

Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Кузбасский государственный технический университет
имени Т. Ф. Горбачева»

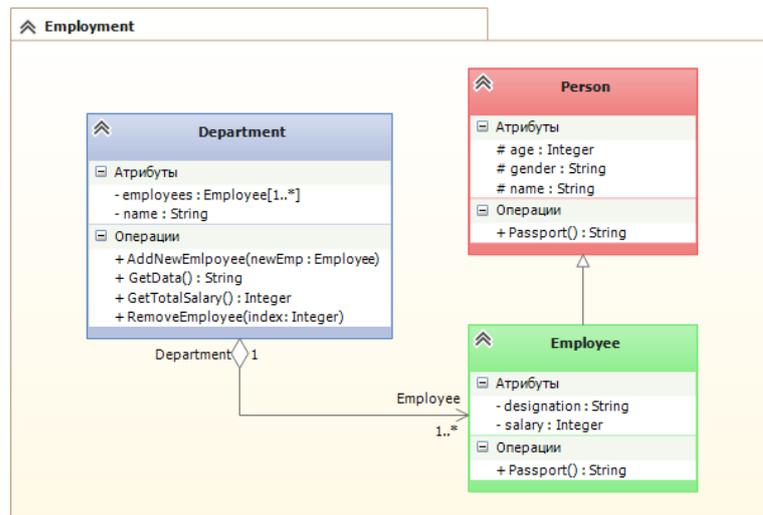
Кафедра информационных
и автоматизированных производственных систем

Составители:
Д. Е. Турчин, О. Н. Ванеев

ТЕОРИЯ ИНФОРМАЦИОННЫХ ПРОЦЕССОВ И СИСТЕМ

Лабораторный практикум

Рекомендовано учебно-методической комиссией
направления 09.03.02 «Информационные системы и технологии»
в качестве электронного издания для использования в учебном процессе



Рецензенты:

Чичерин И. В. – заведующий кафедрой информационных и автоматизированных производственных систем

Сыркин И. С. – доцент, кандидат технических наук кафедры информационных и автоматизированных производственных систем

Турчин Денис Евгеньевич

Ванеев Олег Николаевич

Теория информационных процессов и систем : лабораторный практикум: для студентов очной формы обучения направления подготовки бакалавров 09.03.02 «Информационные системы и технологии» / Кузбасский государственный технический университет им. Т. Ф. Горбачева ; Кафедра информационных и автоматизированных производственных систем ; составители: Д. Е. Турчин, О. Н. Ванеев. – Кемерово : КузГТУ, 2024. – 1 файл (3212 КБ). – Текст : электронный.

В данных методических указаниях изложены содержание лабораторных работ, порядок и примеры их выполнения, а также контрольные вопросы к ним.

© КузГТУ, 2024

© Ванеев О. Н., Турчин Д. Е.,
составление, 2024

СОДЕРЖАНИЕ

Лабораторная работа №1. Основы объектно-ориентированного моделирования структуры системы.....	6
1. Цель и задачи работы.....	6
2. Основные теоретические сведения.....	6
3. Порядок выполнения работы	22
4. Требования к содержанию отчёта.....	25
5. Контрольные вопросы.....	25
Лабораторная работа №2. Отношения между классами и их обозначение на диаграммах классов UML	26
1. Цель и задачи работы	26
2. Теоретические положения	26
3. Порядок выполнения работы	34
Варианты заданий.....	35
4. Контрольные вопросы.....	37
Лабораторная работа №3. Основы разработки классов в приложениях на языке C#.....	38
1. Цель и задачи работы	38
2. Основные теоретические сведения.....	38
3. Порядок выполнения работы	52
4. Контрольные вопросы.....	52
Лабораторная работа №4. Использование свойств.....	54
1. Цель работы.....	54
2. Теоретические сведения	54
3. Порядок выполнения работы	56
4. Контрольные вопросы.....	56
5. Содержание отчёта	57
Лабораторная работа №5. Работа с библиотеками классов в Visual Studio	58
1. Цель работы.....	58

2. Теоретические положения	58
3. Задание и порядок выполнения.....	60
4. Контрольные вопросы.....	60
Лабораторная работа №6. Статические классы	61
1. Теоретические положения	61
2. Порядок выполнения работы	63
3. Контрольные вопросы.....	65
Лабораторная работа №7. Работа со структурами и перечислениями на языке С#.....	67
1. Цель и задачи работы	67
2. Основные теоретические сведения.....	67
3. Порядок выполнения работы	74
4. Контрольные вопросы.....	76
Лабораторная работа №8. Разработка семейства классов с использованием композиции и наследования на языке С#	77
1. Цель и задачи работы	77
2. Основные теоретические сведения.....	77
3. Порядок выполнения работы и варианты заданий	92
4. Контрольные вопросы и задачи	93
Лабораторная работа №9. Работа в проекте Visual Studio Windows Forms.....	94
1. Цель работы.....	94
2. Теоретические положения	94
3. Задания для работы	94
Лабораторная работа №10. Основы использования полиморфизма в семействах классов на языке С#.....	95
1. Цель и задачи работы	95
2. Основные теоретические сведения.....	95
3. Порядок выполнения работы и варианты заданий	100
4. Варианты заданий.....	101

5. Контрольные вопросы и задачи	102
Лабораторная работа №11. Абстрактные классы и методы. Запечатанные классы и методы	103
1. Теоретические положения	103
2. Порядок выполнения работы и варианты заданий	110
Лабораторная работа №12. Работа с интерфейсами в приложениях на языке С#.....	113
1. Цель и задачи работы	113
2. Основные теоретические сведения.....	113
3. Порядок выполнения работы и варианты заданий	125
4. Контрольные вопросы и задачи	127
Лабораторная работа №13. Работа с событиями средствами С#.....	128
1. Цель и задачи работы	128
2. Основные теоретические сведения.....	128
3. Порядок выполнения работы и варианты заданий	131
4. Контрольные вопросы и задачи	133

ЛАБОРАТОРНАЯ РАБОТА №1. ОСНОВЫ ОБЪЕКТНО-ОРИЕНТИРОВАННОГО МОДЕЛИРОВАНИЯ СТРУКТУРЫ СИСТЕМЫ

1. ЦЕЛЬ И ЗАДАЧИ РАБОТЫ

Цель работы – приобрести умение производить объектно-ориентированное моделирование структуры разрабатываемой программной системы с помощью диаграмм классов языка UML.

Основные задачи работы:

- освоить использование основных элементов языка UML для построения диаграмм классов;
- научиться строить диаграммы классов с помощью MS;
- Office Visio.

Работа рассчитана на 4 часа.

2. ОСНОВНЫЕ ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

2.1. Определение объекта и класса. Общая характеристика языка UML

В ООП система представляется в виде совокупности взаимодействующих объектов, каждый из которых является экземпляром определенного класса. Базовыми понятиями ООП являются понятия объекта и класса.

Объект – конкретный идентифицируемая сущность (элемент) реальная или абстрактная, играющая определенную роль в некоторой предметной области.

С точки зрения ООП, объект представляет собой совокупность данных, характеризующих его состояние и функций их обработки, определяющих его поведение.

Например, у объекта «Студент» имеются такие характеристики, как имя, пол, номер зачётки, группа и т.д. Студент также обладает поведением, то есть он ходит, говорит, обучается и т.д.

Класс – описание множества объектов, которые имеют одинаковый смысл, структуру данных и поведение. Термины «объект» и «экземпляр класса» взаимозаменяемы. Если использовать

аналогию с деталью и ее чертежом, то класс – это чертеж, а объект – конкретная деталь, изготовленная по чертежу.

До того как начать программирование классов, необходимо определить, сколько и какие классы нужны для решения поставленной задачи, какими характеристиками и поведением должны обладать экземпляры классов, а также установить взаимосвязи между классами. Указанные задачи решаются в ходе **объектно-ориентированного анализа и проектирования**.

Язык UML.

Для создания объектно-ориентированных моделей систем используют языки визуального моделирования, наиболее популярным из которых в настоящее время является UML.

UML (*Unified Modeling Language* – унифицированный язык моделирования) – стандартный язык для написания моделей анализа, проектирования и реализации объектно-ориентированных программных систем.

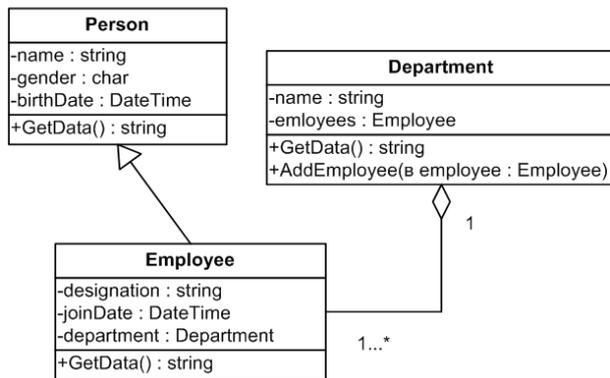
Модели UML могут преобразовываться в код на языках программирования (C++, Java, C# и др.) и таблицы реляционных баз данных.

Основным средством представления объектной модели в UML являются диаграммы.

Диаграмма UML – это нагруженный связный граф, в котором вершины нагружаются элементами модели, а дуги (ребра) – отношениями между элементами.

Одна диаграмма способна изобразить лишь отдельный аспект системы, поэтому используют набор диаграмм. Этот набор обеспечивает визуальное представление программной системы с разных точек зрения. Объединение всех точек зрения дает полную картину, достаточную для решения конкретных задач разработки программного обеспечения. Примеры диаграмм UML представлены на рисунке (Рисунок 1.1).

a)



б)

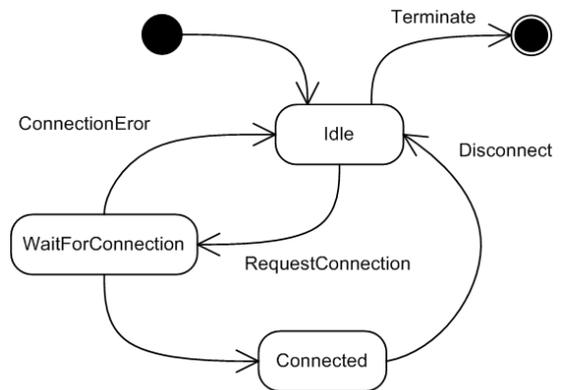


Рисунок 1.1. Примеры диаграмм UML: *a* – диаграмма классов; *б* – диаграмма состояний (конечных автоматов)

Диаграммы UML разделяются на две группы:

- **структурные диаграммы**, которые отражают статическую структуру элементов в программной системе (диаграммы классов, пакетов, компонентов, и др.);
- **диаграммы поведения**, которые описывают динамику программной системы на различных этапах ее разработки (диаграммы вариантов использования, деятельности, последовательности, состояний и др.).

Для создания диаграмм UML существуют различные среды моделирования, например Visual Paradigm.

Построение совокупности моделей в среде рассматривается как отдельный проект. Для построения нового проекта UML в меню **Project** следует выбрать команду **New**.

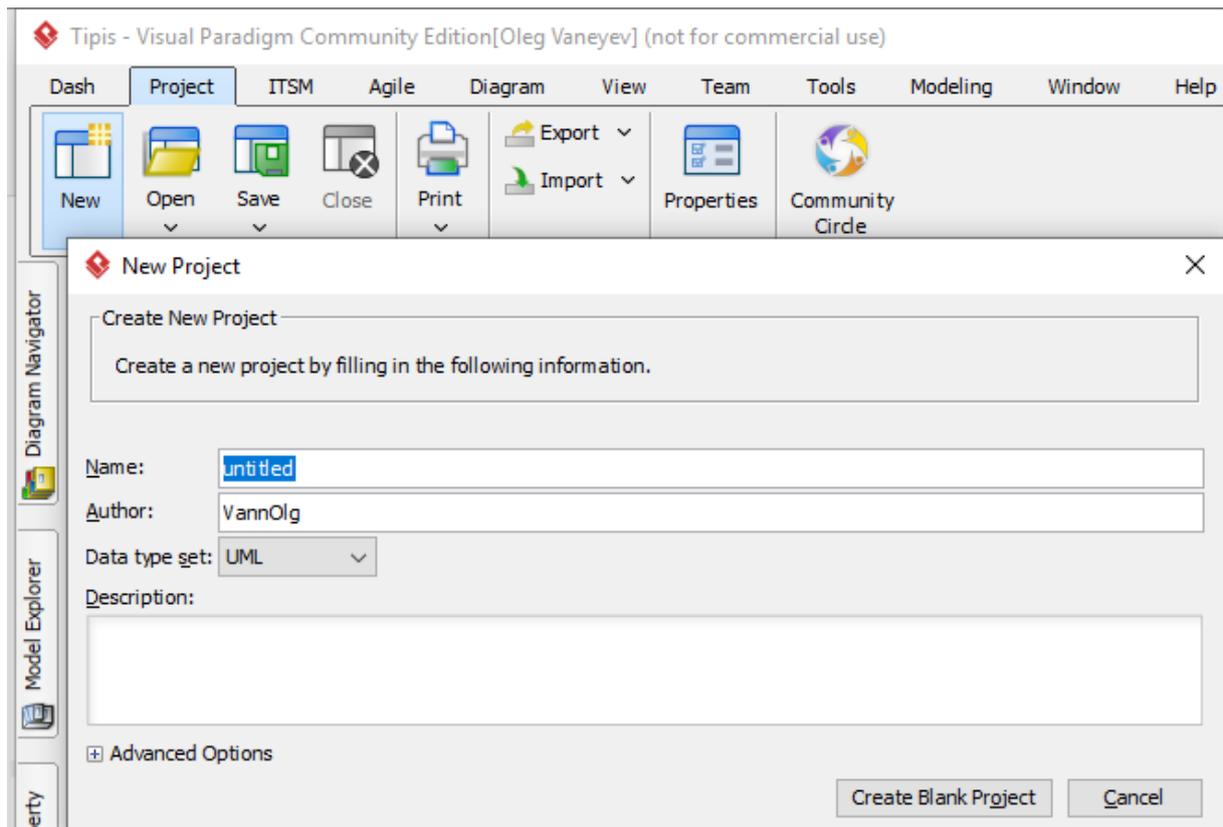


Рисунок 1.2. Создание нового проекта

В работе предполагается создание пустого проекта (Create blank project).

Для просмотра содержимого проекта и включённых в него моделей в среде Paradigm предусмотрено специальное средство обозреватель модели (Model Explorer). По умолчанию Model Explorer располагается на закладке слева. Обозреватель моделей отображает содержание моделей включённых в проект в виде древовидной структуры. Верхний узел соответствует всему проекту. Добавление элементов модели возможно через контекстное меню (Рисунок 1.3). Обычно структура проекта включает вложенные модели, в которых содержатся пакеты (которые могут содержать другие пакеты) и конечные элементы модели, представленные в виде структурных сущностей и диаграмм (Рисунок 1.4).

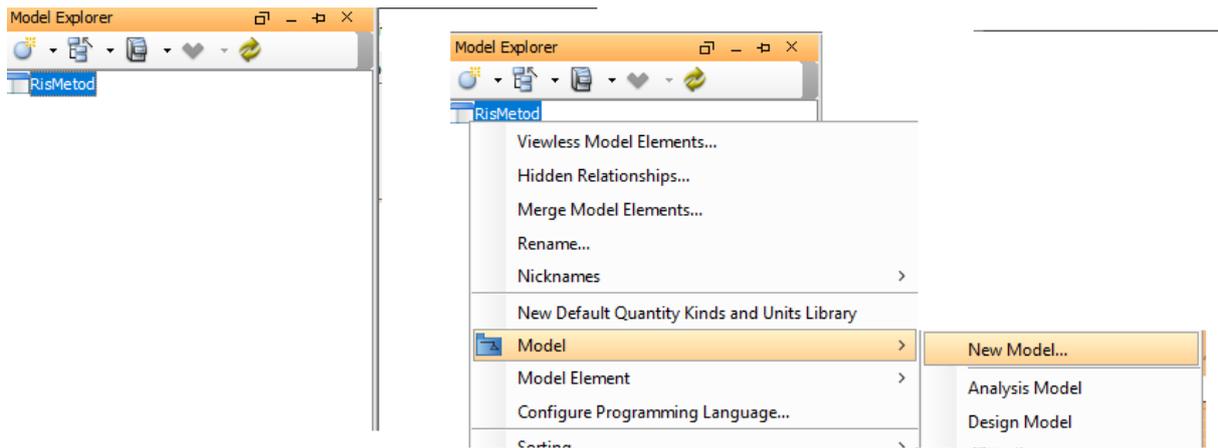


Рисунок 1.3. Обзорщик модели. Добавление содержимого

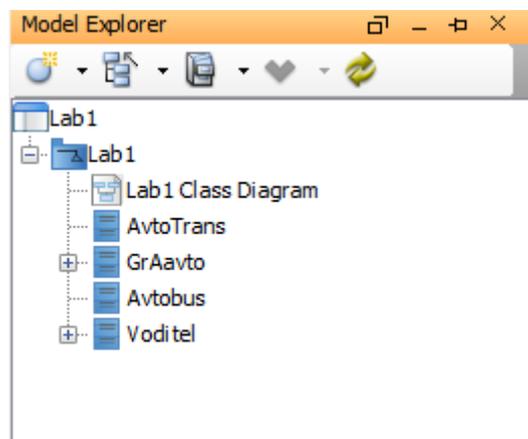


Рисунок 1.4. Содержание проекта Visual Paradigm

Работа с элементами модели может производиться как непосредственно в обзорщике модели, так и в окне диаграммы. В обзорщике модели элемент модели, например новый класс, можно добавить, как отмечалось ранее, через контекстное меню.

После добавления элемента модели открывается окно для задания его параметров спецификации (Рисунок 1.5).

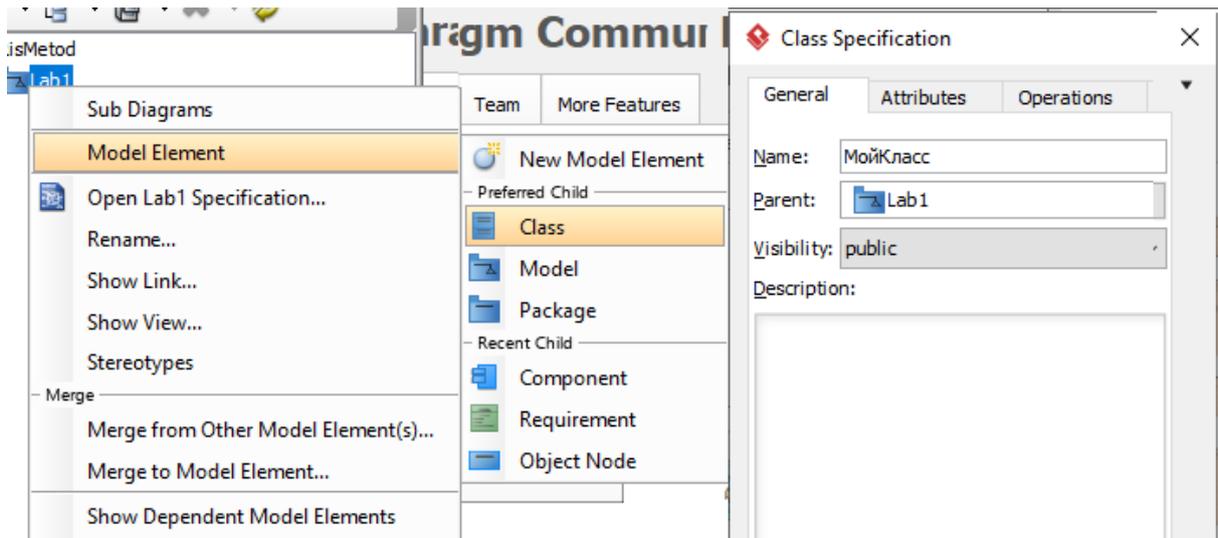


Рисунок 1.5. Добавление нового класса в модель
(в составляющую модель проекта Lab1)

Добавление диаграмм производится аналогично структурным сущностям модели – через контекстное меню.

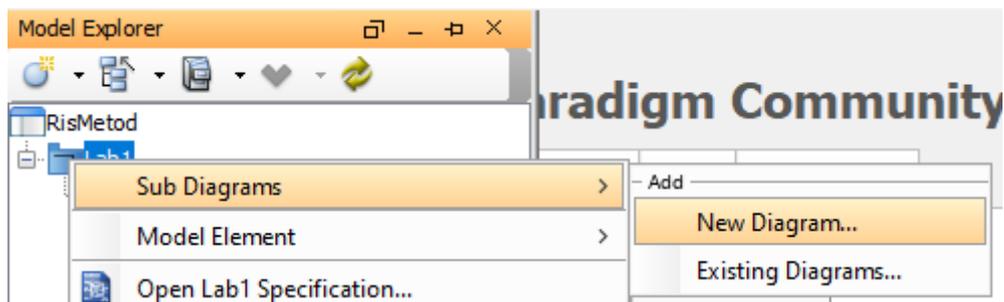


Рисунок 1.6. Добавление диаграммы
в составляющую модель проекта

После добавление диаграммы открывается диалог выбора типа диаграммы (Рисунок 1.7).

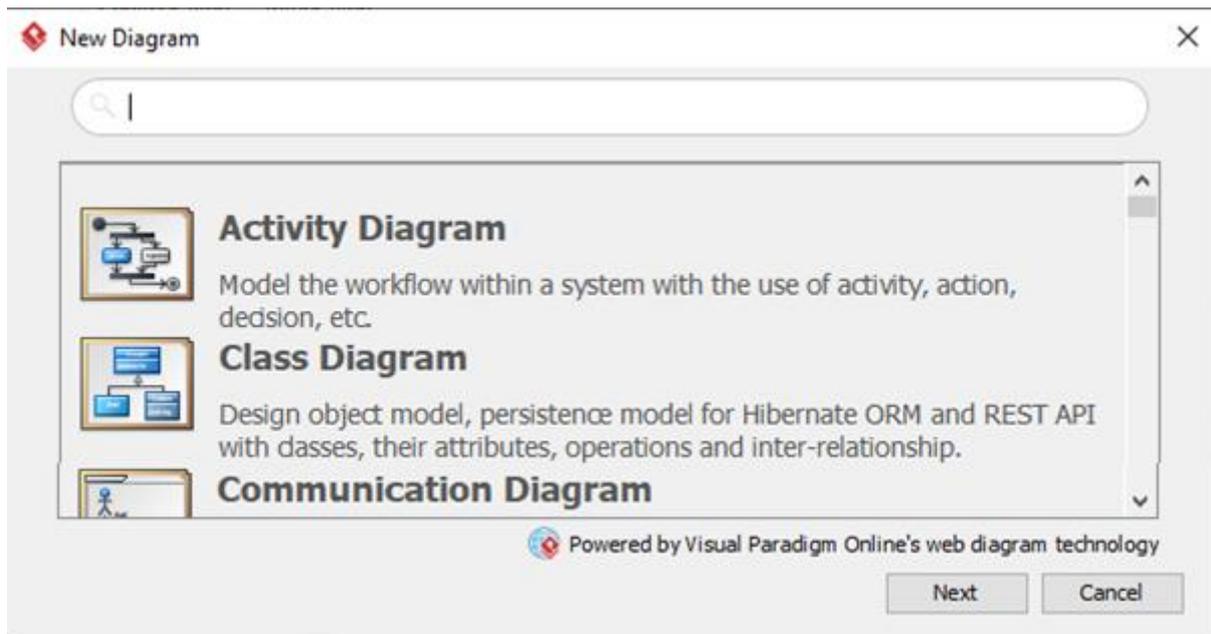


Рисунок 1.7. Диалог выбора типа диаграммы.

В данной работе рассматриваются диаграмма классов (Class Diagram). При выборе типа диаграммы открывается рабочее окно диаграммы и панель элементов, которые могут быть использованы на данной диаграмме (Рисунок 1.8).

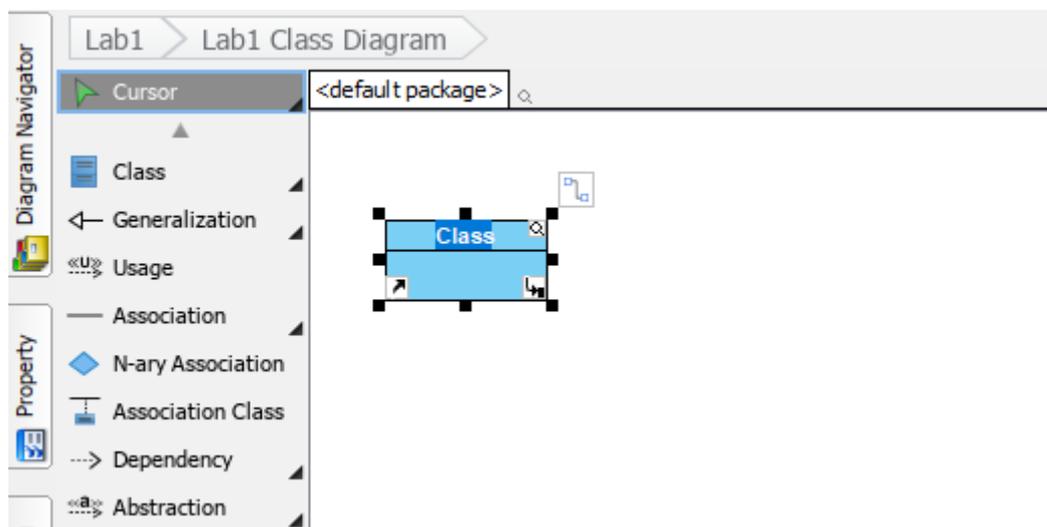


Рисунок 1.8. Рабочее окно диаграммы классов

Построение диаграммы UML можно выполнять путём перетаскивания мышью элементов из панели элементов в рабочую область, где они определенным образом компоуются.

Кроме того в рабочую область диаграммы можно перетащить элемент из обозревателя модели (Рисунок 1.9).

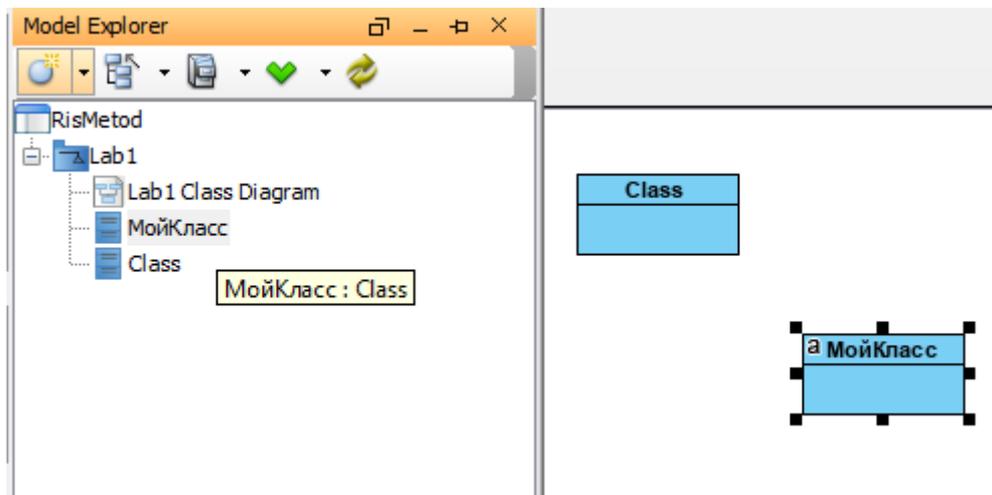


Рисунок 1.9. Вставка в диаграмму существующих элементов моделей

2.2. Диаграммы классов. Описание атрибутов и операций классов в языке UML

Диаграммы классов

Основным средством для представления логической структуры программных систем в языке UML являются **диаграммы классов** (*class diagram*)

Диаграмма классов – это статическая структурная диаграмма, узлами которой являются классы, а дугами – отношения между классами.

Отдельный класс изображается на диаграмме классов в виде прямоугольника, называемого **классификатором**.

В UML элементы класса, задающие характеристики его экземпляров, называют **атрибутами**, а элементы, реализующие поведение экземпляров класса, – **операциями**.

В программных классах атрибуты реализуются через константы, переменные, свойства; операции реализуются через процедуры, функции.

В языке C# для атрибута используется термин поле (*field*), а для операции – метод (*method*).

Каждый класс на диаграмме классов должен обладать именем, отличающим его от других классов. К имени класса может быть добавлено имя пакета, в котором этот класс располагается.

Имя класса должно быть уникальным в пределах пакета. Оно указывается в первой верхней секции классификатора, записывается по центру секции имени и должно начинаться с заглавной буквы. Пробелы в имени класса опускаются.

Имя класса на диаграммах классов указывается всегда, атрибуты и операции – выборочно. На начальных этапах разработки класс может обозначаться простым прямоугольником с указанием только имени класса (Рисунок 1.5, а). В дальнейшем диаграммы классов дополняются атрибутами (Рисунок 1.5, б) и операциями (Рисунок 1.5, в).

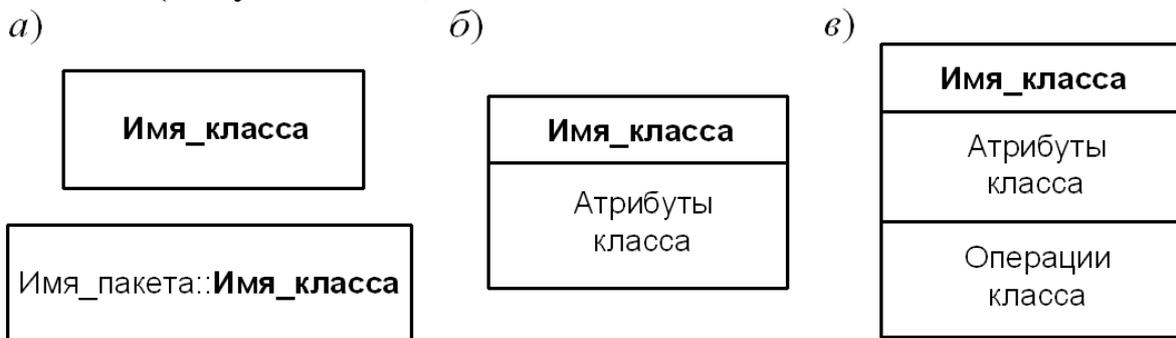


Рисунок 1.10. Представление класса на диаграмме классов

По *уровню видимости* (*visibility*) атрибуты и операции класса в языке UML разделены на 4 группы. Для указания уровня видимости определены четыре модификатора видимости, приведенные в (Таблица 1.1).

Таблица 1.1.

