргументы как бы накладываются на параметры и замещают их); выполняется тело метода; если метод возвращает значение, оно передается в точку вызова, если метод имеет тип void, управление передается на оператор, следующий после вызова. При вызове метода проверяется соответствие типов аргументов и параметров и при необходимости выполняется их преобразование. При несоответствии типов выдается *диагностическое сообщение*. Пример 3 иллюстрирует этот процесс.

```
namespace ConsoleClasses4
{
    class Example
    {
        // Метод выбора минимального значения.
        public static int Min(int a, int b)
        {
            if ( a < b ) return a;
            else         return b;
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            int a = 2, b = 4;
            int r = Example.Min( a, b ); // Вызов метода Min.
            Console.WriteLine( r ); // Результат: 2.
            short c = 3, d = 4;
            int w = Example.Min(c, d); // вызов метода Min.
            Console.WriteLine( w ); // Результат: 3.
            // вызов метода Min.
            int v = Example.Min(a + c, d / 2 * b);
            Console.WriteLine( v ); // Результат: 5.
        }
    }
}
```

Пример 3. Передача параметров методу.

Примечание. Главное требование при передаче параметров состоит в том, что аргументы при вызове метода должны записываться в том же порядке, что и в заголовке метода, и должно существовать неявное преобразование типа каждого аргумента к типу соответствующего параметра. Количество аргументов должно соответствовать количеству параметров.

Существует два основных способа передачи параметров: *по значению и по ссылке.* При передаче *по значению* метод получает копии значений аргументов и *операторы* метода работают с этими копиями. Доступа к исходным значениям аргументов у метода нет, а следовательно, нет и возможности их изменить. При передаче *по ссылке* (*по адресу*) метод получает копии *адресов аргументов*, он осуществляет *доступ* к ячейкам памяти *по этим адресам* и может изменять исходные значения аргументов, модифицируя параметры. В C# для обмена данными между вызывающей и вызываемой функцией предусмотрено четыре типа параметров: параметры-значения; параметры-ссылки – описываются с помощью ключевого слова ref; выходные параметры – описываются с помощью ключевого слова out; параметры-массивы – описываются с помощью ключевого слова params. *Ключевое слово* предшествует описанию типа параметра. Если оно опущено, *параметр* считается параметром-значением.

Параметр-массив может быть только один и должен располагаться последним в списке, например:

```
public int Execute( int a, ref int b, out int c,
                    params int[] d ) …
```

Параметры-значения

Этот параметр описывается в заголовке метода следующим образом: **`тип имя`**

Пример заголовка метода, имеющего один параметр-значение целого типа: **`void P( int x )`**

Имя параметра может быть произвольным. Параметр **`x`** представляет собой локальную переменную, которая получает свое значение из вызывающей функции при вызове метода. В метод передается копия значения аргумента. Механизм передачи следующий: из ячейки памяти, в которой хранится переменная, передаваемая в метод, берется копия ее значения и записывается в специальную область памяти – область параметров. Метод работает с этой копией. По завершении работы область параметров освобождается. Этот способ годится только для

передачи в метод исходных данных. При вызове метода на месте параметра, передаваемого по значению, может находиться выражение, для типа которого существует *неявное преобразование типа* выражения к типу параметра. Например, пусть в вызывающей функции описаны переменные и им до вызова метода присвоены значения:

```
int    x = 1; sbyte  c = 1; ushort y = 1;
```

Тогда следующие вызовы метода **P**, заголовок которого был описан ранее, будут синтаксически правильными:

```
P( x );    P( c );    P( y );    P( 10 );    P( x * 5 + 1 );
```

Параметры-ссылки

Признаком параметра-ссылки служит ключевое слово **ref** перед описанием параметра: `ref тип имя`

Пример заголовка метода, имеющего один параметр-ссылку целого типа:

```
void P( ref int x )
```

При вызове метода в область параметров копируется адрес аргумента и метод через него имеет доступ к ячейке, в которой хранится аргумент. Метод работает непосредственно с переменной из вызывающей функции и, следовательно, может ее изменить, поэтому если в методе требуется изменить значения параметров, они должны передаваться только по ссылке. При вызове метода на месте параметра-ссылки может находиться только ссылка на инициализированную переменную точно того же типа. Перед именем параметра указывается ключевое слово ref. Проиллюстрируем передачу параметров-значений и параметров-ссылок на примере (пример 4).

```
namespace ConsoleMethodParams
{
    class Program
    {
        static void M(int c, ref int d)
        {
            c = 11; d = 22;
            Console.WriteLine("В теле метода \tc = {0} \td =
{1}", c, d);
```

```
        }
        static void Main(string[] args)
        {
            int c = 2, d = 4;
            Console.WriteLine("Перед вызовом \tc = {0} \td =
{1}", c, d);
            M(c, ref d);
            Console.WriteLine("После вызова \tc = {0} \th =
{1}", c, d);
        }
    }
}
```

Пример 4. Параметры-значения и параметры-ссылки.
Результаты работы программы:

```
Перед вызовом   c = 2   d = 4
В теле метода   c = 11  d = 22
После вызова    c = 2   h = 22
Для продолжения нажмите любую клавишу . . .
```

Если в метод передаются объекты классов, т. е. величин ссылочных типов, можно считать, что объекты всегда передаются по ссылке.

Выходные параметры

Довольно часто возникает необходимость в методах, которые формируют несколько величин. В этом случае становится неудобным ограничение параметров-ссылок: необходимость присваивания значения аргументу до вызова метода. Это ограничение снимает спецификатор out. Параметру, имеющему этот спецификатор, необходимо присвоить значение внутри метода. Изменим описание второго параметра в примере 4 так, чтобы он стал *выходным* (пример 5).

```
{
    class Program
    {
        static void M(int c, out int d)
        {
            c = 11; d = 22;
            Console.WriteLine("В теле метода \tc = {0} \td =
{1}", c, d);
```

```
        }
        static void Main(string[] args)
        {
            int c = 2, d;
            M(c, out d);
            Console.WriteLine("После вызова \tc = {0} \td =
{1}", c, d);
        }
    }
}
```

Пример 5. Выходные параметры.

При вызове метода перед соответствующим параметром тоже указывается ключевое слово out.

```
В теле метода   c = 11  d = 22
После вызова    c = 2   d = 22
Для продолжения нажмите любую клавишу . . .
```

КЛЮЧЕВОЕ СЛОВО THIS

Каждый *объект* содержит свой набор полей класса. Методы находятся в памяти в единственном экземпляре и используются всеми объектами совместно, поэтому необходимо обеспечить работу методов с полями именно того объекта, для которого они были вызваны. Для этого в любой нестатический метод автоматически передается скрытый *параметр* this, в котором хранится *ссылка* на вызвавший функцию объект. В явном виде *параметр* this применяется для того, чтобы возвратить из метода ссылку на вызвавший *объект*, а также для идентификации поля, если его имя совпадает с именем параметра метода, например:

```
class Simple
{
    double y;
    // Метод возвращает ссылку на экземпляр.
    public Simple T() { return this; }
    public void Sety(double y)
    {
```

```
            // Полю y присваивается значение параметра y.
            this.y = y;
        }
    }
```

КОНСТРУКТОРЫ

Конструктор – это метод, предназначенный для инициализации полей объекта. Его вызывают при создании объекта класса с помощью операции new. Конструктор должен иметь то же имя, что и класс. Конструктор имеет следующие свойства: не возвращает значение, даже типа void; класс может иметь несколько конструкторов с разными параметрами для разных видов инициализации; при отсутствии в классе конструктора или если какие-то поля не были инициализированы, в поля значимых типов заносится нуль, в поля ссылочных типов – значение null; конструктор, вызываемый без параметров, называется конструктором по умолчанию. До сих пор мы задавали начальные значения полей класса при описании класса. Это удобно в том случае, когда для всех объектов класса начальные значения некоторого поля одинаковы. Если же при создании объектов требуется присваивать полю разные значения, это следует делать в конструкторе. В примере 6 в класс Simple добавлен *конструктор*, а поля сделаны закрытыми.

```
namespace ConsoleConstructor
{
    class Simple
    {
        // Конструктор с параметрами.
        public Simple ( int a, double z )
        {
            this.a = a;
            this.z = z;
        }
        // Метод получения поля y
        public double Get_z() { return z; }
        int a;
        double z;
    }
    class Program
```

```
    {
        static void Main(string[] args)
        {
            // Вызов конструктора.
            Simple a = new Simple ( 10, 2.2 );
            // Результат: 2,2.
            Console.WriteLine( a.Get_z() );
            // Вызов конструктора.
            Simple b = new Simple ( 1, 6.54 );
            // Результат: 6,54.
            Console.WriteLine( b.Get_z() );
        }
    }
}
```

Пример 6. Конструктор.

