Министерство образования Республики Беларусь
Учреждение образования
«Белорусский государственный университет
информатики и радиоэлектроники»

Факультет компьютерных систем и сетей

Кафедра программного обеспечения информационных технологий

**Ю. О. Герман, О. В. Герман**

# *ПРОЕКТИРОВАНИЕ И РАЗРАБОТКА ИНФОРМАЦИОННЫХ СИСТЕМ*

*Рекомендовано УМО по образованию в области информатики и радиоэлектроники в качестве учебно-методического пособия для специальности 1-40 01 01 «Программное обеспечение информационных технологий»*



Минск БГУИР 2020

УДК 004.42(075)
ББК 32.973.26-018.2я73
Г38

Р е ц е н з е н т ы:

кафедра программного обеспечения информационных систем и технологий
Белорусского национального технического университета»
(протокол №6 от 11.12.2019);

доцент кафедры информационных систем и технологий
учреждения образования
«Белорусский государственный технологический университет»
кандидат физико-математических наук, доцент Н. И. Гурин

**Герман, Ю. О.**

Г38 Проектирование и разработка информационных систем : учеб.-метод. пособие / Ю. О. Герман, О. В. Герман. – Минск : БГУИР, 2020. – 128 с. : ил.

ISBN 978-985-543-567-0.

Содержит материалы лекций и лабораторных работ, а также краткие теоретические сведения о современных технологиях проектирования программного обеспечения. Рассмотрены средства проектирования на базе языка UML в Rational Rose, паттерны проектирования, разработка кода на основе спецификаций и правил, основы WWF-технологии. Представлено описание семи лабораторных работ, содержащих теоретическую часть, описание средств C# для решения поставленных задач и указания по выполнению работ.

**УДК 004.42(075)**
**ББК 32.973.26-018.2я73**

**ISBN 978-985-543-567-0**

## СОДЕРЖАНИЕ

# 1 ЭВОЛЮЦИЯ ТЕХНОЛОГИЙ РАЗРАБОТКИ ПРОГРАММ

В первых электронных машинах программы вводились покомандно в двоичном формате с пульта управления оператором. Очевидным шагом вперед стало использование псевдомашинного языка (Ассемблера), где вместо двоичных кодов стали писать текстовые названия, например,

ADD A,5

По этой команде к переменной А добавляется 5, т. к. ADD обозначает «прибавить».

В создании компиляторов решающую роль сыграла разработка Ноэмом Хомским формальной теории языков. Хомский выделяет четыре класса языков: автоматные (регулярные), контекстно-свободные (КС-языки), контекстно-зависимые и языки со свободной структурой. Для целей программирования наибольшее значение имеют первые два класса. Для каждого языка используются определенные математические средства их разбора. Для автоматных языков такими средствами являются детерминированные автоматы без памяти, для КС-языков – недетерминированные автоматы с памятью. Применяются также деревья разбора (например, деревья нисходящего синтаксического разбора) и другие формализмы.

По сути эра автоматического программирования началась с появления первого компилятора – программы, которая переводит код с языка высокого уровня в машинный код. В конце 50-х годов прошлого века первый компилятор был создан для языка Фортран американцами Бэкусом и Науром. В 1960-х годах число языков стремительно растет: ПЛ/1, Алгол, Бейсик, Кобол, РПГ и др. Никлаус Вирт (Швейцария) создает язык Паскаль. В 1970-е годы появляется язык Си (авторы Кен Томпсон и Денис Ритчи). Все эти языки относились к категории процедурно-ориентированных языков. Минимальной единицей кода была процедура (функция, модуль). В языках Паскаль и Си использовались сложные типы данных: записи – в Паскале, структуры – в Си. Преимуществами Си стали наличие указателей (на ячейки памяти), возможность динамически выделять память под переменные и освобождать ее.

Поскольку физические параметры ЭВМ были сравнительно ограниченными, то разработчики старались писать экономные коды с очень неочевидной логикой. Ситуация отягощалась широким использованием оператора goto, способного «перебрасывать» управление в любую точку программы. Основное внимание сконцентрировалось на отладке. Если условно обозначить затраты времени на создание свободной от ошибок программы как 100 %, то отладка стала занимать порядка 80 %. Вопрос о технологии(ях) разработки программ встал в полную силу.

При этом важно учитывать, на какой этап разработки ориентирована та или иная технология. Этапы разработки образуют жизненный цикл программы. Он включает в себя:

• **анализ требований**: определяются и уточняются требования заказчика, разрабатываются соответствующие документы (техническое задание, описание применения и др.);

• **проектирование**: формируются модели будущей системы или другие спецификации, определяются компоненты архитектуры, условия взаимодействия компонентов, представление данных, алгоритмы работы;

• **реализацию** (**кодирование**): осуществляются выбор и обоснование языка(ов) программирования, разработка программных модулей и интерфейса, интеграция модулей;

• **тестирование и отладку**;

• **документирование**: разрабатываются документы на программное обеспечение, включающие описание применения, руководство пользователя, руководство системного программиста, тексты программ и др.;

• **внедрение**;

• **эксплуатацию**;

• **сопровождение**.

Основным нормативным документом, регламентирующим жизненный цикл ПО, является международный стандарт ISO/IEC 12207 (ISO – International Organization of Standardization – Международная организация по стандартизации, IEC – International Electrotechnical Commission – Международная комиссия по электротехнике).

Соответственно этапам жизненного цикла программного обеспечения разрабатывались технологии, которые их поддерживали. Известны следующие основные модели жизненного цикла программы: водопадная (каскадная), водопадная с возвратами и спиральная [1].

Каждый этап водопадной модели завершается выпуском полного комплекта документации, достаточной для продолжения разработки другой командой разработчиков. Недостаток водопадной модели состоит в невозможности возвратов на ранние этапы с целью коррекции или исправления части проекта, которая не была предусмотрена техническим заданием или иной документацией на проектируемую систему. Кроме того, такая модель не предусматривает возможность изменения в ходе разработки концепции системы, что существенно при создании сложных программных комплексов, для которых трудно предусмотреть все аспекты работы. Водопадная модель ориентирована в большей степени на небольшие и средние по сложности проекты программно-информационных систем. В связи с этим появляется водопадная модель с возвратами (рисунок 1.1).

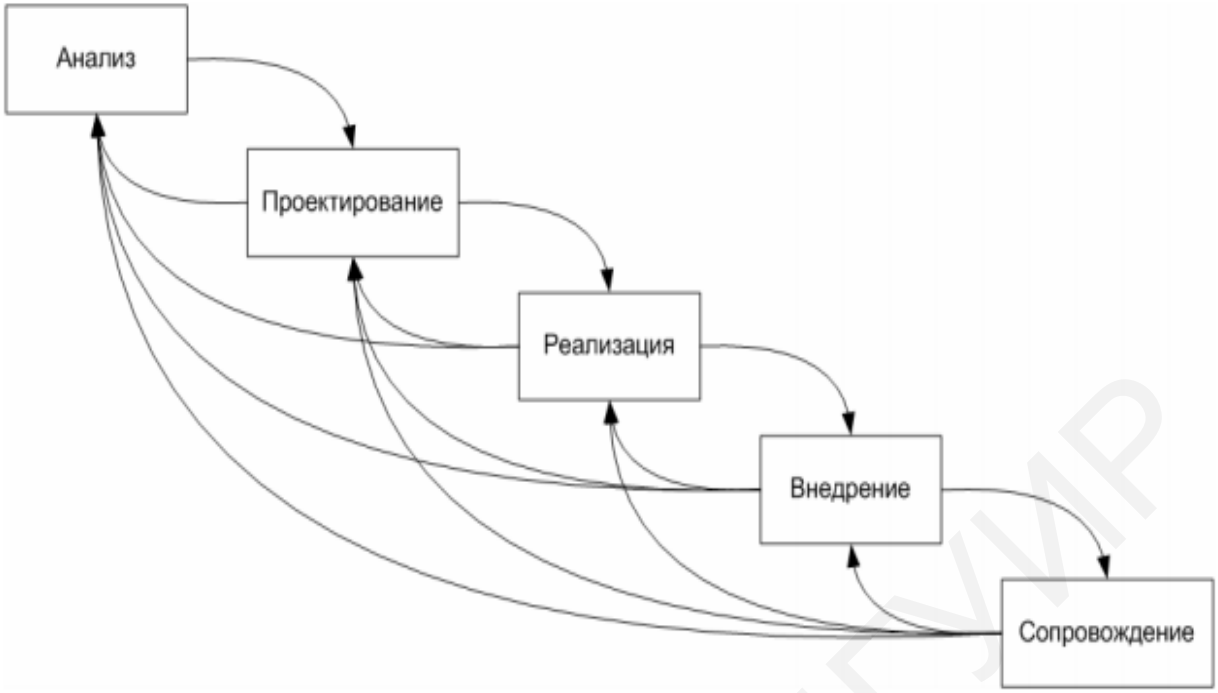



Рисунок 1.1 – Водопадная модель с возвратами

Основной минус водопадной модели с возвратами – проект выполняется по-прежнему достаточно медленно в связи с его этапностью (переход на следующий этап нельзя осуществить до полного завершения предыдущего этапа). Поэтому появляется более прогрессивная форма жизненного цикла – спиральная модель (рисунок 1.2) [1].

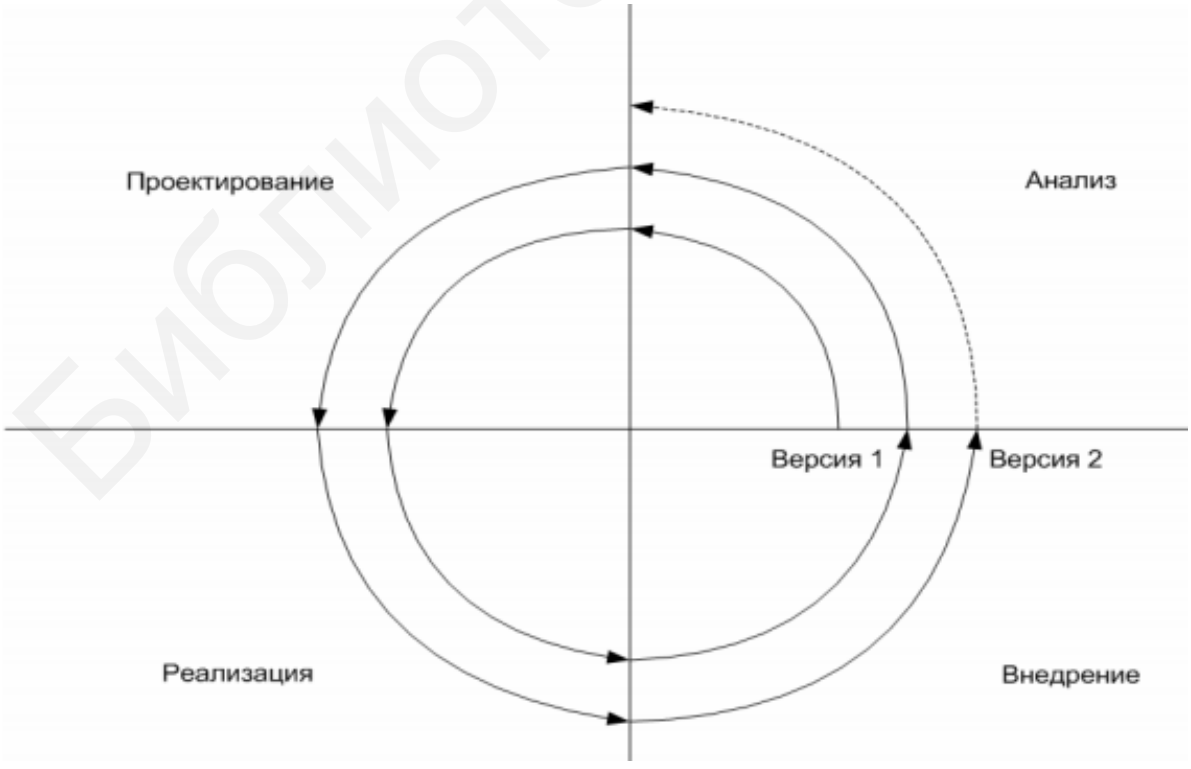



Рисунок 1.2 – Спиральная модель жизненного цикла программы

В спиральной модели сначала создается первая версия системы, затем выполняется доводка компонентов разработанной архитектуры, модернизация и повышение эффективности программных модулей, т. е. реализуется следующий виток, и т. д. Каждый виток спирали соответствует поэтапной модели создания фрагмента или версии программного изделия, на нем уточняются цели и характеристики проекта, определяется его качество, планируются работы следующего витка спирали. Таким образом, углубляются и последовательно конкретизируются детали проекта, и в результате выбирается обоснованный вариант, который доводится до реализации [1].