Модификаторы видимости в языке UML

Название	Обозначение	Описание
Открытая (public)	+	Доступен для всех элементов
Защищенная (protected)	#	Доступен только для элементов класса и подклассов (потомков)

Закрытая (private)	–	Доступен только для элементов класса
Пакетная (package)	~	Доступен только для элементов пакета

Область действия атрибута (операции). Если атрибут (операция) подчеркивается, то его областью действия является класс, в противном случае – областью действия является экземпляр.

Описание атрибутов и операций классов на диаграммах UML.

Общий синтаксис представления атрибута в языке UML имеет следующий вид:

[Мод_видим] имя_атрибута: Тип [Кратность] [= Нач_знач] [{Свойство}]

Модификатор видимости задаёт уровень видимости с помощью символов: «-», «+», «#», «~». Если модификатор видимости не указан, то считают, что атрибут объявлен с закрытым уровнем видимости.

Тип атрибута может быть выбран из типов конкретного языка программирования или являться именем одного из классов на диаграмме классов.

Имя атрибута должно быть уникальным в пределах класса. Принято начинать имя атрибута со строчной буквы.

Кратность (множественность) атрибута показывает количество конкретных атрибутов данного типа, входящих в состав класса. Кратность атрибута может быть задана следующими способами:

- 5 – определенное число;
- 0..* – ноль или более;
- 1..* – один или более;
- 0..1 – ноль или один;
- 3..12 – определенный диапазон.

Если кратность атрибута не указана, то по умолчанию принимается ее значение равное 1.

Начальное значение атрибута является значением, которое атрибут получает по умолчанию при создании экземпляра класса.

Свойство является строкой, которая указывает на особое свойство атрибута. Атрибутам можно присвоить свойство уникальности с помощью свойства **{unique}**. Например, если атрибут представляет собой ключ хэш-таблицы или первичный ключ в таблице реляционной базы данных.

Описание операции в языке UML имеет следующий вид:

```
[Мод_видим] Имя_Операции([Список_парам]) [: Тип_возвр] [{Свойство}]
```

Имя операции является единственным обязательным элементом и должно быть уникальным в пределах данного класса.

Для именованной операции рекомендуется использовать глаголы, соответствующие ожидаемому поведению объектов данного класса.

Модификатор видимости операции может принимать такие же значения, как и для атрибутов класса.

Тип возвращаемого значения является зависимой от языка реализации спецификацией типа. Операция может не возвращать никакого значения.

Свойство служит для определения особых свойств данной операции. Например, если операция не может изменять состояние объекта, то она обозначается как **{isQuery}** (запрос).

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

```
[Вид_парам] имя_параметра [: Тип_парам] [= Знач_по_умолч]
```

Вид параметра – это одно из ключевых слов:

- **in** – входной параметр; не может модифицироваться (по умолчанию);
- **out** – выходной параметр; может модифицироваться для передачи информации в вызывающий объект;
- **inout** – входной параметр; может модифицироваться.

Атрибуты можно задавать через служебное окно спецификации класса. Окно можно вызвать через контекстное меню класса в рабочем окне диаграммы класса или окне обозревателя моделей (Рисунок 1.11). Окно имеет несколько закладок для задания атрибутов, операций класса и других данных (Рисунок 1.12).

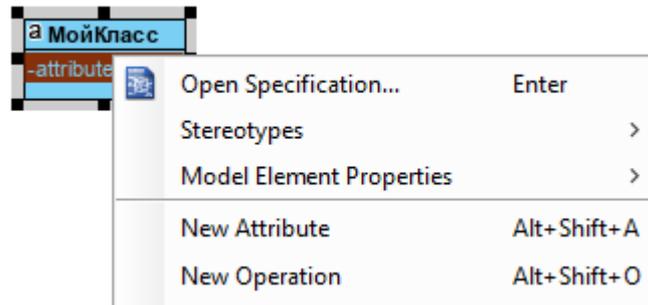


Рисунок 1.11. Вызов окна спецификации класса из рабочего окна диаграммы классов

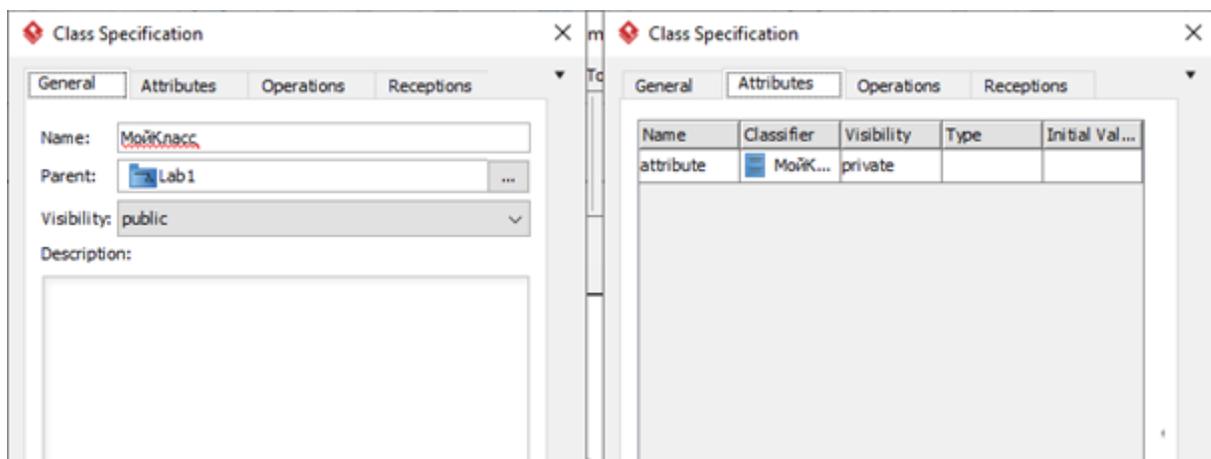


Рисунок 1.12. Окно спецификации класса. Закладки «Общая» и «Атрибуты»

Через контекстное меню на закладки для задания атрибутов через контекстное меню отдельного атрибута можно открыть окно задания его подробной спецификации (), в котором для выделенного атрибута можно указать область владения (экземпляр или класс), а также сделать его текстовое описание в поле **Документация**.

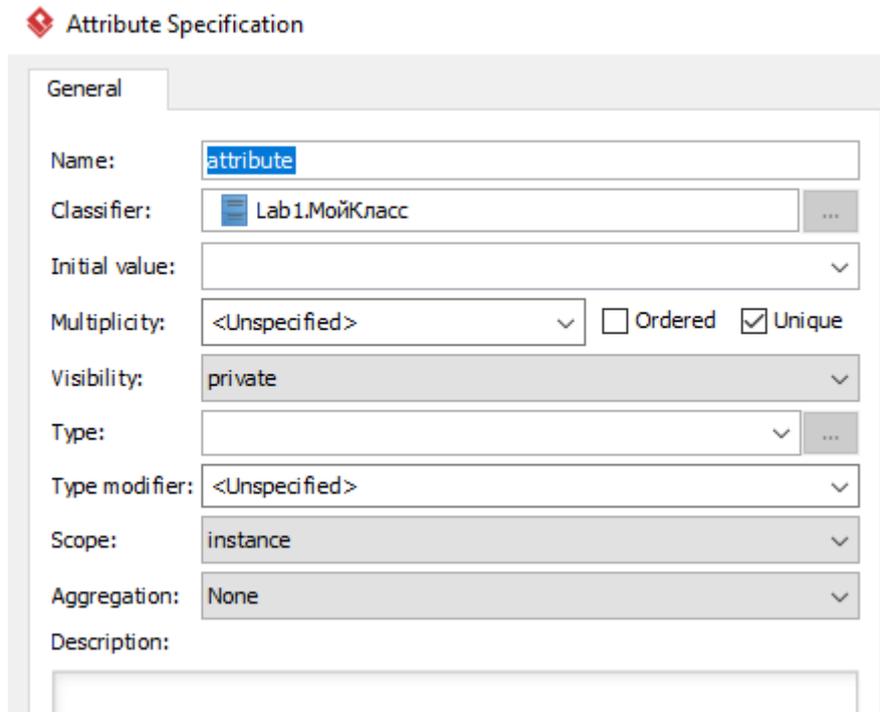


Рисунок 1.13. Окно спецификации атрибута

Аналогично на закладке, предназначенной для работы с операциями, можно просмотреть и задать операции класса.

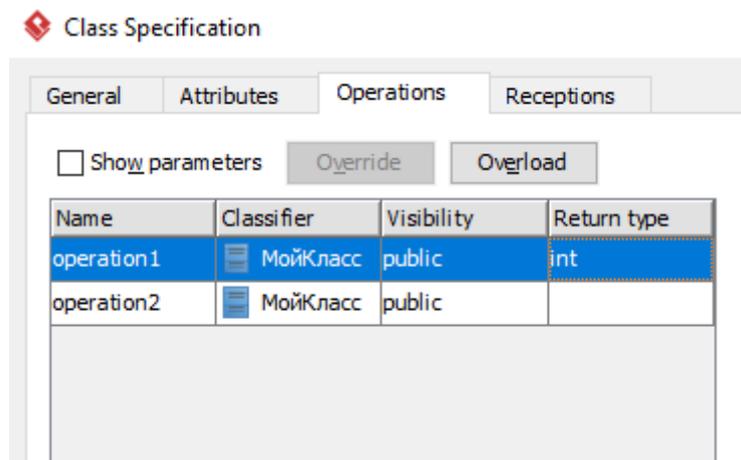


Рисунок 1.14. Окно спецификаций класса (закладка Операции)

Для добавления и редактирования параметров операции класса необходимо через контекстное меню вызвать окно спецификации операций

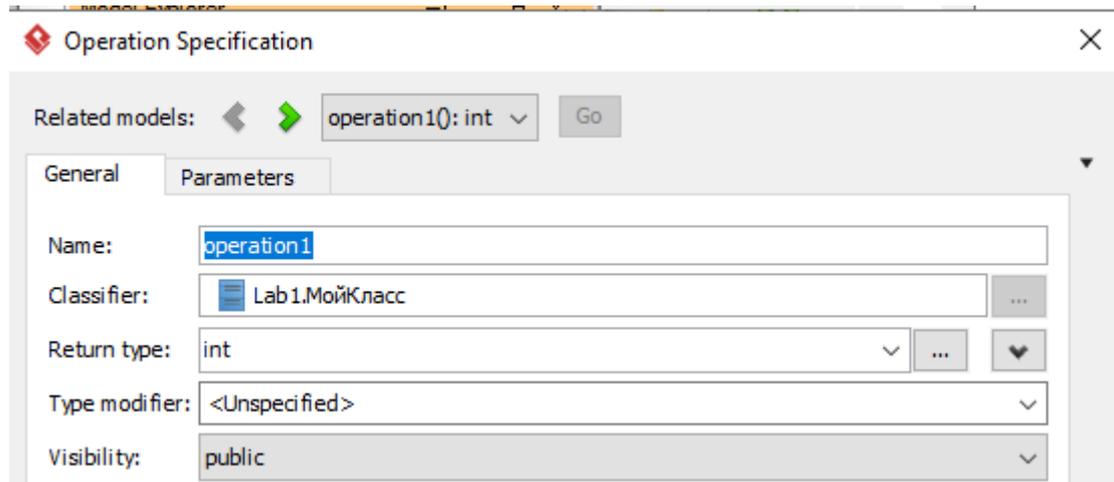


Рисунок 1.15. Окно спецификации операции

Описание атрибутов и операций классов с помощью диаграмм классов UML

Требуется получить описание атрибутов и операций заданного класса (**Автомобиль**), а также отобразить этот класс на диаграммах классов UML.

Класс **Автомобиль** представляет автомобили, для которых важно знать текущее и заданное местоположение, скорость движения и уровень топлива. Примем, что движение автомобиля в заданное место осуществляется по прямой линии. При достижении заданного места автомобиль останавливается. Также остановка происходит при нулевом уровне топлива. Заправить автомобиль можно только при его полной остановке.

Описание атрибутов класса **Автомобиль** приведено в табл. 1.2, а операций – в табл. 1.3.

Таблица 1.2.

Описание атрибутов класса **Автомобиль**

Видим.	Имя	Тип	Область	Крат.	Нач. знач.
private	колич	int	Класс	1	0
private	марка	string	Экземпляр	1	-
private	треб_X	double	Экземпляр	1	-
private	тек_X	double	Экземпляр	1	-
private	тек_V	int	Экземпляр	1	-
private	тек_F	int	Экземпляр	1	-

Атрибуты класса **Автомобиль**, содержат следующие данные:

- **колич** – количество созданных экземпляров класса;
- **марка** – марка автомобиля (например, «Газель 3221»);
- **треб_X** – координата заданного места по оси X, км;
- **тек_X** – текущая координата автомобиля по оси X, км;
- **тек_V** – текущая скорость, км/ч;
- **тек_F** – текущий уровень топлива в баке, л.

Таблица 1.3.

Описание операций класса **Автомобиль**

Видим.	Имя	Тип возврата	Область	Параметры		
				Вид	Имя	Тип
public	Опред_колич	double	Класс	–	–	–
public	Получ_характ	string	Экземпляр	–	–	–
public	Опред_коорд	string	Экземпляр	–	–	–
public	Измер_скор	int	Экземпляр	–	–	–
public	Измер_топл	int	Экземпляр	–	–	–
public	Задать_коорд	void	Экземпляр	in	x	double
				in	y	double
public	ИзменСкор	void	Экземпляр	in	dV	double
public	Заправить	void	Экземпляр	in	dF	double
public	Ехать	void	Экземпляр	in	dT	double
				in	v	double

Описание операций для класса **Автомобиль**:

- **Опред_колич()** – определить число созданных экземпляров класса; возвращает значение атрибута **колич**;
- **Получ_характ()** – получить данные о характеристиках автомобиля; возвращает значения атрибутов **марка**, **max_V**, **расход** и **max_F**, в форме строки «**марка: ...; макс. скорость: ... км/час; расход топлива: ...**».
- **Опред_коорд()** – текущие координаты автомобиля (например с помощью спутникового навигатора); возвращает значение атрибутов **тек_X**
- **Измер_скор()** – измерить текущую скорость автомобиля (с помощью спидометра); возвращает значение атрибута **тек_V**;
- **Измер_топл()** – измерить уровень топлива (с помощью датчика уровня топлива); возвращает значение атрибута **тек_F**;
- **ИзменСкорость(dV)** – увеличить скорость на заданную величину **dV** (с помощью педали «Газ»); прибавляет к значению атрибута **тек_V** величину **dv**;
- **Заправить(dF)** – заправить автомобиль топливом на величину **dF**; увеличивает значение атрибута **тек_F** на величину **dF**; не может производиться при **тек_V > 0**;

Представление класса **Автомобиль** на диаграмме классов показано на рис. 1.16.

Автомобиль
-Колич : int = 0
+Марка : string
-треб_X : double
-тек_X : double
-тек_V : double
-тек_F : double
+ОпредКолич() : int
+ПолучХаракт() : string
+ОпредКоорд() : double
+ИзмерСкор() : double
+ИзмерТопл() : double
+ИзмерСкорость() : double
+Заправить() : double

Рисунок 1.16. Класс Автомобиль на диаграмме классов

3. ПОРЯДОК ВЫПОЛНЕНИЯ РАБОТЫ

Данная лабораторная работа предполагает выполнение следующих этапов:

1. Изучить методические указания к лабораторной работе.
2. Для заданных классов (табл. 1.4) требуется описать атрибуты и операции, используя язык UML. Отобразить классы с их атрибутами и операциями на диаграмме классов с помощью MS Visio. К заданным атрибутам необходимо добавить статические атрибуты «количество» и «время», а к операциям – статические операции «определить количество» и «определить время».
3. Оформить и защитить отчет по лабораторной работе.

Таблица 1.4.

Варианты заданий для описания атрибутов и операций классов на диаграмме классов

№ вар.	Описание классов
1, 9, 17	<p>Класс 1: Принтер. <i>Атрибуты:</i> модель, скорость печати, максимальное и текущее число чистых листов в лотке, объём картриджа и текущее количество краски. <i>Операции:</i> Определить число чистых листов в лотке; Определить количество чернил; Заправить картридж; Положить определённое число листов в лоток; Печатать заданное число листов.</p> <p>Класс 2: Трёхмерный вектор. <i>Атрибуты:</i> Компоненты вектора. <i>Операции:</i> Получить данные; Определить длину, Сложить два вектора, Найти скалярное произведение двух векторов.</p>
2, 10, 18	<p>Класс 1: Пулемёт (оснащён сменным стволом). <i>Атрибуты:</i> скорострельность, максимальная и текущая температура ствола, нагревание ствола при одиночном выстреле, скорость остывания ствола, температура окружающей среды. <i>Операции:</i> Измерить температуру ствола; Сменить ствол; Выпустить заданное число пуль; Остановить стрельбу на определённое время.</p> <p>Класс 2: Прямоугольник (стороны параллельны осям координат).</p>

№ вар.	Описание классов
	<p>Атрибуты: координаты верхнего левого и нижнего правого углов.</p> <p>Операции: Получить данные; Определить площадь и периметр; Переместить на заданные величины по координатным осям.</p>
3, 11, 19	<p>Класс 1: Грузовой лифт.</p> <p>Атрибуты: максимальный и текущий этаж, максимальный и текущий вес груза, объём камеры лифта и объём груза, время перемещения лифта между соседними этажами.</p> <p>Операции: Определить местоположение лифта; Вызвать лифт на заданный этаж; Загрузить в лифт груз заданного веса и объёма; Перевезти в лифте груз на заданный этаж.</p> <p>Класс 2: Рациональное число (отношение двух целых чисел).</p> <p>Атрибуты: значение числителя и знаменателя.</p> <p>Операции: Получить данные; Сложить/Вычесть два рациональных числа; Умножить/Разделить два рациональных числа.</p>
4, 12, 20	<p>Класс 1: Электрочайник (оснащён термореле).</p> <p>Атрибуты: вместимость и текущее количество воды, максимальная и текущая температура воды, скорость нагревания и скорость остывания воды в чайнике.</p> <p>Операции: Измерить объём и температуру воды; Налить/Слить заданное количество воды; Нагреть воду в течение определённого времени; Ожидать в течение заданного времени.</p> <p>Класс 2: Круг.</p> <p>Атрибуты: координаты центра, радиус.</p> <p>Операции: Получить данные; Определить площадь круга и длину окружности; Определить находится ли заданная точка внутри круга.</p>
5, 13, 21	<p>Класс 1: Подъёмный кран</p> <p>Атрибуты: максимальная высота подъёма груза, текущая высота подъёма груза, скорость подъёма/спуска, грузоподъёмность, вес груза.</p> <p>Операции: Определить положение грузозахватного приспособления; Переместить приспособление на заданную высоту; Прицепить груз заданного веса; Отцепить груз.</p> <p>Класс 2: Точечный электрический заряд</p> <p>Атрибуты: значение заряда; координаты на плоскости.</p> <p>Операции: Получить данные; Определить силу электростатического взаимодействия с точечным зарядом, имеющим</p>

№ вар.	Описание классов
	указанные координаты; Определить расстояние между двумя зарядами.
6, 14, 22	<p>Класс 1: Смеситель <i>Атрибуты:</i> максимальный и текущий угол поворота крана с холодной (горячей) водой; максимальная пропускная способность (л/мин); положение переключателя (вода через излив, вода через душевую лейку); <i>Операции:</i> Повернуть кран с холодной (горячей) водой на заданный угол; Переключить воду на излив; Переключить воду на лейку; Определить расход воды за определённое время.</p> <p>Класс 2: Квадратное уравнение ($ax^2 + bx + c = 0$). <i>Атрибуты:</i> значения параметров a, b, c. <i>Операции:</i> Получить данные; Определить дискриминант; Найти корни.</p>
7, 15, 23	<p>Класс 1: Маневровый локомотив. <i>Атрибуты:</i> число рельсовых путей и текущий путь; наибольшее и текущее перемещение от станции; максимальное и текущее число перевозимых вагонов; средняя скорость движения. <i>Операции:</i> Определить местоположение локомотива; Переместить в заданное место; Прицепить/Отцепить заданное число вагонов.</p> <p>Класс 2: Отрезок прямой. <i>Атрибуты:</i> координаты начальной и конечной точек. <i>Операции:</i> Получить данные; Определить длину отрезка и координаты центральной точки; Повернуть на заданный угол относительно выбранной точки.</p>
8, 16, 24	<p>Класс 1: Холодильник. <i>Атрибуты:</i> текущая температура, заданная температура и допустимое отклонение от заданной температуры, скорость охлаждения и скорость выравнивания температуры с окружающей средой. <i>Операции:</i> Измерить температуру воздуха в камере; Охлаждать в течение заданного времени; Ожидать в течение заданного времени.</p> <p>Класс 2: Квадратный многочлен ($ax^2 + bx + c$). <i>Атрибуты:</i> значения параметров a, b, c. <i>Операции:</i> Получить данные; Вычислить значение для заданного аргумента x; Сложить два многочлена.</p>
25	<p>Класс 1. Паровой котёл. <i>Атрибуты:</i> давление текущее, текущая температура, объём</p>

№ вар.	Описание классов
	<p>ём.</p> <p>Операции: добавить воды, повысить температуру (на заданную величину), понизить температуру, понизить давление, получить текущее значение давления, получить текущее значение температуры.</p> <hr/> <p>Класс2. Материальный объект</p> <p>Атрибуты: масса, объём, плотность</p> <p>Операции: Сложить два объекта (массы и объём складываются, плотность пересчитывается); проверить равенство объектов (объекты равны, когда объёмы и массы равны).</p>

4. ТРЕБОВАНИЯ К СОДЕРЖАНИЮ ОТЧЁТА

Отчёт по работе должен содержать.

1. Данные о студенте, выполнившем работу: Группа, ФИО.
2. Содержание исходного задания к работе с указанием номера задания.
3. Описание в виде таблиц атрибутов и операций классов.
4. Диаграмма классов, отображающая разработанные классы в среде Visual Paradigm.

5. КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Что понимают под объектом и классом в ООП?
2. Каково назначение диаграмм классов UML?
3. Что понимают под атрибутами и операциями на диаграммах классов?
4. Какие выделяют уровни видимости для атрибутов и операций и как они обозначаются на диаграммах классов?
5. Каков синтаксис представления атрибута на диаграмме классов?
6. Как описывается операция на диаграмме классов?
7. Каким образом записывают параметры операции на диаграмме классов?

ЛАБОРАТОРНАЯ РАБОТА №2. ОТНОШЕНИЯ МЕЖДУ КЛАССАМИ И ИХ ОБОЗНАЧЕНИЕ НА ДИАГРАММАХ КЛАССОВ UML

1. ЦЕЛЬ И ЗАДАЧИ РАБОТЫ

Целью работы является получение навыков отображения связей классов объектов через отношения UML при объектно-ориентированном моделировании структуры системы.

- Задачи работы.
- Изучить отношения, используемые в UML для отображения связей сущностей.
- Научится отображать отношения в среде Visual Studio при построении диаграмм классов.

2. ТЕОРЕТИЧЕСКИЕ ПОЛОЖЕНИЯ

Виды отношений между классами. Ассоциация, агрегация и композиция.

Кроме внутреннего устройства классов важную роль при проектировании программной системы имеют различные отношения между классами. В этих отношениях классы играют различные роли, которые определяют цель или силу взаимодействия между классами.

Выделяют следующие основные виды отношений между классами на диаграммах классов:

- ассоциация (*association*);
- агрегация (*aggregation*);
- композиция (*composition*);
- зависимость (*dependency*);
- обобщение (*generalization*);
- реализация (*realization*).

Каждое из этих отношений имеет собственное графическое обозначение в виде линии, соединяющей классы. На концах линий отношения, могут располагаться текстовые метки, которые уточняют характер связи.

Ассоциация обозначает смысловую зависимость классов и соответствует произвольному отношению между классами. Ассоциация используется, когда точное отношение между классами неизвестно.

Ассоциация изображается сплошной линией, соединяющей два класса.

Концы этой линии называются **полюсами ассоциации**. Полюса ассоциации обладают такой характеристикой, как **кратность**, которая показывает, как много экземпляров могут быть связаны отношением ассоциации на данном полюсе. Кратность полюсов ассоциации обозначается аналогично кратности атрибута класса.

Роль сущности в ассоциации отражается с помощью **имени полюса ассоциации**.

Отношение ассоциации между классами **Покупатель** и **Товар** показано на рисунке ниже (Рисунок 2.1)

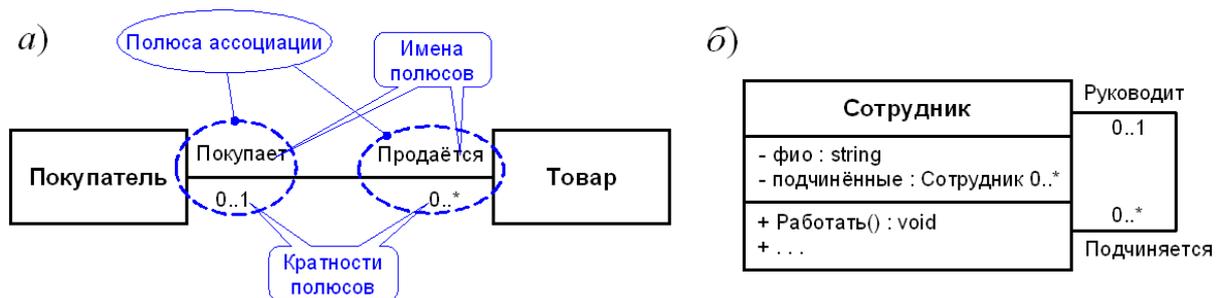


Рисунок 2.1. Отношение ассоциации: *а* – элементы ассоциации; *б* – рефлексивная ассоциация

Класс может иметь ассоциацию с самим собой, называемую **рефлексивной ассоциацией** (рис. 2.1, б).

Выделяют двунаправленную и однонаправленную ассоциацию. Например, классы рейс и самолёт связаны двунаправленной ассоциацией, а классы человек и кофейный автомат – однонаправленной.

Частным случаем отношения ассоциации является агрегация.

Агрегация задаёт отношение «часть-целое» («*has a*») между классами. При этом класс, представляющий целое, называется

агрегатом, а класс, представляющий составную часть – *компонентом*.

Агрегация описывает разбиение (декомпозицию) сложной системы на более простые составные части.

На диаграммах классов агрегация обозначается сплошной линией с ромбом на конце, соединённым с классом-агрегатом (рис. 2.2).

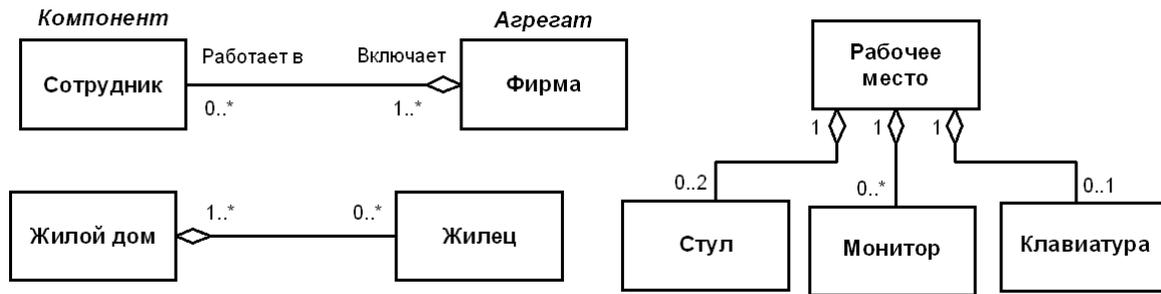


Рисунок 2.2. Примеры отношения агрегации

Время существования классов-компонентов не зависит от времени существования класса-агрегата. Если агрегат будет уничтожен, то его содержимое продолжит существовать.

Усиленной формой агрегации является *композиция*. В случае композиции агрегат и его компоненты не могут существовать отдельно. Продолжительность существования составных частей в композиции совпадает с продолжительностью существования целого.

Обозначение композиции отличается от агрегации закрашенным ромбом на конце связи (Рисунок 2.3).

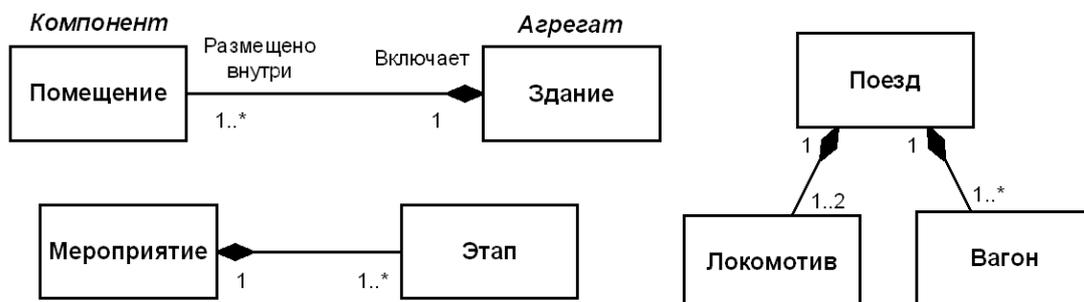


Рисунок 2.3. Примеры отношения композиции

Отношения зависимости, обобщения и реализации.

Зависимость является отношением, которое показывает, что изменение в одном классе (независимом) может влиять на другой класс (зависимый).

Графически зависимость изображается как пунктирная стрелка, направленная на класс, от которого зависят (Рисунок 2.4).

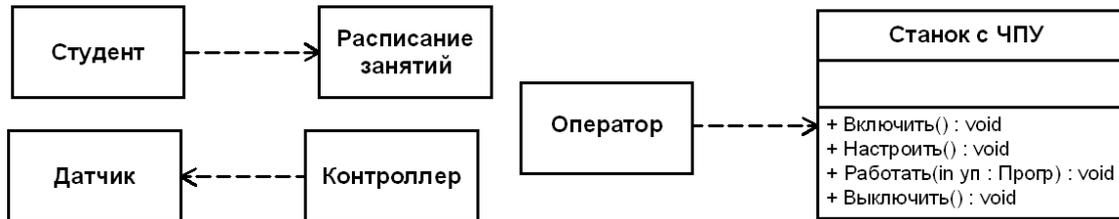


Рисунок 2.4. Примеры отношения зависимости

С помощью зависимости уточняют, какой класс является клиентом, а какой – поставщиком определенной услуги. То есть пунктирная стрелка зависимости направлена от клиента к поставщику.

Обобщение – отношение между общим классом (базовым, родительским, суперклассом) и специализированным классом (производным, дочерним, подклассом).

Обобщение задаёт отношение «общее-частное» («*is a*») между классами и показывает, что один из двух связанных классов является частным случаем более общего класса, называемого обобщением первого.