Для инициализации объектов разными способами необходимо задать в классе несколько конструкторов. Все конструкторы должны иметь разные сигнатуры. Из одного конструктора можно вызвать другой. Это делается с помощью ключевого слова this, например:

```
class Simple
{
    // Конструктор 1.
    public Simple ( int a )
    { this.a = a; }
    // Вызов конструктора 1.
    public Simple ( int a, double y ) : this( a )
    { this.y = y; }
}
```

Конструкция, находящаяся после двоеточия, называется инициализатором. Все классы в C# имеют общего предка – класс object. Конструктор любого класса, если не указан инициализатор, автоматически вызывает конструктор своего предка. Для класса, содержащего только статические данные, создавать экземпляры такого класса не имеет смысла. Вы можете в этом случае описать статический класс, т. е. класс с модификатором static. Для такого класса объекты создавать нельзя и, кроме того, от него нельзя наследовать. Все элементы такого

класса должны объявляться с модификатором static (константы и вложенные типы классифицируются как статические элементы автоматически). Статический класс приведен в примере 7.

```
namespace ConsoleStatClass
{
    static class Simple
    {
        static int a = 10;
        static double b = 2.5;
        const double pi = 3.1415;
        public static void Print ()
        {
            Console.WriteLine("a = " + a );
            Console.WriteLine("b = " + b);
            Console.WriteLine("pi = " + pi);
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            Simple.Print();
        }
    }
}
```

Пример 7. Статический класс (начиная с версии 2.0).

В качестве примера, на котором будет демонстрироваться работа с различными элементами класса, создадим *класс*, моделирующий простую дробь. Для этого требуется задать значения числителя и знаменателя, операцию сложения, сокращения дроби и преобразования ее в формат строки.

```
namespace ConsoleFrac1
{
    class TFrac
    {
        int num;
```

```
int denom;
public override string ToString()
{
    string s = string.Format("{0}/{1}", num, denom);
    return s;
}
public TFrac(int n, int d = 1)
{
    this.num = n;
    this.denom = d;
    Reduce();
}
public TFrac(string s = "0/1")
{
    int n = s.IndexOf('/');
    if (n == -1)
    {
        this.num = int.Parse(s);
        this.denom = 1;
    }
    else
    {
        this.num = int.Parse(s.Substring(0, n));
        this.denom = int.Parse(s.Substring(n + 1));
    }
    Reduce();
}
public TFrac Add(TFrac b)
{
    int n = this.num * b.denom + b.num * this.denom;
    int d = this.denom * b.denom;
    return new TFrac(n, d);
}
static int nod(int a, int b)
{
    int t;
    a = Math.Abs(a);
    b = Math.Abs(b);
    if (a > b) { t = a; a = b; b = t; }
```

```
            while (a != 0)
            {
                if (a <= b) { b = b % a; }
                else { t = a; a = b; b = t; }
            }
            return b;
        }
        void Reduce()
        {
            int t = nod(this.num, this.denom);
            this.num /= t;
            this.denom /= t;
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            TFrac a = new TFrac(1, 5);
            Console.WriteLine(a.ToString());
            TFrac b = new TFrac("3/5");
            Console.WriteLine(b.ToString());
            TFrac c = a.Add(b);
            Console.WriteLine(c.ToString());
        }
    }
}
```

Пример 8. Класс TFrac.
Результат работы программы:

```
1/5
3/5
4/5
Для продолжения нажмите любую клавишу . . .
```

СВОЙСТВА

Свойства служат для организации доступа к полям класса. Как правило, свойство связано с закрытым полем класса и определяет методы его получения и установки. *Синтаксис* свойства:

```
[ атрибуты ] [ спецификаторы ] тип имя_свойства
{
     [ get код_доступа ]
     [ set код_доступа ]
}
```

Значения спецификаторов для свойств и методов аналогичны. Чаще всего свойства объявляются со спецификатором public. *Код доступа* представляет собой блоки операторов, которые выполняются при получении (get) или установке (set) свойства. Может отсутствовать либо часть get, либо set, но не обе одновременно. Если отсутствует часть set, свойство доступно только для чтения (*read-only*), если отсутствует часть get, свойство доступно только для записи (write-only). В версии C# 2.0 введена возможность задавать разный уровень доступа для частей get и set.

Пример описания свойств:

```
public class Button: Control
{
    // Закрытое поле, с которым связано свойство
    private string caption;
    // свойство
    public string Caption
    {
       // способ получения свойства
       get { return caption;       }
       // способ установки свойства
       set { if (caption != value) { caption = value; } }
    }
}
```

Метод для записи может содержать действия по проверке допустимости устанавливаемого значения, метод для чтения может содержать действия по преобразованию формата хранящихся в объекте данных. В программе свойство выглядит как *поле* класса, например:

```
Button ok = new Button();
// вызывается метод установки свойства
ok.Caption = "OK";
// вызывается метод получения свойства
string s = ok.Caption;
```

При обращении к свойству автоматически вызываются указанные в нем методы чтения и установки. Синтаксически чтение и *запись* свойства выглядят почти как методы. Метод get должен содержать оператор return. В методе set используется *параметр* со стандартным именем value, который содержит устанавливаемое *значение*.

В *класс* TFrac, описанный в примере 8, добавим свойства, позволяющие работать с закрытыми полями этого класса. Код класса станет больше, но использование класса станет проще.

```
namespace ConsoleFrac1
{
    class TFrac
    {
        int num;
        int denom;
        // Свойство для работы с полем num.
        public int Num
        {
            get { return num; }
            set { if (num != value) num = value; }
        }
        // Свойство для работы с полем denom.
        public int Denom
        {
            get { return denom; }
            set { if (denom != value) denom = value; }
        }
        // Свойство для работы с полями num, denom.
        public string Frac
        {
            get
            {
             string s = string.Format("{0}/{1}", num, denom);
```

```
        return s;
    }
}
public TFrac(int n, int d = 1)
{
    this.num = n;
    this.denom = d;
    Reduce();
}
public TFrac(string s = "0/1")
{
    int n = s.IndexOf('/');
    if (n == -1)
    {
        this.num = int.Parse(s);
        this.denom = 1;
    }
    else
    {
        this.num = int.Parse(s.Substring(0, n));
        this.denom = int.Parse(s.Substring(n + 1));
    }
    Reduce();
}
public TFrac Add(TFrac b)
{
    int n = this.num * b.denom + b.num * this.denom;
    int d = this.denom * b.denom;
    return new TFrac(n, d);
}
static int nod(int a, int b)
{
    int t;
    a = Math.Abs(a);
    b = Math.Abs(b);
    if (a > b) { t = a; a = b; b = t; }
    while (a != 0)
    {
        if (a <= b) { b = b % a; }
```

```
                else { t = a; a = b; b = t; }
            }
            return b;
        }
        void Reduce()
        {
            int t = nod(this.num, this.denom);
            this.num /= t;
            this.denom /= t;
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            TFrac a = new TFrac(1, 5);
            // Используем свойство.
            Console.WriteLine(a.Frac);
            // Используем свойство.
            a.Num = 3;
            // Используем свойство.
            a.Denom = 7;
            // Используем свойство.
            Console.WriteLine(a.Frac);
        }
    }
}
```

Пример 9. Класс TFrac со свойствами.
Результат работы программы:

```
1/5
3/7
Для продолжения нажмите любую клавишу . . .
```

КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Перечислите и опишите элементы класса в C#.
2. Опишите способы передачи параметров в методы.
3. Для чего в классе может потребоваться несколько конструкторов?
4. Как можно вызвать один конструктор из другого? Зачем это нужно?
5. Что такое this? Что в нем хранится, как он используется?
6. Какие действия обычно выполняются в части set свойства?
7. Может ли свойство класса быть не связанным с его полями?
8. Можно ли описать разные спецификаторы доступа к частям get и set свойства?

ПРАКТИЧЕСКИЕ ЗАДАНИЯ ДЛЯ ЗАКРЕПЛЕНИЯ

Практическая работа

Проектирование конвертера p1_p2

Цель. Объектно-ориентированный анализ, проектирование и реализация приложения Конвертер p1_p2 под Windows для преобразования действительных чисел, представленных в системе счисления с основанием p1, в действительные числа, представленные в системе счисления с основанием p2. В процессе выполнения работы студенты изучают: отношения между классами (ассоциация, агрегация, зависимость), их реализацию средствами языка программирования высокого уровня; этапы разработки приложений в технологии ООП; элементы технологии визуального программирования; диаграммы языка UML для документирования разработки.

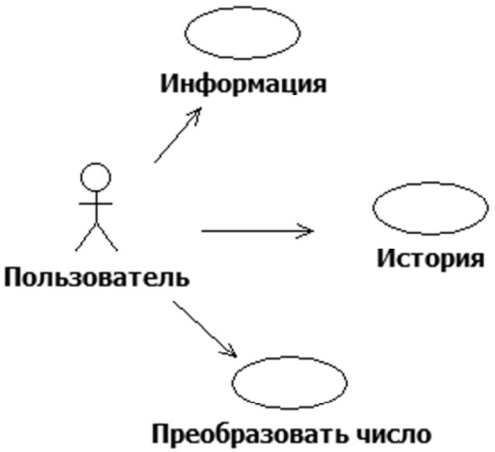
Функциональные требования к приложению

Интерфейс приложения может выглядеть так (рис. 17).