Рассматривая технологии, поддерживающие этапы жизненного цикла программы, отметим, например, систему SREM, которая использовалась для автоматизации создания и анализа требований к программному проекту и содержала язык определения требований (RSL) к данным и операциям их обработки, посредством которого устанавливались связи между объектами. Предложения RSL обрабатывались с помощью специализированного процессора.

Значительная часть технологий ориентировалась на этап проектирования. Никлаус Вирт предложил концепцию структурного программирования. Оператор goto «изгонялся» из языка программирования. Принципы структурного программирования включали нисходящее проектирование, модульность и пошаговую детализацию. Идеи структурного программирования так или иначе проявились в таких технологиях проектирования программ, как SADT, SREM, методике Джексона и др.

Графический язык системы SADT – это иерархический структурированный набор диаграмм, причем каждый блок диаграммы раскрывается более детально с помощью другой диаграммы. Структура проекта представляется со все большей степенью детализации по мере его разработки (рисунок 1.3) [2].

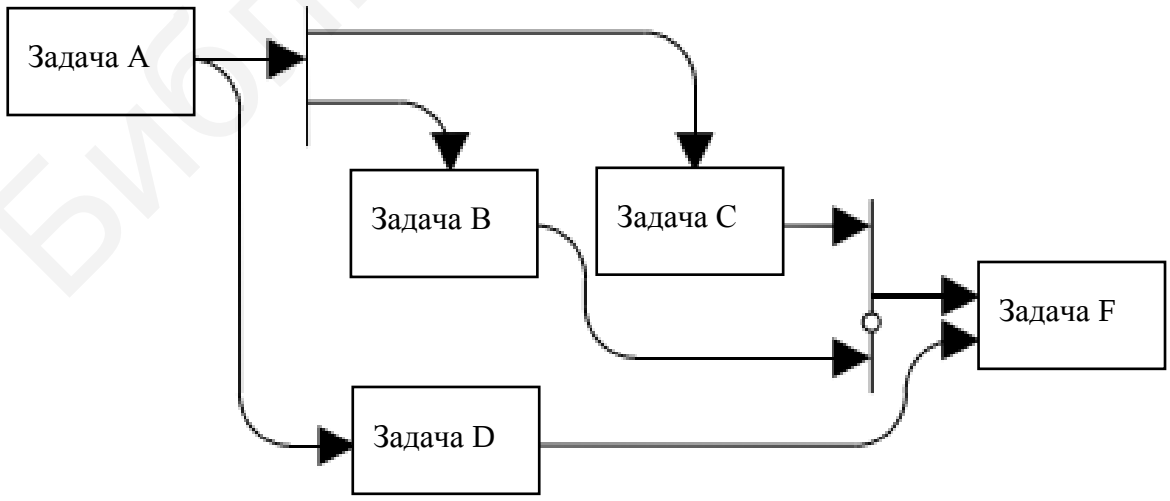



Рисунок 1.3 – Пример пошагового раскрытия взаимосвязей задач в SADT

Примерно в это же время появились работы, связанные с логическим анализом программ и их верификацией [3]. Идея заключалась в том, чтобы наделить операторы логическими условиями, определяющими их срабатывание и завершение (такие условия назывались соответственно предусловиями и постусловиями). Проблема в определенной мере «сосредоточилась» на инварианте цикла – логическом условии, которое характеризует каждый цикл и которое нельзя определить формально на основе алгоритма. Таким образом, трудности определения инвариантов циклов и полуразрешимость логической теории как таковой не позволили поставить технологии верификации на эффективную практическую основу.

Абсолютным прорывом в развитии технологий программирования можно считать создание объектно-ориентированной платформы (ООП) в программировании (Б. Страуструп) и проектировании (Г. Буч). Эта эра длится уже почти 40 лет. Можно считать, что ООП ориентирована главным образом на этапы проектирования и реализации. Необходимость создания ООП продиктована следующими обстоятельствами:

- плохо структурированные программы не позволяли эффективно их модернизировать в последующем;
- вероятность ошибки оставалась достаточно высокой, особенно при изменении условий функционирования;
- тестировать программы было достаточно сложно;
- написание программ выполнялось без привязки к ясной и четко определенной модели предметной области, понятной не только программисту, но и заказчику;
- программы были «привязаны» к разработчику.

Объектно-ориентированная платформа в значительной степени устраняла эти недостатки. Проектирование программ на основе классов привело к появлению паттернов проектирования – типовых шаблонов, которые служили образцами решения задач в достаточно общих ситуациях. Классы выполняли роль строительных блоков и одновременно служили моделью некоторой предметной области. Программирование получило в этом смысле средства для моделирования, а также стало более структурированным и производительным. Концепция объектно-ориентированного программирования была реализована еще в 1960-е годы в языках SmallTalk и Simula, но стала востребована только в недавнее время.

Для конструирования и разработки современных программно-информационных систем используются различные среды и платформы, объединяемые общим термином CASE-системы. Среди наиболее известных отметим такие системы, как Rational Rose, Erwin, BPWin, MS Visio, Power Designer и др. Они позволяют описывать архитектуру создаваемой программно-информационной системы, используя различные диаграммы, блок-схемы и графики, и даже писать программный код «под блоки» диаграмм (например,

такая возможность реализована в системе бизнес-проектирования WWF – Windows Workflow Foundation в среде Visual Studio NET).

Получили развитие технологии тестирования программного обеспечения, например, Rational (Visual Test, Rational Robot, Team Test и др.), Mercury Interactive (WinRunner), Segue Software (QA Partner). Процесс тестирования в таких системах в значительной степени автоматизирован. Подобные системы приближают «покрытие» тестами программы к 100 %.

# 2 ДИАГРАММЫ UML

## 2.1 Краткие теоретические сведения

При проектировании сложных программных систем используются определенные математические и информационно-программные средства, позволяющие описать структуру проекта и его функционально-логическое наполнение. Таковыми средствами являются: языки спецификаций [4, 5], математические модели [6], диаграммы, шаблоны (паттерны) проектирования [7], программные фреймворки типа WWF (Windows Workflow Foundation) и др. Рассмотреть их все в этом пособии не представляется возможным, поэтому мы ограничились языком UML [8], программными паттернами и рекурсивными спецификациями с учетом их доступности, относительной простоты и возможности использования на практических занятиях.

Важной составляющей технологий проектирования программного обеспечения является язык диаграмм. Так, один из ведущих языков проектирования – UML (Universal Modeling Language) содержит следующие типы диаграмм:

- вариантов использования;
- классов;
- объектов;
- активности (деятельности/activity);
- последовательности;
- коммуникаций и др.;
- графы состояний (state-diagram).

Каждый тип диаграмм «схватывает» некоторую характеристику проектируемой системы, но в общем случае диаграммы используют объекты, классы, сообщения, отношения, условия и методы. Диаграммы не содержат способа реализации метода или класса. Они являются графическими моделями. По замыслу разработчиков (прежде всего, отметим Гради Буча), диаграммы нужны для целостного восприятия системы, ее архитектуры и внутренних связей. «Исключение» составляют диаграммы классов, которые используются для создания программных классов при конвертации UML-диаграмм в языки типа C++.

## 2.2 Диаграммы вариантов использования

Эти диаграммы показывают, как разрабатываемая система взаимодействует с ее пользователями. Пользователи называются акторами (actors). Каждый актор использует систему по-своему (реализует вариант использования). Данная диаграмма носит достаточно общий характер, но она уже содержит будущие классы. Рассмотрим пример диаграммы вариантов использования (рисунок 2.1).

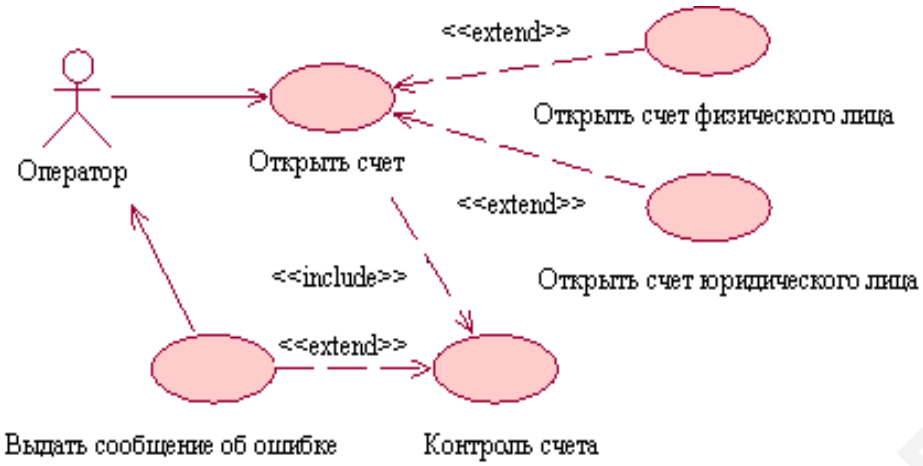



Рисунок 2.1 – Пример диаграммы вариантов использования

В примере на рисунке 2.1 указан только один актор – оператор отделения банка. Определено пять вариантов использования, обозначенных закрашенными эллипсами. Каждый вариант использования указывает некторое действие (операцию), выполняемое актором или системой. Если операция выполняется системой, то стрелка рисуется от эллипса к актору. Если операция выполняется актором, то направление стрелки противоположное. Стрелки передают ассоциативные связи между акторами и вариантами использования либо между самими вариантами использования.

### 2.3 Диаграммы классов

Диаграммы классов являются важнейшим типом диаграмм UML. На этой диаграмме отображаются один или несколько классов и связей между ними. Класс на диаграмме представляется так, как показано на рисунке 2.2.

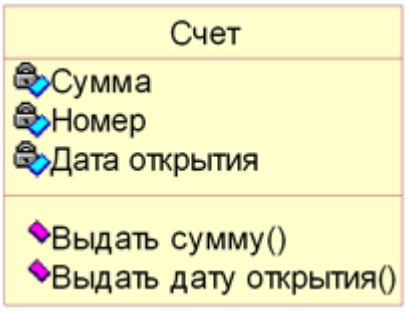



Рисунок 2.2 – Представление класса

Определены три группы полей: имя класса (Счет), переменные класса (Сумма, Номер, Дата открытия) и методы (Выдать сумму и Выдать дату открытия). Между классами могут возникать связи (отношения). Важнейшими

отношениями являются обобщение (быть родовым классом для) и ассоциация (какие-то объекты одного класса могут входить в множество объектов другого класса или иметь с ними какую-то общую характеристику). Отношение обобщения можно передать рисунком 2.3.