На диаграммах классов обобщение показывают в виде сплошной линии со стрелкой в форме пустого треугольника, указывающей на общий класс (Рисунок 2.5).

Класс может иметь одного или нескольких родителей и потомков либо не иметь их вовсе. Класс, не имеющий родителей, но имеющий одного или несколько потомков, называется **корневым** (*root*). Класс, не имеющий потомков, называется **листовым** (*leaf*).

Отношение обобщения в объектно-ориентированном программировании реализуется с помощью наследования.

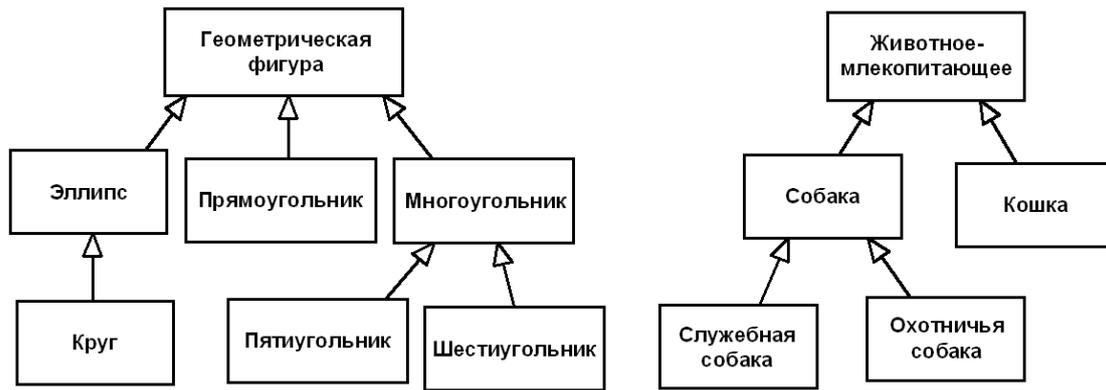


Рисунок 2.5. Примеры отношения обобщения

Реализацией в UML называют отношение между классификаторами, один из которых определяет некий контракт, а другой обязуется его выполнять. При этом одна сущность реализует поведение, заданное другой сущностью. Чаще всего реализация применяется для описания отношения между классом и интерфейсом.

По смыслу реализация представляет собой нечто среднее между зависимостью и обобщением. На диаграмме классов реализация изображается пунктирной линией с не закрашенной треугольной стрелкой, указывающей на сущность, которая определяет контракт (Рисунок 2.6, а). Кроме того, реализация может быть представлена в сокращенной форме с использованием нотации «леденец» (англ. *lollipop notation*) для предоставляемого интерфейса (Рисунок 2.6, б).

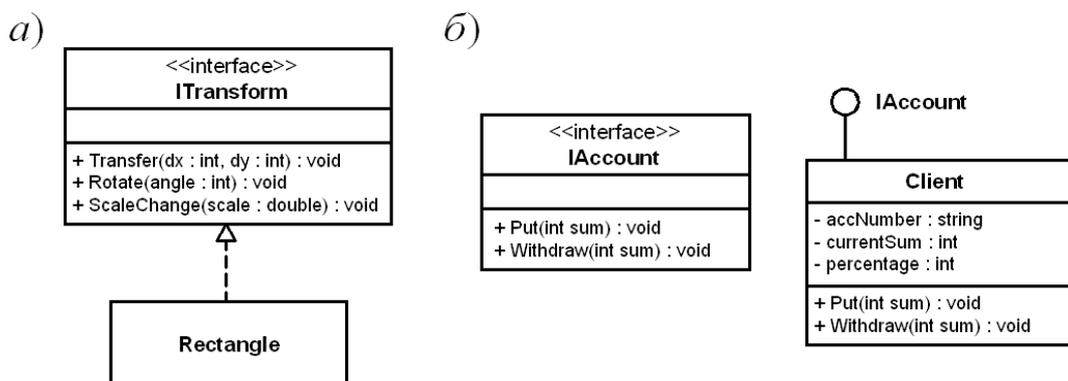


Рисунок 2.6. Примеры отношения реализации

Задание отношений между классами в Visual Paradigm.
 Отношения отображаются на панели элементов соответствующей диаграммы.

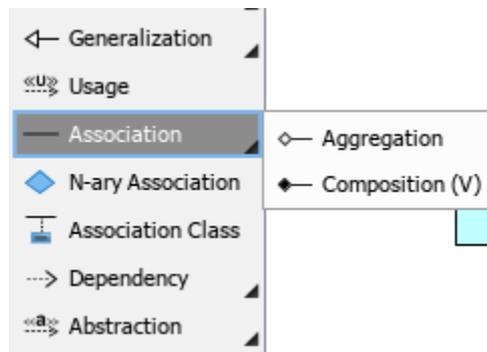


Рисунок 2.7. Отношения на панели элементов диаграммы классов

Для отношения можно настроить его свойства с помощью пункта контекстного меню Open Specification на диаграмме.

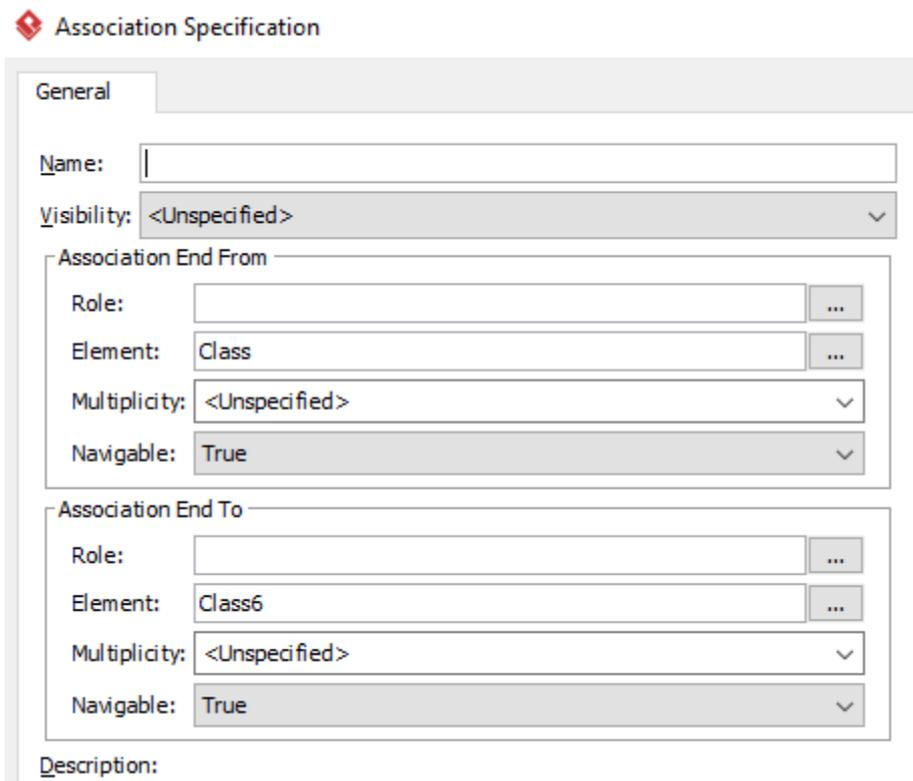


Рисунок 2.8. Свойства отношения Ассоциации

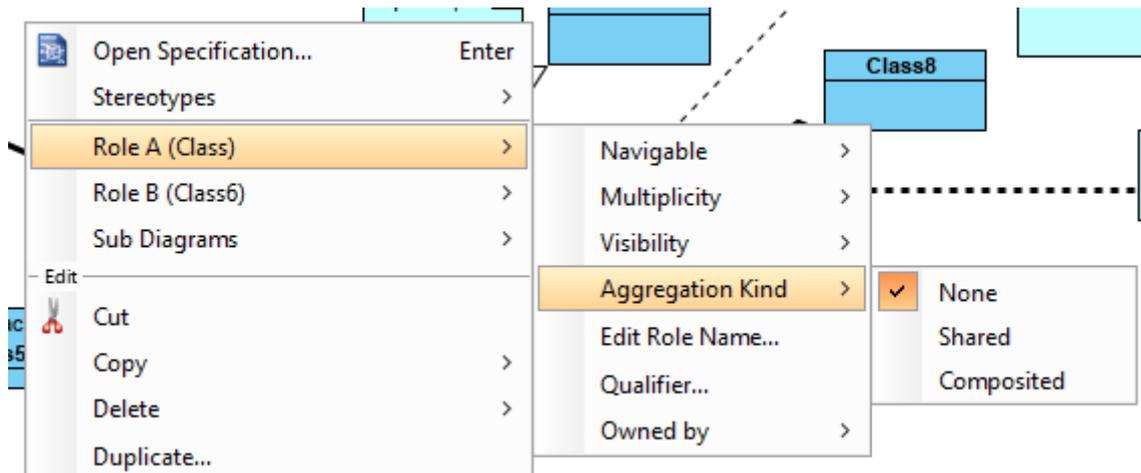


Рисунок 2.9. Свойство ассоциации

Свойства ассоциации UML (рис. 2.7), в котором можно указать имя ассоциации, роли (**Имя окончания**), вид окончания (**Агрегат**), кратность и др.

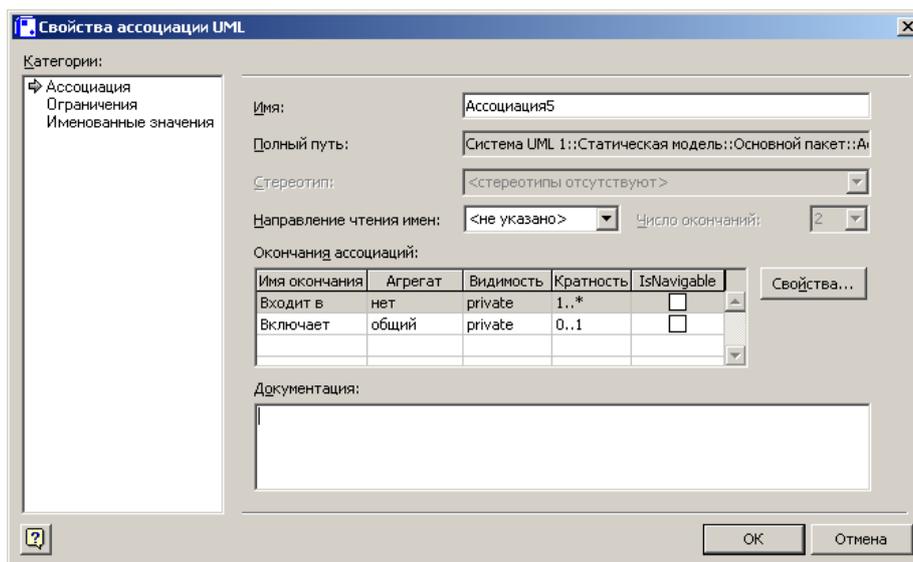


Рисунок 2.10. Окно Свойства ассоциации (категория Ассоциация)

В поле **Агрегат** задается вид окончания ассоциации путём выбора следующих вариантов:

- **нет** – прямая линия;
- **общий** – пустой ромб (агрегация);
- **составное** – закрашенный ромб (композиция).

□Пример 2.1. Описание отношений между классами на диаграмме классов UML.

Требуется описать отношения между заданными классами на диаграмме классов. В качестве рассматриваемых классов выступают:

- **Небесное тело** (название, масса, координаты X, Y, Z).
- **Звезда** (название, масса, координаты X, Y, Z, радиус, спектральный класс).
- **Планета** (название, масса, координаты X, Y, Z, экваториальный диаметр, орбитальный радиус).
- **Планетная система** (название, звезда, планеты; Добавить новую планету).
- **Звёздная система** (название, звёзды; Добавить новую звезду, Удалить звезду).

В качестве операции для указанных классов будем использовать метод **Получить_данные()**, который возвращает строку со значениями всех атрибутов экземпляра класса.

Очевидно, что класс **Небесное тело** является обобщением классов **Звезда** и **Планета**. Все атрибуты класса **Небесное тело** должны иметь модификатор доступа **# (protected)** для обеспечения доступа к ним классов-потомков.

Класс **Звезда** специализирует класс **Небесное тело** и находится в ассоциации с классом **Планета**:

Звезда 1 ↔ 0...* Планета

Кроме того, класс **Звезда** может являться компонентом для классов-агрегатов **Планетная система** и **Звёздная система**:

Звезда 1 ↔ 0...1 Планетная система

Звезда 1...* ↔ 0...1 Звёздная система

Класс **Планета** специализирует класс **Небесное тело** и находится в ассоциации с классом **Звезда**. Также класс **Планета** должен рассматриваться как компонент агрегата **Планетная система** (композиция):

Планета 1...* ↔ 1 Планетная система

Диаграмма классов, соответствующая выявленным отношениям между классами, показана на рис. 2.8.

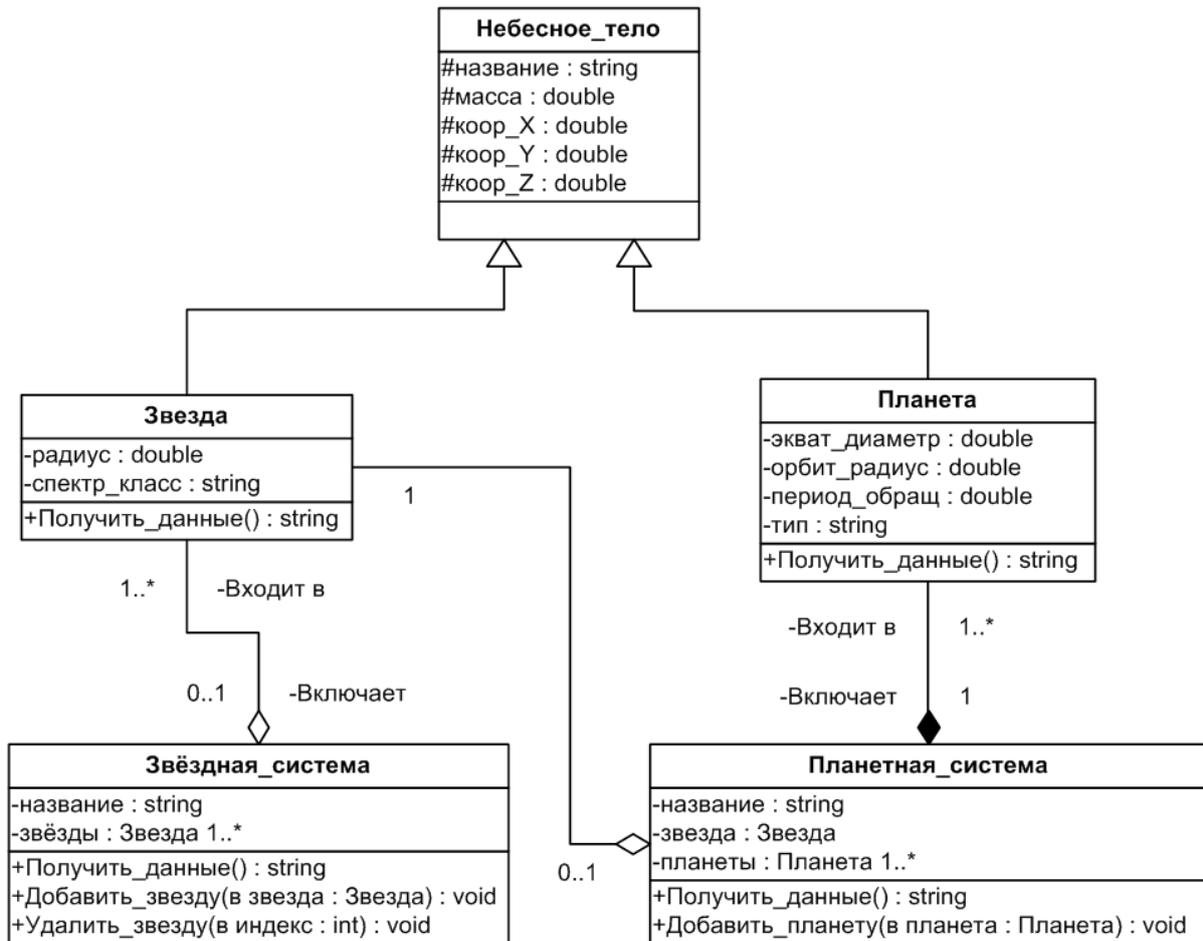


Рисунок 2.11. Диаграмма классов с заданными отношениями между классами

3. ПОРЯДОК ВЫПОЛНЕНИЯ РАБОТЫ

Данная лабораторная работа предполагает выполнение следующих этапов:

1. Изучить методические указания к лабораторной работе.
2. Описать отношения между заданными классами (табл. 2.1), используя средства языка UML. Представить с помощью MS Visio отношения между классами на диаграмме классов. Для отношения ассоциации и его разновидности требуется определить кратность и роли полюсов.
3. Оформить и защитить отчет по лабораторной работе.

Таблица 2.1.

ВАРИАНТЫ ЗАДАНИЙ

№ вар.	Описание классов
1, 13	<ul style="list-style-type: none"> • Автомобильный транспорт (регистрационный номер, марка, пробег). • Грузовой автомобиль (грузоподъёмность, тип кузова). • Автобус (тип, число мест). • Водитель (ФИО, категория, дата устройства на работу; Добавить автомобиль, Удалить автомобиль). • Автотранспортная компания (название, адрес, телефон; Добавить водителя, Удалить водителя).
2, 14	<ul style="list-style-type: none"> • Товар (наименование, размер, производитель, цена). • Обувь (цвет, материал верха). • Одежда (материал, утеплитель). • Магазин (название, адрес, телефон; Добавить обувь, Добавить одежду, Продать обувь, Продать одежду). • Покупатель (код карты клиента, ФИО, общая сумма покупок, скидка).
3, 15	<ul style="list-style-type: none"> • Человек (ФИО, пол, дата рождения). • Врач (специальность, должность, дата начала работы; Добавить обращение, Удалить обращение). • Пациент (номер страхового полиса, группа крови). • Обращение (дата, пациент, результаты анализов, диагноз). • Поликлиника (номер, адрес, телефон; Добавить пациента, Удалить пациента).
4, 16	<ul style="list-style-type: none"> • Печатное издание (код, название, число страниц). • Книга (автор, издательство). • Журнал (номер, год). • Библиотека (название, адрес, телефон; Добавить читателя, Удалить читателя). • Читатель (номер читательского билета, ФИО, выданные книги; Добавить книгу, Удалить книгу).
5, 17	<ul style="list-style-type: none"> • Человек (ФИО, пол, дата рождения). • Клиент (паспортные данные, дата заселения, число дней проживания). • Сотрудник (должность, оклад). • Гостиничный номер (номер, вместимость, вид (люкс, полулюкс, эконом и др.); Добавить клиента, Удалить клиента). • Гостиница (название, адрес, число звёзд; Добавить сотрудника, Удалить сотрудника).
6, 18	<ul style="list-style-type: none"> • Человек (ФИО, пол, дата рождения).

№ вар.	Описание классов
	<ul style="list-style-type: none"> • Студент (номер студенческого билета, группа). • Преподаватель (должность, кафедра; Добавить дисциплину, Удалить дисциплину). • Институт (название, адрес, телефон; Добавить студента, Удалить студента). • Учебная дисциплина (название, семестр, число часов, форма контроля).
7, 19	<ul style="list-style-type: none"> • Банковская услуга (дата, сумма, процент); • Кредит (вид, срок погашения). • Депозит (номер счета, вид, клиент). • Банк (название, адрес центрального офиса, телефон; Добавить клиента, Удалить клиента). • Клиент (ФИО, паспортные данные, телефон; Добавить кредит; Удалить кредит).
8, 20	<ul style="list-style-type: none"> • Человек (ФИО, пол, дата рождения). • Рабочий (должность, разряд; Добавить станок, Удалить станок). • Начальник цеха (дата устройства на работу, оклад). • Станок (наименование, модель). • Цех (номер, название; Добавить рабочего, Удалить рабочего).
9, 21	<ul style="list-style-type: none"> • Недвижимость (адрес, владелец, стоимость). • Квартира (число комнат, площадь, этаж). • Коттедж (число этажей, площадь земельного участка). • Агентство недвижимости (название, телефон, клиенты; Добавить квартиру, Удалить квартиру, Добавить коттедж, Удалить коттедж). • Клиент (ФИО, паспортные данные, телефон).
10, 22	<ul style="list-style-type: none"> • Человек (ФИО, пол, дата рождения). • Сотрудник (должность, дата устройства на работу, зарплата). • Клиент (паспортные данные, телефон). • Фирма (название, адрес, телефон; Добавить сотрудника, Удалить сотрудника, Добавить заказ, Удалить заказ). • Заказ (наименование, клиент, дата выполнения, стоимость).
11, 23	<ul style="list-style-type: none"> • Товар (код, производитель, модель, цена). • Ноутбук (размер экрана, процессор, оперативная память). • Смартфон (операционная система, ёмкость батареи, вес). • Магазин (название, адрес, телефон и др.; Добавить ноутбук, Продать ноутбук, Добавить смартфон, Продать смартфон). • Покупатель (код карты клиента, ФИО, общая сумма поку-

№ вар.	Описание классов
	пок, скидка).
12, 24	<ul style="list-style-type: none"> • Человек (ФИО, пол, дата рождения). • Спортсмен (вид спорта, рост, вес). • Тренер (звание, тренируемые спортсмены; Добавить спортсмена, Удалить спортсмена). • Соревнование (название, дата проведения, этапы). • Этап соревнования (название, время, победитель; Добавить участника, Удалить участника).

4. КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Какие выделяют виды отношений между классами на диаграмме классов?
2. Что характеризует и как обозначается отношение ассоциации на диаграмме классов?
3. Какое отношение задаёт агрегация и как она отображается на диаграмме классов?
4. В чем заключаются особенности отношения композиции?
5. Что показывает и как обозначается отношение зависимости на диаграмме классов?
6. Что задаёт и как обозначается отношение обобщения между классами?
7. Какое отношение определяют реализация?

ЛАБОРАТОРНАЯ РАБОТА №3. ОСНОВЫ РАЗРАБОТКИ КЛАССОВ В ПРИЛОЖЕНИЯХ НА ЯЗЫКЕ C#

1. ЦЕЛЬ И ЗАДАЧИ РАБОТЫ

Цель работы – приобрести умение разрабатывать собственные классы в приложениях на языке C#.

Основные задачи работы:

- освоить синтаксис описания классов и их основных элементов (полей, методов, свойств, индексаторов) на языке C#;
- научиться разрабатывать классы и создавать их экземпляры в приложениях на языке C#;
- научиться разрабатывать статические классы на языке C#.

Работа рассчитана на 4 часа.

2. ОСНОВНЫЕ ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

2.1. Синтаксис и состав класса. Поля и константы класса

Назначение и состав класса.

Объектно-ориентированное программирование построено на *классах*. Любую программную систему, построенную в объектном стиле можно рассматривать как совокупность классов, объединённых в проекты и решения.

В программной системе классы могут играть две роли:

- *класс как модуль*, представляющий архитектурную единицу программной системы; модульность построения является основным средством борьбы со сложностью системы;
- *класс как тип данных*, который определяет характеристики и поведение некоторого множества конкретных объектов этого класса, называемых *экземплярами класса*; в этой роли класс выступает в виде чертежа, по которому создаются объекты.

В хорошо спроектированной объектно-ориентированной программе классы играют обе роли.

Язык C# допускает как классы, являющиеся типами данных, так и классы, играющие единственную роль модуля. В последнем случае класс называется *статическим классом*. В библиотеке

FCL примерами статических классов являются классы **Console**, **Convert**, **Math**.

В языке C# в состав класса могут входить различные элементы (члены), которые разделяют на две группы (рис. 3.1):

- **элементы-данные** (*data members*), определяющие характеристики класса или его экземпляров (поля и константы);
- **элементы-функции** (*function members*), определяющие поведение класса или его экземпляров (методы, свойства, события и др.).

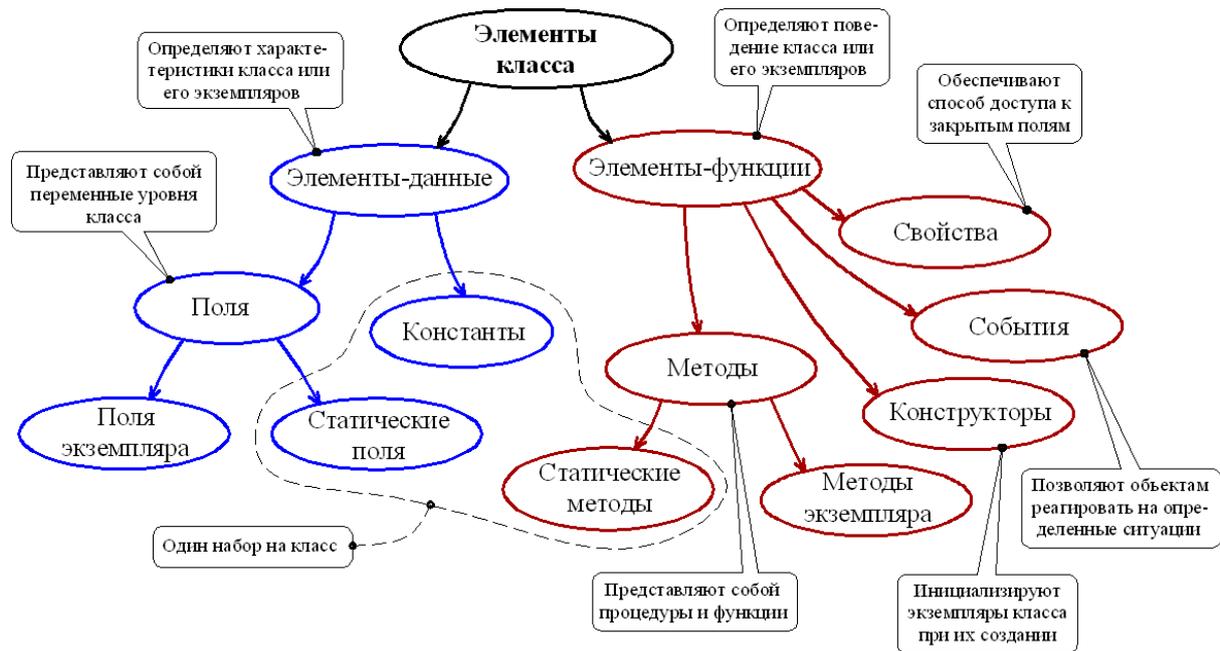


Рисунок 3.1. Основные элементы класса в языке C#

В классе могут присутствовать **статические элементы**, которые существуют в единственном числе для всех экземпляров класса.

К основным элементам класса относятся:

- **поля**, которые хранят данные класса и представляют собой переменные, объявленные на уровне класса;
- **константы**, которые хранят неизменяемые значения, связанные с классом;
- **методы**, которые реализуют действия, выполняемые классом или экземплярами;

- **конструкторы**, которые обеспечивают инициализацию экземпляров класса (присвоение начальных значений полям) при их создании;
- **свойства**, которые предоставляют способ доступа к полям класса или объекта и, по сути, являются методами для работы с полями;
- **события**, которые представляют собой автоматические уведомления от объекта о том, что произошло некоторое действие.

Кроме перечисленных элементов к элементам класса также относятся индексомеры, операции, деструкторы и вложенные типы.

Синтаксис класса. Поля и константы класса. Принцип инкапсуляции.

Объявление класса состоит из двух частей: объявление заголовка класса и объявление тела класса.

Заголовок состоит из атрибутов, модификаторов, ключевого слова **class**, имени самого класса и списка предков класса.

Тело класса представляет собой список описаний его элементов (полей, методов и др.). Этот список может быть пустым, если класс не содержит ни одного элемента. В теле класса могут быть размещены объявления других классов, называемых **вложенными классами**.

Общий синтаксис объявления класса в C# выглядит следующим образом:

```
[атрибуты]
[модификаторы] class Имя_класса [: Предки_класса]
{ тело_класса }
```

Атрибуты класса задают дополнительную декларативную информацию о классе. Эта информация используется различными классами для анализа и может влиять на ход выполнения программы.

Модификаторы класса определяют различные характеристики класса (доступность, особенности наследования и др.). Модификаторами класса могут являться ключевые слова **abstract** и **sealed**, которые используются при наследовании, слово **static**,

обозначающее статический класс, а также четыре модификатора доступа: **public**, **internal**, **private** и **protected**.

По умолчанию класс имеет модификатор доступа **internal**, который обеспечивает доступ к классу в пределах одного проекта (сборки). Модификаторы **private** и **protected** можно задать только для вложенных классов, то есть классов, объявленных внутри других классов.

Предком класса является другой класс, на основе которого строится данный класс через механизм наследования. Кроме того, предками класса могут выступать интерфейсы, реализуемые данным классом.

Описание данных класса (переменных и констант). Первыми при описании тела класса в языке С# идут элементы-данные (переменные класса и константы класса). Переменные класса могут называться *полям*.

Синтаксис описания элемента данных класса имеет следующий вид:

[атрибуты] [модификаторы] [const] Тип имя [= нач_значение];
--

Обращение к данным класса осуществляется с помощью операции доступа – точка (например `exempl.dnn`)

Справа от точки задается имя переменной или константы (`dnn`), слева – имя экземпляра класса (`exempl`).

Ключевое слово **const** используется при объявлении именованных констант (констант класса или константных полей). При этом в объявлении константы ей обязательно должно быть присвоено начальное значение.

Похожими на константы являются поля, доступные только для чтения, которые не могут быть изменены в процессе работы программы. В отличие от констант, значение такому полю может быть присвоено только в конструкторе.

Задание начальных значений для статических переменных выполняется при инициализации класса, для обычных переменных – при инициализации экземпляра.

Реализация принципа инкапсуляции. Согласно данному принципу хорошо спроектированный класс должен скрывать детали своей реализации от других классов. К деталям реализации в первую очередь относятся поля класса. Закрытие полей класса достигается их объявление с модификаторами **private** или **pro-**

tected. Заккрытие данных защищает данные объекта от нежелательного изменения в процессе работы программы. При этом доступ к данным должен осуществляться опосредованно, с применением одного из двух приёмов:

- определение пары открытых методов для чтения и записи;
- определение открытого свойства .NET.

Допускается иметь открытые константы и открытые поля, предназначенные только для чтения.

2.2. Методы класса. Конструкторы и создание экземпляров класса. Свойства и индексы

Методы класса. Конструкторы и создание экземпляров класса.

После описания элементов-данных в теле класса описывают элементы-функции, среди которых наиболее универсальными являются методы.