Рис. 17. Главная форма приложения

Приложение должно обеспечивать пользователю: преобразование действительного числа, представленного в системе счисления с основанием p1, в число, представленное в системе счисления с основанием p2; основания систем счисления p1, p2 для исходного числа и результата преобразования выбираются пользователем из диапазона от 2…16; возможность ввода и редактирования действительного числа, представленного в системе счисления с основанием p2 с помощью командных кнопок и мыши, а также с помощью клавиатуры; контекстную помощь по элементам интерфейса и справку о назначении приложения; просмотр истории сеанса (журнала) работы пользователя с приложением – исходные данные, результат преобразования и основания систем счисления, в которых они представлены; дополнительные повышенные требования: автоматический расчет необходимой точности представления результата.

Функциональные требования представлены диаграммой прецедентов (use-case диаграммой), расположенной ниже.



Сценарий для прецедента «Преобразовать число»

Основной поток событий

1. Пользователь выбирает основание системы счисления p1 исходного числа.

2. Пользователь выбирает основание системы счисления p2 результата.

3. Пользователь вводит действительное число, представленное в системе счисления с основанием p1.

4. Пользователь вводит команду «Преобразовать».

5. Система выводит введенное пользователем число, представленное в системе счисления с основанием p2.

6. Система сохраняет исходные данные и результат преобразования в Историю.

Альтернативный поток событий 1. Введенное пользователем число выходит за границы допустимого диапазона (*к пункту 3 основного потока событий*)

3.1. Пользователь получает окно с сообщением.

3.2. Приложение переходит в режим Ввод и редактирование.

Альтернативный поток событий 2. Количество разрядов в результате превышает размер поля вывода визуального компонента (*к пункту 4 основного потока событий*)

4.1. Пользователь получает окно с сообщением.

4.2. Приложение переходит в режим Ввод и редактирование.

Можно осуществить декомпозицию прецедента Преобразовать, в результате мы получим для него следующую диаграмму:

Сценарий для прецедента «Преобразовать»

Предусловие

Завершен ввод и редактирование исходного числа.

Основной поток событий

1. Пользователь вводит команду «Преобразовать».
2. Система выводит введенное пользователем число, представленное в системе счисления с основанием p2.
3. Система сохраняет исходные данные и результат преобразования в Историю.

Альтернативный поток событий 1. Количество разрядов в результате превышает размер поля вывода визуального компонента (*к пункту 1 основного потока событий*)

1.1. Пользователь получает окно с сообщением.
1.2. Приложение переходит в режим Ввод и редактирование.

Сценарий для прецедента «Выбрать основание с. сч. p2»

Предусловие

Прецедент «Преобразовать» завершен.

Основной поток событий

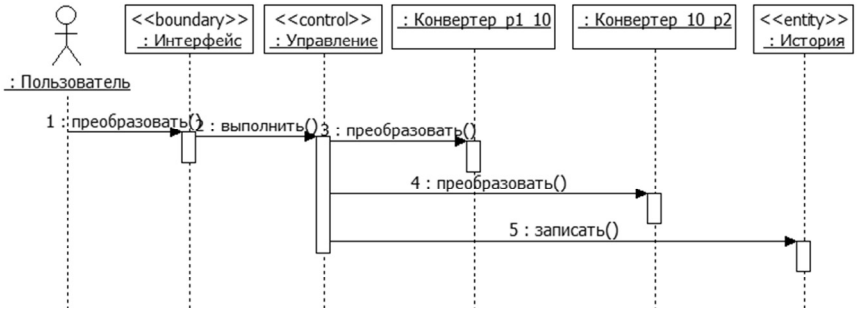
1. Пользователь изменяет основания систем счисления p2.
2. Введенное пользователем число отображается в системе счисления с выбранным основанием.

Диаграмма классов модели объектно-ориентированного анализа

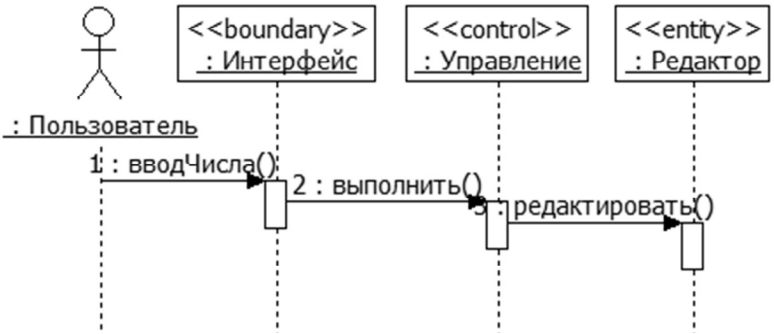
Проанализировав прецеденты, можно выделить следующие классы анализа приложения. Они представлены на диаграмме классов анализа ниже.

Обмен сообщениями между объектами
Диаграмма последовательностей

Спроектируем обмен сообщениями между объектами в процессе выполнения прецедента «Преобразовать». Добавим объект класса «Управление» для организации обмена сообщениями между объектами в ходе выполнения прецедента. На диаграмме, приведенной ниже, показана последовательность сообщений между объектами в основном потоке событий прецедента «Преобразовать».



На диаграмме, приведенной ниже, показана последовательность сообщений между объектами в процессе реализации прецедента «Ввести число».



На диаграмме, приведенной ниже, показана последовательность сообщений между объектами в процессе реализации прецедента «История».

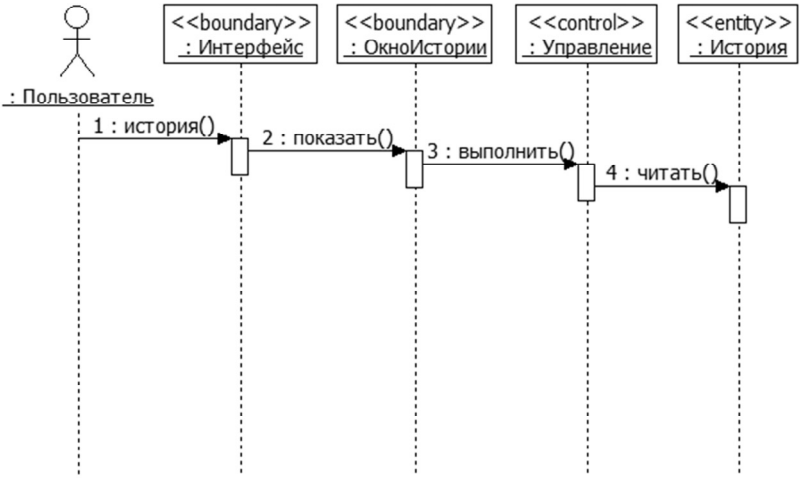
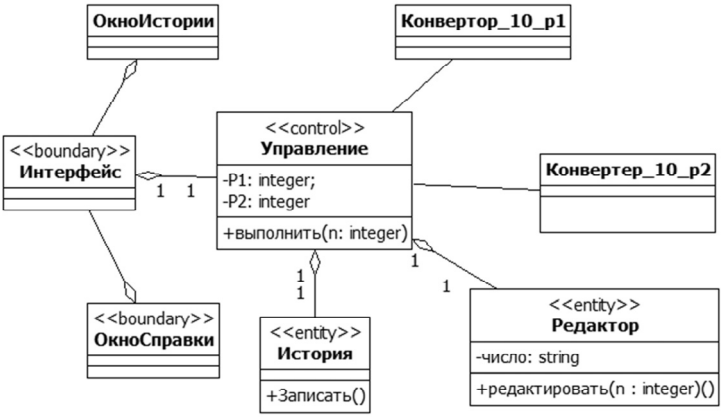
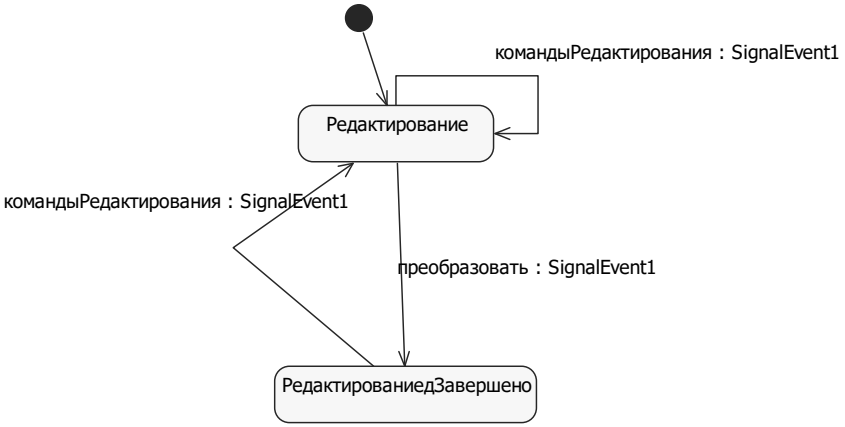


Диаграмма классов проекта

Проанализировав сообщения, которыми обмениваются классы в процессе выполнения прецедентов, можно построить следующую диаграмму классов проекта. Для упрощения взаимодействия между классами в процессе работы приложения добавим класс «Управление». Тогда наша диаграмма примет следующий вид:



Из диаграммы классов видно, что объект класса «Интерфейс» вызывает методы класса «Управление» и «Справка». Объект же класса «Управление» в свою очередь вызывает методы объектов классов «Редактор», «История» и «Конвертер_p1_10», «Конвертер_10_p2». Диаграмма состояний для объекта класса «Управление» представлена ниже:



Объект класса «Управление» может находиться в двух состояниях: «Редактирование» и «Редактирование завершено».