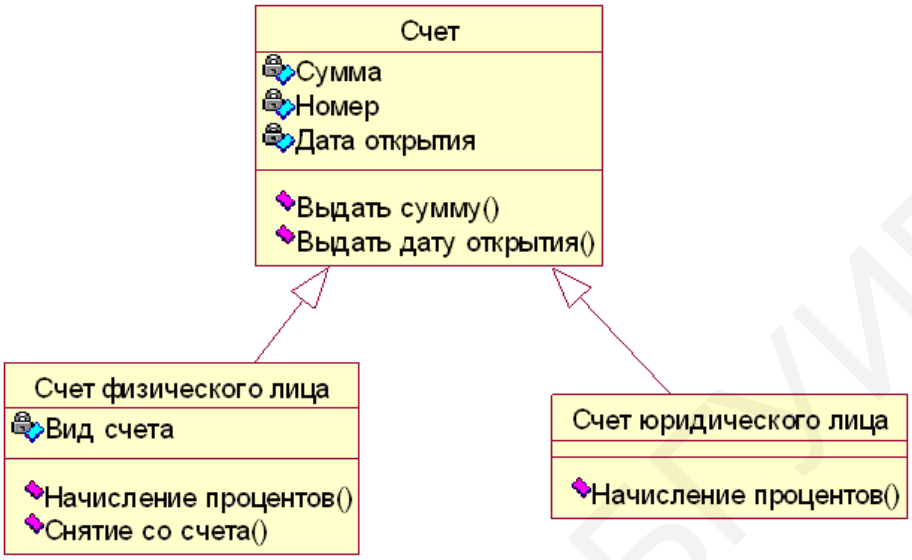



Рисунок 2.3 – Отношение обобщения

Для передачи отношения обобщения используется стрелка с пустым трегольником. Стрелка направлена к родительскому (общему) классу. Так, счет физического лица есть частный случай по отношению к классу Счет. Каждый дочерний класс может наследовать какие-то методы и переменные родительского класса и добавлять к ним свои собственные.

Ассоциация устанавливает семантическую связь между классами. Пример показан на рисунке 2.4.

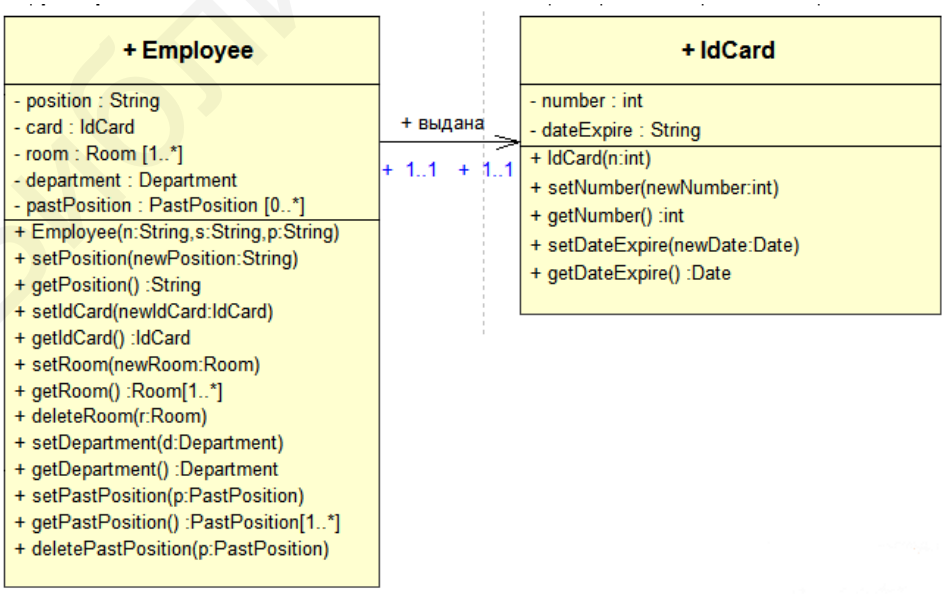



Рисунок 2.4 – Отношение ассоциации

На рисунке 2.4 класс Employee (работник) ассоциирован (связан) с классом IdCard (персональная учетная карточка). Ясно, что каждому работнику соответствует уникальный учетный номер (в данном случае верно и обратное). Отношение ассоциации отображается обычной стрелкой.

Частным вариантом ассоциации является агрегация и зависимость. Агрегация – особая разновидность отношения ассоциации, представляющая структурную связь целого с его частями. Как тип ассоциации, агрегация может быть именованной. Одно отношение агрегации не может включать более двух классов (контейнер и содержимое). Агрегация встречается, когда один класс является коллекцией или контейнером других. Пример агрегации дан на рисунке 2.5.

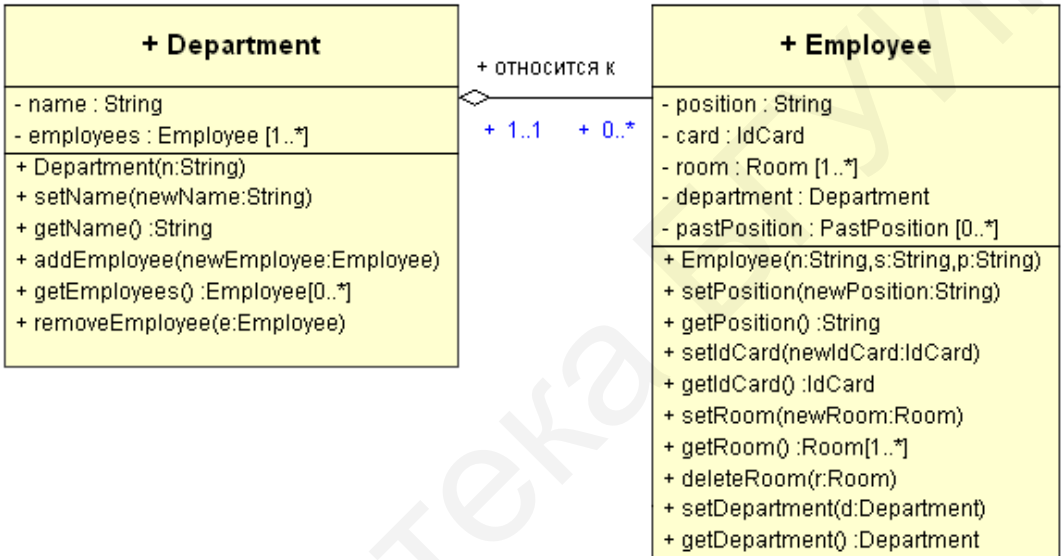



Рисунок 2.5 – Агрегация

Здесь отдел (Department) может содержать несколько работников (объектов класса Employee). Простая агрегация передается ромбом. Другим видом агрегации является композиция. Композиция рисуется с помощью закрашенного ромба. В отличие от простой агрегации, в случае композиции объекты одного класса входят как составная часть в объекты другого, поэтому когда уничтожается контейнер, уничтожаются все содержащиеся в нем объекты. Например: этажи являются частью дома и если дом сносят, то этажи также пропадут.

Наконец, последний вид связи – это связь реализации, показывающая, что один класс реализует «поведение» второго, т. е. реализует методы интерфейса.

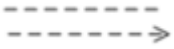
Отношение зависимости передается стрелкой с пунктирной линией, иногда со стрелкой, направленной к той сущности, от которой зависит данная сущность: ---------->

Зависимость – это связь, указывающая, что изменение одной сущности может повлиять на другие сущности, которые используют ее.

Суммируем связи между классами:

 Агрегация (Aggregation) – тип связи «часть – целое», в котором «часть» может существовать отдельно от «целого».

 Композиция (Composition) – тип агрегации, в котором «части» не существуют вне «целого».

 Зависимость (Dependency) – изменение одной сущности (независимой) может влиять на поведение другой сущности (зависимой). Стрелка направлена на независимую сущность.

### 2.4 Диаграммы активности (деятельности)

Диаграмма вариантов использования призвана ответить на вопрос «Что должна делать система?» Диаграмма активности отвечает на вопрос «Как должна система что-то делать». Из названия ясно, что в основе диаграммы активности находятся процессы (операции). Процессы связываются стрелками, устанавливающими порядок их выполнения. Также используются условные вершины для выбора переходов между процессами. Начало и конец диаграммы активности задаются кружками. Пример дан на рисунке 2.6.

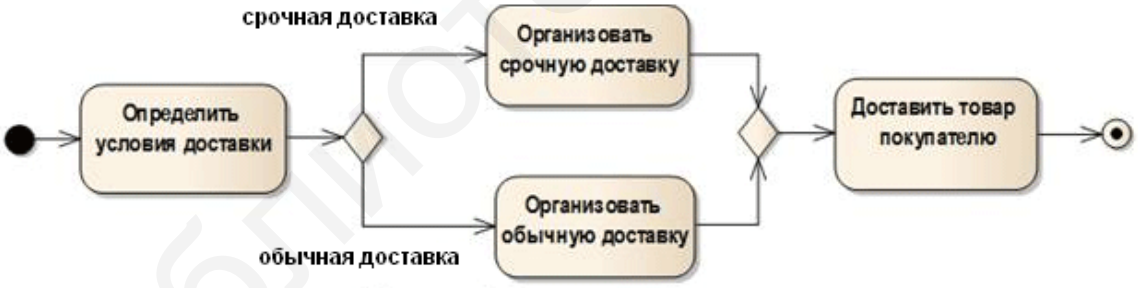



Рисунок 2.6 – Пример диаграммы активности

Начало диаграммы отображается черным кружком, конец – кружком с черной фишкой. Активности представляются овалами. Ромбики представляют логические условия. Очевидно, за данной диаграммой могут стоять классы товара, заказчика, отправителя и др. Диаграмма активности в какой-то мере соответствует обычной блок-схеме. Следующий тип диаграмм – граф состояний – логически связан с диаграммой активности.

## 2.5 Граф состояний (State Diagram)

Диаграммы состояний определяют все возможные состояния, в которых может находиться конкретный объект, а также процесс смены состояний объекта в результате влияния некоторых событий и действий. На рисунке 2.7 показана диаграмма состояний UML, отражающая процесс формирования отчета, включая различные состояния, в которых может находиться этот процесс.

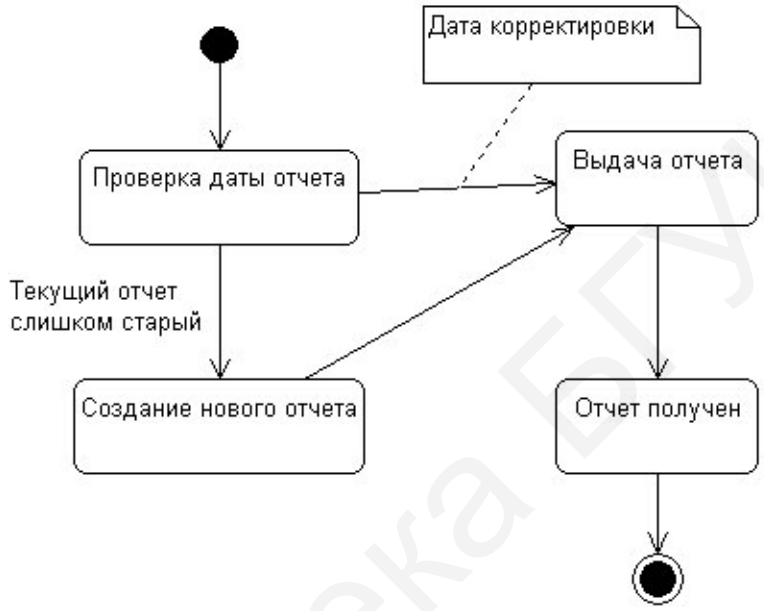



Рисунок 2.7 – Пример диаграммы состояний

Легко усмотреть тесную связь (общность) диаграмм состояний и диаграмм активностей. Эта связь того же рода, что и отношение между блок-схемами алгоритмов и автоматами (автоматы представляются состояниями и переходами между ними под действием входных сигналов (условий)).