Методы являются функциональными элементами класса, которые реализуют действия (например, вычисления), выполняемые классом или экземпляром. Методы определяют поведение класса и представляют собой обычные процедуры и функции. Все экземпляры одного класса имеют один и тот же набор методов.

Синтаксис метода в языке C# имеет следующий вид:

```
[атрибуты]
[модификаторы] Тип_возврата Имя_метода ([параметры])
{ тело_метода }
```

Чаще всего для метода задается модификатор доступа **public**, так как методы составляют интерфейс класса.

Тип возврата определяет, значение какого типа возвращается с помощью метода. Если метод не возвращает значения, то в его заголовке указывается тип **void**.

Имя метода вместе с количеством и типами его параметров представляют собой *сигнатуру метода* – то, чем один метод отличается от других. В классе не должно быть методов с одинаковыми сигнатурами. Однако имена у методов с различными параметрами могут быть одинаковыми. Методы с одинаковыми име-

нами удобно использовать для реализации одного алгоритма для различных данных.

Использование нескольких методов с одним и тем же именем, но с различными параметрами называется *перегрузкой методов*.

Статические методы. Выделяется особый тип методов класса – статические методы, то есть методы, объявленные с модификатором `static`. Статический метод (метод класса) связан не с экземпляром класса, а непосредственно с классом, соответственно он вызывается через имя класса, обычный же метод вызывается через имя экземпляра.

Конструкторы классов. Перед тем как работать с объектом, требуется присвоить начальные значения полям этого объекта (инициализировать объект). Для этого в языке C# поддерживается механизм конструкторов.

Конструктором называется специальный метод, предназначенный для инициализации объекта в момент его создания. Данный метод вызывается автоматически при создании объекта с помощью ключевого слова **new**.

Конструкторы обладают следующими свойствами:

- в отличие от обычных методов конструктор не возвращает значение (даже **void**);
- имя конструктора должно совпадать с именем класса;
- класс может иметь несколько конструкторов с разными параметрами для разных видов инициализации объекта;
- если какие-то поля экземпляра класса не были инициализированы, то полям значимых типов присваивается ноль, полям ссылочных типов – значение **null**.

В том случае если явно конструкторы не определены, класс в C# снабжается *конструктором по умолчанию*, который не принимает аргументов. Данный конструктор инициализирует все поля объекта значениями по умолчанию. Конструктор по умолчанию может быть переопределен.

Обычно помимо конструктора по умолчанию в классах определяются дополнительные конструкторы с параметрами, предоставляющие дополнительные способы инициализации объекта.

Конструктор с параметрами, который инициализирует все поля класса, называют полным конструктором. Обычно у полного конструктора число параметров совпадает с числом полей.

В языке C# имеется ключевое слово **this**, которое обеспечивает доступ к текущему экземпляру. Одно из возможных применений ключевого слова **this** состоит в том, чтобы устранять неоднозначность, когда входной параметр назван так же, как поле класса.

Другое применение ключевого слова **this** состоит в проектировании класса, использующего *цепление конструкторов*. В этом случае в классе должно присутствовать множество конструкторов с разными наборами параметров. Из этих конструкторов выбирается конструктор с максимальным числом аргументов (главный конструктор) и в нём реализуется вся необходимая логика. Остальные конструкторы используют ключевое **this** слово для передачи входных параметров главному конструктору. При внесении изменений в поля класса придётся менять только главный конструктор, в то время как остальные конструкторы останутся в основном пустыми.

Статические конструкторы класса.

При наличии в классе статических полей в нём может быть объявлен *статический конструктор*, который предназначен для инициализации статических полей. Для статического конструктора недоступны нестатические элементы класса.

Статический конструктор вызывается общеязыковой средой выполнения (CLR) автоматически перед первым обращением к любому статическому полю класса или перед созданием экземпляра класса.

Класс может иметь только один статический конструктор. Для статического конструктора не указываются параметры. Кроме того, у статического конструктора не может быть модификаторов доступа.

Работа с классами в среде Visual Studio. В среде Visual Studio в программе (приложение) разрабатывается в виде *решения*, которое может включать один или несколько проектов. Проект, это единица разработки. Один из проектов должен быть ав-

томатически запускаемым. Состав решения отображается в окне обозревателя решений.

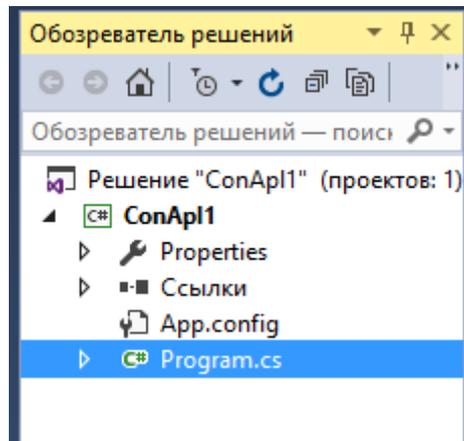


Рисунок 3.2. Окно обозревателя решений

На примере, приведённом на рисунке, решение `СonAp11` включает проект `СonAp11` разрабатываемый на языке `С#`. Проект является отдельным продуктом разработке выполняемым в среде `Visual Studio` и может включать несколько составляющих узлов. В данном случае выделяются следующие узлы:

- узел `Property`, описывает свойства проект в целом,
- `Ссылки`, задаёт компоненты, библиотеки и другие ресурсы, используемые при разработке проекта.
- `App.config`, содержит один файл, описывающий конфигурацию приложения.
- и один или несколько классов, отображаемых в виде исходный файлов описания классов на используемом языке программирования, в данном случае это файл класса `Program.cs`

По умолчанию файл класса `Program.cs` имеет следующее содержимое.

Листинг 3.1. Содержимое класса Programs

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ConAp11
{
    class Program
    {
        static void Main(string[] args)
        {

        }
    }
}

```

Файл содержит, как и другие файлы классов, директивы `using`, определяющие подключение необходимых пространств имён, описание пространства имён, соответствующего проекту, и описание класса программ. Данный класс содержит описание метода `Main(string[] args)`, автоматически запускаемого при запуске на выполнение проекта. Работа консольного приложения собственно и состоит в запуске этого метода.

Добавить класс в проект можно, просто включив его описание в файл `Program.cs`. Например, вот так.

Листинг 3.2. Включение класса в Program.cs

```

namespace ConAp11
{public class Car
    {static double t; // время моделирования
      static int num = 0; // Число экземпляров класса Car
      .....
    }
    class Program
    {
        static void Main(string[] args)
        {

        }
    }
}

```

Но более правильным, является добавление класса непосредственно в проект.

Добавление класса в проект можно выполнить через контекстное меню на узле «проект». Команда "класс", или "создать элемент". При выборе "создать элемент" необходимо из предложенных элементов выбрать "класс".

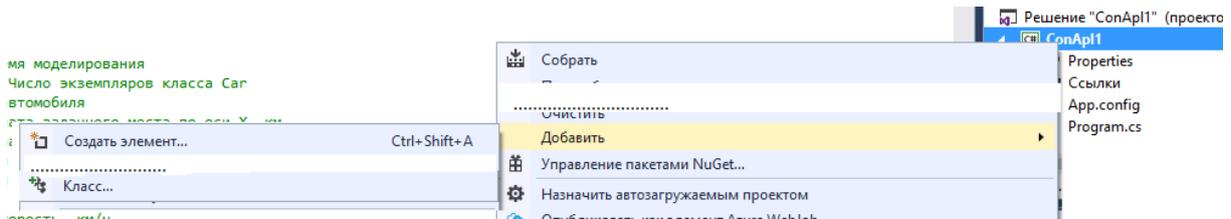


Рисунок 3.3. Добавление класса в проект

При создании класса необходимо задать его имя. При этом в проекте будет создан файл, одноимённый классу, в котором будет содержаться шаблон создания класса.

Листинг 3.3. Шаблон класса созданного в обозревателе решений

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
namespace ConApi1
{
    class avto
    {
    }
}
```

□Пример 3.1. Реализация класса на языке C#

Требуется реализовать класс **Автомобиль**, который был описан в примере 1.1, используя средства языка C#.

Для работы с классом создадим проект консольного приложения **ConsoleApplication** в составе решения **ClassAndStruct**.

Это приложение будет выводить в окно консоли данные об объектах класса **Автомобиль**.

Листинг 3.4. Пример класса

```
public class Car
{
    static int num = 0; // Число экземпляров класса Car
    string mark; // Марка автомобиля
    double ordX; // Координата заданного места по оси X, км
    double curX; // Текущая координата по оси X, км
    int maxV;
    int curV; // Текущая скорость, км/ч
    int curF; // Текущий уровень топлива в баке, л
}
```

В качестве переменных класса отображены атрибуты класса, который был определён для автомобиля в примере 1.1. Обратите внимание, что атрибут колич: `int` представлен в виде статической переменной (`static int num = 0`), то есть переменной, принадлежащей не отдельному автомобилю, а всему классу автомобилей, в классе определяется начальное значение этой переменной.

Для создания экземпляра класса «автомобиль» переопределим конструктор по умолчанию `public Car()`.

Листинг 3.5. Переопределение конструктора класса по умолчанию

```
public Car()
{
    num++;
    mark = "Газель 3212";
    ordX = 100; curX = 0;
    curV = 0;
    curF = 35;
}
```

В данном конструкторе задаются начальные значения переменным экземпляра класса `Car` и увеличивается на единицу значение статической переменной `num`, отображающей количество созданных экземпляров автомобилей.

Для того чтобы отслеживать значение переменных экземпляра класса «автомобиль», добавим ему метод, распечатываю-

щий значение переменных. Пусть этот метод обозначается как `Passport()`.

Листинг 3.6. Метод `Passport()`

```
public string Passport()
    { return string.Format("Характеристики автомобиля:\n" +
"- колич авто : {0} км/ч\n - марка: {1}\n - объём топливного бака:
{2} л\n", num, mark, maxF); }
```

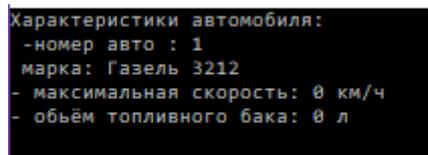
Данный метод содержит вызов метода `Format` объекта `string` из пространства имён `System`. В методе в качестве первого параметра передаётся форматная строка, в которой определяются места для выводимых данных в виде символов `{n}` (`n` – это номер выводимого данного) и задаётся вспомогательный текст. Сами выводимые данные задаются следующими параметрами.

Проверим работу методов созданного класса, добавив в метод `Main(string[] args)` класса `Program` консольного приложения создание экземпляра класса и вызов метода `passport`.

Листинг 3.7. Консольное приложение

```
{ Car car1 = new Car();
  Console.WriteLine(car1.Passport() + "\n");
  Console.Read();
}
```

Результат работы приложения будет примерно следующим



```
Характеристики автомобиля:
-номер авто : 1
марка: Газель 3212
- максимальная скорость: 0 км/ч
- объём топливного бака: 0 л
```

Рисунок 3.4. Вывод на консоль результатов работы приложения

Добавим в класс конструктор с параметрами, позволяющий инициализировать значение переменных `string mark`, `int coordX`, `int maxF`.

Листинг 3.8. Конструктор с параметрами

```
public Car(string mark, int coordX, int maxF)
{
```

```

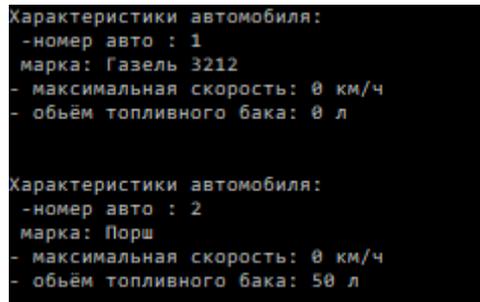
num++;
ordX = coordX;
this.mark = mark;
this.maxF = maxF;
}

```

Обратите внимание, что для переменных, имя которых совпадает с именем входного параметра, используется объект `this`, указывающий на текущий экземпляр класса.

Добавим в вызывающем методе `Main` консольного приложения создание другого экземпляра класса с помощью конструктора с параметрами и вызов метода `Passport`, созданного экземпляра.

Результат будет примерно следующим.



```

Характеристики автомобиля:
-номер авто : 1
марка: Газель 3212
- максимальная скорость: 0 км/ч
- объём топливного бака: 0 л

Характеристики автомобиля:
-номер авто : 2
марка: Порш
- максимальная скорость: 0 км/ч
- объём топливного бака: 50 л

```

Рисунок 3.5. Окно вывода консольного приложения при создании двух экземпляров класса.

Пусть требуется создать метод `SpeedIzm(int dV)`, реализующий изменение скорости на заданную величину.

Листинг 3.9. Метод изменение скорости

```

public void SpeedIzm(int dV)
{
    curV = curV+ dV;
}

```

Так же добавим метод `Move(double dT)`, реализующий движение автомобиля с постоянной скоростью, заданное время.

Листинг 3.10. Метод перемещение

```

public void Move(double dT)
{
    ordX = ordX + curV * dT;
}

```

Для отслеживания текущего значения переменных автомобиля создадим метод `GetCoordV()`, аналогичный методу `Passport()`.

Листинг 3.11. Метод получение координат

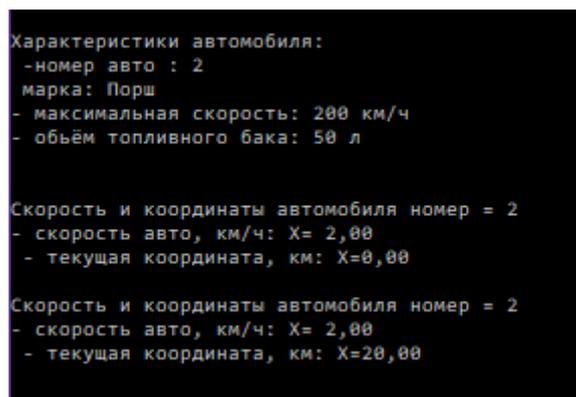
```
public string GetCoordinates()
{turn string.Format("Скорость и координаты автомобиля номер =
{0}\n" +
    "- скорость авто, км/ч: X= {1:f2} \n " +
    "- текущая координата, км: X={2:f2}",num, curV, curX);
}
```

Выполним вызов этих методов для одного из созданных экземпляров в методе `Main`.

Листинг 3.12. Содержание метода Main

```
Car car2 = new Car("Порш",100,50);
Console.WriteLine(car2.Passport() + "\n");
car2.SpeedIzm(20);
car2.Move(1);
Console.WriteLine(car2.Passport() + "\n");
Console.Read();
```

Результат работы консольного приложения с экземплярами класса **Car** показан на рис. 2.4.



```
Характеристики автомобиля:
-номер авто : 2
марка: Порш
- максимальная скорость: 200 км/ч
- объём топливного бака: 50 л

Скорость и координаты автомобиля номер = 2
- скорость авто, км/ч: X= 2,00
- текущая координата, км: X=0,00

Скорость и координаты автомобиля номер = 2
- скорость авто, км/ч: X= 2,00
- текущая координата, км: X=20,00
```

Рисунок 3.6. Данные об автомобиле 2, вызов метода до изменения скорости и выполнения движения и после

3. ПОРЯДОК ВЫПОЛНЕНИЯ РАБОТЫ

Данная практическая работа предполагает выполнение следующих этапов:

1. Создать консольное приложение.
2. Создать в консольном приложении класс в соответствии с заданием лабораторной работы 1 (класс 1). Для инициализации объектов класса переопределить конструктор по умолчанию,
3. Создать метод для вывода данных класса.
4. Создать в методе Main класса Program экземпляр класса.
5. Отобразить данные созданного экземпляра с помощью метода вывода данных класса.
6. Создать полный конструктор
7. Создать экземпляр класса с помощью полного конструктора. Отобразить данные экземпляра с помощью метода вывода данных.
8. Создать конструктор инициализирующий не все данные класса.
9. Создать экземпляр класса с помощью данного конструктора. Отобразить данные экземпляра с помощью метода вывода данных.
10. Создать методы, реализующие заданные операции.
11. Продемонстрировать их работу.

4. КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Какую роль играют классы в объектно-ориентированной программе?
2. На какие две группы разделяют элементы класса в языке С#?
3. Какие элементы класса называются статическими?
4. Каков синтаксис объявления класса в языке С#?
5. Какие модификаторы доступа можно применять к классу в языке С#?
6. Как объявляются данные класса?
7. Что такое константа класса, чем она отличается от переменной класса? Какие особенности объявления константы можно выделить?

8. В чём заключается принцип инкапсуляции? Как реализуется инкапсуляция для классов C#?

9. Что называют конструкторами, и какими свойствами они обладают?

10. Что такое решение в среде Visual Studio? Какова структура решения?

11. Как можно включить класс в проект в среде Visual Studio?

12. Зачем в рассмотренном примере используется метод "`string Passport()`". Какие данные он возвращает?

13. Какие параметры у метода "`string.Format(...)`"? Раскройте структуру первого параметра?

ЛАБОРАТОРНАЯ РАБОТА №4. ИСПОЛЬЗОВАНИЕ СВОЙСТВ

1. ЦЕЛЬ РАБОТЫ

Получить навыки использования свойств при работе с классами.

2. ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

Свойства

Свойства представляют собой специальный тип методов, который служит для организации доступа к полям класса. Как правило, свойство связано с закрытым полем класса и определяет методы его получения и установки.

В языке C# свойства имеют следующий синтаксис:

```
[атрибуты]
[модификаторы] Тип Имя_свойства
{
    [атрибуты] [модификаторы] get { код_доступа }
    [атрибуты] [модификаторы] set { код_доступа }
}
```

Обращение к свойству выглядит, как обращение к переменной класса. Метод `get` выполняется при получении значения свойства. Метод `set` выполняется при присвоении значения свойству.

Например, пусть для класса `Car` лабораторной работы 1 будет определено свойство `Velocity`.

```
Class Car {
    . . . .
    public int Velocity {get {.. ..} set { .. .. } }
    . . . .
}
```

Пусть будет создан экземпляр класса `Car`
`cr1 = new Car();`

Выражение `cr1.Velocity=45` подразумевает вызов метода `set`, определённого в свойстве `Velocity`,

Выражение `int pr1 = cr1.Velocity` подразумевает вызов метода `get`, определённого в свойстве `Velocity`, и присвоение переменной `pr1` значения, которое вернёт метод `get`.

В методе свойства `set` используется специальная величина `value` – она указывает на значение присваиваемое свойству, то есть на значение, стоящее справа от знака `'='`. В выражении `cr.Velocity=45` величина `value` получает значение `45`.

Метод `set` обычно содержит различные проверки и преобразования величины `value` при присвоении преобразованного значения переменной класса, соответствующей свойству.

Метод `get` обычно содержит преобразования значения переменной класса, соответствующей свойству и возвращение преобразованного значения инструкцией `return`.

Листинг 4.1. Пример свойства

```
public int Velocity
{ get { return curV; }
  set
    {if (value >= 0 && value <= maxV)
      curV = value;
     else if (value < 0)
      curV = 0;
     else
      curV = maxV;
    }
}
```

Пример использования данного свойства в консольном приложении.

Пусть будет создан новый экземпляр класса автомобиль `car3`. Получим значение его координат и скорости с помощью метода `GetCoordinates()` (см. лабораторную работу 3). Затем с помощью свойства производится получение его скорости и задание новой скорости.

Листинг 4.2. Использование свойств

```
Car car3 = new Car("Заз965", 100, 200, 0,30);
Console.WriteLine(car3.GetCoordinates() + "\n");
Console.WriteLine("Текущая скорость экземпляра car3 {0}
\n",car3.Velocity);
```

```

Console.WriteLine("устанавливаем новую скорость car3 м = {0}
\n", 50);
car3.Velocity = 50;
Console.WriteLine("Изменённая скорость car3 {0} \n",
car3.Velocity);
Console.Read();

```

```

Скорость и координаты автомобиля номер = 3
- скорость авто, км/ч: X= 0,00
- текущая координата, км: X=200,00

Текущая скорость экземпляра car3 0
устанавливаем новую скорость car3 м = 50
Изменённая скорость car3 0

```

Рисунок 4.1. Результат выполнения фрагмента кода.

3. ПОРЯДОК ВЫПОЛНЕНИЯ РАБОТЫ

1. Скопировать каталог приложения лабораторной работы 3 в новый каталог. Например SidorovLb4.

2. Создать свойства, реализующие операции, связанные с получением и изменением значений переменных класса.

3. Продемонстрировать в методе программ получение значений и задание значений переменных класса с помощью свойств.

4. КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Что такое свойство класса?
2. Как работает свойство?
3. Какие специальные величины могут использоваться в методе set свойства?
4. Что обычно выполняется в методе get? Когда он выполняется?
5. Когда выполняется методы set?

5. СОДЕРЖАНИЕ ОТЧЁТА

1. Отчёт должен содержать описание задания для работы.
2. Фрагменты программного кода с пояснением, реализующие заданные свойства.
3. Фрагменты кода с пояснением, реализующие выполнение программного кода демонстрирующего работу свойства.
4. Снимок экрана, демонстрирующего выполнение заданного программного кода, демонстрирующего работу свойства.

ЛАБОРАТОРНАЯ РАБОТА №5. РАБОТА С БИБЛИОТЕКАМИ КЛАССОВ В VISUAL STUDIO

1. ЦЕЛЬ РАБОТЫ

Целью работы является освоение навыков работы с библиотеками классов в среде Visual Studio.

2. ТЕОРЕТИЧЕСКИЕ ПОЛОЖЕНИЯ

При создании приложений удобно организовать многочисленные классы в специальные библиотеки классов.

Для создания в Visual Studio проекта библиотеки классов следует в окне **New Project (Создать проект)** выбрать шаблон **Class Library (Библиотека классов)** (рис. 5.1).

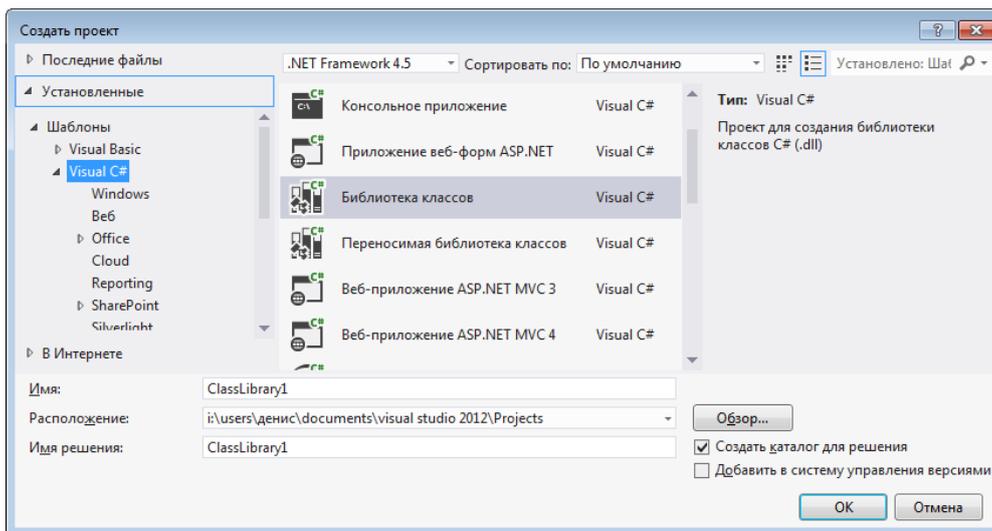


Рисунок 5.1. Окно **Создать проект**
с выбранным шаблоном **Библиотека классов C#**

Если предполагается добавить библиотеку классов в разрабатываемое решение, необходимо выбрать опцию – *добавить в решение*.

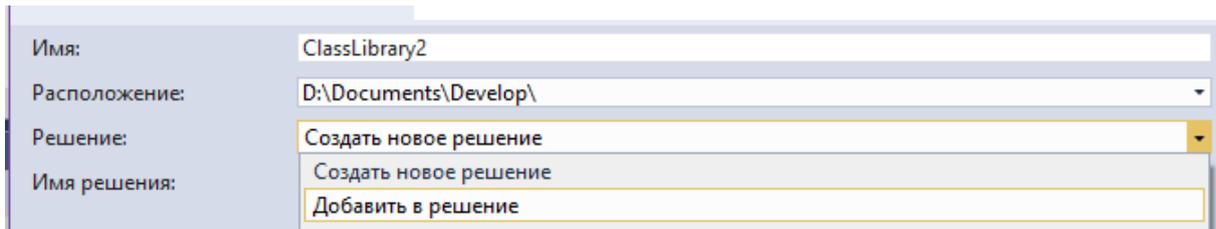


Рисунок 5.2. Включение библиотеки классов в разрабатываемое решение

В результате компиляции такого проекта будет получена сборка, называемая **DLL** (*Dynamic Link Library*) – совместно используемая библиотека. Главной особенностью DLL является то, что в них содержится код, который связывается с программами во время выполнения, а не во время компиляции. Такие сборки имеют расширение **.dll** и не могут быть непосредственно выполнены.

Для использования в проекте классов, определённых во внешней DLL, необходимо установить ссылку на соответствующий двоичный файл. Это можно сделать с помощью диалогового окна **Add Reference (Менеджер ссылок)** (Рисунок 4.1), которое открывается через пункты меню **Project | Add Reference (Проект | Добавить ссылку...)**.

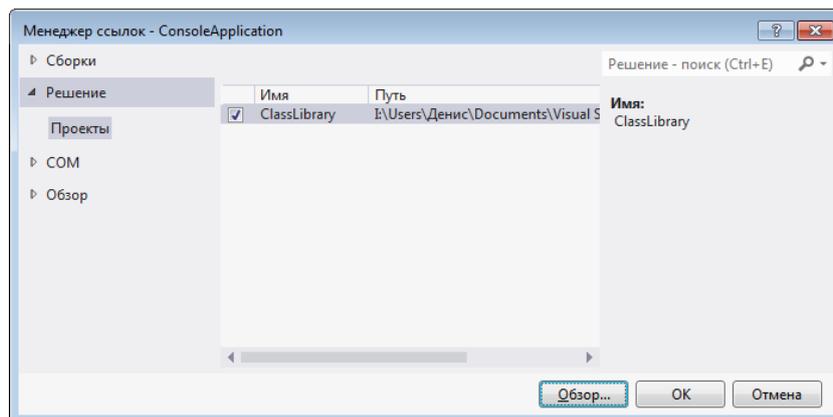


Рисунок 5.3. Окно Менеджер ссылок с выбранной внешней сборкой

Добавим в решение **ClassAndStruct** проект библиотеки классов **ClassLibrary**. В этом проекте создадим модуль **Car.cs** с описанием класса **Car** (**Автомобиль**).

После добавления ссылки необходимо в запускаемый проект, использующий разрабатываемую библиотеку, добавить директиву подключения модулей, содержащихся в библиотеке, например `using biblClass;`

3. ЗАДАНИЕ И ПОРЯДОК ВЫПОЛНЕНИЯ

Выполнить лабораторную работу 6 с использованием библиотеки классов.

4. КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Что такое библиотека классов?
2. Каким образом создаётся библиотека классов?
3. Каким образом подключается библиотека классов?

ЛАБОРАТОРНАЯ РАБОТА №6. СТАТИЧЕСКИЕ КЛАССЫ

1. ТЕОРЕТИЧЕСКИЕ ПОЛОЖЕНИЯ

Статическим классом в языке C# является класс, объявленный с модификатором **static** и выполняющий единственную роль – роль модуля. Статические классы применяют для логического объединения функционально близких статических методов.

Статический класс не может рассматриваться как тип данных.

Статический класс обладает следующими основными особенностями:

- Все элементы статического класса (поля, методы, свойства и события), кроме констант и вложенных классов, должны быть объявлены с модификатором **static**, то есть являются статическими.

- Экземпляр (объект) статического класса нельзя создать при помощи операции **new**. При выполнении программы автоматически создаётся единственный объект статического класса, имя которого совпадает с именем класса.

- Статический класс может иметь только статический конструктор. Конструкторы экземпляров для статического класса не нужны, так как экземпляры статического класса не создаются.

- Статические классы не могут служить базовыми классами при наследовании. Статический класс не может иметь класса-предка. Статический класс имеет только один базовый класс **System.Object**.

Классы, которые имеют только статическую функциональность, часто называют *сервисными (утилитными) классами*. Такие классы предоставляют свои сервисы классам-клиентам. Например, методы класса **Math** позволяют другим классам вычислять математические функции, а методы класса **Convert** позволяют осуществлять преобразование одного типа данных в другой, не принадлежа ни одному из этих типов.

Пример 6.1. Разработка статического класса на языке C#.

Требуется разработать статический класс **DecBinConverter**, который выполняет преобразование чисел из десятичной системы счисления в двоичную систему и обратно. Функциональность данного класса будет использоваться в классе **Program** консольного приложения.

Определим в классе следующие статические элементы:

- `binBase = 2` – основание двоичной системы счисления;
- `ConvertBinToDec(binNumber : string) : int` – переводит число `binNumber` из двоичной системы в десятичную;
- `ConvertDecToBin (decNumber : int) : string` – перевести из десятичной системы в двоичную.

Представление класса `DecBinConverter` на диаграмме классов UML показано на рис. 6.1.

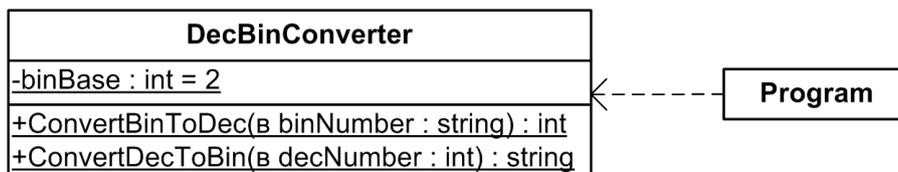


Рисунок 6.1. Диаграмма классов

*Листинг 6.1. Исходный код класса **DecBinConverter***

```

public static class DecBinConverter
{
    const int binBase = 2;
    public static string ConvertDecToBin(int decNumber)
    {
        string binString = "";
        while (decNumber > 0)
        { binString = Convert.ToString(decNumber % binBase) +
binString;
            decNumber /= binBase;
        }
        return binString;
    }
    public static int ConvertBinToDec(int binNumber)
    { int decNumber = 0; // формируемое десятичное число
        string binString = Convert.ToString(binNumber);
        int n = binString.Length; //число цифр в строке bin-
String
        string binS = "";
        for (int i = 0; i < n; i++)
        { binS = Convert.ToString(binString[i]);
  
```

```

        decNumber += Convert.ToInt32(Convert.ToByte(binS) *
Math.Pow(binBase, n - i - 1));
    }
    return decNumber;
}
}

```

Исходный код метода консольного приложения, в котором вызываются статические методы класса **DecBinConverter**, приведён в листинге.