Разработка приложения разбита на практические работы, в каждой из которых вы реализуете один или несколько логически связанных классов приложения.

Практическая работа

Конвертер чисел из десятичной системы счисления в систему счисления с заданным основанием

Цель. Сформировать практические навыки реализации классов на языке C#.

Задание

1. Реализовать преобразователь действительных чисел со знаком из десятичной системы счисления в систему счисления с заданным основанием p в соответствии с приведенной ниже спецификацией, используя класс C#. Основание системы счисления p принадлежит диапазону значений от 2 до 16.
2. Протестировать каждый метод класса.

Спецификация класса «Преобразователь чисел из десятичной системы счисления в систему счисления с заданным основанием p»

Данные

Преобразователь действительных чисел из десятичной системы счисления в систему счисления с заданным основанием (тип Conver_10_p). Основание системы счисления p – это целое число со значением, принадлежащим диапазону от 2 до 16, и целое число c, определяющее точность представления результата, выраженную в количестве разрядов.

Операции

Do(double n, int p, int c)	*Выполнение преобразования*
Вход	Десятичное действительное число n. Основание системы счисления p. Точность преобразования дроби, заданная числом разрядов дробной части результата c. Например: Do(-17.875,16,3) = “-A1.E”
Процесс	Выполняет преобразование десятичного действительного числа *n* в систему счисления с основанием p и точностью c. Например: Do (”-17.875”,16,3) = “-A1.E”
Выход	Строка результата. Например: Do(”-17.875”) = “-A1.E”

Окончание таблицы

Do(double n, int p, int c)	*Выполнение преобразования*
int_to_Char(int d)	Преобразование целого значения в цифру системы счисления с основанием p
Вход	d – значение типа int – целое, соответствующее цифре в системе счисления с основанием p
Предусловия	Нет
Процесс	Преобразует целое d в соответствующую ему цифру в системе счисления с основанием p, значение типа Char Например: int_to_Char (14) = “E”
Выход	Значение типа char
Постусловия	Нет
int_to_P(int n, int p)	Преобразование целого в строку
Вход	n – целое число в системе счисления с основанием 10. p – основание системы счисления результата
Предусловия:	Нет
Процесс	Преобразует целое n в строку, содержащую целое число в системе счисления с основанием p Например: int_to_P(161, 16) = “A1”
Выход	Строка
Постусловия	Нет
flt_to_P(double n, int p, int c)	Преобразование дроби в строку
Вход	*n* – дробь в системе счисления с основанием 10, p – основание системы счисления, c – точность представления дроби
Предусловия	Нет
Процесс	Преобразует дробь n в строку, содержащую дробь в системе счисления с основанием p с точностью c. Например: flt_to_P(0.9375, 2, 4) «1111»
Выход	Строка
Постусловия	Нет

Рекомендации к выполнению

1. Тип данных реализовать, используя статический класс.
2. Тип данных сохранить в отдельном файле «Conver_10_p».

Ниже представлена заготовка описания класса Conver_10_p. Вам необходимо написать код методов и протестировать методы.

```
namespace Конвертор
{
    class Conver_10_P
    {
     //Преобразовать целое в символ.
     public static char int_to_Char(int n) {        }
     //Преобразовать десятичное целое в с.сч. с основанием p.
     public static string int_to_P(int n, int p) {        }
     //Преобразовать десятичную дробь в с.сч. с основанием p.
        public static string flt_to_P(double n, int p, int c)
{        }
       //Преобразовать десятичное
      //действительное число в с.сч. с осн. p.
      public static string Do(double n, int p, int c) {
}
    }
}
```

Содержание отчета

1. Задание.
2. Текст программы.
3. Тестовые наборы данных для тестирования класса.

Контрольные вопросы

1. Что такое инкапсуляция?
2. Как синтаксически представлено поле в описании класса?
3. Как синтаксически представлен метод в описании класса?
4. Как синтаксически представлено простое свойство в описании класса?
5. Особенности описания методов класса.
6. Видимость идентификаторов в описании класса.
7. В чем особенности статических методов?
8. В чем особенности статических классов?

Практическая работа

Класс «Конвертер p_10» – преобразователь чисел из системы счисления с основанием p в десятичную систему счисления

Цель. Сформировать практические навыки реализации классов на языке C#.

Задание

1. Реализовать преобразователь действительных (Конвертер p_10) чисел из системы счисления с основанием p в десятичную систему счисления в соответствии с приведенной ниже спецификацией, используя класс. Основание системы счисления p принадлежит диапазону значений от 2 до 16.
2. Протестировать каждый метод класса.

Спецификация класса «Конвертер p_10» – преобразователь действительных чисел со знаком из системы счисления с основанием p в десятичную систему счисления.

Данные

Преобразователь действительных чисел из заданной системы счисления с основанием p в десятичную систему счисления (тип Conver_p_10). Основание системы счисления со значением, принадлежащим диапазону от 2 до 16.

Операции

dval(string P_num, int P)	*Выполнение преобразования*
Вход	P_num – строковое представление действительного числа в системе счисления с основанием p. Например: dval(“A5.E”, 16)
Процесс	Выполняет преобразование действительного числа, представленного строкой, в числовое представление. Например: dval(“A5.E”, 16) = -165.875
Выход	Вещественное число
Постусловия	Нет
char_To_num(char ch)	*Преобразование символа в целое*
Вход	ch – значение типа char – символ, изображающий цифру системы счисления с основанием p
Предусловия	Нет
Процесс	Преобразует символ ch в значение целого типа. Например: PCharToInt(‘A’) = 10
Выход	Вещественное число
Постусловия	Нет
convert(string P_num, int P, double weight)	*Преобразование строки в вещественное число*
Вход	P_num – строка, изображающая цифры целой и дробной частей вещественного числа в системе счисления с основанием p без разделителя. weight – вес единицы старшего разряда целой части числа
Предусловия	Нет
Процесс	Преобразует строку P_num, содержащую цифры целой и дробной частей вещественного числа в системе счисления с основанием p без разделителя, в вещественное число. Например: convert (“A5E1”, 16, 16)
Выход	Вещественное число
Постусловия	Нет

Рекомендации к выполнению

1. Описание класса может выглядеть следующим образом:

```
namespace Конвертер
{
    public class Conver_P_10
    {
        //Преобразовать цифру в число.
        static double char_To_num(char ch) {        }
        //Преобразовать строку в число
        private static double convert(string P_num, int P,
        double weight) { }
        //Преобразовать из с.сч. с основанием p
        //в с.сч. с основанием 10.
        public static double dval(string P_num, int P) {
}
    }
}
```

2. Тип данных реализовать, используя статический класс.
3. Сохранить класс в отдельном файле «Conver_p_10».

Содержание отчета

1. Задание.
2. Текст программы.
3. Тестовые наборы данных для тестирования класса.

Контрольные вопросы

1. Что такое инкапсуляция?
2. Как синтаксически представлено поле в описании класса?
3. Как синтаксически представлен метод в описании класса?
4. Как синтаксически представлено простое свойство в описании класса?

5. Особенности описания методов класса.
6. Видимость идентификаторов в описании класса.
7. В чем особенности статических методов?
8. В чем особенности статических классов?
9. Как вызываются статические методы?

Практическая работа

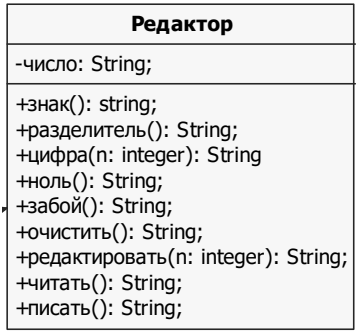
Редактор чисел в системе счисления с основанием p

Цель. Формировать практические навыки реализации классов средствами объектно-ориентированного языка программирования C#.