## 2.6 Диаграммы последовательности

Диаграмма последовательности включает в себя хронологические (временные) переходы (последовательности действий), а также последовательности сообщений и их хронологический порядок. В диаграмме последовательности показываются экземпляры объектов и субъектов, а также сообщения, описывающие их взаимодействие. Рассмотрим следующий пример (рисунок 2.8).

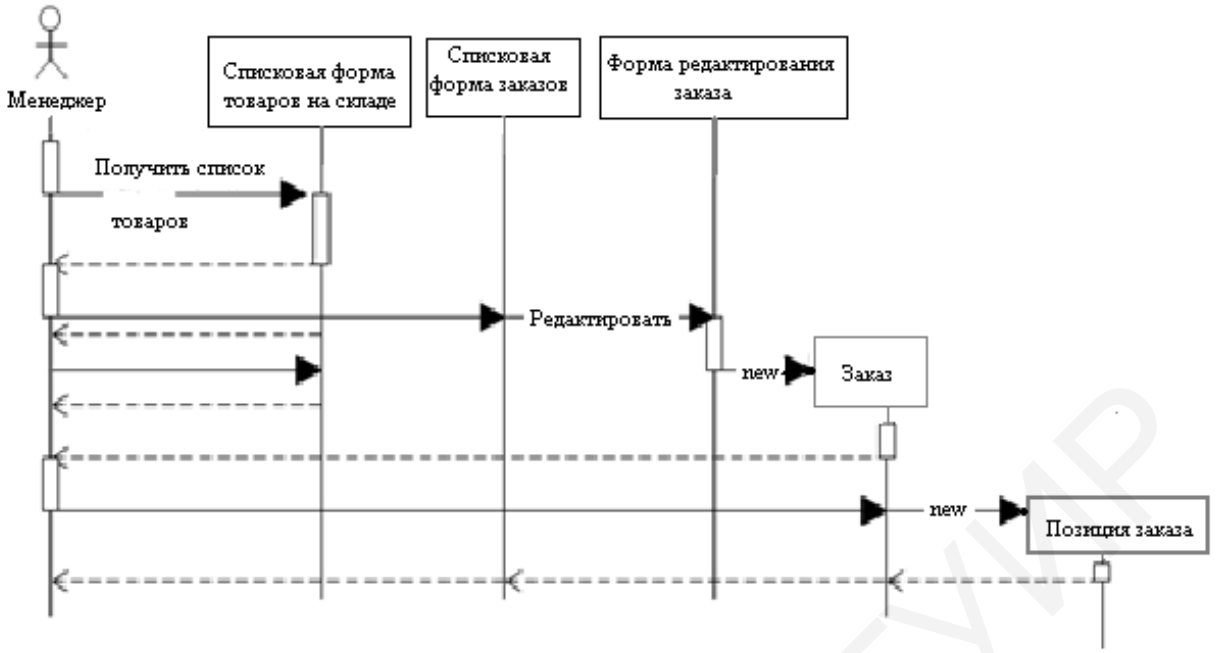



Рисунок 2.8 – Пример диаграммы последовательности

Данная диаграмма последовательности отражает хронологию действий, выполняемых менеджером. Каждое действие передается стрелкой, направленной от линии жизни одного объекта к линии жизни другого. Линия жизни – это вертикальная линия от каждого объекта. На ней прямоугольники соответствуют некоторым операциям (или временным задержкам). Так, первым действием менеджера является получение списка товаров. Соответствующее сообщение от менеджера идет к линии жизни объекта, который именуется как «списковая форма товаров на складе».

## 2.7 Создание диаграмм в среде Rational Rose

Окно Rational Rose имеет вид, показанный на рисунке 2.9. На панели 1 указаны классы диаграмм (View), объединенные по некоторому общему признаку. Так, Use Case View содержит диаграммы вариантов использования, Logical View – диаграммы классов, активностей, состояний и др. Logical View показывает, как система будет реализовывать поведение, описанное в вариантах использования, и содержит описание составных частей системы и их взаимодействие. Логическое представление включает конкретные классы, диаграммы классов и диаграммы состояний. С их помощью конструируется детальный проект создаваемой системы. Component View позволяет строить компоненты для компиляции в языки типа C++. Панель 2 – это панель инструментов (элементов), которые выбираются и перетаскиваются в окно диаграммы 3. Окно 4 – это окно документации (предполагается, что разработчик может сопровождать диаграммы текстовыми документами). Нижнее окно 5 – это окно диагностики (например, сообщений об ошибках).

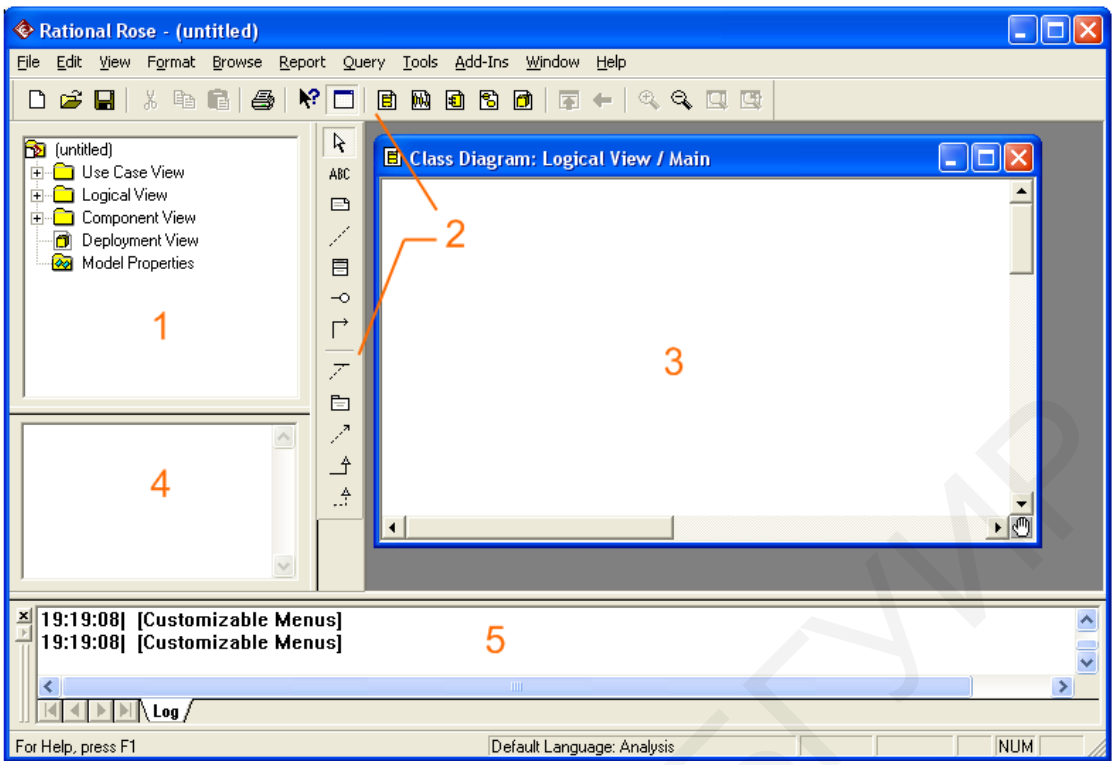



Рисунок 2.9 – Окно Rational Rose

Далее продемонстрируем, как создавать диаграммы вариантов использования и классы в окне диаграмм классов.

### 2.7.1 Создание диаграмм вариантов использования в среде Rational Rose

Как правило, с этой диаграммы начинается создание проекта. В Rational Rose создайте новый проект через меню File → New Model. Сначала нарисуйте диаграмму с помощью панели инструментов (рисунок 2.10).


Рисунок 2.10 – Панель инструментов

Основным элементов является актор – субъект, выполняющий определенные действия и операции. Ему соответствует значок  . На диаграмме может быть несколько акторов. Актор реализует некоторый вариант использования (или несколько вариантов использования). Вариант использования ассоциируется с процессом (используется также название прецедент) и отображается значком  . Если актор управляет процессом, то стрелка рисуется от актора к прецеденту (рисунок 2.11).

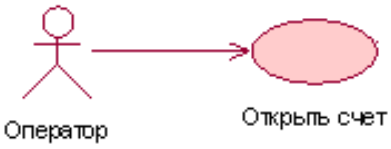



Рисунок 2.11 – Актор управляет процессом

Если актор получает информацию или результаты от процесса, то стрелка рисуется в обратном направлении (рисунок 2.12).


Рисунок 2.12 – Актор получает сообщение от процесса

### 2.7.2 Создание классов в среде Rational Rose

Диаграмма классов является важнейшей среди диаграмм Rational Rose. Классы представляются отдельными блоками и могут вступать в различные связи (ассоциации). На основании диаграммы классов можно создавать скеллетон программы, например на C++.

Для создания класса на диаграмме классов откройте узел Logical View (окно 1), представленный на рисунке 2.13.

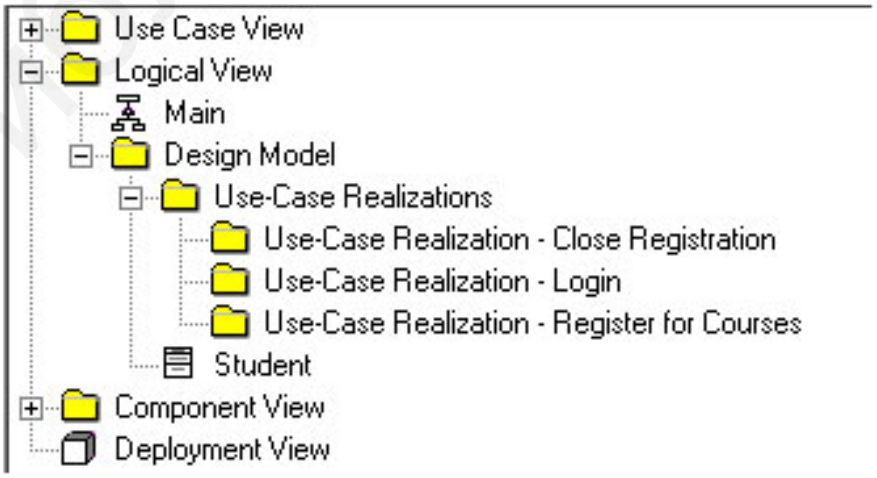



Рисунок 2.13 – Открытый узел Logical View

Для создания классов выполните следующие действия:

1. Щелкните правой кнопкой мыши по пакету Design Model.
2. Выберите пункт New → Class в открывшемся меню. Новый класс под названием NewClass появится в браузере.
3. Выделите его и введите его имя, например Student.
4. Создайте аналогичным образом другие классы: Lector, Schedule, Courses.
5. Щелкните правой кнопкой мыши по пакету Design Model.
6. Выберите пункт New → Class Diagram в открывшемся меню.
7. Присвойте имя диаграмме классов.
8. Чтобы расположить вновь созданные классы на диаграмме классов, откройте ее и перетащите классы на открытую диаграмму кнопкой мыши.

Создайте описание каждого класса, включив в него атрибуты и методы, для этого выберите правой кнопкой мыши класс, а затем опцию Open Specification. Откроется окно, показанное на рисунке 2.14.