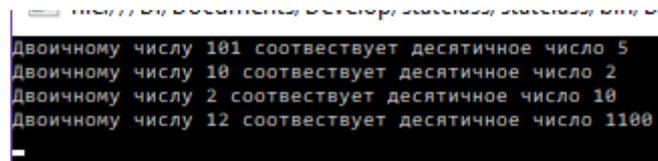
Листинг 6.2. Исходный код консольного приложения

```

int binNum=101;
    Console.WriteLine(String.Format("Двоичному числу {0} со-
ответствует десятичное число {1}", binNum,
DecBinConverter.ConvertBinToDec(binNum)));
    binNum = 10;
    Console.WriteLine(String.Format("Двоичному числу {0} со-
ответствует десятичное число {1}", binNum,
DecBinConverter.ConvertBinToDec(binNum)));
    binNum = 2;
    Console.WriteLine(String.Format("Двоичному числу {0} со-
ответствует десятичное число {1}", binNum,
DecBinConverter.ConvertDecToBin(binNum)));
    binNum = 12;
    Console.WriteLine(String.Format("Двоичному числу {0} со-
ответствует десятичное число {1}", binNum,
DecBinConverter.ConvertDecToBin(binNum)));
    Console.ReadKey();

```

Результат работы консольного приложения показан на рис. 6.2. □



```

Двоичному числу 101 соответствует десятичное число 5
Двоичному числу 10 соответствует десятичное число 2
Двоичному числу 2 соответствует десятичное число 10
Двоичному числу 12 соответствует десятичное число 1100

```

Рисунок 6.2. Результат работы консольного приложения

2. ПОРЯДОК ВЫПОЛНЕНИЯ РАБОТЫ

Данная практическая работа предполагает выполнение следующих этапов:

1. Разработать заданный статический класс (табл. 6.1). Изобразить заданный класс на диаграмме классов UML. Разместить код статического класса в библиотеке классов.

2. Добавить в решение с библиотекой классов консольное приложение, демонстрирующее работу с полученными классами.

3. Оформить и защитить отчет по лабораторной работе.

Таблица 6.1.

Данные для разработки статического класса

Double P1=0.2;

Double X2=4.5;

Double P2=0.2;

.....

```
Public double matOg(double x1, double x2,///)
{return=
}
```

```
Public double disp(double x1, double x2,///)
{ double matOQ= matOg(X1*X1,x2*x2...)
  QmatO= matOg(X1,x2)* matOg(X1,x2)
```

```
return=
}
```

№ вар.	Элементы статического класса	Пояснения
1, 7, 13, 19	<p>Статистический калькулятор.</p> <ul style="list-style-type: none"> • Значения дискретной случайной величины (ДСВ) • Определить математическое ожидание ДСВ для заданных вероятностей • Определить дисперсию ДСВ • Определить среднеквадратическое отклонение (СКО) 	<p>Формулы:</p> <p>Математическое ожидание:</p> $M(X) = \sum_{i=1}^n p_i x_i$ <p>Дисперсия ДСВ:</p> $D(X) = M(X^2) - [M(X)]^2$ <p>СКО:</p> $\sigma(X) = \sqrt{D(X)}$
2, 8, 14, 20	<p>Преобразователь единиц длины.</p> <ul style="list-style-type: none"> • Единицы измерения в м 	<p>Соотношение единиц:</p> <p>1 дюйм = 0,0254 м</p>

№ вар.	Элементы статического класса	Пояснения
	<ul style="list-style-type: none"> • Перевести из дюймов в сантиметры • Перевести из километров в морские мили • Перевести из футов в метры 	1 миля морская = 1852 м 1 фут = 0,3048 м
3, 9, 15, 21	Калькулятор времени. <ul style="list-style-type: none"> • Определить возраст по дате рождения • Определить число дней между датами • Определить дату по числу дней назад/вперёд 	Использовать класс DateTime:
4, 10, 16, 22	Преобразователь единиц веса. <ul style="list-style-type: none"> • Единицы измерения в кг • Перевести из килограммов в фунты • Перевести из аптекарских унций в граммы • Перевести из килограммов в слэги 	Соотношение единиц: 1 фунт = 0,4536 кг 1 унция аптекарская = 31,1035 · 10 ⁻³ кг 1 слэг = 14,5939 кг
5, 11, 17, 23	Валютный калькулятор. <ul style="list-style-type: none"> • Курсы валют по отношению к рублю • Перевести из рублей в доллары • Перевести из долларов в евро • Перевести из юаней в рубли 	Курсы валют (22.02.2015): 1 доллар = 61,87 руб. 1 евро = 70,44 руб. 1 юань = 9,89 руб.
6, 12, 18, 24	Преобразователь единиц объёма. <ul style="list-style-type: none"> • Единицы измерения в м³ • Перевести из метров кубических в нефтяные баррели • Перевести из галлонов (Великобр.) в литры • Перевести из метров кубических в бушели (США) 	Соотношение единиц: 1 баррель нефтяной = 0,158988 м ³ 1 бушель (США) = 3,52393 · 10 ⁻² м ³ 1 галлон (Великобр.) = 4,54609 · 10 ⁻³ м ³

3. КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Какие элементы класса называются статическими?
2. Каков синтаксис объявления статического класса в языке

C#?

3. Можно ли создавать экземпляры статического класса?
4. Как используется статический класс?
5. Что называют конструкторами, и какими свойствами они обладают?
6. Для чего предназначены свойства в языке C#?
7. Что понимают под совместно используемой библиотекой DLL?
8. Для чего предназначен статический конструктор?
9. Каковы основные особенности статических классов?

ЛАБОРАТОРНАЯ РАБОТА №7. РАБОТА СО СТРУКТУРАМИ И ПЕРЕЧИСЛЕНИЯМИ НА ЯЗЫКЕ C#

1. ЦЕЛЬ И ЗАДАЧИ РАБОТЫ

Цель работы – приобрести умение работать со структурами и перечислениями, а также выполнять перегрузку операций на языке C#.

Основные задачи работы:

- освоить работу со структурами;
- научиться применять перегрузку операций;
- научиться использовать перечисления.

Работа рассчитана на 2 часа.

2. ОСНОВНЫЕ ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

2.1. Особенности типов-значений и типов-ссылок. Структуры. Перегрузка операций

Особенности типов-значений и типов-ссылок.

При выполнении программы все её данные хранятся в оперативной памяти компьютера. Для хранения данных программы используются два участка оперативной памяти, которые называются стеком и хипом.

Стек (англ. *stack*) является линейным участком памяти, который работает по принципу «последним пришёл – первым вышел» (*Last In First Out – LIFO*). Последний записанный в стек элемент данных будет извлечён из стека первым. Данные могут добавляться только в вершину стека и удаляться из вершины. Добавление и удаление данных из произвольного места стека невозможно.

Операции по добавлению и удалению элементов из стека выполняются очень быстро. Обычно размер стека является небольшим и время хранения в нём данных зависит от времени жизни переменной.

Хип (куча, динамическая область памяти, англ. *heap*) – это область оперативной памяти, в разных частях которой могут вы-

деляться участки для хранения данных. В отличие от стека данные в хипе могут выделяться и освобождаться в любом порядке.

Для удаления данных из хипа используется компонент среды CLR, называемый сборщиком мусора (*garbage collector*).

В языке C# различают две категории типов данных: *типы-значения* (*значащие типы* – англ. *value types*) и *типы-ссылки* (*ссылочные типы* – англ. *reference types*). Основное различие между ними состоит в том, что типы-значения хранят непосредственно свои значения в стеке, а типы-указатели хранят в стеке ссылки на свои значения, которые размещены в хипе.

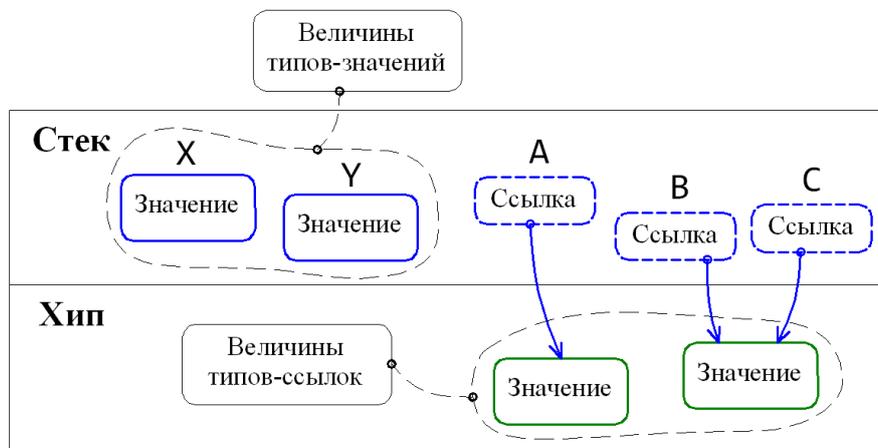


Рисунок 7.1. Размещение типов-значений и типов-ссылок в оперативной памяти

К значащим типам в языке C# относятся *структуры* и *перечисления*. Ссылочными типами в C# являются рассмотренные ранее классы, интерфейсы и делегаты.

Одни и те же действия над величинами значащего и ссылочного типов выполняются по-разному. Величины значащих типов равны, когда равны их значения. Величины ссылочных типов равны, если они ссылаются на одни и те же данные.

Каждый значащий и ссылочный тип в C# неявно наследуют класс **Object**. Поэтому каждый объект в языке C# получает метод **ToString()**, который возвращает строковое представление данного объекта.

Переопределить метод **ToString()** можно при помощи следующего объявления метода:

```
public override string ToString()
{ ... }
```

Понятие и синтаксис структуры.

Структура (*structure*) – тип, аналогичный классу, но имеющий ряд важных отличий от него:

- структура не может участвовать в иерархиях наследования; она может наследовать только интерфейсы;
- структура не может быть статической, то есть объявленной с модификатором **static**;
- в структуре запрещено определять конструктор по умолчанию, поскольку он определен неявно;
- экземпляры структуры можно создавать без использования ключевого слова **new**;
- для структур нельзя объявлять деструктор, поскольку это бессмысленно (структура – значащий тип).

Любой значащий тип C# (например, **int**, **double**, **bool**, **char**) является структурой.

Синтаксис объявления структуры имеет следующий вид:

```
[атрибуты]
[модификаторы] struct Имя_Структуры [: Интер-
фейсы]
{ тело_структуры }
```

Модификаторы структуры имеют такой же смысл, как и для класса, причем из модификаторов доступа можно использовать только **public**, **internal** и **private**. Из-за отсутствия механизма наследования для структур не применимы модификаторы **abstract** и **sealed**.

Для структур не может быть задан родительский класс или родительская структура, у структуры не может быть наследников. Структура может наследовать один или несколько интерфейсов.

Тело структуры может состоять из полей, констант, конструкторов, методов, свойств, операций и вложенных типов. Правила их описания и использования аналогичны соответству-

Ющим элементам классов, за исключением следующих особенностей:

- поскольку структуры не могут участвовать в иерархиях наследования, то для их элементов нельзя использовать модификаторы **protected** и **protected internal**;
- методы структур не могут быть абстрактными и виртуальными (то есть не могут иметь модификаторы **abstract** и **virtual**);
- переопределяться (то есть описываться с модификатором **override**) могут только методы, унаследованные от базового класса **object**;
- для полей структуры нельзя задавать значения по умолчанию (это ограничение не относится к статическим полям).

При выборе между классом и структурой можно руководствоваться следующими правилами:

- если требуется отнести класс к значимому типу, то его следует сделать структурой;
- если у класса число полей относительно невелико, а число возможных объектов относительно велико, то используют структуры; в этом случае память используется более эффективно (объекты размещаются только в стеке, не создаются лишние ссылки).

С помощью структур можно отобразить действия с некоторыми сложными объектами. Например с комплексными числами (состоит из действительной и мнимой части).

Листинг 7.1. Пример структуры

```
public struct Complex
{
    double re; // действительная часть
    double im; // мнимая часть
    ...
}
```

Для создания экземпляра комплексного числа необходимо объявить конструктор. Например, такой

Листинг 7.2. Конструктор структуры

```
public Complex(double rre, double iim) // конструктор с параметрами.
{
    this.re = rre;
```

```

        this.im = iim;
    }

```

Для доступа к действительной и мнимой части пусть будут использованы свойства `Real` (- для доступа к действительной части) и `Image` (- для доступа к мнимой части).

Описание свойства `Real` будет выглядеть следующим образом.

Листинг 7.3. Пример свойства используемого в структуре

```

public double Real
{
    get { return re; }
    set { re = value;}
}

```

Метод `get` просто возвращает значение действительной части, а метод `set` устанавливает в качестве значения действительной части величину, присваиваемую свойству. Свойство `Image` имеет аналогичное содержание, но производится работа с мнимой частью `im`.

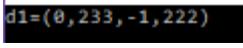
Протестируем работу со структурой в консольном приложении. Пусть будет создан экземпляр структуры и выведены с помощью свойств значение его действительной и мнимой части.

Листинг 7.4. Создание комплексного числа и обращение к его свойствам

```

Complex d1 = new Complex(0.2333, -1.2222);
Console.WriteLine("d1=({0:f3},{1:f3})", d1.Real, d1.Image);
Console.ReadKey();

```

Результат будет следующим 

Для реализации работы со структурами необходимо определить основные методы, соответствующие базовым операциям. Например, сложение комплексных чисел, вычитание, произведение.

Сложение комплексных чисел подразумевает сложение их действительных и мнимых частей.

Реализуем сложение с помощью статического метода, то есть метода, относящегося не к экземплярам, а ко всей структуре. Данный метод должен возвращать значение типа `Complex` и в ка-

честве параметров принимать два комплексных числа (суммируемые числа), то есть сигнатура метода должна иметь вид –

```
public static Complex Summ(Complex Ch1,Complex Ch2).
```

В теле метода с помощью конструктора должен создаваться новый экземпляр структуры Complex, в качестве значений действительной и мнимой части должны приниматься соответственно сумма действительных и мнимых складываемых комплексных чисел.

```
{Complex Ch= new Complex(Ch1.Real + Ch2.Real, Ch1.Image +
Ch2.Image);
return Ch; }
```

Можно обойтись и без переменной ch, сразу подставив в качестве возвращаемого значения - new Complex(Ch1.Real + Ch2.Real, Ch1.Image + Ch2.Image).

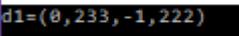
Тогда тело метода будет выглядеть следующим образом.

```
{return new Complex(Ch1.Real + Ch2.Real, Ch1.Image + Ch2.Image; }
```

Протестируем работу метода в консольном приложении

Листинг 7.5. Тестирование консольного приложения

```
Complex d1 = new Complex(0.233, -1.222);
Complex d2 = new Complex(-5.233, 1.333);
Console.WriteLine("d1={({0:f3},{1:f3}},d2={({2:f3},{3:f3}})",
d1.Real, d1.Image, d2.Real, d2.Image);
Complex d3 =Complex.Summ(d1, d2);
Console.WriteLine("d1+d2={({0:f3},{1:f3}})", d3.Real, d3.Image);
Console.ReadKey();
```

Результат выполнения будет следующим 

Работа с методами рассматриваемой структуры была бы более удобно, если бы использовались стандартные обозначения для операций. Например, была бы удобна и наглядна запись сложения комплексных чисел в виде операции «+». То есть чтобы при записи d3=d1+d2, где d1, d2, d3 – экземпляры структуры Complex, вызывался метод Summ. Это может быть реализовано через *перегрузку операций*.

Перегрузка операций.

Язык C# позволяет придавать другой смысл стандартным операциям («+», «-», «*», «==» и др.) при работе с типами, определяемыми пользователем.

Определение операций для пользовательских типов называют *перегрузкой операций*. Перегрузка обычно применяется для типов, описывающих математические или физические понятия, то есть типов, для которых использование операций делает программу более понятной.

Операции описываются с помощью специальных методов (функций-операций), содержащих в заголовке ключевое слово **operator**. Синтаксис перегрузки операции имеет следующий вид:

```
[атрибуты]
public static [тип_возвр] operator опера-
ция([параметры])
{ тело операции }
```

Использование модификаторов **public** и **static** при перегрузке операций является стандартным, то есть операция должна быть описана только как открытый статический метод класса.

Тело операции определяет действия, которые выполняются при использовании операции в выражении, и представляет собой блок, аналогичный телу других методов.

Обозначение перегружаемой операции выбирается из набора стандартных операций. Новые обозначения для собственных операций вводить нельзя.

В списке параметров хотя бы один из параметров должен относиться к типу, в котором определяется данная операция. Параметры в операцию требуется передавать только по значению (то есть параметры не должны предваряться ключевыми словами **ref** и **out**).

Операции «++» (инкремент) «--» (декремент) перегружаются парами. Операции сравнения также перегружаются парами. Если перегружается операция «==», то должна быть перегружена операция «!=».

К операциям, не подлежащим перегрузке, относятся «=», «.», «->», **new**, **is**, **sizeof**, **typeof**.

Пусть будет перегружена операция "+" для суммирования комплексных чисел, её реализация будет совпадать с методом Summ.

Листинг 7.6. Перегрузка операции «+»

```
public static Complex operator +(Complex Ch1, Complex Ch2)
{
    return new Complex(Ch1.Real + Ch2.Real, Ch1.Image + Ch2.Image);
}
```

Протестируем работу перегруженной операции «+»

Листинг 7.7. Тестирование перегруженной операции

```
Complex d4 = d1 + d2;
Console.WriteLine("d1+d2=({0:f3},{1:f3})", d4.Real,
d4.Image);
Console.ReadKey();
```

Результат выполнения данного фрагмента кода будет следующим

3. ПОРЯДОК ВЫПОЛНЕНИЯ РАБОТЫ

Данная практическая работа предполагает выполнение следующих этапов:

1. Изучить методические указания к практической работе.
2. Добавить в библиотеку классов структуру, соответствующую 2 заданию из лабораторной работы 1 (табл. 1.1, класс 2).
3. Реализовать структуру средствами языка C#. Перегрузить две заданные операции (табл. 7.1) для работы с экземплярами структуры. Разработать консольное приложение, демонстрирующее работу со структурой и её экземплярами.
4. Оформить и защитить отчет по лабораторной работе.

Таблица 7.1. Варианты заданий для перегрузки операций

№ вар.	Операция	Описание
1, 9, 17	==	Равенство двух треугольников. Возвращает значение true , если длины сторон одного треугольника попарно равны длинам сторон другого треугольника, в противном случае возвращает false

	!=	Неравенство двух треугольников. Возвращает значение true , если длины сторон одного треугольника попарно не равны длинам сторон другого треугольника, в противном случае возвращает false
2, 10, 18	+	Сложение двух векторов. Возвращает вектор, компоненты которого равны сумме соответствующих компонент двух исходных векторов
	*	Умножение вектора на число. Возвращает вектор, все компоненты которого умножены на заданное число.
3, 11, 19	==	Равенство двух кругов. Возвращает значение true , если радиус одного круга равен радиусу другого круга, в противном случае возвращает false
	!=	Неравенство двух кругов. Возвращает значение true , если радиус одного круга не равен радиусу другого круга, в противном случае возвращает false
4, 12, 20	+	Сложение двух квадратных многочленов. Возвращает квадратный многочлен, компоненты которого a , b , c равны сумме соответствующих компонент двух исходных многочленов
	-	Вычитание одного квадратного многочлена из другого. Возвращает квадратный многочлен, компоненты которого a , b , c равны разности соответствующих компонент двух исходных многочленов
5, 13, 21	==	Равенство двух зарядов. Возвращает значение true , величина зарядов равна и они находятся на одинаковом расстоянии от начала координат.
	!=	Неравенство двух зарядов. Возвращает значение true , величина зарядов различная.
6, 14, 22	+	Сложение двух рациональных чисел. Возвращает рациональное число, числитель и знаменатель которого вычисляются по правилам сложения рациональных чисел
	*	Умножение двух рациональных чисел. Возвращает рациональное число, числитель и знаменатель которого соответственно равны произведению числителей и знаменателей исходных чисел
7	==	Равенство, когда длины отрезков равны
		Не равенство отрезков – когда их длины не равны
7, 15, 23	==	Равенство двух прямоугольников. Возвращает значение true , если ширина и высота одного пря-

		моугольника совпадают с шириной и высотой другого прямоугольника, в противном случае возвращает false
	!=	Неравенство двух прямоугольников. Возвращает значение true , если ширина и высота одного прямоугольника не равны длине и ширине другого прямоугольника; иначе возвращает false
8, 16, 24	+	Сложение двух квадратных уравнений. Возвращает квадратное уравнение, компоненты которого <i>a</i> , <i>b</i> , <i>c</i> равны сумме соответствующих компонент двух исходных уравнений
	-	Вычитание одного квадратного уравнения из другого. Возвращает квадратное уравнение, компоненты которого <i>a</i> , <i>b</i> , <i>c</i> равны разности соответствующих компонент двух исходных уравнений

4. КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Какие области оперативной памяти называют стеком и хипом?
2. В чём заключается различие между типами-значениями и типами-ссылками?
3. Чем структура отличается от класса?
4. Каков синтаксис объявления структуры в C#?
5. В каких случаях следует создавать структуру вместо класса?
6. Что такое «Перегрузка операций»?
7. Как в языке C# задаётся перегрузка операции?

ЛАБОРАТОРНАЯ РАБОТА №8. РАЗРАБОТКА СЕМЕЙСТВА КЛАССОВ С ИСПОЛЬЗОВАНИЕМ КОМПОЗИЦИИ И НАСЛЕДОВАНИЯ НА ЯЗЫКЕ C#

1. ЦЕЛЬ И ЗАДАЧИ РАБОТЫ

Цель работы – приобрести умение строить семейства классов, связанных отношениями наследования и композиции, в приложениях на языке C#.

Основные задачи работы:

- освоить особенности композиции и наследования в языке C#;
- научиться использовать наследование и композицию для создания семейств классов в приложениях на языке C#.

Работа рассчитана на 4 часа.

2. ОСНОВНЫЕ ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

2.1. Отношения между классами. Реализация отношения композиции на C#.

Отношения наследования и композиции между классами C#.

Классы программной системы находятся в определенных отношениях друг с другом.

В языке C# наиболее важными видами отношений между классами являются отношения наследования (*inheritance*) и композиции (*composition*).

Данные отношения в языке C# отображают отношения UML соответственно «обобщения» и «агрегации» (см. лабораторная работа 2). Однако надо понимать, что отношения UML это отношения уровня моделей, а отношения, вводимые в языке программирования, это отношения уровня реализации, они имеют другой уровень абстракции.

Классы **A** и **B** находятся в отношении **наследования** (**предок-наследник**), если при объявлении класса **A** класс **B** указан в

качестве предка (то есть класс *A* наследует элементы класса *B*). При этом класс *B* называют **базовым классом** (родительским классом, классом предком, суперклассом) а класс *A* – **производным классом** (дочерним классом, классом потомком, подклассом).

С помощью наследования строятся иерархии классов (рис. 4.1). Классы, расположенные ближе к началу иерархии, объединяют общие черты всех нижележащих классов.

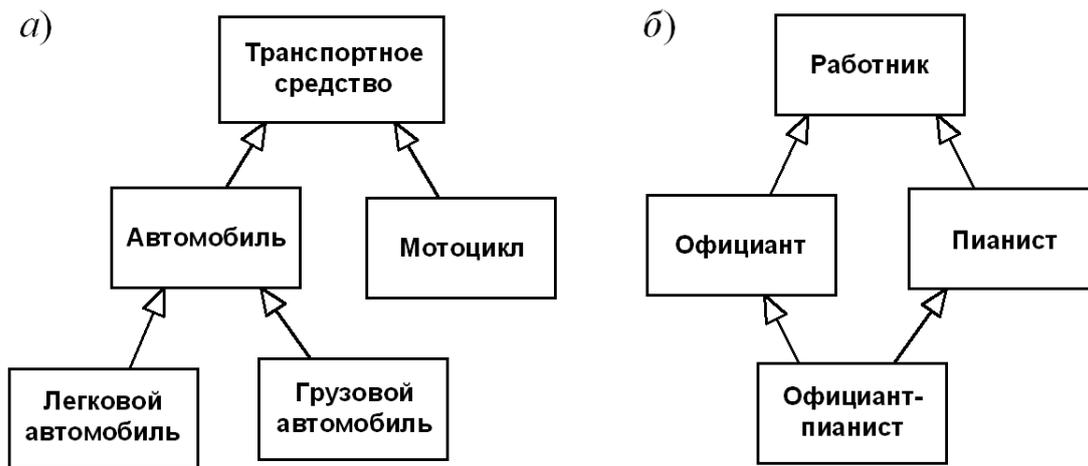


Рисунок 8.1. Иерархии классов на основе наследования: *a* – с одиночным наследованием; *б* – с множественным наследованием

Отношение композиции. Реализует отношение UML агрегации и композиции. Классы *C* и *P* находятся в отношении **композиции** (включения, делегации, клиент-поставщик), если в классе *C* создаются объекты класса *P* (поля или локальные переменные) или вызываются статические элементы (методы или поля) класса *P*.

При этом класс *C* называется **клиентом** (*client*) класса *P*, а класс *P* – **поставщиком** (*provider*) класса *C*.

Случай, когда в классе клиенте создаются объекты класса поставщика реализует отношение композиции UML (экземпляр класса-поставщика не может существовать без экземпляра класса-клиента).

Когда объекты методы или поля класса поставщика только вызываются реализует отношение агрегации UML (экземпляр класса поставщика создаётся вне экземпляра класса-клиента).

Отношение композиции может рассматриваться как отношение «имеет» («*has a*»), содержит («*contains*») или «является частью» («*part of*»). Например, в отношении **включения** находятся классы **Точка** и **Отрезок**. В классе **Отрезок** могут быть сделаны ссылки на экземпляры класса **Точка**, представляющие концы отрезка.

Отношение композиции является динамическим, то есть может быть определено на этапе выполнения программы.

Реализация отношения композиции на С#. Определение вложенных типов.

Отношение композиции (включения) между классами А и В на языке С# обеспечивается (задаётся) путём определения в классе А переменной, значением которой является экземпляр класса В.

Класс может находиться в отношении композиции с самим собой, то есть класс может содержать экземпляры данного класса. Например, в классе **Person** (Человек) можно объявить поля **father** и **mother**, которые задают родителей человека, и поле-массив **children[]**, задающее его детей. Эти объекты также могут быть экземплярами класса **Person**:

```
class Person { ...
    Person father, mother;
    Person[] children;
    ... }
```

Разновидностью отношения включения (композиции) является концепция **вложенных типов** (англ., *nested types*). В С# допускается определять тип (класс, структуру, перечисление, интерфейс или делегат) непосредственно внутри класса или структуры.

Реализация отношения наследования на С#.

Мощь объектно-ориентированного программирования во многом основывается на механизме наследования, позволяющем строить иерархии классов.

Наследование представляет собой возможность выделить элементы одного класса (базового) и приписать их другому классу (производному), иногда с их модификацией. При этом вновь созданные классы как бы наследуют характеристики и поведение исходного класса.

Базовые классы используются для определения характеристик и поведения, общих для всех наследников. Производные классы расширяют общую функциональность, добавляя дополнительную специфическую функциональность.

Наследование применяется для следующих целей:

- исключение из программы повторяющихся фрагментов кода;
- упрощение модификации программы;
- упрощение создания новых программ на основе существующих.

В языке C# класс может иметь произвольное количество наследников и только один базовый класс. То есть язык C# **не поддерживает множественное наследование**. При этом класс в C# может наследовать любое число интерфейсов – специальных типов, не имеющих реализации.

При описании класса имя его базового класса записывается после двоеточия:

```
[модификаторы] class Имя_производ_класса :
Имя_баз_класса
{ тело_производ_класса }
```

Если имя предка не указано, то предком считается исходный базовый класс всей иерархии **System.Object**.

Для обеспечения доступа наследников к элементам базового класса следует использовать для этих элементов модификаторы **public**, **protected** или **protected internal**. Элементы базового класса, определенные как **private**, в производном классе недоступны.

Для обращения в производном классе к элементу базового класса используется ключевое слово **base**. Это ключевое слово не допускается применять в статических методах, поскольку **base** открывает доступ к реализациям экземпляров базового класса.

Для передачи полей базового класса потомкам следует снабжать поля базового класса модификатором **protected**.

Производный класс не может изменить модификаторы или типы полей, наследованных от базового класса – он может только добавить собственные поля.

Поле базового класса можно *скрыть* в производном классе путём задания в нём поля с тем же именем и типом, а также с ключевым словом **new**.

Производный класс никогда не наследует конструкторы своего базового класса, даже если они определены как открытые (**public**). Поэтому производный класс должен иметь собственные конструкторы.

При создании экземпляра производного класса выполняются следующие действия:

- 1) полям присваиваются значения по умолчанию;
- 2) вызывается конструктор текущего класса;
- 3) конструктор текущего класса вызывает конструктор базового класса;
- 4) инициализируются поля базового класса, и управление передается в конструктор производного класса;
- 5) инициализируются поля производного класса;
- 6) создаётся экземпляр производного класса.

Для конструктора без параметров вызов аналогичного конструктора базового класса подразумевается по умолчанию. Для конструктора с параметрами вызов конструктора базового класса должен быть явным.

Вызов конструктора базового класса происходит не в теле конструктора, а в заголовке, пока ещё не создан экземпляр класса. Для вызова используется ключевое слово **base**, именуемое базовый класс. Этот вызов следует сразу за списком параметров конструктора и отделяется от этого списка символом двоеточия. Например, для класса **Employee** (Сотрудник), являющегося наследником класса **Person** (Человек), конструктор может быть задан следующим образом:

```

class Employee : Person
{ ...
    string post;
    public Employee(string post string name) :
base(name)
    { this.post = post; }
... }

```

Класс-потомок может изменять наследуемые им методы. Если производный класс создаёт метод с именем, совпадающим с именем метода базового класса, то возможны три ситуации:

- *перегрузка метода (overload)*, которая возникает, когда сигнатура создаваемого метода отличается от сигнатуры наследуемых методов предков; в этом случае в производном классе будет несколько перегруженных методов с одним именем, и вызов нужного метода определяется правилами перегрузки методов;
- *сокрытие метода (hide)*, при котором метод базового класса не является виртуальным или абстрактным, а в производном классе создаётся метод с той же сигнатурой и модификатором **new**; в этом случае при вызове метода предпочтение будет отдаваться методу потомка; метод базового класса можно вызвать через ключевое слово **base** и имя метода;
- *переопределение метода (override)*, при котором сигнатура наследуемого метода сохраняется; метод базового класса в этом случае должен иметь модификатор **virtual** или **abstract**, а метод производного класса должен быть снабжён модификатором **override**; данный случай рассматривается в следующем подразделе.