Задание

1. Разработать и реализовать класс «Editor» (Редактор действительных чисел, представленных в системе счисления с основанием p), используя класс языка высокого уровня. Основание системы счисления p принимает значение из диапазона 2…16. Все команды редактора удобно пронумеровать, начиная с команды «Добавить 0» целыми числами от 0. При реализации интерфейса номера команд удобно хранить в свойстве Tag, которое имеется у визуальных компонентов.

Атрибуты и операции класса представлены ниже.



Ответственность класса Editor (редактор) – хранение, ввод и редактирование строкового представления числа, в системе счисления с основанием p. Класс должен обеспечивать: добавление символов (AddDigit), соответствующих p-ичным цифрам (от 2 до 16); добавление нуля (AddZero()); добавление разделителя целой и дробной частей

(AddDelim()); забой символа – удаление символа, стоящего справа (BS); очистку – установку нулевого значения числа (Clear); чтение строкового представления р-ичного числа (Number).

2. Протестировать каждый метод класса.

Рекомендации к выполнению

Описание класса может выглядеть следующим образом:

```
namespace Конвертер
{
    class Editor
    {
        //Поле для хранения редактируемого числа.
        string number = "";
        //Разделитель целой и дробной частей.
        const string delim = ".";
        //Ноль.
        const string zero = "0";
         //Свойствое для чтения редактируемого числа.
        public string Number
        {  get {   }        }
        //Добавить цифру.
        public string AddDigit(int n) {        }
        //Точность представления результата.
        public int Acc(){        }
        //Добавить ноль.
        public string AddZero(){        }
        //Добавить разделитель.
        public string AddDelim(){        }
        //Удалить символ справа.
        public string Bs() {        }
        //Очистить редактируемое число.
        public string Clear() {        }
        //Выполнить команду редактирования.
        public string DoEdit(int j) {        }
    }
}
```

Класс сохраните в файле Editor. В разделе описания констант опишите следующие константы: «разделитель целой и дробной частей» строкового типа; «строковое представление нуля» строкового типа.

Содержание отчета

1. Задание.
2. Текст программы.
3. Тестовые наборы данных для тестирования класса.

Контрольные вопросы

1. В чем состоит особенность разделов описания класса с уровнем доступа protected, с уровнем доступа private, с уровнем доступа public?
2. В чем состоит особенность инициализации полей ссылочного типа и констант в конструкторе?
3. Что такое this?
4. Как описываются поля в классах?
5. Какой операцией создаются объекты классов?
6. Как вызвать нестатический метод класса?

Практическая работа

Класс История

Цель. Сформировать практические навыки реализации классов средствами объектно-ориентированного программирования языка C#; использовать библиотечный класс обобщенной коллекции List<> для обработки данных.

Задание

1. Разработать и реализовать класс History «История», используя класс языка C#. Класс отвечает за документирование выполнения пользователем переводов чисел. Объекты класса хранят исходные числа, результаты преобразования, основания систем счисления исходного числа и результата.

Атрибуты и операции класса представлены ниже.

История
Запись(i: integer): String; ДобавитьЗапись(a: String); Записей(): integer ОчиститьИсторию();
Обязанность: ввод, вывод, хранение данных, введенных пользователем, и полученных результатов.

Класс должен отвечать за ввод, вывод, хранение данных, введенных пользователем, и полученных результатов и обеспечивать:

- добавление записи (ДобавитьЗапись) – строки, содержащей введенное пользователем число, результат его преобразования и основания исходной системы счисления и той, в которую число преобразовано;
- извлечение записи по ее номеру в списке (Запись);
- очистку списка (ОчиститьИсторию);
- конструктор (Запись);
- текущий размер списка в числе записей (Записей);

2. Протестировать каждый метод класса.

Рекомендации к выполнению

1. Создайте консольное приложение, в которое добавьте класс History, и сохраните его в файле History.

2. В этот же файл добавьте структуру (struct) Record с четырьмя полями для хранения исходного числа, результата и оснований их систем счисления. В структуру добавьте конструктор и метод ToString() для преобразования значения в формат строки.

3. Класс History постройте на основе библиотечного класса коллекции List<Record> из пространства имен System.Collections.Generic.

4. В классе History опишите поле List<Record> L – список значений типа Record. Для поля опишите уровень доступа private.

Описания структуры Record и класса History могут иметь следующий вид:

```
    public struct Record
    {
        int p1;
        int p2;
        string number1;
        string number2;
        public Record(int p1, int p2, string n1, string n2) {
}
        public override string ToString() {       }
    }
    public class History
    {
        List<Record> L;
```

```
    public Record this[int i] {        }
    public void AddRecord(int p1, int p2, string n1,
    string n2) {    }
    public void Clear() {       }
    public int Count() {       }
    public History() {       }
}
```

Содержание отчета

1. Задание.
2. Текст программы.
3. Тестовые наборы данных для тестирования класса.

Контрольные вопросы

1. В чем состоит особенность обобщенной коллекции List< >?
2. В чем состоит отличие типа struct от типа class?
3. Как создаются объекты типа struct?
4. В чем состоит особенность разделов описания класса с уровнем доступа private, с уровнем доступа public?
5. В чем состоит особенность инициализации полей ссылочного типа и констант в конструкторе?
6. Что такое this?

Практическая работа

Класс Управление для «Конвертера p1_p2»

Цель. Сформировать практические навыки реализации классов на языке C#.

Задание

1. Реализовать «Управление» для «Конвертера p1_p2».
2. Протестировать каждый метод класса.

Спецификация класса «Управление» для «Конвертера p1_p2».

Данные

Объект класса Control_ (Управление) отвечает за координацию действий между классом «Интерфейс» и классами «Редактор», «Конвертер p1_10», «Конвертер 10_p2», «История». Объект класса Control_

содержат поля: ed типа Editor, his типа «История» и свойства: Pin типа int (основание системы счисления исходного числа), Pout типа int (основание системы счисления результата), St типа State (состояние конвертера). Он может находиться в одном из двух состояний: «Редактирование», «Преобразовано». Объекты этого типа изменяемы.

Операции

Control_	*Конструктор*
Вход	Нет
Процесс	Создает объект Управление типа (тип **Control_**) и инициирует поля объекта начальными значениями
DoCommand	Выполнение команды
Вход	n – целое значение, номер выполняемой команды
Предусловия	Нет
Процесс	В зависимости от значения n и состояния (St) передает сообщение объекту Редактор или Преобразователь и изменяет состояние. Возвращает строку результата: либо отредактированное число, либо результат преобразования
Выход	Строка
Постусловия	Нет

Рекомендации к выполнению

1. Тип данных реализовать, используя класс C#.
2. Для записи и считывания полей «преобразователя» использовать свойства.
3. Тип данных реализовать в отдельном файле Control_.

Пример описания класса «Управление» приведен ниже.

```
namespace Конвертер
{
    class Control_
    {
        //Основание системы сч. исходного числа.
        const int pin = 10;
        //Основание системы сч. результата.
        const int pout = 16;
        //Число разрядов в дробной части результата.
        const int accuracy = 10;
```

```
 public История his = new История();
 public enum State {Редактирование, Преобразовано}
 //Свойство для чтения и записи состояние Конвертера.
 public State St { get; set; }
 //Конструктор.
 public Control_()
 {
     St = State.Редактирование;
     Pin = pin;
     Pout = pout;
 }
 //объект редактор
 public Editor ed = new Editor();
//Свойство для чтения и записи основание системы сч. р1.
 public int Pin { get; set; }
//Свойство для чтения и записи основание системы сч. р2.
 public int Pout { get; set; }
 //Выполнить команду конвертера.
 public string DoCmnd(int j)
 {
     if (j == 19)
     {
         double r = Conver_P_10.dval(ed.Number,
         (Int16)Pin);
         string res = Conver_10_P.Do
         (r, (Int32)Pout,acc());
         St = State.Преобразовано;
        his.ДобавитьЗапись(Pin, Pout, ed.Number, res);
         return res;
     }
     else
     {
         St = State.Редактирование;
         return ed.DoEdit(j);
     }

 }
 //Точность представления результата.
 private int acc()
```

```
        {
            return (int)Math.Round(ed.Acc() * Math.Log(Pin) /
            Math.Log(Pout) + 0.5);
        }
    }
}
```

Содержание отчета

1. Задание.
2. Текст программы.
3. Тестовые наборы данных для тестирования класса.

Контрольные вопросы

1. Что такое инкапсуляция?
2. Как синтаксически представлено поле в описании класса?
3. Как синтаксически представлен метод в описании класса?
4. Как синтаксически представлено простое свойство в описании класса?
5. Особенности описания методов класса.
6. Особенности описания и назначение конструктора класса.
7. Видимость идентификаторов в описании класса.
8. Особенности вызова методов применительно к объектам класса.

Практическая работа

Интерфейс приложения «Конвертер p1_p2»

Цель. Сформировать практические навыки реализации графических интерфейсов пользователя (GUI) на основе библиотеки визуальных компонентов.