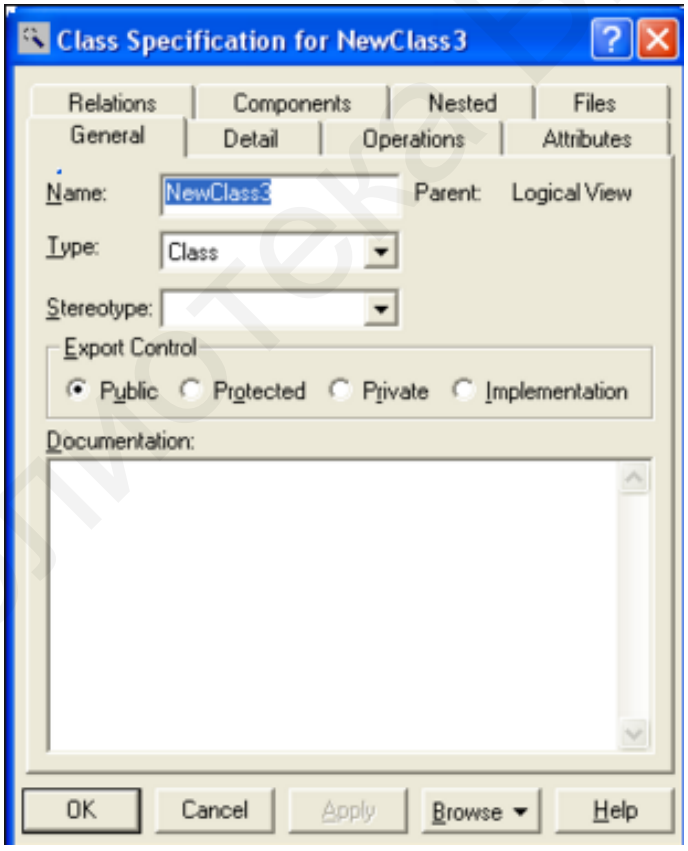



Рисунок 2.14 – Окно Open Specification

Каждый класс имеет имя, атрибуты и методы, а также различные типы связей (ассоциаций) с другими классами.

В поле Name напишите название класса, в поле Type задайте тип (в нашем случае – Class). Укажите область видимости: Public, Protected, Private, либо Implementation. Область видимости Public означает, что элементы внутри класса доступны вне класса; Protected – элементы доступны только внутри данного класса; Private – элементы доступны также и производным классам; Implementation соответствует классу-интерфейсу.

Для задания атрибутов (свойств/полей) класса во вкладке Attributes вызовите правой клавишей мыши контекстное меню и нажмите Insert (добавить атрибут). В результате в этом поле появится новый атрибут с именем по умолчанию. Дважды щелкнув кнопкой мыши по имени атрибута, попадаем в окно свойств данного атрибута (рисунок 2.15). Здесь впишите название (например, count) и определите тип атрибута: целочисленный, логический, текстовый или другой тип во всплывающем меню. Используются только стандартные типы атрибутов.

В поле Documentation можно ввести краткое описание класса, включающее его назначение, используемые методы и поля (атрибуты), а также связи с другими классами.

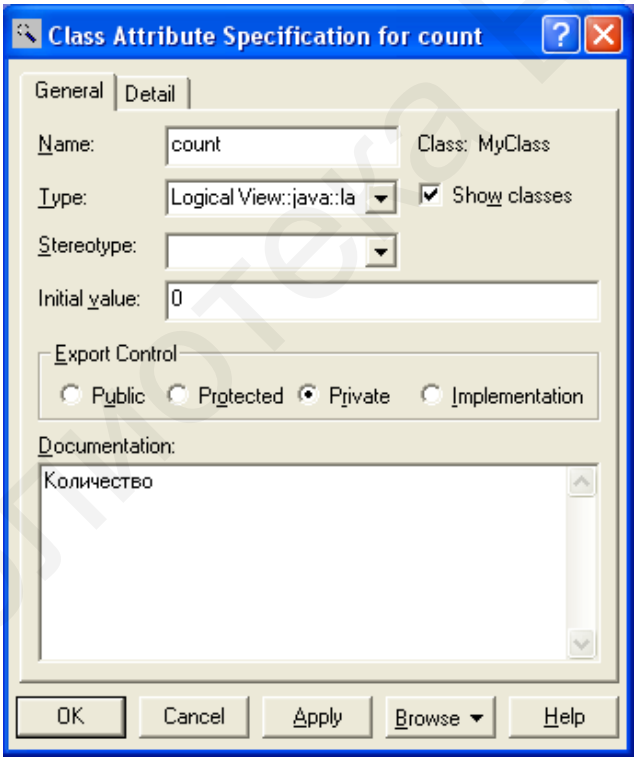



Рисунок 2.15 – Окно добавления атрибута

Аналогичным образом добавляется метод (операция), используется вкладка Operations (см. рисунок 2.14). Окно для добавления операции представлено на рисунке 2.16.

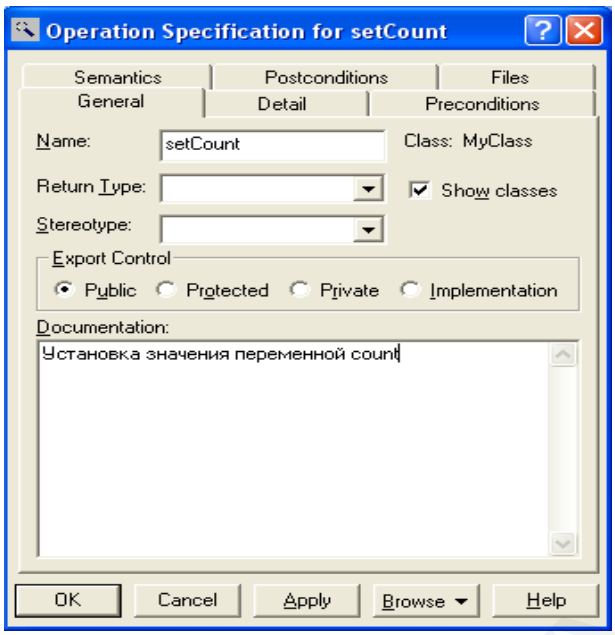



Рисунок 2.16 – Окно для добавления операции

Имя операции (метода) вводится в поле Name. В поле Возвращаемый тип (Return Type) указывается тип данных для значения, возвращаемого в данной операции. Для того чтобы добавить входные аргументы, используется вкладка Detail. Откроется окно, показанное на рисунке 2.17.

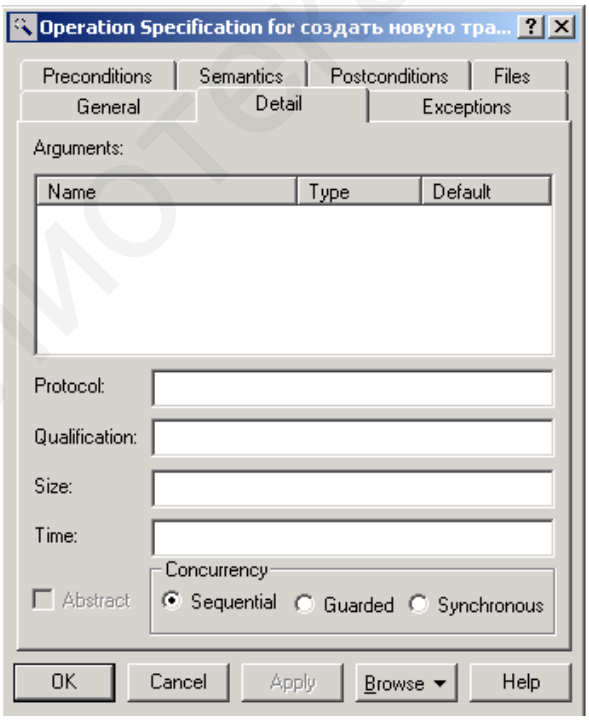



Рисунок 2.17 – Добавление аргументов в операцию

Далее следует щелчком правой кнопки мыши в белом окне открыть контекстное меню и выполнить *операцию меню* **Insert** (Вставить). После чего в этом *поле* появится *аргумент* данной *операции* с именем по умолчанию

**argname**. Для редактирования свойств аргумента предназначено специальное окно свойств аргумента. Для атрибута и метода задаются свойства видимости (+ public, – private, # -protected в нотации UML). В Rational Rose свойства видимости атрибутов и операций задаются согласно таблице 2.1.

Таблица 2.1 – Нотация UML

| Графическое изображение | Текстовый аналог |
|---|---|
|  | Public |
|  | Protected |
|  | Private |
|  | Implementation |

Пример некоторого реализованного варианта диаграммы классов показан на рисунке 2.18.

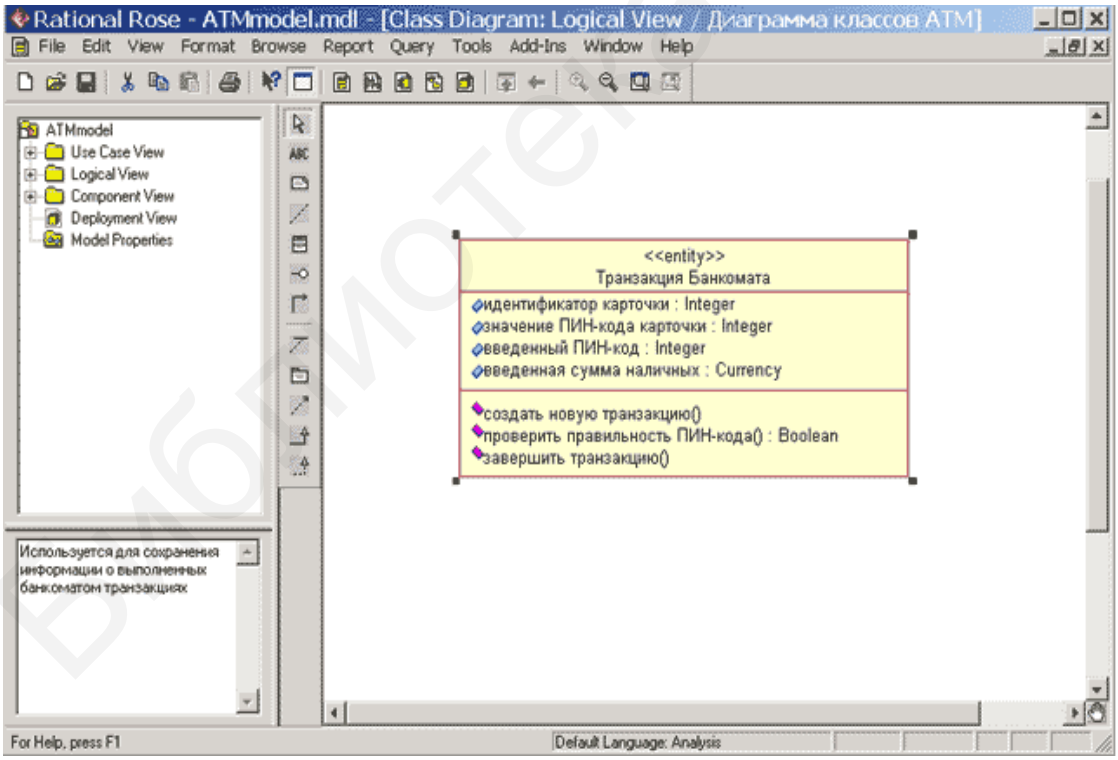



Рисунок 2.18 – Реализованный вариант диаграммы классов

Обычно на диаграмме классов отображается несколько классов, которые связываются стрелками и линиями, передающими отношения. Для выбора стрелки можно использовать палитру инструментов (Tools).