Диаграммы классов в проектах Visual Studio.

В Visual Studio присутствуют инструменты, позволяющие строить диаграммы классов проекта.

Для добавления диаграммы классов следует выбрать меню **Project | Add New Item (Проект | Добавить новый элемент)** и в открывшемся окне (рис. 4.2) указать пункт **Class Diagram (Схема классов)**.

Также для построения диаграммы в окне **Solution Explorer** (Обозреватель решений) можно выбрать имя проекта и в контекстном меню выполнить команду «Перейти к схеме классов» 

Каждый класс на диаграмме может быть раскрыт, отображая его поля, свойства и методы. Элементы класса можно редактировать через окно **Class Details** (Сведения о классах).

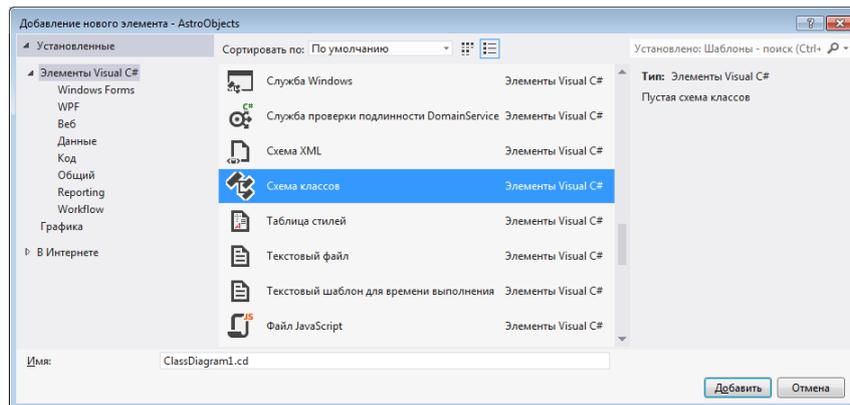


Рисунок 8.2. Окно **Add New Item** (Добавление нового элемента) с выбранным элементом **Class Diagram** (Схема классов)

Пример 8.1. Разработка семейства классов, связанных отношениями наследования и композиции, на языке C#.

Требуется реализовать семейство классов, спроектированное в примере 1.3, с помощью средств языка C#.

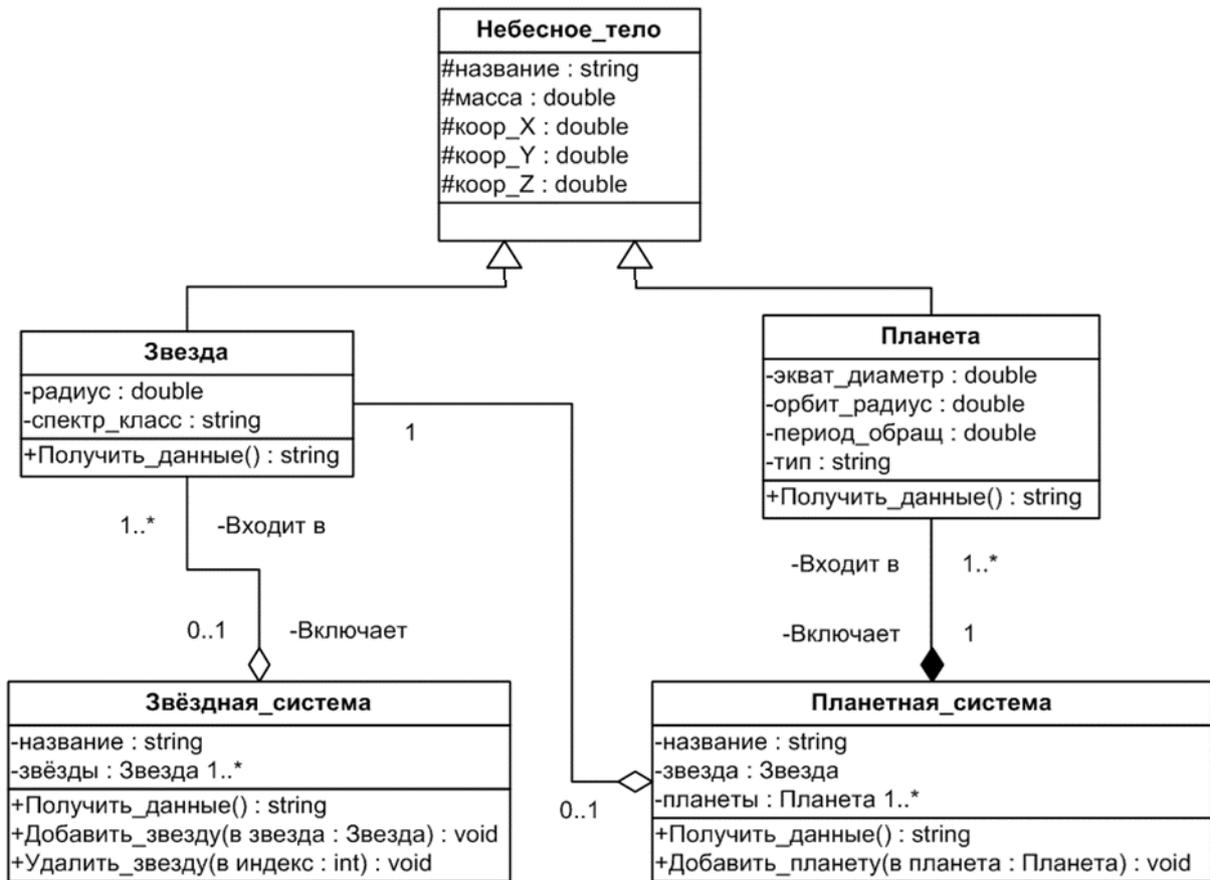


Рисунок 8.3. Диаграмма классов

Создадим решение **ClassRelations** в котором будет проект консольного приложения **ConsoleApp**. Для размещения классов добавим в решение проект библиотеки классов.

В библиотеке классов **AstroObjectsLib** будут присутствовать следующие классы:

- **SkyBody** – представляет небесные тела;
- **Planet** – представляет планеты;
- **Star** – представляет звёзды;
- **PlanetSystem** – представляет планетные системы;
- **StarSystem** – представляет звёздные системы.

Классы **SkyBody**, **Planet** и **Star** связаны отношением наследования: класс **SkyBody** – базовый, а **Planet** и **Star** – производные классы.

Класс **PlanetSystem** связан отношением композиции с классами **Planet** и **Star** и содержит ссылки на список экземпляров класса **Planet** и экземпляр класса **Star**. Класс **StarSystem** хранит

ссылки на список экземпляров класса **Star** и находится в отношении композиции с ним.

Исходный код классов **SkyBody**, **Planet**, **Star**, **PlanetSystem** и **StarSystem** представлен в листингах 4.1 – 4.5.

Сначала описывается класс **SkyBody** – базовый класс, от которого наследованием образуются другие классы.

В классе определены переменные `name` и `mass` с уровнем доступа `protected`, таким образом, они будут доступны в классах – наследниках от данного класса. Данный класс содержит два конструктора. По умолчанию и конструктор с параметрами.

*Листинг 8.1. Исходный код класса **SkyBody***

```
public class SkyBody
{
    protected string name;
    protected double mass;
    // константы гравитационная постоянная, астрономическая единица, световой год
    public const double gravConst = 6.6726e-11;
    public const double astroUnit = 1.496e+11;
    public const double lightYera = 9.46e+15;
    // Координаты небесного тела
    public double CorX { get; set; }
    public double CorY { get; set; }
    public double CorZ { get; set; }
    public SkyBody() // конструктор по умолчанию
    {
        name = "неизвестное небесное тело";
        mass = 1;
    }
    public SkyBody(string name, double mass) // конструктор с параметрами
    {
        this.name = name;
        this.mass = mass;
    }
}
```

Для классификации типов планет в проект введено перечисление **PlanetType**

*Листинг 8.2. Исходный код перечисления **PlanetType***

```
public enum PlanetType
{
    Major, // гигант
    Terra, // Земной группы
    Minor // карлик
}
```

Класс планет определён как наследник от класса небесное тело. Содержит данные о планете (экваториальный диаметр, период обращения, тип планеты.), конструктор по умолчанию, полный конструктор, полный конструктор, и метод печати данных объекта - `pasport()`.

*Листинг 8.3. Исходный код класса **Planet** (часть 1 содержание конструкторов вынесено на отдельный листинг)*

```
public class Planet : SkyBody
{
    static int count;
    double ekvatDiam;
    double orbitRad;
    double rotPeriod;
    PlanetType type;
    // константы Экв диаметр земли и масса земли - через них будут относительные
размеры планет
    public const double earthEkvD = 1.27e+7;
    public const double earthMass = 5.95e+24;
    public const double lightYera = 9.46e+15;

    public Planet() : base() // конструктор по умолчанию
    {
        ...
    }
    public Planet(string name, double mass, double diametr, double orbitRad,
        double rotPeriod, PlanetType type) : base(name, mass) // конструктор полный
    {
        ...
    }
    public string pasport()
    { return string.Format("Характеристики планеты:\n" +
        "- название:{0}\n"+
        "- масса(относительно земли) :{1:f2}\n" +
        "- экват.диаметр (относительно земли) :{2:f2}\n" +
        "- орбитальный радиус (в астрономических единицах):{3:f2}\n" +
        "- период обращения (год) :{4}\n" +
        "- тип :{5}\n", name,mass,earthEkvD,orbitRad,rotPeriod,type);
    }
}
```

Оба конструктора класса `Planet` вызывают конструкторы базового класса (`:base()`, `:base(name, mass)`). При этом полный конструктор передаёт в конструктор базового класса переданные в него значения параметров `name` и `mass`. Конструктор без параметров формирует имя планеты на основе данных о текущем количестве планет.

Листинг 8.4. Исходный код конструкторов класса Planet

```

public Planet() : base() // конструктор по умолчанию
{
    base.name = string.Format("Планета {0}", count);
    this.ekvatDiam = 1;
    this.orbitRad = 0;
    this.rotPeriod = 0;
    this.type = PlanetType.Terra;
    count++;
}
public Planet(string name, double mass, double diametr,
double orbitRad,
double rotPeriod, PlanetType type) : base(name, mass)
// конструктор полный
{
    base.name = name;
    this.ekvatDiam = diametr;
    this.orbitRad = orbitRad;
    this.rotPeriod = rotPeriod;
    this.type = type;
    count++;
}
}

```

Класс Star (звезда) также определён как наследник от класса SkyBody (небесное тело).

Содержит переменные для хранения данных о звезде (радиус, спектральный класс и т.д.) конструктор по умолчанию и конструктор с параметрами. Как и в другие классы, в данный класс включён метод печати данных объекта - passport(). Необходимо отметить, что в конструкторе по умолчанию формируется наименование планеты и передаётся в переменную базового класса name. Обращение к переменной базового класса производится через объект base -

```
(base.name= String.Format("Звезда № {0}", count);
```

Листинг 8.5. Исходный код класса Star

```

class Star:SkyBody
{
    static int count;//число созданных звёзд
    double radius; //радиус звезды
    string spectrClass; // спектральный класс (М, К, G и др.)
    public const double sunMass=1.99e+30; // масса относительно солнца
    public const double sunRadius = 6.96e+8; // радиус относительно солнечного

```

```

public Star() : base() // конструктор по умолчанию
{
    base.name = String.Format("Звезда № {0}", count);
    this.radius = 1;
    this.spectrClass = "G0";
    count++;
}
public Star(string name, double mass, double radius, string spectrClass) :
base(name, mass) // конструктор по умолчанию
{
    this.radius = radius;
    this.spectrClass = spectrClass;
    count++;
}
public string pasport()
{ return string.Format("Характеристики звезды:\n" +
    "- название:{0}\n" +
    "- масса(относительно Солнца) :{1:f1}\n" +
    "- радиус (относительно Солнца):{2:f1}\n" +
    "- спектральный класс :{3}\n", name, mass, radi-
us, spectrClass);
}
}

```

Класс, отображающий планетные системы PlanetSyst, должен обеспечивать включение в себя звезды и перечня планет, которые входят в данную систему. Для этого в данный класс должна быть включена переменная ранее определённого класса Star и список, элементами которого являются планеты List<planet> .

Листинг 8.6. Исходный код класса PlanetSystem

```

class PlanetSystem
{
    static int count;
    string name;
    Star star;
    List<Planet> planets;
    public PlanetSystem()
    { this.name=String.Format("Планетная система {0}", name);
      this.star = new Star();
      this.planets = new List<Planet>() { new Planet() };
      count++;
    }
    public PlanetSystem(string name, Star star, List<Planet> planets)
    { this.name = name;
      this.star = star;
      this.planets = planets;
      count++;
    }
    public string pasport()
    { return string.Format("Характеристики планетной системы:\n" +
        "- название:{0}\n" +
        "- число планет :{1}", name, planets.Count());
    }
}

```

```

public void addPlanet(Planet newPlanet)
{ planets.Add(newPlanet);
}
}

```

Класс, отображающий звёздные системы `StarSystem`, должен обеспечивать включение в себя некоторого набора звёзд. Для этого в данный класс должна быть включен список, элементами которого являются экземпляры класса `Star List<Star>`

Листинг 8.7. Исходный код класса `StarSystem`

```

class StarSystem
{
    static int count;
    string name;
    Star star;
    List<Star> stars;
    public StarSystem()
    {
        this.name = String.Format("Звёздная система {0}",
count);
        this.stars = new List<Star>() { new Star() };
        count++;
    }
    public StarSystem(string name, List<Star> stars)
    {
        this.name = name;
        this.stars = stars;
        count++;
    }
    public string passport()
    {
        return string.Format("Характеристики звёздной
системы:\n" +
                                "- название:{0}\n" +
                                "- число звёзд :{1}", name,
stars.Count());
    }
    public void addStar(Star newStar)
    {
        stars.Add(newStar);
    }
    public void removeStar(int index)
    {
        if (index >= 0 && index < stars.Count)
stars.RemoveAt(index);
    }
}

```

Для демонстрации полученных классов добавим в метод `Main()` консольного приложения программный код, в котором создаются экземпляры классов и выполняется вызов их методов.

Исходный код метода **Main()** консольного приложения представлен в Листинг 8.8.

*Листинг 8.8. Исходный код метода `static void Main(string[] args)` класса **Program** консольного приложения*

```

    Console.Title = "Лабораторная работа 9 Наследование ком-
позиция в семействах классов с#";
    Console.WriteLine("Астрономические объекты");
    Star star1 = new Star();
    Star sun = new Star("Солнце", 1, 1, "G2V");
    Star proxCen = new Star("Проксима Центавра", 0.12,
0.15, "Ms.SVe");
    Star alfaCenA = new Star("АльФА Центавра А", 1.2, 1.23,
"G2V");
    Star alfaCenB = new Star("АльФА Центавра В", 0.85, 0.9,
"K1V");
    Star antares = new Star("Антарес", 16, 883,
"M1.SI");
    Console.WriteLine(sun.pasport());
    Console.WriteLine(proxCen.pasport());
    Console.WriteLine(alfaCenA.pasport());
    Console.WriteLine(antares.pasport());

    Planet planet1 = new Planet();
    Planet mercury = new Planet("Меркурий", 0.06, 0.382,
0.39, 0.24, PlanetType.Minor);
    Planet venus = new Planet("Венера", 0.82, 0.949, 0.72,
0.62, PlanetType.Terra);
    Planet earth = new Planet("Земля", 1, 1, 1, 1, Planet-
Type.Terra);
    Planet mars = new Planet("Марс", 0.11, 0.532, 1.52,
1.88, PlanetType.Terra);
    Planet jupiter = new Planet("Юпитер", 317.8, 11.21,
5.2, 11.86, PlanetType.Major);

    // формирование списка планет солнечной системы список
планет солнечной системы
    List<Planet> solarPlanet = new List<Planet>() { mercur-
y, venus, earth, mars, jupiter };
    // вывод планет солнечной системы
    foreach (Planet p in solarPlanet)
    { Console.WriteLine(p.pasport()); }
    PlanetSystem solarPlSystem = new
PlanetSystem("Солнечная система", sun, solarPlanet);
    Console.WriteLine(solarPlSystem.pasport());
    solarPlSystem.AddPlanet(new Planet("Плутон", 0.002,

```

```

0.19, 49.5, 248.09, PlanetType.Minor));
    Console.WriteLine(solarPlSystem.pasport());

    List<Star> alfaCentStars = new List<Star>() { proxCen,
alfaCenA, alfaCenB };
    StarSystem alfaCenSyst = new StarSystem("Система Альфа
Центавра", alfaCentStars);
    Console.WriteLine(alfaCenSyst.pasport());
    alfaCenSyst.AddPStar(star1);
    Console.WriteLine(alfaCenSyst.pasport());
Console.ReadKey();

```

Результат работы консольного приложения показан на рис. 8.4.

The image shows two screenshots of a console application window titled "Наследование и композиция в семействах классов на C#". The window displays the output of a program that prints characteristics of various astronomical objects.

The first screenshot shows the following output:

```

***** Астрономические объекты *****
Характеристики звезды:
- название: Солнце
- масса (относ. Солнца): 1.0
- радиус (относ. Солнца): 1.0
- спектральный класс: G2V
Характеристики звезды:
- название: Проксима Центавра
- масса (относ. Солнца): 0.1
- радиус (относ. Солнца): 0.2
- спектральный класс: M5.5Ve
Характеристики звезды:
- название: Сириус B
- масса (относ. Солнца): 2.1
- радиус (относ. Солнца): 1.7
- спектральный класс: B10
Характеристики планеты:
- название: Меркурий
- масса (относ. Земли): 0.06
- экватор. диаметр (относ. Земли): 0.38
- орбитальный радиус, а.е.: 0.39
- период обращения, год: 0.24
- тип: земной группы
Характеристики планеты:
- название: Венера
- масса (относ. Земли): 0.82
- экватор. диаметр (относ. Земли): 0.95
- орбитальный радиус, а.е.: 0.72
- период обращения, год: 0.62
- тип: земной группы

```

The second screenshot shows the following output:

```

- орбитальный радиус, а.е.: 9.54
- период обращения, год: 29.46
- тип: газовый гигант
Характеристики планеты:
- название: Уран
- масса (относ. Земли): 14.60
- экватор. диаметр (относ. Земли): 4.01
- орбитальный радиус, а.е.: 19.22
- период обращения, год: 84.01
- тип: газовый гигант
Характеристики планеты:
- название: Нептун
- масса (относ. Земли): 17.20
- экватор. диаметр (относ. Земли): 3.88
- орбитальный радиус, а.е.: 30.06
- период обращения, год: 164.8
- тип: газовый гигант
Характеристики планетной системы:
- название: Солнечная система
- число планет: 8
Характеристика звездной системы:
- название: Система Альфа Центавра
- число звезд: 3
Характеристика звездной системы:
- название: Система Сириуса
- число звезд: 3
Характеристика звездной системы:
- название: Система Сириуса
- число звезд: 2

```

Рисунок 8.4. Результат работы консольного приложения

3. ПОРЯДОК ВЫПОЛНЕНИЯ РАБОТЫ И ВАРИАНТЫ ЗАДАНИЙ

Этапы выполнения работы.

Данная лабораторная работа предполагает выполнение следующих этапов:

1. Изучить методические указания к лабораторной работе.
2. Построить библиотеку классов с семейством классов из работы №1 (табл. 1.7), связанных отношениями композиции и наследования. Разработать консольное приложение для работы с экземплярами полученных классов.
3. Дополнительно разработать заданные классы, связанные отношением наследования (табл. 4.1). Продемонстрировать в консольном приложении использование операций **is** и **as** при работе с экземплярами классов.
4. Оформить и защитить отчет по лабораторной работе.

Варианты заданий.

Варианты заданий в качестве задания необходимо взять классы из лабораторной работы 2 своего варианта

Требования к отчёту.

Отчёт по лабораторной работе должен содержать следующие пункты:

1. Титульный лист с указанием названия работы.
2. Цель и задачи работы.
3. Исходный код библиотеки классов, содержащей семейство классов, которые связаны отношениями композиции и наследования.
4. Исходный код консольного приложения, демонстрирующего работу с экземплярами классов из библиотеки.

4. КОНТРОЛЬНЫЕ ВОПРОСЫ И ЗАДАЧИ

1. Каковы основные виды отношений между классами в программе на языке С#?
2. Какое отношение между классами называют отношением композиции?
3. Что понимают под наследованием в ООП?
4. Для каких задач используют наследование при разработке объектно-ориентированных программ?
5. Как задают отношение наследования между классами в коде на языке С#?
6. В чём заключаются особенности работы с конструкторами при наследовании?
7. Что называют сокрытием метода базового класса в производном классе?
8. В чём заключается принцип подстановки Лисков?

ЛАБОРАТОРНАЯ РАБОТА №9. РАБОТА В ПРОЕКТЕ VISUAL STUDIO WINDOWS FORMS

1. ЦЕЛЬ РАБОТЫ

Цель работы – получить практические навыки работы с управляющими элементами Windows Forms

2. ТЕОРЕТИЧЕСКИЕ ПОЛОЖЕНИЯ

Современные среды разработки предоставляют готовые компоненты диалоговых приложений, включая наборы стандартных элементов управления, то есть элементов позволяющим управлять работой приложения.

Базовым элементом управления является форма.

Элементами для ввода управляющих воздействий – кнопки.

Элементами для ввода данных – поля ввода.

Отображать данные можно с помощью меток, окон сообщений.

3. ЗАДАНИЯ ДЛЯ РАБОТЫ

Преобразовать операции вывода данных так, чтобы они выводили данные в виде окон сообщений MessageBox, в виде текста меток.

Преобразовать приложение для лабораторной работы 9 в виде формы.

ЛАБОРАТОРНАЯ РАБОТА №10. ОСНОВЫ ИСПОЛЬЗОВАНИЯ ПОЛИМОРФИЗМА В СЕМЕЙСТВАХ КЛАССОВ НА ЯЗЫКЕ C#

1. ЦЕЛЬ И ЗАДАЧИ РАБОТЫ

Цель работы – приобрести умение использовать полиморфизм в семействах классов, связанных отношением наследования, путём использования в классах виртуальных и абстрактных методов.

Основные задачи:

- освоить создание виртуальных методов в базовых классах и их переопределение в производных классах;
- научиться использовать абстрактные классы для построения иерархии классов.

Работа рассчитана на 4 часа.

2. ОСНОВНЫЕ ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

Понятие полиморфизма. Виртуальные методы.

В общем случае под *полиморфизмом* понимают изменение поведения базового класса в производных классах. При этом операции производных классов имеют одинаковую сигнатуру, но разные реализации.

Полиморфизм наряду с инкапсуляцией и наследованием представляет одну из важнейших концепций объектно-ориентированного программирования. Применение полиморфизма позволяет строить легко расширяемое и гибкое программное обеспечение.

В программировании с полиморфизмом тесно связаны такие понятия, как

- виртуальные методы,
- переопределение методов,
- абстрактные классы и абстрактные методы.

Виртуальными методами называют методы, у которых в заголовке присутствует ключевое слово **virtual**:

```
public virtual void GetTotalValue() ...
```

Виртуальные методы создаются в базовых классах и переопределяются в производных классах.

Под *переопределением* (англ. *override*) виртуального метода понимают изменение реализации этого метода в производных классах. Производные классы не должны в обязательном порядке переопределять виртуальные методы базового класса.

Объявление метода «виртуальным» означает, что все ссылки на этот метод будут разрешаться на стадии выполнения программы, а не на стадии компиляции. Такой механизм называется *поздним связыванием* (англ., *late binding*).

Для обеспечения позднего связывания необходимо, чтобы адреса виртуальных методов хранились там, где ими можно в любой момент воспользоваться. С этой целью компилятор формирует *таблицу виртуальных методов* (англ., *Virtual Method Table, VMT*), в которую записываются адреса виртуальных методов в порядке описания в классе.

Если в производном классе требуется переопределить виртуальный метод, то в заголовке соответствующего метода используется ключевое слово **override**:

```
public override void GetTotalValue()
{ ...
    base.GetTotalValue()
... }
```

Переопределенный виртуальный метод должен обладать таким же набором параметров, как и одноименный метод базового класса.

Виртуальные методы базового класса определяют интерфейс всей иерархии классов. Этот интерфейс может расширяться в потомках за счет добавления новых виртуальных методов.

При описании классов рекомендуется определять в качестве виртуальных те методы, которые в производных классах должны реализовываться по-другому.

В Visual Studio после набора слова **override** внутри класса и нажатии клавиши пробела автоматически отображается список всех переопределяемых элементов базового класса (рис. 5.1). По-

сле выбора элемента и нажатия клавиши **Enter** среда выполнит автоматическое заполнение шаблона метода.

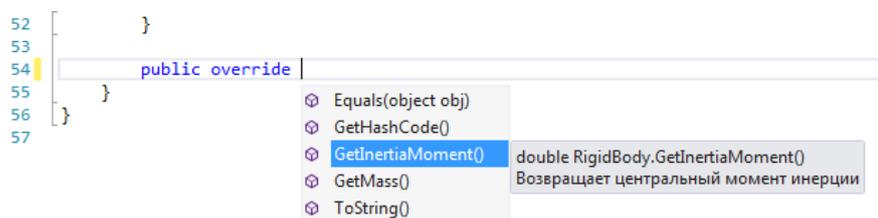


Рисунок 10.1. Просмотр списка переопределяемых элементов в Visual Studio

□ Пример 10.1. Разработка классов с виртуальными методами и их переопределением.

Требуется разработать класс **BaseCar** (Легковой автомобиль в базовой комплектации), который содержит виртуальный метод **int GetCost()**, определяющий стоимость автомобиля на основе года выпуска, и переопределяет следующие методы базового класса **System.Object**:

- **string ToString()** – возвращает строку с данными о состоянии автомобиля;
- **bool Equals(object obj)** – возвращает результат сравнения автомобилей по марке и году выпуска (если они совпадают, то результат – **true**, иначе – **false**);
- **int GetHashCode()** – возвращает хеш-код на основе строки, формируемой методом **ToString()**.

Кроме того, требуется разработать производный от него класс **ForcedCar** (Легковой автомобиль с гидроусилителем руля), который будет переопределять методы **int GetCost()** и **string ToString()**.

Исходный код указанных классов представлен в листингах 10.1, 10.2 и 10.3.

Исходный код класса **Program** консольного приложения приведён в листинге 10.4.

Листинг 10.1. Исходный код класса BaseCar

```

public class BaseCar
{
    public string Mark { get; set; }
    public int Year { get; set; }
    public int Cost { get; set; }
    public BaseCar()
    {
        Mark = "Lada Vesta";
        Year = 2020;
        Cost = 800000;
    }
    public BaseCar(string Mk, int Yr, int Cst)
    {
        Mark = Mk;
        Year = Yr;
        Cost = Cst;
    }
    public override string ToString() // переопределяется метод
    класса object
    {
        return String.Format("Данные об автомобиле :\n Модель-{0}
        Год выпуска-{1} Базовая стоимость {2}",
            Mark, Year, Cost);
    }
    public virtual int GetCost()
    {
        // пусть уценка определяется возрастом авто. На каждый год
        уменьшение на 1/85
        int disCost = Convert.ToInt32(Cost*(DateTime.Today.Year -
        Year)/85);
        int newCost = Cost - disCost;
        return newCost;
    }
}

```

Листинг 10.2. Исходный код класса ForcedCar (часть 1)

```

public class ForcedCar : BaseCar
{
    public int dopCmpCost; // стоимость доп компонентов
    public int dopMontCost; // стоимость доп монтажа компонен-
    тов
    public ForcedCar() : base() // конструктор по умолчанию
    {
        dopCmpCost = 20000;
        dopMontCost = 10000;
    }
    public ForcedCar(string Mrk, int Yr, int Cst, int dopCmp-
    Cst, int dopMntCst): base(Mrk, Yr, Cst) // полный конструктор
    {
        dopCmpCost = dopCmpCst;
        dopMontCost = dopMntCst;
    }

    public override string ToString()// переопределяется метод

```

класса object

```

    { return base.ToString() + String.Format("\n стоимость доп
компонентов {0} стоимость доп монтажа-{1} ", dopCmpCost, dop-
MontCost);
    }
    // переопределение метода базового класса расчёта стоимости
как метод базового класса к результатом которого
// добавляется стоимость доп компонентов и доп монтажа
public override int GetCost()
{ return base.GetCost() + dopCmpCost + dopMontCost;
}
}

```

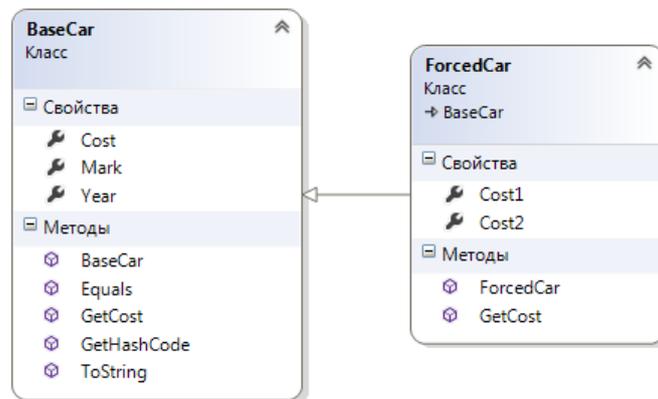


Рисунок 10.2. Диаграмма классов проекта

*Листинг 10.3. Исходный код класса **Program** консольного приложения*

```

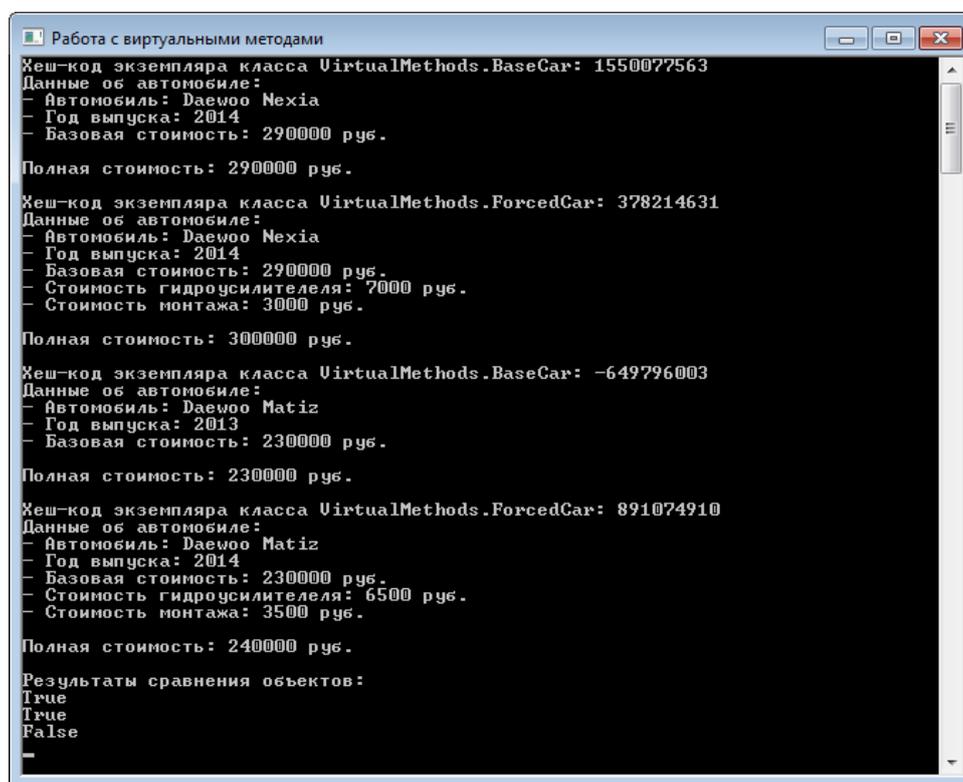
class Program
{
    static void Main(string[] args)
    {
        BaseCar bc1 = new BaseCar("Lada XRay", 2018, 80000);
        ForcedCar fc1 = new ForcedCar("Lada xray", 2021, 80000,
20000, 10000);
        BaseCar bc2 = new BaseCar();
        ForcedCar fc2 = new ForcedCar();
        BaseCar[] cars = new BaseCar[] { bc1, fc1, bc2, fc2 };
        foreach (BaseCar cr in cars)
        { Console.WriteLine(cr.ToString());
          Console.WriteLine("Стоимость авто с учётом износа и
улучшений{0} ", cr.GetCost());
        }

        Console.ReadKey();
    }
}

```

```
}  
}
```

Результат работы консольного приложения представлен на рис. 10.3.



```
Работа с виртуальными методами  
Хеш-код экземпляра класса VirtualMethods.BaseCar: 1550077563  
Данные об автомобиле:  
- Автомобиль: Daewoo Nexia  
- Год выпуска: 2014  
- Базовая стоимость: 290000 руб.  
Полная стоимость: 290000 руб.  
Хеш-код экземпляра класса VirtualMethods.ForcedCar: 378214631  
Данные об автомобиле:  
- Автомобиль: Daewoo Nexia  
- Год выпуска: 2014  
- Базовая стоимость: 290000 руб.  
- Стоимость гидроусилителя: 7000 руб.  
- Стоимость монтажа: 3000 руб.  
Полная стоимость: 300000 руб.  
Хеш-код экземпляра класса VirtualMethods.BaseCar: -649796003  
Данные об автомобиле:  
- Автомобиль: Daewoo Matiz  
- Год выпуска: 2013  
- Базовая стоимость: 230000 руб.  
Полная стоимость: 230000 руб.  
Хеш-код экземпляра класса VirtualMethods.ForcedCar: 891074910  
Данные об автомобиле:  
- Автомобиль: Daewoo Matiz  
- Год выпуска: 2014  
- Базовая стоимость: 230000 руб.  
- Стоимость гидроусилителя: 6500 руб.  
- Стоимость монтажа: 3500 руб.  
Полная стоимость: 240000 руб.  
Результаты сравнения объектов:  
True  
True  
False  
-
```

Рисунок 10.3. Работа консольного приложения

3. ПОРЯДОК ВЫПОЛНЕНИЯ РАБОТЫ И ВАРИАНТЫ ЗАДАНИЙ

Данная лабораторная работа предполагает выполнение следующих этапов:

1. Изучить методические указания к лабораторной работе.
2. Разработать базовый класс, содержащий виртуальный метод (табл. 5.1), и производный от него класс, в котором этот метод переопределён. Реализацию метода придумать самостоятельно. Дополнительно требуется переопределить методы класса **System.Object**.
3. Оформить и защитить отчет по лабораторной работе.