Задание

1. Реализовать «Интерфейс» приложения «Конвертер p1_p2», используя библиотечный класс формы и визуальные компоненты.
2. Протестировать методы класса.

Спецификация класса «Интерфейс».

Интерфейс приложения представлен на рис. 17.

Данные

«Интерфейс» приложения p2 предназначен: для выбора оснований систем счисления p1, p2 из диапазона от 2 до 16; ввода и редактирования действительного числа со знаком в системе счисления с выбранным основанием p1; отображения результата – представления введенного числа в системе счисления с основанием p2; отображения справки о приложении; отображения истории текущего сеанса работы пользователя с приложением.

«Интерфейс» несет на себе визуальные компоненты, реализующие выполнение команд преобразователя, и объект «Управление» класса Control_.

Операции

Наименование	Пояснение
trackBar1_Scroll	Обработчик события Scroll для компонента trackBar1
Вход	object sender, EventArgs e. sender – указатель на объект, который стал инициатором события Scroll e. – это объект базового класса для классов, содержащих данные о событии
Предусловия	Пользователь перетаскивает бегунок компонента trackBar1
Процесс	Обновляет свойства визуальных компонентов формы, связанных с изменением основания системы счисления p1 исходного числа. Устанавливает новое значение основания системы счисления. Обновляет состояние командных кнопок
Постусловия	Обновления выполнены
Выход	Нет
numericUpDown1_ValueChanged	Обработчик события ValueChanged для компонента numericUpDown1
Вход	object sender, EventArgs e. sender – указатель на объект, который стал инициатором события ValueChanged
Предусловия	Пользователь изменяет p1 с помощью компонента numericUpDown1

Продолжение таблицы

Наименование	Пояснение
Процесс	Обновляет свойства визуальных компонентов формы, связанных с изменением основания системы счисления p1 исходного числа. Устанавливает новое значение основания системы счисления. Обновляет состояние командных кнопок
Постусловия	Обновления выполнены
Выход	Нет
trackBar2_Scroll	Обработчик события Scroll для компонента trackBar2
Вход	object sender, EventArgs e. sender – указатель на объект, который стал инициатором события Scroll
Предусловия	Пользователь перетаскивает бегунок компонента trackBar2
Процесс	Обновляет свойства визуальных компонентов формы, связанных с изменением основания системы счисления p2 исходного числа. Устанавливает новое значение основания системы счисления
Постусловия	Обновления выполнены
Выход	Нет
numericUpDown2_ValueChanged	Обработчик события ValueChanged для компонента numericUpDown2
Вход	object sender, EventArgs e. sender – указатель на объект, который явился инициатором события ValueChanged
Предусловия	Пользователь изменяет p2 с помощью компонента numericUpDown2
Процесс	Обновляет свойства визуальных компонентов формы, связанных с изменением основания системы счисления p2 результата. Устанавливает новое значение основания системы счисления. Обновляет состояние командных кнопок
Постусловия	Обновления выполнены
Выход	Нет
numericUpDown1_ValueChanged	Обработчик события ValueChanged для компонента numericUpDown1

Продолжение таблицы

Наименование	Пояснение
Вход	object sender, EventArgs e. sender – указатель на объект, который стал инициатором события ValueChanged
Предусловия	Пользователь изменяет p1 с помощью компонента numericUpDown1
Процесс	Обновляет свойства визуальных компонентов формы, связанных с изменением основания системы счисления p1 результата. Устанавливает новое значение основания системы счисления. Обновляет состояние командных кнопок
Постусловия	Обновления выполнены
Выход	Нет
TPanelp_p_Load	Обработчик события Load для компонента TPanelp_p
Вход	(object sender, EventArgs e.) sender – указатель на объект, который стал инициатором события Load
Предусловия	Форма загружается в память
Процесс	Устанавливает начальные значения свойств визуальных компонентов формы после загрузки формы
Выход	Нет
Постусловия	Установка свойств выполнена
DoCmnd(int j)	Выполнить команду
Вход	j – значение целого типа – номер команды преобразователя
Предусловия	Пользователь нажал командную кнопку команды с номером j
Процесс	Передает сообщение объекту «Управление» и отображает возвращаемый им результат. Вызывает метод объекта «Управление» и передает номер набранной пользователем команды «Конвертера»
Выход	Нет
Постусловия	Обновляется состояние «Интерфейса»
button_Click	Обработчик события Click для командных кнопок

Продолжение таблицы

Наименование	Пояснение
Вход	object sender, EventArgs e. sender: object – указатель на объект, который стал инициатором события Click
Предусловия	Пользователь нажал командную кнопку
Процесс	Извлекает из свойства Tag командной кнопки номер соответствующей ей команды. Вызывает метод DoCmnd Интерфейса и передает в него номер команды
Выход	Нет
Постусловия	Нет
TPanelp_p_KeyPress	Обработчик события KeyPress для алфавитно-цифровых клавиш клавиатуры
Вход	object sender, KeyPressEventArgs e.
Предусловия	Пользователь нажал алфавитно-цифровую клавишу клавиатуры
Процесс	Определяет по нажатой алфавитно-цифровой клавише номер соответствующей ей команды. Вызывает метод DoCmnd Интерфейса и передает в него номер команды
Выход	Нет
Постусловия	Команда пользователя вызвана
TPanelp_p_KeyDown	Обработчик события KeyDown для клавиш управления клавиатуры
Вход	(object sender, KeyEventArgs e.)
Предусловия	Пользователь нажал клавишу управления клавиатуры
Процесс	Определяет по нажатой клавише управления номер соответствующей ей команды. Вызывает метод DoCmnd Интерфейса и передает в него номер команды
Выход	Нет
Постусловия	Команда пользователя вызвана
UpdateP1	Выполнить обновления, связанные с изменением p1
Вход	Нет

Продолжение таблицы

Наименование	Пояснение
Предусловия	Изменено основание с. сч. p1 исходного числа
Процесс	Выполняет необходимые обновления при смене ос. с. сч. p1
Выход	Нет
Постусловия	Состояние кнопок обновлено
UpdateP2	Выполнить обновления, связанные с изменением p2
Вход	Нет
Предусловия	Изменено основание с. сч. p2 результата
Процесс	Выполняет необходимые обновления при смене ос. с. сч. p2
Выход	Нет
Постусловия	Состояние кнопок обновлено
UpdateButtons	
Вход	Нет
Предусловия	Изменено основание с. сч. p1 исходного числа
Процесс	Обновляет состояния командных кнопок, предназначенных для ввода цифр выбранной системы счисления p1
Выход	Нет
Постусловия	Состояние кнопок обновлено.
выходToolStrip MenuItem_Click	Команда Выход основного меню класса TPanelp_p формы
Вход	object sender, EventArgs e.
Предусловия	Пользователь кликает мышью на пункте Выход основного меню формы
Процесс	Завершает работу приложения
Выход	Нет
Постусловия	Приложение завершено
справкаToolStrip MenuItem_Click	Команда Справка основного меню класса TPanelp_p формы
Вход	object sender, EventArgs e.
Предусловия	Пользователь кликает мышью на пункте Справка основного меню формы

Окончание таблицы

Наименование	Пояснение
Процесс	Показывает окно со справкой по приложению
Выход	Нет
Постусловия	Отображено окно справки
историяToolStrip MenuItem_Click	Команда История основного меню класса TPanelp_p формы
Вход	object sender, EventArgs e.
Предусловия	Пользователь кликает мышью на пункте История основного меню формы
Процесс	Открывает окно История
Выход	Нет
Постусловия	Окно История – открыто

Рекомендации к выполнению

1. Интерфейс приложения будет состоять из трех форм: основная форма, класс TPanel_p_p, HistoryForm – форма для отображения истории, AboutBox – форма, информирующая о приложении. Все они наследники класса Form.

2. Для реализации «Интерфейса» приложения разместите на форме компоненты, описанные в таблице ниже.