# 3 ПАТТЕРНЫ ПРОЕКТИРОВАНИЯ

## 3.1 Краткие теоретические сведения

Паттерны проектирования – это некоторые обобщенные приемы реализации типовых программных задач. Имея «под руками» библиотеку паттернов, можно существенно ускорить процесс создания кода и снизить число ошибок.

Различают следующие три основные группы паттернов (шаблонов) проектирования: поведенческие (B), порождающие (C), структурные (S) (рисунок 3.1).


Рисунок 3.1 – Классификация паттернов проектирования

Паттерны проектирования определяют типовые схемы программных (проектных) решений, которые пригодны для многих ситуаций. Каждый паттерн проектирования реализует некоторую типовую UML-диаграмму.

Порождающие паттерны используют некоторые общие принципы проектирования, а именно:

- наследование от родительского класса должно быть одинаковым для любого дочернего класса;
- доступ к классу должен выполняться через его интерфейс;
- следует стремиться изменять не методы, а данные;
- любое изменение класса должно быть максимально упрощено и др.

Начнем рассмотрение с порождающих паттернов. В качестве рабочего языка выбран язык C#, демонстрирующий реализацию паттернов.

### 3.2 Фабричный метод (Class Factory)

Этот паттерн позволяет порождать объекты различных (нужных разработчику) классов. Каждый класс определяется своим интерфейсом. Имеется некоторый общий шаблон для построения конкретных производителей классов, которые порождают конкретные объекты. UML-диаграмма этого паттерна показана на рисунке 3.2.

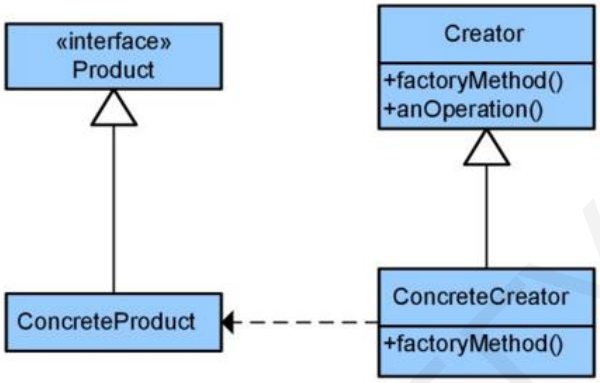



Рисунок 3.2 – UML-диаграмма паттерна Class Factory

Данный паттерн:

- существенно упрощает создание новых объектов;
- позволяет увеличить число классов в фабрике классов на основе однотипных программных решений;
- содержит «общую часть», характерную всем классам фабрики.

Рассмотрим следующий код, иллюстрирующий возможности рассматриваемого паттерна:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace ClassFactoryLab
{
        public class CheckBook
        {
           protected decimal _Amount;
           public decimal getExpence()
           {
              return _Amount;
           }
        }

        public class Travel : CheckBook
        {
```

```
        public Travel()
        {
            _Amount = 10000.20M;
          // Console.WriteLine("Travel Expences are:{0}",
          //     _Amount);
        }
    }
    public class Health : CheckBook
    {
        public Health()
        {
            _Amount = 1000.60M;
        }
    }

    public class _Factory
    {
        public CheckBook produceObject(int typeobj)
        {
            switch (typeobj)
            {
                case 1:
                    return new Travel();

                case 2:
                    return new Health();

                default: return null;
            }
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            _Factory facr = new _Factory();

            Travel trvl = (Travel)facr.produceObject(1);
            Health heal = (Health)facr.produceObject(2);
            Console.WriteLine("Health costs:{0}",
                heal.getExpence());

            Console.WriteLine("Travel costs:{0}",
                trvl.getExpence());
            Console.ReadLine();

        }
    }
}
```

В этом примере объявлены два предметных класса: Travel и Health. Они наследуются от общего родителя – CheckBook (чековая книжка). Различие у них в стоимости (Amount). Также различия могут быть и в свойствах, и в методах. Шаблон фабрики классов позволяет создавать объекты (в данном примере) как класса Travel, так и класса Health. Для этого используется базовый креатор – класс _Factory. Рассмотрим этот класс:

```
public class _Factory
{
   public CheckBook produceObject(int typeobj)
   {
      switch (typeobj)
      {
         case 1:
            return new Travel();

          case 2:
             return new Health();

          default: return null;
       }
    }
 }
```

В нем описан метод produceObject(int typeobj), который возвращает объект либо типа Travel, либо типа Health в зависимости от переданного аргумента typeobj – целого числа. Выходное окно показано на рисунке 3.3.

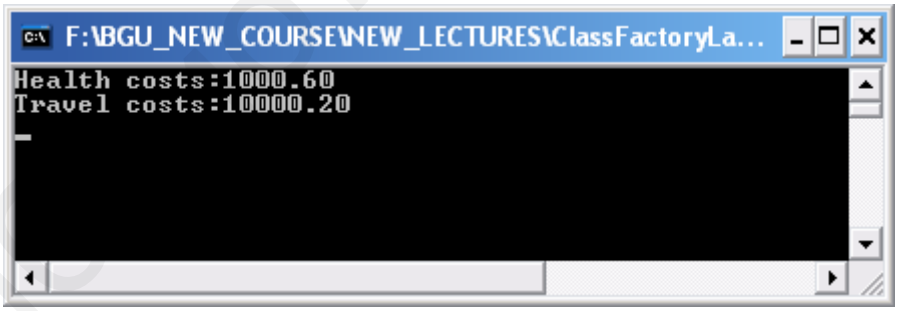



Рисунок 3.3 – Окно программы Class Factory

Заметим, что в данном варианте реализации мы использовали базовый класс CheckBook. Согласно рисунку 3.2, его можно заменить интерфейсом. Это демонстрирует следующий код:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
```

```
namespace ClassFactoryLab
{
   public interface CheckBook
   {
      void setExpence(decimal Amount);
      decimal getExpence();
   }

   public class Travel : CheckBook
   {
      protected decimal _Amount=10000.20M;
      public decimal getExpence()
      {
           return _Amount;
      }
         public void setExpence(decimal x)
         { _Amount=x;
         }
   }

   public class Health : CheckBook
   {
      protected decimal _Amount;

      public decimal getExpence()
      {
         return _Amount;
      }
      public void setExpence(decimal x)
      {
         _Amount = x;
      }


      public Health()
      {
         _Amount = 1000.60M;

      }
   }

   public class _Factory
   {
      public CheckBook produceObject(int typeobj)
      {
         switch (typeobj)
         {
            case 1:
               return new Travel();
            case 2:
               return new Health();
```

```
            default: return null;
          }
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            _Factory facr = new _Factory();

            Travel trvl = (Travel)facr.produceObject(1);
            Health heal = (Health)facr.produceObject(2);
            Console.WriteLine("Health costs:{0}",
                heal.getExpence());

            Console.WriteLine("Travel costs:{0}",
                trvl.getExpence());
            Console.ReadLine();
        }
    }
}
```

Выполнение этого кода дает тот же результат, что и ранее. Но теперь мы используем интерфейс

```
public interface CheckBook
    {
       void setExpence(decimal Amount);
        decimal getExpence();
    }
```

На практике вместо интерфейса могут также быть использованы абстрактные классы.

### 3.3 Абстрактная фабрика классов (Abstract Class Factory)

Этот паттерн позволяет использовать несколько фабрик классов, базирующихся на одном абстрактном классе, который и называется абстрактной фабрикой классов. Клиент может заказывать нужную ему фабрику классов и порождать объекты (продукты) из этой фабрики. UML-диаграмма абстрактной фабрики классов представлена на рисунке 3.4.

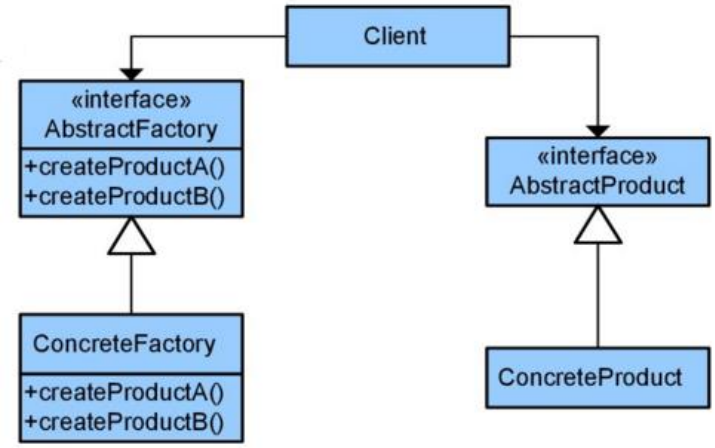



Рисунок 3.4 – UML-диаграмма Abstract Class Factory

Обратимся к программной реализации:

```
using System;
using System.Windows.Forms;
using System.Drawing;
using System.Drawing.Drawing2D;

namespace AbstractFactory
{
   abstract class AbstractFactory
   {
      public abstract AbstractProduct CreateProduct(int i);

   }

   class ConcreteFactoryGeometry : AbstractFactory
   {
      public override AbstractProduct CreateProduct(int i)
      {
         return new Geometry(i);
      }
   }

   class ConcreteFactoryColor : AbstractFactory
   {
      public override AbstractProduct CreateProduct(int i)
      {
         return new Color(i);
      }
   }

   abstract class AbstractProduct
   {
      public abstract void getInfo(AbstractProduct a);
   }
```

```
class Geometry : AbstractProduct
{
    public string name = "Geometry";
    public string descript = "Define geometric figures";

    public Geometry(int i)
    {
        switch (i)
        {
            case 1:
                name = "ellipse";
                break;

            case 2:
                name = "rectangle";
                break;
            default:
                break;
        }

    }

     public override void getInfo(AbstractProduct a)
    {
        Console.WriteLine("Product name {0} \n Product description {1}\n ",
             name,descript);
    }

}

  class Color : AbstractProduct
{
    public string name = "Color";
    public string descript = "Define color of figure";

     public Color(int i)
    {
        switch (i)
        {
            case 1:
                name = "red";
                break;

            case 2:
                name = "blue";
                break;
            case 3:
                name = "green";
                break;
            default:
                break;
```

```
        }

    }

    public override void getInfo(AbstractProduct a)
    {

        Console.WriteLine("Product name {0} \n Product description {1}",
            name, descript);
    }
}

class Client
{

    private AbstractProduct _prod;

    public Client(AbstractFactory factory, int i)
    {
        _prod = factory.CreateProduct(i);
    }

    public void Run()
    {
        _prod.getInfo(_prod);
    }
}

class Program
{
    [STAThread]
    static void Main(string[] args)
    {
        ConcreteFactoryGeometry factorygeometry1 = new
        ConcreteFactoryGeometry();
        Console.WriteLine("input 1- ellipse; 2- rectangle");
        string s = Console.ReadLine();
        int i = 0;
        try
        {
          i = Convert.ToInt32(s);
        }
        catch (Exception)
        { }

        Geometry _prod;

            _prod =(Geometry) factorygeometry1.CreateProduct(i);

       Console.WriteLine("Geometry product:{0}  {1}", _prod.name,
          _prod.descript);
```

```
        Console.WriteLine("input 1- red; 2- blue; 3 - green");
        s = Console.ReadLine();
        i = 0;
        try
        {
           i = Convert.ToInt32(s);
        }
        catch (Exception)
        { }

        AbstractFactory factorycolor2 = new
        ConcreteFactoryColor();
        Client client2 = new Client(factorycolor2, i);
        client2.Run();
        Console.ReadKey();
      }
   }
}
```

Здесь абстрактная фабрика определена как

```
abstract class AbstractFactory
  {
    public abstract AbstractProduct CreateProduct(int i);

  }
```

Она содержит единственный абстрактный метод CreateProduct(int i);
Класс абстрактного продукта представлен ниже:

```
abstract class AbstractProduct
  {
    public abstract void getInfo(AbstractProduct a);
  }
```

У класса абстрактного продукта в данном примере есть только один абстрактный метод. Теперь мы определяем конкретные классы продуктов на базе этого абстрактного класса:

```
class Geometry : AbstractProduct
   {
        .....
```

и класс

```
class Color : AbstractProduct
   {
 .....}
```

Каждый из этих классов описывает свои собственные объекты, разумеется, в общем случае он может их и порождать (в рассматриваемом примере этого нет). Например, класс Geometry описывает объекты эллипс, прямоугольник. Эти объекты порождаются конкретными фабриками. В нашем примере эти фабрики следующие:

```
class ConcreteFactoryGeometry : AbstractFactory
  {
     public override AbstractProduct CreateProduct(int i)
     {
        return new Geometry(i);
     }
  }

class ConcreteFactoryColor : AbstractFactory
  {
     public override AbstractProduct CreateProduct(int i)
     {
        return new Color(i);
     }
  }
```

Обе приведенные выше конкретные фабрики базируются на абстрактной фабрике и переписывают единственный абстрактный метод CreateProduct(int i). В зависимости от номера *i* возвращается описание нужного объекта.