4. ВАРИАНТЫ ЗАДАНИЙ

Таблица 10.1.

Варианты заданий для разработки классов, содержащих виртуальные методы и их переопределение

№ вар.	Классы	Виртуальный метод
1, 7, 13, 19	<i>Базовый.</i> Здание (площадь, высота, число этажей, износ) <i>Производный.</i> Жилое здание (общая жил-площадь)	Определить стоимость ремонта
2, 8, 14, 20	<i>Базовый.</i> Кредит (дата выдачи, сумма, срок, процент) <i>Производный.</i> Кредит с залогом имущества (стоимость имущества).	Определить сумму возврата
3, 9, 15, 21	<i>Базовый.</i> Квартира (адрес дома, номер, площадь, число комнат, этаж). <i>Производный.</i> Квартира с лоджией (площадь лоджии)	Определить цену недвижимости
4, 10, 16, 22	<i>Базовый.</i> Перевозка грузов (дата, адрес доставки, тип груза, вес груза). <i>Производный.</i> Перевозка грузов за пределы города (расстояние от города).	Определить стоимость перевозки
5, 11, 17, 23	<i>Базовый.</i> Вклад (номер счёта, дата предоставления, сумма, процент). <i>Производный.</i> Срочный вклад (срок депонирования суммы).	Определить начисления по процентам
6, 12, 18, 24	<i>Базовый.</i> Деталь (наименование, материал, масса, трудоёмкость изготовления). <i>Производный.</i> Деталь с покрытием (стоимость покрытия).	Определить себестоимость

5. КОНТРОЛЬНЫЕ ВОПРОСЫ И ЗАДАЧИ

1. Что понимают под полиморфизмом в объектно-ориентированном программировании?
2. Какие методы называют виртуальными?
3. Что в объектно-ориентированном программировании называют поздним связыванием?
4. Какую роль играет таблица виртуальных методов?
5. Как в коде на C# осуществляется переопределение виртуальных методов?
6. Какие виртуальные методы содержит базовый класс **System.Object**?

ЛАБОРАТОРНАЯ РАБОТА №11. АБСТРАКТНЫЕ КЛАССЫ И МЕТОДЫ. ЗАПЕЧАТАННЫЕ КЛАССЫ И МЕТОДЫ

1. ТЕОРЕТИЧЕСКИЕ ПОЛОЖЕНИЯ

Возможны случаи, когда базовый класс для нескольких типов дочерних сущностей представляет очень общую сущность, и создание экземпляров такого класса не будет иметь смысла. В этом случае класс объявляют как абстрактный (*abstract*) и в нём присутствуют абстрактные методы.

На диаграммах классов UML абстрактные операции записываются курсивом (Рисунок 11.1). Имя абстрактного класса также записывается курсивом.

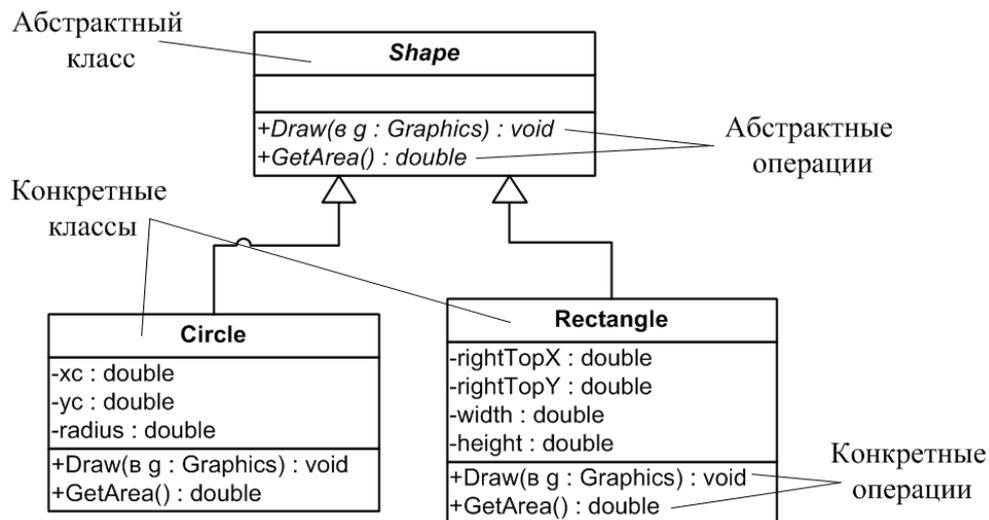


Рисунок 11.1. Абстрактный класс на диаграмме классов UML

При программной реализации средствами С# абстрактные классы UML отображаются абстрактными С#. Абстрактный программный класс С# обозначается модификатором **abstract**.

```

public abstract class Shape
{
    ...
    public abstract void Draw();
    ...
}
  
```

Абстрактный класс должен иметь хотя бы один *абстрактный* метод. Метод называется *абстрактным*, если при его опре-

делении задана его сигнатура, но не задана реализация метода. Абстрактный метод представляет собой виртуальный метод, переопределяемый производными классами.

Абстрактными могут быть только методы экземпляра, но не в статические методы.

Так как абстрактные классы содержат абстрактные методы, они не являются полностью определёнными и для них нельзя создавать объекты.

Кроме абстрактных методов, абстрактные классы могут содержать и полностью определённые методы (в отличие от сходного с ним по назначению специального вида классов, называемых интерфейсом).

Абстрактные классы служат только для создания классов-потомков. Обычно в абстрактном классе задается набор методов, которые каждый из потомков будет реализовывать по-своему. Если класс, являющийся производным от абстрактного класса, не переопределяет все абстрактные методы, то он также является абстрактным и тоже должен иметь модификатор **abstract**.

Запечатанные классы и методы.

Полезным видом классов являются *запечатанные* (бесплодные) классы, для которых запрещается строить производные классы путём наследования. Примером запечатанного класса является класс **String** встроенной библиотеки .NET Framework.

Запечатанный класс задается с помощью ключевого слова **sealed**.

□Пример 11.1. Разработка иерархии классов, начиная с абстрактного класса.

Требуется разработать семейство из трёх классов, связанных иерархией наследования:

- **Твёрдое тело:** плотность материала тела; Определить объём; Определить массу; Определить центральный момент инерции;
- **Сегмент параболоида вращения:** радиус основания, высота (Рисунок 11.2);
- **Усечённый параболоид вращения:** радиус малого основания.

Формулы, по которым определяются объёмы и центральные моменты инерции заданных фигур, имеют следующий вид:

- *сегмент параболоида вращения:*

$$V = \frac{1}{2} \pi R^2 H ;$$

$$I_{zz} = \frac{1}{5} m R^2 .$$

- *усечённый параболоид вращения:*

$$V = \frac{1}{2} \pi H (r^2 + R^2) ;$$

$$I_{zz} = \frac{1}{5} m R r .$$

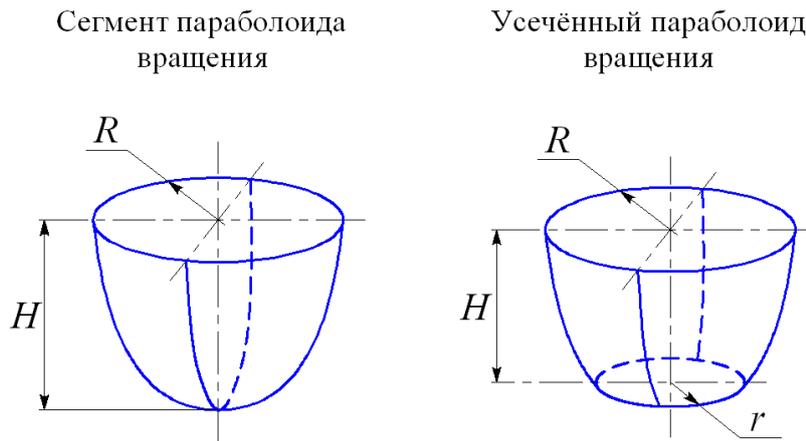


Рисунок 11.2. Параболоиды вращения

Пусть в Visual Studio будет создано решение **Polymorphism**, с проектом консольного приложения **ConsoleApplication**. В решение добавлен проект библиотеки классов **PolymorphClasses**.

В библиотеке классов **PolymorphClasses** реализована следующая иерархия классов:

- **RigidBody** (Твёрдое тело) – абстрактный класс;
- **Paraboloid** (Параболоид вращения) – наследник класса **RigidBody**:

• **TruncParab** (Усечённый параболоид вращения) – наследник класса **Paraboloid**.

Исходный код классов **RigidBody**, **Paraboloid** и **TruncParab** приведён в листингах 5.5 – 5.7.

*Листинг 11.1. Исходный код абстрактного класса **RigidBody***

```
public RigidBody()
    {   dencity = 1000.0;
        count++;
    }
    public RigidBody(double dencity )
    {   this.dencity = dencity;
        count++;
    }
    public static int Count //свойство для получения количества
//тел
    { get { return count; } }

    public double Dencity //свойство для получения задания
//плотности
    { get { return dencity; }
      set { if (value < 0) dencity = 0; }
    }
    public abstract double getMass();
    public abstract double getInertiaMoment();
}
```

В классе определяется переменная `dencity` (плотность тела), свойство `Dencity` для установки и чтения значения данной переменной, полный конструктор и два абстрактных метода `getMass()`, `getInertiaMoment()`.

*Листинг 11.2. Исходный код класса **Paraboloid** (часть 1)*

```
public class Paraboloid:RigidBody
    {
        protected double radius;
        protected double height;
        public Paraboloid() : base()
        {   radius = 1;
            height = 1;
        }
        public Paraboloid(double dencity, double radius, double
height) : base(dencity)
        {   this.radius = radius;
            this.height = height;
        }
    }
```

```
    }
    public virtual double Radius // Свойство для доступа к
//радиусу (получение и установка значения)
    {   get { return radius; }
        set { if (value <=0) radius = 1.0; }
    }
    public virtual double Height// Свойство для доступа к
//высоте (получение и установка значения)
    {   get { return height; }
        set { if (value < 0) height = 1.0; }
    }
    public override string ToString() // перегрузка метода
//Object.ToString() для вывода данных экземпляра класса
    {   return String.Format("Характеристики параболоида
вращения: \nплотн осьть {0:f2} кг/м3 радиус основания {1:f2} м
высота {2:f2}", dencity, radius, height);
    }
    public virtual double getVolume() //виртуальный метод
//расчёта объёма
    {   return Math.PI * radius * radius * 5;
    }
    public override double getMass() // перегруженный метод
//расчёта массы
    {   return getVolume()*dencity;
    }
    public override double getInertiaMoment() // перегруженный
//метод расчёта центрального момента инерции
    {   return getMass() *radius*radius/5;
    }
}
```

В классе вводятся переменные `radius` и `height`, свойства для доступа к данным переменным. Кроме того в классе добавляются методы, перегружающие (`override`) виртуальные методы `getMass()`, `getInertiaMoment()`, определённые в родительском классе и добавляется виртуальный метод `getVolume()`.

*Листинг 11.3. Исходный код класса **TruncParaboloid***

```

public sealed class TruncParaboloid : Paraboloid
{
    double radiusSmoll;//
    public TruncParaboloid() : base()
    { radiusSmoll = base.radius / 2; }
    public TruncParaboloid(double dencity, double radius, dou-
ble height, double radiusSmall) : base(dencity, radius, height)
    { this.radiusSmoll = radiusSmall; }
    public override double Radius // Свойство для доступа к
радиусу (получение и установка значения)
    {
        get { return radius; }
        set { if (value <= radiusSmoll) radius = radiusSmoll; }
    }
    public double RadiusSmall // Свойство для доступа к малому
радиусу (получение и установка значения)
    {
        get { return radiusSmoll; }
        set { if (value <= 0) radiusSmoll = 0; }
    }

    public override string ToString() // перегружается метод
Object для вывода данных экземпляра класса
    { return base.ToString()+String.Format("радиус малого
основания {0:f2}:", radiusSmoll);
    }
    public override double getVolume() // перегруженный метод
расчёта объёма
    { return (base.getVolume()+Math.PI*height*radiusSmoll/2);
    }
    public override double getMass() // перегруженный метод
расчёта массы
    { return getVolume() * dencity;
    }
    public override double getInertiaMoment() // перегруженный
метод расчёта центрального момента инерции
    { return getMass() * radiusSmoll * radiusSmoll / 2;
    }
}

```

Схема классов, добавленных в проект библиотеки, показана ниже (Рисунок 11.3).

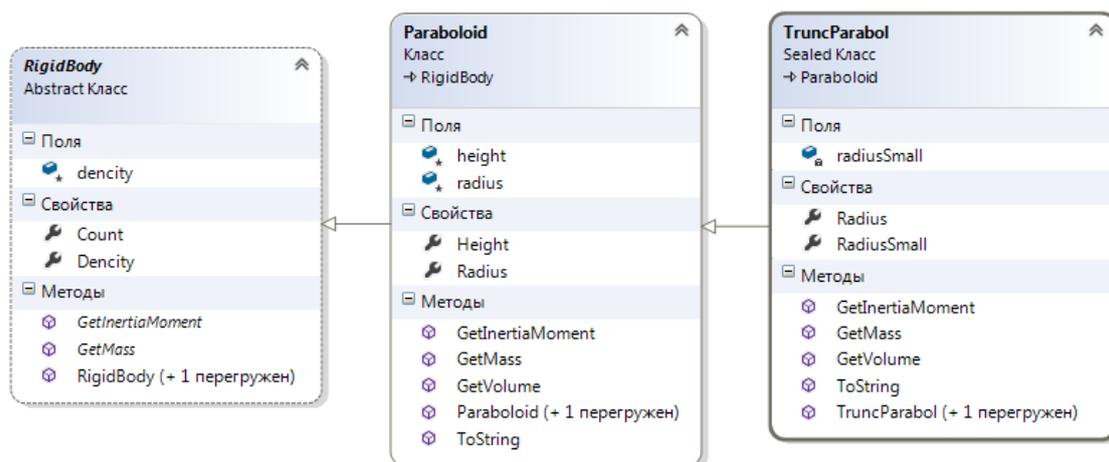


Рисунок 11.3. Схема классов проекта

Исходный код консольного приложения представлен в листинге 11.4.

*Листинг 11.4. Исходный код метода **Main()** консольного приложения*

```

static void Main(string[] args)
{ Console.WriteLine("Работа с абстрактными классами и методами");
    Console.WriteLine("Твёрдые тела");
    RigidBody pb1 = new Paraboloid();
    RigidBody pb2 = new Paraboloid(800,0.7,1.2);
    RigidBody pb3 = new TruncParaboloid();
    RigidBody pb4 = new TruncParaboloid(750, 0.6,
0.25,0.9);
    RigidBody[] bodies = new RigidBody[] { pb1, pb2, pb3,
pb4 };
    foreach (RigidBody rb in bodies)
    {
        Console.WriteLine(rb.ToString());
        Console.WriteLine("Характеристика очередного"+
"тела\n масса {0:f2} кг\n момент инерции {1:f2} кг*м2\n",
            rb.getMass(), rb.getInertiaMoment());
    }
    Console.ReadKey();
}
  
```

Обратите внимание, на использование в данном случае принципа подстановки Лисков. Классы разных типов `Paraboloid` и `TruncParaboloid` создаются в переменные типа базового класса

RigidBody. При вызове в цикле перегруженного метода `rb.ToString()`, используется перегрузка соответствующая классу экземпляра. Аналогично с вызовами методов `rb.getMass()`, `rb.getInertiaMoment()`.

Результат работы консольного приложения показан на (Рисунок 11.4) □

```

Работа с абстрактными классами и методами
Твёрдые тела
Характеристики параболоида вращения:
плотность 1000,00 кг/м3 радиус основания 1,00 м высота 1,00
Характеристика очередного тела
масса 15707,96 кг
момент инерции 3141,59 кг*м2

Характеристики параболоида вращения:
плотность 800,00 кг/м3 радиус основания 0,70 м высота 1,20
Характеристика очередного тела
масса 6157,52 кг
момент инерции 603,44 кг*м2

Характеристики параболоида вращения:
плотность 1000,00 кг/м3 радиус основания 1,00 м высота 1,00
Доп. для учённого - Радиус малого основания 0,50:
Характеристика очередного тела
масса 16493,36 кг
момент инерции 2061,67 кг*м2

Характеристики параболоида вращения:
плотность 750,00 кг/м3 радиус основания 0,60 м высота 0,25
Доп. для учённого - Радиус малого основания 0,90:
Характеристика очередного тела
масса 4506,22 кг
момент инерции 1825,02 кг*м2

```

Рисунок 11.4. Результат работы консольного приложения

2. ПОРЯДОК ВЫПОЛНЕНИЯ РАБОТЫ И ВАРИАНТЫ ЗАДАНИЙ

1. Изучить методические указания к лабораторной работе.
2. Построить иерархию классов, начиная с абстрактного класса (табл. 11.2). Предусмотреть виртуальные методы в проектируемых классах, а также переопределение этих методов в классах-потомках. Разместить полученные классы в DLL. Разработать консольное приложение, демонстрирующее полиморфизм построенного семейства классов.

3. Оформить и защитить отчет по лабораторной работе.
Варианты заданий.

Таблица 11.1.

Варианты заданий для разработки иерархии классов, начинающая от абстрактного класса

№ вар.	Данные для разработки иерархии классов
1, 9, 17	Класс 1. Объёмная фигура (Определить объём, Определить площадь поверхности). Класс 2. Шар (радиус). Класс 3. Полый шар (внутренний радиус).
2, 10, 18	Класс 1. Плоская фигура (Определить периметр, Определить площадь). Класс 2. Круг (радиус). Класс 3. Круглое кольцо (внутренний радиус).
3, 11, 19	Класс 1. Объёмная фигура (Определить объём, Определить площадь поверхности). Класс 2. Прямой круговой конус (радиус основания, высота). Класс 3. Усечённый прямой круговой конус (радиус малого основания).
4, 12, 20	Класс 1. Плоская фигура (Определить периметр, Определить площадь). Класс 2. Трапеция (верхнее основание, нижнее основание, высота). Класс 3. Параллелограмм (угол параллелограмма).
5, 13, 21	Класс 1. Объёмная фигура (Определить объём, Определить площадь поверхности). Класс 2. Прямой круговой цилиндр (радиус основания, высота). Класс 3. Полый цилиндр (внутренний радиус).
6, 14, 22	Класс 1. Объёмная фигура (Определить объём, Определить площадь поверхности). Класс 2. Правильная пирамида (площадь основания, высота, апофема). Класс 3. Правильная усечённая пирамида (площадь малого основания).
7, 15, 23	Класс 1. Объёмная фигура (Определить объём, Определить площадь поверхности). Класс 2. Шаровой сегмент (высота, радиус шара, радиус основания). Класс 3. Шаровой слой (радиус малого основания).
8,	Класс 1. Плоская фигура (Определить периметр, Определить

№ вар.	Данные для разработки иерархии классов
16, 24	площадь). Класс 2. Круговой сектор (радиус, центральный угол). Класс 3. Круговой сегмент (высота).

Практические задачи.

Требуется разработать иерархию классов, начиная с абстрактного класса (табл. 11.1).

Таблица 11.2.

Варианты заданий для разработки семейства классов

Вар.	Диаграмма классов
Нечёт- чёт- ный	<pre> classDiagram class Продавец { - сумма_продаж : int + Получить_данные() : string + Работать() : void } class Работник { - фио : string - дата_устройства : DateTime - оклад : int + Получить_данные() : string + Работать() : void } class Грузчик { - суммар_вес_груза : int + Получить_данные() : string + Работать() : void } Продавец < -- Работник Работник < -- Грузчик </pre>
Чёт- ный	<pre> classDiagram class Собака { - обученность : bool + Получить_данные() : string + Издавать_звуки() : string } class Домаш_животное { - имя : string - возраст : int - вес : int + Получить_данные() : string + Издавать_звуки() : void } class Кошка { - пушистость : bool + Получить_данные() : string + Издавать_звуки() : void } Собака < -- Домаш_животное Домаш_животное < -- Кошка </pre>

ЛАБОРАТОРНАЯ РАБОТА №12. РАБОТА С ИНТЕРФЕЙСАМИ В ПРИЛОЖЕНИЯХ НА ЯЗЫКЕ C#

1. ЦЕЛЬ И ЗАДАЧИ РАБОТЫ

Цель работы – приобрести умение работать с интерфейсами при разработке приложений на языке C#.

Основные задачи:

- научиться создавать собственные интерфейсы на C# и реализовывать их в классах и структурах.

2. ОСНОВНЫЕ ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

2.1. Объявление и реализация интерфейсов

Объявление интерфейса в C#. Сравнение интерфейсов и абстрактных классов.

Одним из наиболее важных средств языка C# являются интерфейсы, которые позволяют определить, что именно должен делать класс (структура), но не как он должен это делать. Для этого интерфейс должен быть объявлен предком выбранного класса.

Благодаря поддержке интерфейсов в C# может быть в полной мере реализовано главное положение полиморфизма: один интерфейс – много реализаций.

Под ***интерфейсом*** в языке C# понимают именованный набор сигнатур методов. То есть внутри интерфейса следует перечисление заголовков методов без их реализации.

Интерфейсы в языке C# объявляются с помощью ключевого слова **interface**. Синтаксис объявления интерфейса аналогичен синтаксису класса и имеет следующий вид:

```

[атрибуты]
[модификаторы] interface Имя_интерфейса [:
Предки]
{
```

```

    Возвр_тип Имя_метода1([список_параметров]);
    Возвр_тип Имя_метода2([список_параметров]);
    . . .
    Возвр_тип Имя_методаN([список_параметров]);
}

```

Для интерфейсов могут быть указаны модификаторы **new**, **public** (по умолчанию), **protected**, **internal** и **private**. Модификатор **new** применяется для вложенных интерфейсов.

По соглашению имена всех интерфейсов .NET снабжаются префиксом в виде заглавной буквы «I». При создании собственных специальных интерфейсов также рекомендуется следовать этому соглашению.

Предками интерфейса могут выступать другие интерфейсы.

Помимо методов, в интерфейсах также можно указывать свойства, индексаторы и события.

Модификатор доступа у элементов интерфейса не указывается, поскольку элементы интерфейса всегда являются открытыми. Кроме того, ни один из элементов интерфейса не может быть объявлен с модификаторами **virtual** и **static**.

С точки зрения синтаксиса интерфейсы подобны абстрактным классам. Можно отметить следующие отличия интерфейса от абстрактного класса:

- если класс наследует абстрактный класс, то он может реализовать лишь некоторые методы родительского абстрактного класса, оставаясь абстрактным классом; класс, реализующий интерфейс, обязан полностью реализовать все методы интерфейса;
- абстрактный класс представляет собой начальный этап проектирования иерархии классов, которые в будущем получат конкретную реализацию; интерфейсы задают дополнительные свойства разных иерархий классов;
- элементы интерфейса по умолчанию имеют модификатор доступа **public** и не могут иметь модификаторов, заданных явным образом;
- интерфейс не может иметь полей и обычных методов — все элементы интерфейса должны быть абстрактными;

- интерфейс не может иметь перегруженных операций, что вызвано возможной несовместимостью с другими языками .NET (например, с Visual Basic .NET, который не поддерживает перегрузку операций).

Интерфейсы чаще всего используются для задания общего поведения объектов различных иерархий. Если определённый набор действий имеет смысл только для какой-то конкретной иерархии классов, реализующих эти действия разными способами, то уместнее задать этот набор в виде виртуальных методов абстрактного базового класса иерархии.

В настоящее время интерфейсы являются незаменимым инструментом в различных шаблонах проектирования, которые позволяют создавать большие, но при этом очень гибкие и расширяемые приложения.

Реализация интерфейса. Интерфейсные свойства и индексы.

Как только интерфейс будет определён, он может быть указан в качестве предка одного или нескольких классов:

```
class Car : IControl
{ ... }
```

Принято говорить, что класс, построенный на базе интерфейса, *реализует* этот *интерфейс*.

Класс, реализующий интерфейс, воплощает методы, описанные в интерфейсе. Класс не может не реализовать какой-либо из методов интерфейса.

В отличие от наследования класс может реализовывать более одного интерфейса. В этом случае все реализуемые в классе интерфейсы указываются списком через запятую. Кроме того, у класса может быть указан базовый класс и в тоже время реализован один или более интерфейсов. В таком случае имя базового класса должно быть указано перед списком интерфейсов:

```
class CivilBuilding : Building, IConstructable, IComparable
{ ... }
```

Интерфейсы могут наследовать друг друга точно так же, как классы. При этом интерфейс может иметь сколько угодно интерфейсов-предков. Базовые интерфейсы должны быть доступны не в меньшей степени, чем их потомки. Например, нельзя использовать интерфейс, объявленный с модификатором **private** или **internal**, в качестве базового для открытого (**public**) интерфейса.

В интерфейсе-потомке можно указывать элементы, переопределяющие унаследованные элементы с той же сигнатурой. В этом случае перед элементом указывается ключевое слово **new**.

При реализации элемента интерфейса имеется возможность указать его имя полностью вместе с именем самого интерфейса. В этом случае получается *явная реализация* элемента интерфейса. При этом модификаторы доступа не указываются. К таким элементам можно обращаться в программе только через объект типа интерфейса. Простой пример явной реализации элемента интерфейса представлен Листинг 12.1.

Листинг 12.1. Пример явной реализации метода интерфейса

```

// Интерфейс
interface IMyInterface
{ double MyMethodA(double x);
}

// Класс, реализующий интерфейс
class MyClass : IMyInterface
{
    // Обычная реализация метода
    Public double MyMethodA(double x)
    { return (1 / x);
    }
    // Явная реализация метода
    Double IMyInterface.MyMethodA(double x)
    { return (1 / (x * x));
    }
}

class Program
{static void Main(string[] args)
    {IMyInterface i1;
      MyClass m1 = new MyClass();
      Console.WriteLine("вызов не явной реализации интерфейса
{0}", m1.MyMethodA(2));
    }
}

```

```
        i1=new MyClass();
        Console.WriteLine("вызов явной реализации интер-
фейса {0}", i1.MyMethodA(2));
        Console.ReadKey();

    }
}
```

Явная реализация элементов интерфейса может использоваться по двум причинам:

- При явной реализации элемент интерфейса не становится открытым элементом класса. Это позволяет закрыть в классе элементы интерфейса, не требуемые конечному пользователю.
- В одном классе могут быть реализованы два интерфейса с методами, имеющими одинаковые имена и сигнатуры. Неоднозначность в этом случае устраняется благодаря указанию в именах этих методов их соответствующих интерфейсов.