Имя компонента	Имя класса	Назначение
label1	Label	Исходное число
label2	Label	Результат
label3	Label	Подпись к исходному числу
label4	Label	Подпись к результату
trackBar1	TrackBar	Изменять основание с. сч. p1
trackBar2	TrackBar	Изменять основание с. сч. p2
numericUpDown1	NumericUpDown	Изменять основание с. сч. p1
numericUpDown2	NumericUpDown	Изменять основание с. сч. p2
button1 – button10	Button	Ввод цифр от 0 – 9
button11 – button16	Button	Ввод цифр от A – F
button17	Button	Разделитель целой и дробной частей (,)
button18	Button	Забой крайнего правого символа (BS)

Окончание таблицы

Имя компонента	Имя класса	Назначение
button19	Button	Удалить исходное число (CL)
button20	Button	Выполнить (Execute)
menuStrip1	menuStrip1	Основное меню главной формы
выходToolStripMenuItem	ToolStripMenuItem	Завершение работы приложения
историяToolStripMenuItem	ToolStripMenuItem	Просмотр журнала сеанса работы
справкаToolStripMenuItem	ToolStripMenuItem	Справка по приложению

Для этого перейдите на вкладку «Конструктор формы» окна редактора и сделайте форму активной. Добавьте на форму из вкладки «Панели элементов» из раздела «Все формы Windows Forms» командные кнопки Button. Кнопки помещайте на форму снизу вверх слева направо, начиная с 0. Порядок добавления кнопок на форму должен совпадать с порядком их нумерации. Выделите все кнопки и в окне «Свойства» на закладке «Свойства» раскройте свойство Font («Шрифт») и установите требуемые вам свойства для шрифта. Затем в свойство Text («Текст») занесите соответствующие цифры. Затем в свойство Tag каждой кнопки занесите целое число соответствующее кнопке: кнопки для ввода цифр нумеруйте от 0 до 15; кнопку для разделения целой и дробной части – 16; для удаления крайнего символа слева (BS) – 17; для очистки (CL) – 18; для выполнения (Enter) – 19. Измените размер кнопок (свойство Size), если это необходимо.

Добавьте на форму два компонента TrackBar из вкладки «Панель элементов» из раздела «Все формы Windows Forms». С помощью окна «Свойства» установите в их свойства Minimum значение 2, а в Maximum – значение 16. Рядом с каждым из них разместите компонент NumericUpDown «числовое поле со стрелками вверх/вниз». Добавьте на форму два компонента Label. В одном будем отображать основание системы счисления p1, в другом – подпись к нему. Добавьте на форму еще два компонента Label для основания системы счисления результата p2.

Добавьте на форму компонент MenuStrip из вкладки «Панель элементов» из раздела «Все формы Windows Forms». Добавьте в компонент MenuStrip три пункта меню MenuItem. С помощью окна «Свойства» установите в их свойство Text значение «Выход», «История»,

«Справка». Полученная форма будет иметь вид, как представлено на рис. 17.

Для настройки отклика приложения на действия пользователя необходимо создать обработчики событий для тех компонентов интерфейса, которыми будет манипулировать пользователь. Обработчик события для выделенного компонента создается с помощью окна «Свойства» закладки «События». Правее выбранного события в свободном поле необходимо сделать двойной клик мышью, и в классе формы появится текст шаблона обработчика этого события. Затем необходимо описать в обработчике необходимые действия. Обработчики событий для компонентов главной формы приведены в таблице ниже.

Имя компонента	Событие	Обработчик
TPanelp_p (главная форма)	Load	TPanelp_p_Load
TPanelp_p (главная форма)	KeyDown	TPanelp_p_KeyDown
TPanelp_p (главная форма)	KeyPress	TPanelp_p_KeyPress
trackBar1	Scroll	trackBar1_Scroll
trackBar2	Scroll	trackBar1_Scroll
numericUpDown1	ValueChanged	numericUpDown1_ValueChanged
numericUpDown2	ValueChanged	numericUpDown1_ValueChanged
button1 – button19	Click	button_Click
ВыходToolStripMenuItem	Click	ВыходToolStripMenuItem_Click
ИсторияToolStripMenuItem	Click	ИсторияToolStripMenuItem1_Click
СправкаToolStripMenuItem	Click	Справка ToolStripMenuItem_Click

Теперь можно запустить приложение и протестировать работу командных кнопок для ввода цифр действительного числа, представленного в выбранной системе счисления, а также для изменения оснований систем счисления исходного числа и результата. Ниже приведен текст модуля главного окна приложения.

```
namespace Конвертер
{
    public partial class Form1 : Form
    {
```

```
//Объект класса Управление.
Control_ ctl = new Control_();
public Form1()
{
    InitializeComponent();
}
private void Form1_Load(object sender, EventArgs e)
{
    label1.Text = ctl.ed.Number;
    //Основание с.сч. исходного числа p1.
    trackBar1.Value = ctl.Pin;
    //Основание с.сч. результата p2.
    trackBar2.Value = ctl.Pout;
    label3.Text = "Основание с. сч. исходного числа "
    + trackBar1.Value;
    label4.Text = "Основание с. сч. результата " +
    trackBar2.Value;
    label2.Text = "0";
    //Обновить состояние командных кнопок.
    this.UpdateButtons();
}
//Обработчик события нажатия командной кнопки.
private void button1_Click(object sender, EventArgs e)
{
    //ссылка на компонент, на котором кликнули мышью
    Button but = (Button)sender;
    //номер выбранной команды
    int j = Convert.ToInt16(but.Tag.ToString());
    DoCmnd(j);
}
//Выполнить команду.
private void DoCmnd(int j)
{
    if (j == 19) { label2.Text = ctl.DoCmnd(j);  }
    else
    {
        if (ctl.St == Control_.State.Преобразовано)
        {
            //очистить содержимое редактора
```

```
            label1.Text = ctl.DoCmnd(18);
        }
        //выполнить команду редактирования
        label1.Text = ctl.DoCmnd(j);
        label2.Text = "0";
    }
}
//Обновляет состояние командных кнопок по основанию
с. сч. исходного числа.
private void UpdateButtons()
{
    //просмотреть все компоненты формы
    foreach (Control i in this.Controls)
    {
        if (i is Button)//текущий компонент –
         командная кнопка
        {
            int j = Convert.ToInt16
            (i.Tag.ToString());
            if (j < trackBar1.Value)
            {
                //сделать кнопку доступной
                i.Enabled = true;
            }
            if ((j >= trackBar1.Value) && (j <= 15))
            {
                //сделать кнопку недоступной
                i.Enabled = false;
            }
        }
    }
}
//Изменяет значение основания с.сч. исходного числа.
private void trackBar1_Scroll(object sender,
EventArgs e)
{
    numericUpDown1.Value = trackBar1.Value;
    //Обновить состояние командных кнопок.
    this.UpdateP1();
```

```
}
//Изменяет значение основания с.сч. исходного числа.
private void numericUpDown1_ValueChanged(object
sender, EventArgs e)
{
    //Обновить состояние.
    trackBar1.Value =
    Convert.ToByte(numericUpDown1.Value);
    //Обновить состояние командных кнопок.
    this.UpdateP1();
}
//Выполняет необходимые обновления при смене ос.
  с.сч. p1.
private void UpdateP1()
{
    label3.Text = "Основание с. сч. исходного числа "
    + trackBar1.Value;
    //Сохранить p1 в объекте управление.
    ctl.Pin = trackBar1.Value;
    //Обновить состояние командных кнопок.
    this.UpdateButtons();
    label1.Text = ctl.DoCmnd(18);
    label2.Text = "0";
}
//Изменяет значение основания с.сч. результата.
private void trackBar2_Scroll(object sender,
EventArgs e)
{
    //Обновить состояние.
    numericUpDown2.Value = trackBar2.Value;
    this.UpdateP2();
}
//Изменяет значение основания с.сч. результата.
private void numericUpDown2_ValueChanged(object
sender, EventArgs e)
{
    trackBar2.Value =
    Convert.ToByte(numericUpDown2.Value);
    this.UpdateP2();
```

```
 }
 //Выполняет необходимые обновления при смене ос.
с.сч. p2.
 private void UpdateP2()
 {
     //Копировать основание результата.
     ctl.Pout = trackBar2.Value;
     //Пересчитать результат.
     label2.Text = ctl.DoCmnd(19);
     label4.Text = "Основание с. сч. результата " +
     trackBar2.Value;
 }
 //Пункт меню Выход.
 private void выходToolStripMenuItem_Click(object
 sender, EventArgs e)
 {
      Close();
 }
 //Пункт меню Справка.
 private void справкаToolStripMenuItem_Click
 (object sender, EventArgs e)
 {
     AboutBox1 a = new AboutBox1();
     a.Show();
 }
 //Пункт меню История.
 private void историяToolStripMenuItem1_Click
 (object sender, EventArgs e)
 {
     Form2 history = new Form2();
     history.Show();
     if (ctl.his.Записей() == 0)
     {
         history.textBox1.AppendText("История пуста");
         return;
     }
     //Ообразить историю.
     for (int i = 0; i < ctl.his.Записей(); i++)
     {
```

```
                history.textBox1.AppendText(ctl.his[i].
                ToString());
            }
        }
        //Обработка алфавитно-цифровых клавиш.
        private void Form1_KeyPress(object sender,
        KeyPressEventArgs e)
        {
            int i = -1;
            if (e.KeyChar >= 'A' && e.KeyChar <= 'F')
            i = (int)e.KeyChar - 'A' + 10;
            if (e.KeyChar >= 'a' && e.KeyChar <= 'f')
            i = (int)e.KeyChar - 'a' + 10;
            if (e.KeyChar >= '0' && e.KeyChar <= '9')
            i = (int)e.KeyChar - '0';
            if (e.KeyChar == '.') i = 16;
            if ((int)e.KeyChar == 8) i = 17;
            if ((int)e.KeyChar == 13) i = 19;
            if ((i < ctl.Pin)||(i >= 16)) DoCmnd(i);
        }
        //Обработка клавиш управления.
        private void Form1_KeyDown(object sender, KeyEven-
tArgs e)
        {
            if (e.KeyCode == Keys.Delete)
                //Клавиша Delete.
                DoCmnd(18);
            if (e.KeyCode == Keys.Execute)
                //Клавиша Execute Separator.
                DoCmnd(19);
            if (e.KeyCode == Keys.Decimal)
                //Клавиша Decimal.
                DoCmnd(16);
        }
    }
}
```

Содержание отчета

1. Задание.
2. Текст программы.
3. Тестовые наборы данных для тестирования методов класса.

Контрольные вопросы

1. Назначение компонентов класса Button.
2. Назначение компонентов класса Label.
3. Назначение компонентов класса TextBox.
4. Назначение компонентов класса TrackBar.
5. Назначение компонентов класса numericUpDown.
6. Когда возникает событие Load?
7. Когда возникает событие Click?
8. Когда возникает событие Scroll?
9. Когда возникает событие ValueChanged?
10. Когда возникает событие KeyPress?
11. Когда возникает событие KeyDown?

СЛОВАРЬ ТЕРМИНОВ

Объектно-ориентированное программирование (ООП) – совокупность принципов, технологий, а также инструментальных средств для создания программных систем на основе архитектуры взаимодействия объектов.

Предметная область – область человеческой деятельности, для которой разрабатывается программное обеспечение (ПО).

Объектно-ориентированный анализ – это методология, при которой требования формируются на основе понятий классов и объектов, составляющих словарь предметной области.

Объектно-ориентированное проектирование – это методология проектирования, соединяющая в себе процесс объектной декомпозиции и приемы представления как логической и физической, так и статической и динамической модели проектируемой системы.

Объектно-ориентированное программирование – это методология программирования, которая основана на представлении программы в виде совокупности объектов, каждый из которых является реализацией определенного класса, а классы образуют иерархию на принципах наследуемости.

Абстрагирование – выделение существенных характеристик некоторого объекта, которые отличают его от всех других видов объектов, и отвлечение от незначительных деталей.

Инкапсуляция – физическая локализация свойств и поведения в рамках единственной абстракции (рассматриваемой как «черный ящик»), скрывающая их реализацию за общедоступным интерфейсом.

Модульность – свойство системы, которое связано с возможностью ее декомпозиции на ряд внутренне сильно сцепленных, но слабо связанных между собой подсистем (модулей).

Иерархия – ранжированная или упорядоченная система абстракций, расположение их по уровням.

Объект – осязаемая сущность (предмет или явление) (процесс), имеющая четко определяемое поведение.

Состояние объекта – одно из возможных условий, в которых он может существовать (изменяется со временем).

Поведение – действие объекта и его реакция на запросы от других объектов.

Индивидуальность – свойства объекта, отличающие его от всех других объектов.

Класс – множество объектов, связанных общностью свойств, поведения, связей и семантики.

Атрибут – поименованное свойство класса, определяющее диапазон допустимых значений.

Операция – услуга, которую можно запросить у любого объекта данного класса. Воздействие одного объекта на другой с целью вызвать соответствующую реакцию.

Интерфейс – совокупность операций, определяющих набор услуг класса или компонента. Интерфейс не определяет внутреннюю структуру, все его операции имеют открытую видимость.

Полиморфизм – способность скрывать множество различных реализаций под единственным общим интерфейсом.

Компонент – относительно независимая и замещаемая часть системы, выполняющая четко определенную функцию в контексте заданной архитектуры.

Ассоциация – семантическая связь между классами.

Агрегация – форма ассоциации – более сильный тип связи между целым (составным) объектом и его частями (компонентными объектами).

Мощность – число объектов одного класса, связанных с одним объектом другого класса.

Зависимость – связь между двумя элементами модели, при которой изменения в спецификации одного элемента могут повлечь за собой изменения в другом элементе.

Обобщение – связь «тип–подтип», реализующая механизм наследования.

Вариант использования – последовательность действий (транзакций), выполняемых системой в ответ на событие, инициируемое некоторым внешним объектом (действующим лицом).

Действующее лицо (actor) – это роль, которую пользователь играет по отношению к системе. Целью потока событий является подробное документирование процесса взаимодействия действующего лица с системой, реализуемого в рамках варианта использования.

Связь коммуникации – связь между вариантом использования и действующим лицом; она изображается с помощью однонаправленной ассоциации (линии со стрелкой).

Связь включения – средство для описания фрагмента поведения системы (часть потока событий), который повторяется более чем в одном варианте использования.

Связь расширения – средство для описаний отклонения в нормальном поведении системы (описанных в пункте «Расширения»), которые выносятся в отдельный вариант использования.

Диаграмма последовательности – средство для моделирования поведения группы взаимодействующих объектов (для варианта использования или операции класса).

Сообщение (message) – средство, с помощью которого объект-отправитель запрашивает у объекта-получателя выполнение одной из его операций.

Диаграмма классов – средство для описания типов классов системы и различного рода статических связей, которые существуют между ними.

Класс – тип данных, определяемый пользователем, средство объектно-ориентированного программирования. Описание класса содержит ключевое слово class, за которым следует его имя, а далее в фигурных скобках – тело класса, т. е. список его элементов.

Спецификаторы – средства, предназначенные для определения свойства класса, а также для определения доступности класса для других элементов программы.

Константы класса – средство для хранения неизменяемых значений, связанных с классом.

Методы – средства для выполнения вычислений или действий, выполняемых классом или объектом класса.

Свойство – средство определения характеристик объектов класса, включая методы для их чтения и записи.

Конструктор – метод, предназначенный для инициализации объектов класса или класса в целом.

Деструкторы – методы предназначены для выполнения действий, которые необходимо выполнить перед уничтожением объекта.

Индексаторы – элементы класса, обеспечивающие возможность доступа к элементам класса по их порядковому номеру (индексу).

События – уведомления, которые могут генерировать класс.

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

Основной

1. *Леоненков А.В.* Объектно-ориентированный анализ и проектирование с использованием UML и IBM Rational Rose: учеб. пособие / А.В. Леоненков. – М.: Интернет-Университет информационных технологий; БИНОМ. Лаборатория знаний, 2006. – 320 с.: ил. – (Серия «Основы информационных технологий»).

2. *Буч. Г.* Обьектно-ориентированное проектирование с примерами приложений на С++: учебник / пер. с англ. 2-е изд. – М.: Изд-во Бином. – СПб.: Невский Диалект, 1999. – 560 с.

3. *Вендров А.М.* Проектирование программного обеспечения экономических информационных систем. – М.: Финансы и статистика, 2000.

4. *Фаулер М.*, *Скотт К.* UML в кратком изложении. Применение стандартного языка объектного моделирования: пер. с англ. – М.: Мир, 1999.

5. *Вендров А.М*. Практикум по проектированию программного обеспечения экономических информационных систем. – М.: Финансы и статистика, 2002.

6. *Орлов С.А.* Технологии разработки программного обеспечения. Разработка сложных программных систем: учеб. пособие для вузов. – СПб.: Питер, 2003. – 472с.: ил.

7. *Павловская Т.А.* C#. Программирование на языке высокого уровня: учебник для вузов. – СПб.: Питер, 2014. – 432 с.: ил. – (Серия «Учебник для вузов»).

Дополнительная

8. *Коберн А.* Современные методы описания функциональных требований к системам: пер. с англ. – М.: ЛОРИ, 2002.

9. *Библиотека* программиста [Электронный ресурс] URL: http:// www. coders-library.ru/files-view-1029.html (дата обращения 05.12.2016).

10. ИВТ(б)-вики. Структуры и алгоритмы обработки данных [Электронный ресурс] URL: http://xn--90abr5b.xn--p1ai/wiki (дата обращения 25.11.2014).

ОГЛАВЛЕНИЕ

Зайцев Михаил Георгиевич

ОБЪЕКТНО-ОРИЕНТИРОВАННЫЙ АНАЛИЗ И ПРОГРАММИРОВАНИЕ

Учебное пособие

Редактор *И.Л. Кескевич*
Выпускающий редактор *И.П. Брованова*
Корректор *И.Е. Семенова*
Дизайн обложки *А.В. Ладыжская*
Компьютерная верстка *Л.А. Веселовская*

Налоговая льгота – Общероссийский классификатор продукции
Издание соответствует коду 95 3000 ОК 005-93 (ОКП)

Подписано в печать 15.08.2017. Формат 60 × 84 1/16. Бумага офсетная. Тираж 70 экз.
Уч.-изд. л. 4,88. Печ. л. 5,25. Изд. № 361/16. Заказ № 1010. Цена договорная

Отпечатано в типографии
Новосибирского государственного технического университета
630073, г. Новосибирск, пр. К. Маркса, 20