Аналогично методам, свойства в интерфейсе указываются без тела. Общая форма объявления интерфейсного свойства выглядит следующим образом:

```
Тип_возврата Имя_свойства { get; set; }
```

В определении интерфейсного свойства, доступного только для чтения или только для записи, должен присутствовать единственный аксессор: **get** или **set** соответственно.

Несмотря на то, что объявление интерфейсного свойства очень похоже на объявление автоматически реализуемого свойства в классе, между ними существует различие. При объявлении в интерфейсе свойство не становится автоматически реализуемым. В этом случае указывается только имя и тип свойства, а его реализация предоставляется классу. Кроме того, при объявлении свойства в интерфейсе не разрешается указывать модификаторы доступа для аксессоров.

В интерфейсе также можно указать индексаторы. Ниже приведена общая форма интерфейсного индексатора:

```
Тип_возврата this[int индекс] { get; set; }
```

В объявлении интерфейсных индексаторов, доступных только для чтения или только для записи, должен присутствовать единственный аксессор: **get** или **set** соответственно.

В среде Visual Studio добавление модулей интерфейсов в проект осуществляется аналогично модулям классов. Для этого выбираем в меню **Project (Проект)** команду **Add New Item (Добавить новый элемент...)**. В открывшемся диалоговом окне (рис. 12.1) выбираем шаблон **Interface (Интерфейс)** и нажимаем кнопку **Add (Добавить)**.

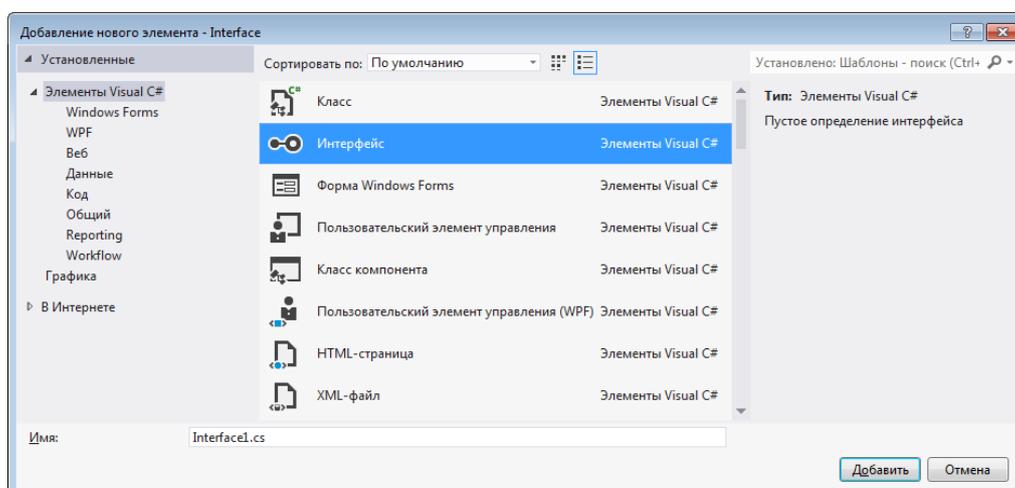


Рисунок 12.1. Окно **Add New Item** (Добавление нового элемента) с выбранным шаблоном **Interface** (Visual Studio 2012)

Пример 12.1. Создание и реализация интерфейсов на языке C#.

Требуется разработать библиотеку классов, в которой заданы классы **Person** (человек), **Client** (клиент банка) и **Firm** (фирма). Класс **Client** является производным от класса **Person**.

Класс **Client** должен реализовывать интерфейс **IAccount** (счёт в банке), в котором должны быть определены следующие элементы:

- **string AccNumber** – свойство, возвращающее номер счёта;
- **int CurrentSum** – свойство, возвращающее текущую сумму;
- **int Percentage** – свойство, возвращающее процент начислений;

- **void Put(int sum)** – метод, обеспечивающий пополнение счёта на сумму **sum**;
- **void Withdraw(int sum)** – метод, позволяющий снять со счёта сумму **sum**.

Класс **Firm** должен реализовывать интерфейсы **ICommercial** (торговая организация) и **IAccount**. В интерфейсе **ICommercial** должны быть определены следующие элементы:

- **int Funds** – свойство, возвращающее размер денежного капитала фирмы;
- **void Buy(int sum)** – метод, позволяющий купить товары на сумму **sum**;
- **void Sell(int sum)** – метод, обеспечивающий продажу товаров на сумму **sum**.

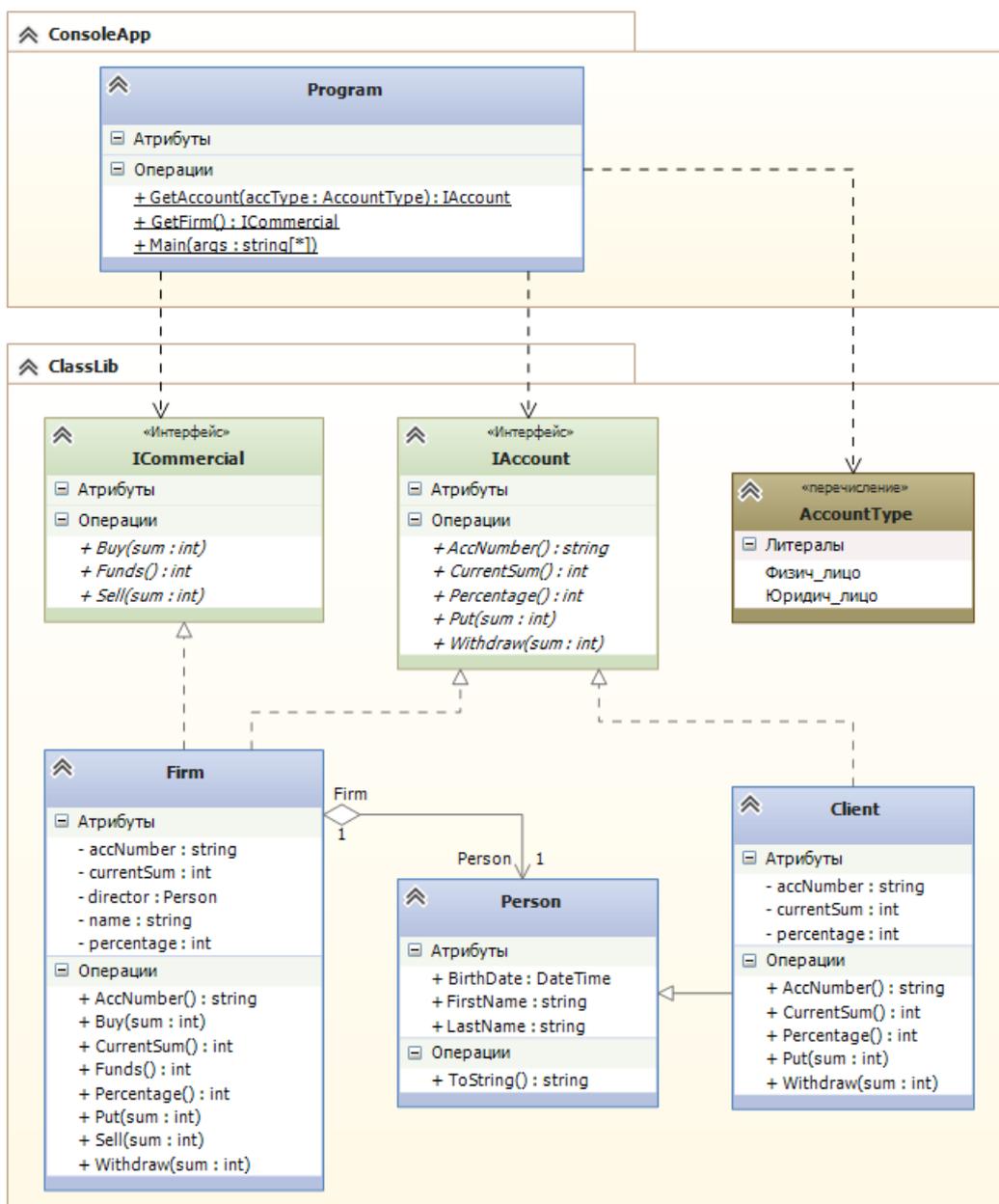


Рисунок 12.2. Диаграмма классов решения

Исходный код интерфейса **IAccount** приведён в Листинг 12.2, а интерфейса **ICommercial** – в Листинг 12.6.

Листинг 12.2. Исходный код интерфейса **IAccount**

И так определяем интерфейс **IAccount**

```

public interface IAccount
{
    string AccNumber { get; } // свойство для номера акканута строковое.
    int CurrentSum { get; } // свойство для текущей суммы
    int Percentage { get; } // .. для процентов
}
  
```

```

void Put(int sum); // метод без реализации для внесения
void Withdraw(int sum); // метод без реализации для снятия
}

```

*Листинг 12.3. Исходный код интерфейса **ICommercial***

```

public interface ICommercial
{
    int Funds { get; }
    void Buy(int sum); // метод без реализации для покупки
    void Sell(int sum); // для продажи
}

```

Исходные коды класса **Person**, а также производного от него класса **Client**, реализующего интерфейс **IAccount**, представлены в Листинг 12.4 и 12.5.

*Листинг 12.4. Исходный код класса **Person***

```

class person
{
    // определяем свойства. Видно без комментариев что за свойства
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public DateTime BirthDate { get; set; }
    // полный конструктор
    public person(string firstName, string lastName, DateTime birthDate)
    {
        FirstName = firstName;
        LastName = lastName;
        BirthDate = birthDate;
    }
    // Переопределяем конструктор по умолчанию
    public person()
    {
        FirstName = "Иван";
        LastName = "Иванов";
        BirthDate = new DateTime(1990, 01, 01);
    }
    // метод для вывода данных о персоне (получаем через переопределение ToString())
    public override string ToString()
    {
        return (string.Format("-- Имя: {0}\n" +
            "-- Фамилия: {1}\n " +
            "-- Дата рождения: {2}",
            FirstName, LastName, BirthDate.ToShortDateString()));
    }
}

```

*Листинг 12.5. Исходный код класса **Client** (часть 1) – наследник от персоны и **IAccount***

```

class Client : person, IAccount
{
    string accNumber;
    int currentSum;
    int percentage;
}

```

```

// Конструктор полный. Обратите внимание в базовый конструктор, то есть конструктор
// person() передаются параметры Fname, IName, birthDate
public Client(string Fname, string IName, DateTime birthDate,
              string accNumber, int currentSum, int persentage)
    : base(Fname, IName, birthDate)
{
    this.accNumber = accNumber;
    this.currentSum = currentSum;
    this.persentage = persentage;
}
// конструктор по умодлчанию
public Client() : this("Иванов", "Иван", new DateTime(1990, 01, 01),
"000000", 0, 1)
{ }
// реализация свойства AccNumber (объявленное в IAccount) возвращает accNumber
// объявленное в данном классе, аналогично остальные свойства
public string AccNumber
{ get { return accNumber; }
}
public int CurrentSum
{ get { return currentSum; }
}
public int Percentage
{ get { return persentage; }
}
// реализуем метод Put, объявленный в IAccount
// реализация простая к текущей currentSum добавл. переданная параметром int sum
public void Put(int sum)
{ currentSum += sum;
}
// аналогично реализуем метод // реализуем метод Put, объявленный в IAccount
// объявленный в IAccount

public void Withdraw(int sum)
{ if (sum <= currentSum) currentSum -= sum;
}
// метод для вывода данных о клиенте
public override string ToString()
{
    return ("Данные о клиенте :\n" + base.ToString() +
string.Format("-- Номкр счёта : {0}\n" +
"-- Текущая сумма : {1} руб.\n " +
"-- Процент начислений: {2} руб",
accNumber, currentSum, persentage));
}
} // Всё просто!!!!!!

```

Исходный код класса **Firm**, реализующего интерфейсы **ICommercial** и **IAccount**, представлен в Листинг 12.6.

Листинг 12.6. Реализация интерфейсов ICommercial и IAccount

```

class Firm : IAccount, ICommercial
{
    string name;
    person director;
    int funds;
    string accNumber;
}

```

```

int currentSum;
int persentage;
public Firm(string name, person director, int funds, string accNumber,
int currentSum, int persantage)
{
    this.name = name;
    this.director = director;
    this.funds = funds;
    this.accNumber = accNumber;
    this.currentSum = currentSum;
    this.persentage = persentage;
}
// реализация свойства получения данных о фондах funds
public int Funds
{ get { return funds; } }
public void Buy(int sum)
{ if (sum <= funds) funds -= sum;
}
public void Sell(int sum)
{ funds += sum;
}
public string AccNumber
{ get { return accNumber; }
}
public int CurrentSum
{ get { return currentSum; }
}
public int Percentage
{ get { return persentage; }
}
public void Put(int sum)
{ currentSum += sum;
}
void IAccount.Withdraw(int sum)
{ if (sum <= currentSum) currentSum -= sum;
}
// методд для получение данных о фирме также переопределяем string ToString()
public override string ToString()
{ return ("Данные о фирме :\n" +
string.Format("-- Название : {0}\n" +
"-- директор {1}\n" +
"-- Капиталл : {2} руб.\n " +
"-- Номер счёта : {3} \n" +
"-- Сумма на счету: {4} руб",
name, director.LastName, funds, accNumber, currentSum));
}
}

```

Исходный код класса **Program** консольного приложения представлен в листинге Листинг 12.7.

Листинг 12.7. Исходный код класса *Program* (часть 1)

```
// создаём нового клиента. Обратите внимание!! создаём в переменную
// объявленную как IAccount acc1
// так можно, но будет доступно только то что есть в IAccount
IAccount acc1 = new Client();
// используем метод ToString() этот метод тоже доступен так как этот метод объявлен
// вообще в Object, но реализация его будет та, которая сделана в классе Client
Console.WriteLine(acc1.ToString());
// используем метод .Put метод доступен, так как он есть в IAccount
acc1.Put(3500);
Console.WriteLine("После добавления суммы {0} руб на счёт {1} ", 3500,
acc1.AccountNumber);
Console.WriteLine(acc1.ToString());
Console.WriteLine("Снимаем сумму {0} руб у клиента {1} ", 1500,
acc1.AccountNumber);
// метод снятия.
acc1.Withdraw(1500);
Console.WriteLine(acc1.ToString());
Console.WriteLine();
Console.WriteLine("Фирму создаём ");
// создаём фирму на переменную Firm acc3
Firm acc3 = new Firm("Простор", new person("Сидор", "Сидоров", new
DateTime(1990, 03, 15)), 2000, "000001", 1, 1);
// переносим фирму на переменную ICommercial acc4
// теперь будет доступно только то что определено в ICommercial
ICommercial acc4 = acc3;
IAccount acc5 = acc3;
Console.WriteLine("Фирма через аккаунт" + acc5.ToString());
Console.WriteLine("Фирма через фирму" + acc3.ToString());
acc5.Put(1000);
Console.WriteLine("Фирма через аккаунт после добавления 1000 " +
acc5.ToString());
Console.WriteLine("Фирма через фирму после добавления 1000" +
acc5.ToString());
acc4.Buy(500);
string s = acc5.ToString();

Console.WriteLine("Фирма через Icommer после покупки чего то в сумме 500"
+ acc5.ToString());
// и т.д.
// ваша задача создать конечные экземпляры и попробовать их методы и показать что они
// работают.

Console.ReadKey();
```

Результат работы консольного приложения – Рисунок 12.3.



```

Данные о клиенте :
-- Имя: Иванов
-- Фамилия: Иван
-- Дата рождения: 01.01.1990-- Номкр счёта : 000000
-- Текущая сумма : 0 руб.
-- Процент начислений: 1 руб
После добавления суммы 3500 руб на счёт 000000
Данные о клиенте :
-- Имя: Иванов
-- Фамилия: Иван
-- Дата рождения: 01.01.1990-- Номкр счёта : 000000
-- Текущая сумма : 3500 руб.
-- Процент начислений: 1 руб
Снимаем сумму 1500 руб у клиента 000000
Данные о клиенте :
-- Имя: Иванов
-- Фамилия: Иван
-- Дата рождения: 01.01.1990-- Номкр счёта : 000000
-- Текущая сумма : 2000 руб.
-- Процент начислений: 1 руб

Фирму создаём
Фирма через аккоунтДанные о фирме :
-- Название : Простор
-- директор Сидоров
-- Капиталл : 2000 руб.
-- Номер счёта : 000001
-- Сумма на счету: 1 руб
Фирма через фирмуДанные о фирме :
-- Название : Простор
-- директор Сидоров
-- Капиталл : 2000 руб.
-- Номер счёта : 000001
-- Сумма на счету: 1 руб
Фирма через аккоунт после добавления 1000 Данные о фирме :
-- Название : Простор
-- директор Сидоров
-- Капиталл : 2000 руб.
-- Номер счёта : 000001
-- Сумма на счету: 1001 руб
Фирма через фирму после добавления 1000Данные о фирме :
-- Название : Простор
-- директор Сидоров
-- Капиталл : 2000 руб.
-- Номер счёта : 000001
-- Сумма на счету: 1001 руб
Фирма через Isomteg после покупки чего то в сумме 500Данные о
-- Название : Простор
-- директор Сидоров
-- Капиталл : 1500 руб.
-- Номер счёта : 000001
-- Сумма на счету: 1001 руб

```

Рисунок 12.3. Работа консольного приложения

3. ПОРЯДОК ВЫПОЛНЕНИЯ РАБОТЫ И ВАРИАНТЫ ЗАДАНИЙ

Основные этапы выполнения работы.

Данная лабораторная работа предполагает выполнение следующих этапов:

1. Изучить методические указания к лабораторной работе.
2. Реализовать заданный интерфейс в указанных классах.

Создать библиотеку с указанными классами и интерфейсами (табл. 12.1). Реализовать интерфейсы в подходящих для них классах. Разработать проект консольного приложения для работы с полученными классами.

Индивидуальные варианты заданий.

Таблица 12.1.

Варианты заданий на реализацию интерфейсов

№ вар.	Классы	Интерфейсы
1, 9, 17	<i>Здание</i> (адрес, площадь, число этажей, дата постройки, полу-	<i>Ремонтируемый</i> (износ, определить стоимость ремонта, ре-

№ вар.	Классы	Интерфейсы
	<p>чить данные).</p> <p><i>Автомобиль</i> (регистр номер, марка, дата выпуска, пробег, получить данные).</p>	<p>монтировать).</p> <p><i>Заправляемый топливом</i> (текущий уровень топлива, объём топливного бака, заправить топливом).</p>
2, 10, 18	<p><i>Книга</i> (название, автор, издательство, год выпуска, число страниц, получить данные).</p> <p><i>Электрочайник</i> (модель, цвет, объём, мощность, получить данные).</p>	<p><i>Товар</i> (цена, скидка, производитель, дата выпуска, определить цену с учётом скидки).</p> <p><i>Потребляющий электроэнергию</i> (напряжение питания, сила тока, подключить к сети, отключить от сети, определить затраты энергии).</p>
3, 11, 19	<p><i>Квартира</i> (адрес дома, номер, этаж, площадь, число комнат, получить данные).</p> <p><i>Локомотив</i> (модель, год выпуска, мощность, максимальная скорость, получить данные).</p>	<p><i>Ремонтируемый</i> (износ, определить стоимость ремонта, ремонтировать).</p> <p><i>Перемещаемый в пространстве</i> (текущие координаты, заданные координаты, двигаться к цели).</p>
4, 12, 20	<p><i>Принтер</i> (модель, скорость печати, объём картриджа, максимальное число листов в лотке, получить данные).</p> <p><i>Обувь</i> (название, сезон, материал, цвет, размер, получить данные).</p>	<p><i>Товар</i> (цена, скидка, производитель, дата выпуска, определить цену с учётом скидки).</p> <p><i>Очищаемый</i> (степень загрязнения, очистить, определить время очистки).</p>
5, 13, 21	<p><i>Трамвай</i> (код, модель, год выпуска, мощность, максимальная скорость, получить данные).</p> <p><i>Станок</i> (тип, модель, мощность привода, точность, получить данные).</p>	<p><i>Потребляющий электроэнергию</i> (напряжение питания, сила тока, подключить к сети, отключить от сети, определить затраты энергии).</p> <p><i>Перемещаемый в пространстве</i> (текущие координаты, заданные координаты, перемещать к цели).</p>
6, 14, 22	<p><i>Дорога</i> (длина, ширина, материал полотна, получить данные).</p> <p><i>Обувь</i> (название, сезон, материал, цвет, размер, получить данные).</p>	<p><i>Ремонтируемый</i> (износ, определить стоимость ремонта, ремонтировать).</p> <p><i>Товар</i> (цена, скидка, производитель, дата выпуска, определить цену с учётом скидки).</p>

№ вар.	Классы	Интерфейсы
7, 15, 23	<p><i>Автомобиль</i> (регистр номер, марка, дата выпуска, пробег, получить данные).</p> <p><i>Квартира</i> (адрес дома, номер, этаж, площадь, число комнат, получить данные).</p>	<p><i>Очищаемый</i> (степень загрязнения, очистить, определить время очистки).</p> <p><i>Заправляемый топливом</i> (текущий уровень топлива, объём топливного бака, заправить топливом).</p>
8, 16, 24	<p><i>Грузовой контейнер</i> (код, перевозчик, получатель груза, вес груза, получить данные).</p> <p><i>Пассажирский самолёт</i> (модель, авиакомпания, макс. число пассажиров, максимальная скорость).</p>	<p><i>Перемещаемый в пространстве</i> (текущие координаты, заданные координаты, перемещать к цели).</p> <p><i>Заправляемый топливом</i> (текущий уровень топлива, объём топливного бака, заправить топливом).</p>

Требования к отчёту.

Отчёт по самостоятельной практической работе должен содержать следующие пункты:

1. Титульный лист с указанием названия работы, фамилии и инициалов выполнившего работу студента и проверившего её преподавателя.
2. Цель и задачи работы.
3. Исходный код заданных интерфейсов и реализующих эти интерфейсы классов на языке C#, а также код консольного приложения, демонстрирующего работу с экземплярами классов.

4. КОНТРОЛЬНЫЕ ВОПРОСЫ И ЗАДАЧИ

Теоретические вопросы.

1. Какой тип называют интерфейсом в языке C#?
2. Для чего применяют интерфейсы?
3. Как объявляется интерфейс в программе на C#?
4. Чем интерфейс отличается от абстрактного класса?
5. Что такое реализация интерфейса?

ЛАБОРАТОРНАЯ РАБОТА №13. РАБОТА С СОБЫТИЯМИ СРЕДСТВАМИ C#

1. ЦЕЛЬ И ЗАДАЧИ РАБОТЫ

Цель работы – приобрести умение работать с событиями разработке приложений на языке C#.

Основные задачи:

- научиться создавать собственные события, их обработчики средствами C#;

2. ОСНОВНЫЕ ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

2.1. Общие сведения

Событие (Event) – это описание существенного факта, который занимает некоторое положение во времени и в пространстве, который может вызвать изменение состояние системы или её элементов.

В контексте автоматов событие – это стимул, который может вызвать переход из одного состояния в другое.

При данном подходе Система становится событийно управляемой, поэтому разработчикам зачастую важно знать, как должен реагировать тот или иной объект на определенные события. Инициаторами событий могут быть как объекты самой Системы, так и её внешнее окружение.

2.2. Программная реализация событий вызова

При описании событий выделяется классы отправитель события и класс-приёмник события.

Класс-отправитель(sender) (или издатель) события (или сообщения), это класс, экземпляры которого включают, вырабатывают событие.

Класс, чьи экземпляры получают сообщения, реагируют на них, называют *классом-получателем* (подписчиком) *сообщения* (receiver).

Результатом оповещения объектов о событии должен быть запуск определённого кода, который называют *обработчиком событий* (англ. *event handler*).

События работают по следующему принципу: объекты, которые должны реагировать на событие содержат специальный метод-обработчик этого события. Когда событие происходит, то вызываются все зарегистрированные обработчики этого события (обработчик может быть не один).

Определение события.

В языке C# каждое событие определяется типом делегата.

Тип делегат задаёт сигнатуру (состав параметров и возвращаемое значение) методов, которые могут быть связаны с событием.

Для определения типа делегата в C# используется ключевое слово **delegate**. Синтаксис объявления делегата имеет следующий вид:

```
[модификаторы]           delegate           Тип_возвр
Имя_делегата([параметры]);
```

Модификаторы типа делегата имеют тот же смысл, что и для класса, причём допускаются только модификаторы **new**, **public**, **protected**, **internal** и **private**.

Тип_возвр описывает возвращаемое значение методов, вызываемых с помощью делегата. *Параметрами* делегата являются параметры соответствующих ему методов.

Объявление делегата можно размещать непосредственно в пространстве имен или внутри класса.

События с точки зрения программной реализации являются экземплярами (переменными) делегата.

События являются элементами класса и объявляются с помощью ключевого слова **event**. Синтаксис объявления события имеет следующий вид:

```
:
```

```
[модификаторы] event [тип_события] имя_события;
```

Тип события – это тип делегата, на котором основано событие, а **имя события** – конкретный экземпляр объявляемого события. Событие по существу является экземпляром делегата, то есть может хранить ссылки на методы.

Создание события состоит из следующих этапов:

1. Объявление типа делегата – функционального класса, задающего сигнатуру. Для некоторых событий можно использовать стандартные делегаты. В этом случае достаточно только знать их имена.
2. Объявление события в классе **sender** как экземпляра соответствующего делегата. При этом добавляется ключевое слово **event**.
3. Объявление методов, инициирующих событие.
4. Связь событий делегата с методами обработчиками в классах объектов, получателей событий.

Пример объявления делегата и события, соответствующего типу этого делегата, а также обработчика события представлен в листинге 13.1.

Листинг 13.1. Пример объявления делегата, события и обработчика

```
// Тип делегата для события с параметром целого типа
public delegate void MyDelegate(int t);

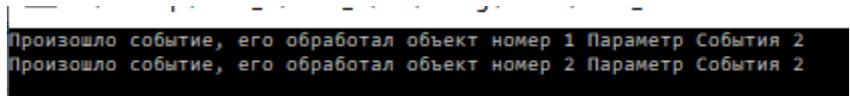
// Объявление события и методов в котором генерируются
события
class ClassWithEvent
{   public event MyDelegate MyEvent; //-Объявление события
    public void MakeEvent(int par)// Метод, выработки события
    {
        if (MyEvent != null) MyEvent(par);
    }
}

// Класс – получатель события
class ClassTargetEvent
{   int n = 0; //номер объекта (как имя используем)
    static int colObj = 0;
}
```

```
//В конструкторе в качестве параметра передаётся экземпляр
объекта с которым связано событие
public ClassTargetEvent(int nn, ClassWithEvent d)
{   n = nn;
    d.MyEvent += MyHandler; //связь события с обработчиком
}

void MyHandler(int p) // обработчик события
{ Console.WriteLine "Событие - параметр {0} - объ {1} ", p, n);
  }
}

//Ниже приведён код из класса Program,
//реализующий создание необходимых экземпляров
// создание экземпляра класса источника события
ClassWithEvent evt = new ClassWithEvent();
// создание экземпляров - приёмников события
ClassTargetEvent d = new ClassTargetEvent(1,evt);
ClassTargetEvent d1 = new ClassTargetEvent(2, evt);
evt.MakeEvent(2); //Вызов метода выработки события с
// с параметром 2
Console.ReadKey();
```



```
Произошло событие, его обработал объект номер 1 Параметр События 2
Произошло событие, его обработал объект номер 2 Параметр События 2
```

Рисунок 13.1. Результаты работы программы

3. ПОРЯДОК ВЫПОЛНЕНИЯ РАБОТЫ И ВАРИАНТЫ ЗАДАНИЙ

Основные этапы выполнения работы.

Данная лабораторная работа предполагает выполнение следующих этапов:

1. Изучить методические указания к лабораторной работе.
2. Реализовать заданные классы события и обработчики событий классов.

Индивидуальные варианты заданий.

Таблица 13.1.

Варианты заданий на события.

В соответствии с индивидуальными заданиями нужно определить классы, определить необходимые переменные класса для отображения результатов событий.

№ вар.	Класс источник	Класс приёмник – метод обработчик
1,10	Нагреватель. Событие – повышение температуры на заданное число градусов	Котёл - повышение давления с задержкой на два градуса При повышении температуры с запаздыванием на 10 градусов. Выше ста градусов происходит разрушение
2, 11, 18	Насос. Событие – подача заданного объёма воздуха с определённым давлением	Мяч - повышение давления и объёма с запаздыванием на 10 градусов При повышении давления выше 10 МПа давление и объём падают до нуля
3, 12, 19	Солнце Событие – увеличение угла над горизонтом на заданное число градусов	Одуванчики Изменяется диаметра цветка (раскрытие). Начальное значение 0, максимальное 20 мм
4, 13, 20	Отопительный радиатор (батарея). Событие – повышение температуры в системе на заданное число градусов	Объект находящийся в комнате - нагревание температуры (с отставанием на 10 градусов)
5, 14, 21	Преподаватель Событие – Выдача задания с определённым номером.	Студент Выполнение заданного задания. При выполнении у студента повышается уровень знания по определённому разделу знаний
6, 15, 22	Окружающая среда Событие – Изменение среднесуточной температуры	Дерево При превышении среднесуточной температуры выше 5 градусов Распускание листьев. При понижении - опадание листьев
7, 16, 23	Окружающая сред Событие – Изменение среднесуточной температуры	Открытый водоём При понижении среднесуточной температуры ниже 5 градусов устанавливается ледяной покров. При превышении +5 градусов ледяной покров исчезает
8, 17,	Окружающая сред Событие – изменение среднесуто-	Заяц Изменение цвета шерсти. При пони-

№ вар.	Класс источник	Класс приёмник – метод обработчик
	чной температуры	жении среднесуточной температуры ниже 0 градусов шерсть становится белой. При превышении среднесуточной температуры выше 0 градусов шерсть становится серой
9,18	Судья футбольного матча Показ карточки (жёлтой или красной)	Футболисты Получение предупреждения (покидает поле если карточка красная или две жёлтых карточки)

Требования к отчёту.

Отчёт по самостоятельной практической работе должен содержать следующие пункты:

1. Титульный лист с указанием названия работы, фамилии и инициалов выполнившего работу студента и проверившего её преподавателя.

2. Цель и задачи работы.

3. Исходный код заданных интерфейсов и реализующих эти интерфейсы классов на языке C#, а также код консольного приложения, демонстрирующего работу с экземплярами классов.

4. КОНТРОЛЬНЫЕ ВОПРОСЫ И ЗАДАЧИ

Теоретические вопросы.

1. Общее определение события C#.
2. Как определяется событие C#?
3. Как происходит выработка события?
4. Что должно выполняться в классе обработчике?