Итак, абстрактная фабрика классов использует абстрактный класс, на котором базируются конкретные фабрики. Каждая конкретная фабрика специализируется на порождении некоторого ряда объектов, специфических именно для этой фабрики.

### 3.4 Одиночка (Singleton)

Этот паттерн проектирования используется для ограничения числа порождаемых объектов класса. В частности, можно ограничить число создаваемых объектов единицей. Для этого конструктор класса делаем закрытым (protected). Вместо конструктора используем какой-либо иной метод (в рассматриваемом далее примере это MakeBook()). Следующий пример иллюстрирует технику реализации данного паттерна. В классе объявлена переменная counter, в которой фиксируется число уже созданных объектов. Эту переменную проверяем каждый раз при создании очередного объекта.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
```

```
namespace Singleton
{
   class CheckBook
   {
      private static int counter=0;

      private CheckBook()
      {
      }
      public static int Counter
      {get
      {
         return counter;
      }
      }

      public static CheckBook MakeBook()
      {
         if (counter <= 1)
         {
            counter++;
            return new CheckBook();
         }

         return null;

      }
   }
   class Program
   {
      static void Main(string[] args)
      {
         CheckBook cb1 = CheckBook.MakeBook();
         CheckBook cb2 = CheckBook.MakeBook();
         CheckBook cb3 = CheckBook.MakeBook();
         Console.WriteLine("Number of books created is {0}",
         CheckBook.Counter);
         Console.ReadLine();
      }
   }
}
```

Конструктор этого паттерна закрыт. Его вызов выполняет метод CheckBook.MakeBook, который контролирует число создаваемых объектов путем проверки значения счетчика counter.

### 3.5 Прототип (Клон/Clone)

Этот прототип позволяет создать копию (клон) объекта. Различают поверхностное и глубинное клонирование. При поверхностном клонировании не копируются внутренние классы, объявленные в данном классе, при глубинном копируются также и внутренние классы. UML-диаграмма показана на рисунке 3.5.

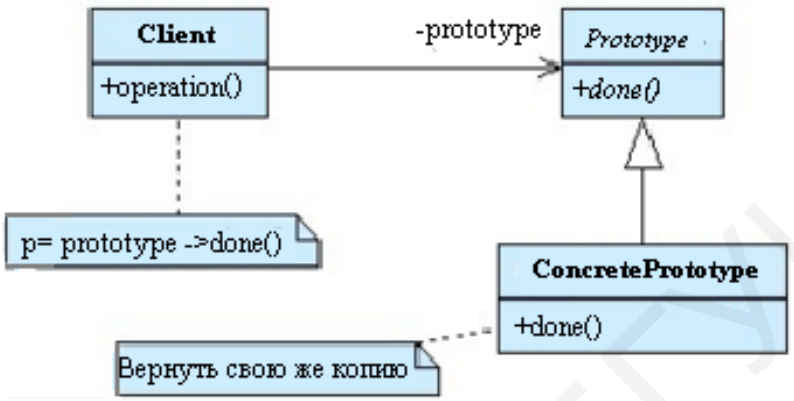



Рисунок 3.5 – Паттерн Clone

Абстрактный класс Prototype содержит объявление метода Clone, конкретная реализация которого возлагается на класс ConcretePrototype. Рассмотрим иллюстративный пример:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace Clone
{
   public abstract class Prime
   {
      private static int age;
      private static String prof;

      public int Age
      {
         get { return age; }
         set { age = value; }
      }
      public String Prof
      {
         get { return prof; }
         set { prof = value; }

      }
```

```
        public abstract Prime Clone();

    }


    class CL : Prime
    {
        public override Prime Clone()
        {
            return (Prime) this.MemberwiseClone() ;
        }

    }

    class Program
    {
        static void Main(string[] args)
        {
            CL x = new CL();
            x.Age = 23;
            x.Prof = "Musician";
            Console.WriteLine("X :  " + x.Prof + "  " + x.Age);
            CL y = x.Clone() as CL;
            Console.WriteLine("Y :  " + y.Prof + "  " + y.Age);

            x.Age = 24;
            x.Prof = "rambler";

            CL z = y.Clone() as CL;

            z.Age = 34;
            Console.WriteLine("Y new :  " + y.Prof + "  " + y.Age+" "+x.Age);
            Console.ReadLine();

        }
    }
}
```

Здесь объявлен абстрактный класс Prime, содержащий объявление метода клонирования – Clone. От класса Prime наследуется класс CL, который уже содержит реализацию метода клонирования:

```
public override Prime Clone()
    {
      return (Prime) this.MemberwiseClone() ;
     }
```

Поверхностное клонирование реализуется стандартным методом MemberwiseClone. При этом будут клонироваться как свойства, так и методы класса Prime. Убеждаемся в этом путем реализации консольного вывода:

```
CL x = new CL();
x.Age = 23;
x.Prof = "Musician";
Console.WriteLine("X :  " + x.Prof + "  " + x.Age);
CL y = x.Clone() as CL;
Console.WriteLine("Y :  " + y.Prof + "  " + y.Age);
```

Окно программы поверхностного клонирования показано на рисунке 3.6.

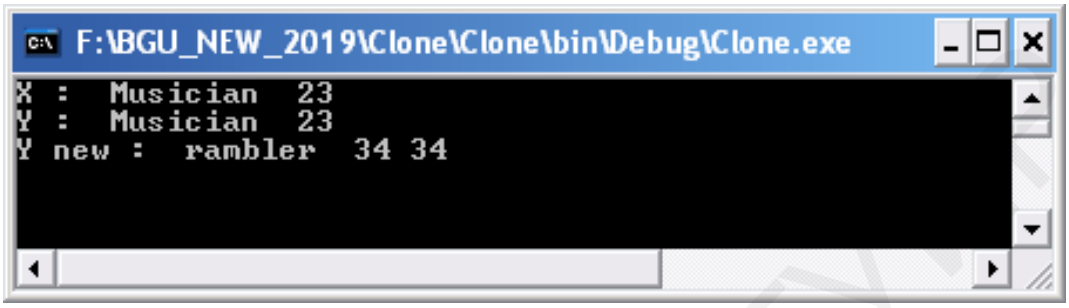



Рисунок 3.6 – Объекты x, y – клоны

### 3.6 Строитель (Builder)

Этот паттерн формирует сложный объект из составных частей. Каждая составная часть определяется условиями задачи. Соответствующая UML-диаграмма приведена на рисунке 3.7.

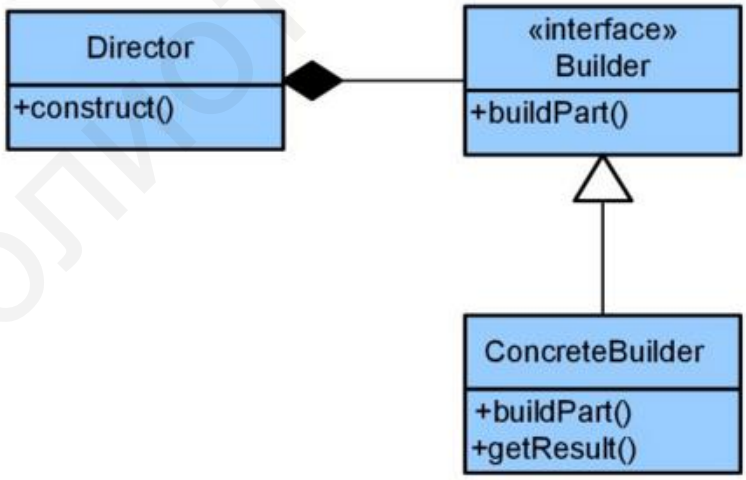



Рисунок 3.7 – UML-диаграмма паттерна Builder

Здесь Director представляет класс, который вызывает нужных строителей. Каждый строитель ответственен за свою собственную часть. Все конкретные строители наследуются от общего интерфейса или абстрактного класса. Рассмотрим следующий пример:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Collections;
namespace BuilderLab
{
   public abstract class Meal
   {
      public abstract void Pizza();
      public abstract void Drink();
      public abstract void Toy();

      public ArrayList parts = new ArrayList();
      public void Show()
      {
         foreach (var z in parts)
         {
            Console.WriteLine(z.ToString());
         }
      }

      public class AdultMeal : Meal
      {
         public AdultMeal()
         {
            this.Pizza();
            this.Drink();
            this.Toy();

         }
         public override void Pizza()
         {
            parts.Add("Mexicano Pizza");
         }

         public override void Toy()
         {
         }

         public override void Drink()
         {
            parts.Add("Dark Coffee");
         }
      }
      public class ChildMeal : Meal
      {
          public ChildMeal()
         {
            this.Pizza();
            this.Drink();
```

```
            this.Toy();

        }


        public override void Pizza()
        {
        }

        public override void Toy()
        {
            parts.Add("Teddy Bear");
        }

        public override void Drink()
        {
            parts.Add("Pepsi");
        }
    }


    class Program
    {
        static void Main(string[] args)
        {
            AdultMeal am = new AdultMeal();
            am.Show();
            Console.WriteLine("\n");
            ChildMeal cm = new ChildMeal();
            cm.Show();
            Console.ReadLine();
        }
    }
  }
}
```

В этом примере используется не интерфейс, а абстрактный класс строителя – Meal. Этот класс применяется для построения конкретных строителей (формируется меню для детей – ChildMeal – и для взрослых – AdultMeal). Комплект для взрослого и для ребенка формируется в строителях однотипно:

```
this.Pizza();
this.Drink();
this.Toy();
```

Однако реализация этих методов для взрослого и ребенка различная. Например, реализация метода Toy() для «взрослого строителя» такова:

```
public override void Toy()
        {
        }
```

Игрушка (Toy) для взрослого не создается.

Основная программа происходит в методе Main:

```
static void Main(string[] args)
    {
      AdultMeal am = new AdultMeal();
      am.Show();
      Console.WriteLine("\n");
      ChildMeal cm = new ChildMeal();
      cm.Show();
      Console.ReadLine();
    }
```

Первая строка AdultMeal am = new AdultMeal(); вызывает конструктор класса AdultMeal, который и создает меню для взрслого. Добавляемые опции меню сохраняются в коллекции ArrayList parts как строковые элементы. Аналогичным образом меню для ребенка создается в строке ChildMeal cm = new ChildMeal(). Содержимое меню выводится в команде cm.Show().

Окно работы прораммы показано на рисунке 3.8.

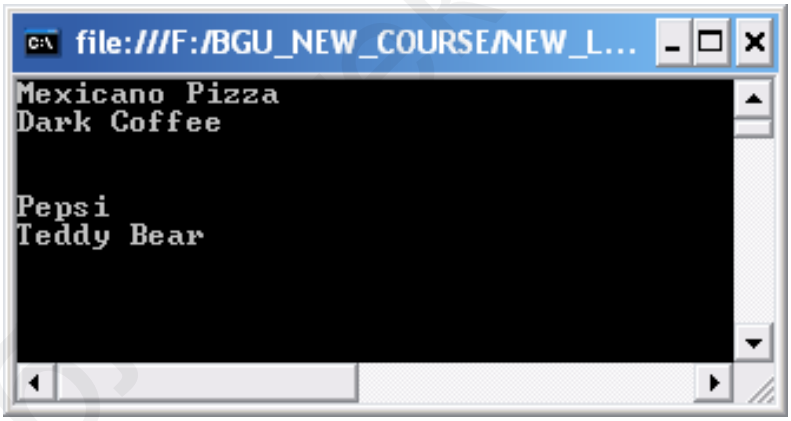



Рисунок 3.8 – Работа паттерна Builder

Рассмотрим далее некоторые структурные паттерны, которые позволяют динамически изменять состав и вид методов класса.

### 3.7 Декоратор (Decorator)

UML-диаграмма этого паттерна может иметь вид, представленный на рисунке 3.9.

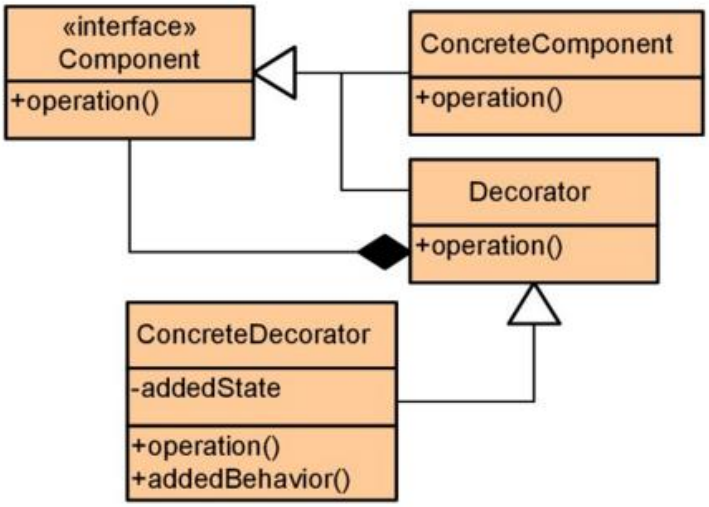



Рисунок 3.9 – UML-диаграмма паттерна Decorator

Родовым классом всей структуры является интерфейс или абстрактный класс Component (имя родового класса, разумеется, условно). Этот класс является обобщающим для класса конкретного компонента (может отсутствовать) и класса Decorator (этот класс в свою очередь является родительским классом для конкретного декоратора). Идею данного паттерна проще пояснить на примере выпечки. Допустим, можно изготовить булочку без каких-либо добавок. Это будет базовый класс. К булочке можно добавить сахарную пудру, орешки или что-либо еще. Это, собственно, и есть декорация. В результате изменится цена булочки и ее функциональные качества. В примере ниже рассматриваем декорацию на основе изюма, ягод и орешков. Базовый класс Bakery (выпечка) является абстрактным и содержит объявления двух методов: public abstract string GetName(); public abstract double GetPrice(). На UML-диаграмме, изображенной на рисунке 3.9, он играет роль интерфейса Component. На основе этого класса объявляется также абстрактный класс Decorator. В этом классе объявляется объект Bakery – в основе будущей выпечки. Кроме того, объявлены поля для названия выпечки и цены.

```
public abstract class Bakery
{
   public abstract string GetName();
   public abstract double GetPrice();

}

public abstract class Decor : Bakery
{
   Bakery base_comp = null;
   protected string base_name = "";
   protected double base_price = 1.0;
```

```
    protected Decor(Bakery bak)
    {
      base_comp = bak;
    }

    public override string GetName()
    {
      return base_name;
    }

    public override double GetPrice()
    {
      return base_price;
    }
  }
```

Затем уже следуют конкретные декораторы. Первый из них – декоратор на основе «ягод»:

```
  class CherryDecor : Decor
  {
    public CherryDecor(Bakery x)
      : base(x)
    {
      base_name = "SweatCherry";
      base_price = 2.0+x.GetPrice();
    }

    public override string GetName()
    {
      return base_name;
    }

    public override double GetPrice()
    {
      return base_price;
    }

    public string ShowInfo()
    {
      return "This is Cherry price=" + base_price;
    }
  }
```

Здесь интерес представляет только конструктор:

```
public CherryDecor(Bakery x)
      : base(x)
    {
      base_name = "SweatCherry";
```

```
        base_price = 2.0+x.GetPrice();
    }
```

Ключевое слово base(x) указывает на обращение к конструктору базового класса, т. е. класса Decor. Если у базового класса есть явно определенный конструктор, то он должен быть вызван в конструкторе дочернего класса. Видим, что конструктор дочернего класса изменяет название выпечки и ее цену. Теперь мы приводим полный текст программы:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace Decorator
{
   public abstract class Bakery
   {
      public abstract string GetName();
      public abstract double GetPrice();

   }

   public abstract class Decor : Bakery
   {
      Bakery base_comp = null;
      protected string base_name = "";
      protected double base_price = 1.0;

      protected Decor(Bakery bak)
      {
         base_comp = bak;
      }

      public override string GetName()
      {
         return base_name;
      }

      public override double GetPrice()
      {
         return base_price;
      }
   }

   class CherryDecor : Decor
   {
      public CherryDecor(Bakery x)
         : base(x)
      {
```

```
        base_name = "SweatCherry";
        base_price = 2.0+x.GetPrice();
    }

    public override string GetName()
    {
        return base_name;
    }

    public override double GetPrice()
    {
        return base_price;
    }

    public string ShowInfo()
    {
        return "This is Cherry price=" + base_price;
    }
}

class RaisinDecor : Decor
{
    public RaisinDecor(Bakery x)
        : base(x)
    {
        base_name = "Raisin";
        base_price = 1.0 + x.GetPrice();
    }

    public override string GetName()
    {
        return base_name;
    }

    public override double GetPrice()
    {
        return base_price;
    }

    public string ShowInfo()
    {
        return "This is Raisin decored price=" + base_price;
    }
}

 class Cake : Bakery
{
    private string cake_name = "Sandwitch";
    private double cake_price = 10.0;
```

```
        public override string GetName()
        {
           return cake_name;
        }

        public override double GetPrice()
        {
           return cake_price;
        }
    }

class DoNut : Bakery
    {
        private string nut_name = "DoNut";
        private double nut_price = 5.0;
        public override string GetName()
   {
      return nut_name;
   }

     public override double GetPrice()
        {
          return nut_price;
        }
    }

class Program
    {
     static void Main(string[] args)
      {
       CherryDecor chr = new CherryDecor(new Cake());
       Console.WriteLine("Done:" + chr.ShowInfo());
       RaisinDecor rsn = new RaisinDecor(new Cake());
       Console.WriteLine("Done:" + rsn.ShowInfo());
       Console.ReadLine();
       }
}}
```

Основные действия «разворачиваются» в методе Main, где реализуются два конкретных декоратора и выполняется вывод о выпечках с этими добавками на консоль (рисунок 3.10).

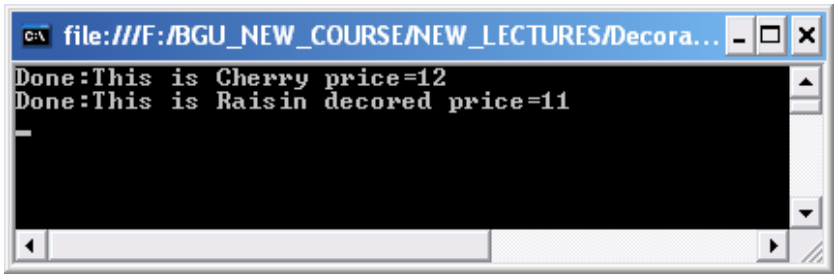



Рисунок 3.10 – Работа паттерна Decorator

### 3.8 Фасад (Facade)

Данный паттерн «прячет» конкретные классы, где определены методы, условно говоря, «за фасадом». Схематически это можно передать так, как показано на рисунке 3.11.

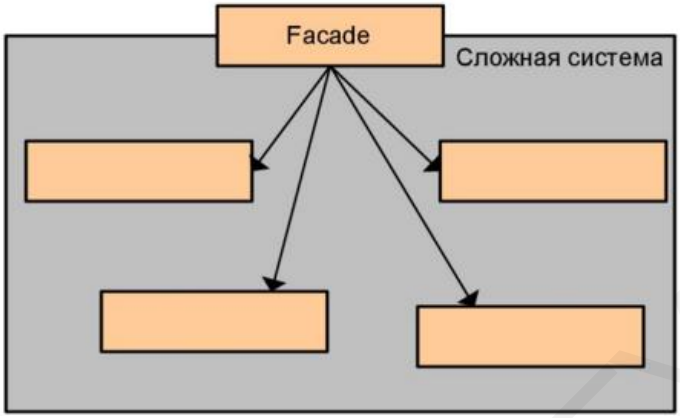



Рисунок 3.11 – UML-диаграмма паттерна Facade

Так, в следующем примере имеется два класса, один из них реализует метод A, второй – метод B:

```
class SystemOne
   {
      public void MethodA()

      { Console.WriteLine("This is SystemOne"); }

   }

class SystemTwo
   {
      public void MethodB()

      { Console.WriteLine("This is SystemTwo"); }

   }
```

Класс фасада создает объекты нужного класса, но скрывает конкретный класс-хозяин:

```
static void Main(string[] args)
       {
         Facade facade = new Facade();
         facade.MethodA();
         facade.MethodB();
```

```
            Console.ReadKey();
        }
```

Здесь команда facade.MethodA() использует за фасадом объект класса SystemOne, а команда facade.MethodB() использует за фасадом объект класса SystemTwo.

Далее представлен полный пример:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace FACADE_LAB
{
    class Program
    {
        static void Main(string[] args)
        {
            Facade facade = new Facade();
            facade.MethodA();
            facade.MethodB();
            Console.ReadKey();
        }
    }

    class SystemOne
    {
        public void MethodA()

        { Console.WriteLine("This is SystemOne"); }

    }

    class SystemTwo
    {
        public void MethodB()

        { Console.WriteLine("This is SystemTwo"); }

    }

    class Facade
    {
        private SystemOne _xone;
        private SystemTwo _xtwo;
        public Facade()
        {
            _xone = new SystemOne();
```

```
        _xtwo = new SystemTwo();
    }
      public void MethodA()
      {
       (new SystemOne()).MethodA();
      }
    public void MethodB()
      {
      _xtwo.MethodB();
      }
    }
}
```

Можно интерпретировать класс фасада как хранилище методов различных классов. Пользователь явно не обращается к конструкторам, детали их реализации спрятаны.

### 3.9 Мост (Bridge)

В этом паттерне абстракция и реализация разделены и могут изменяться независимо. Другими словами, при реализации через шаблон Bridge изменение интерфейса не мешает изменению структуры реализации (рисунок 3.12).

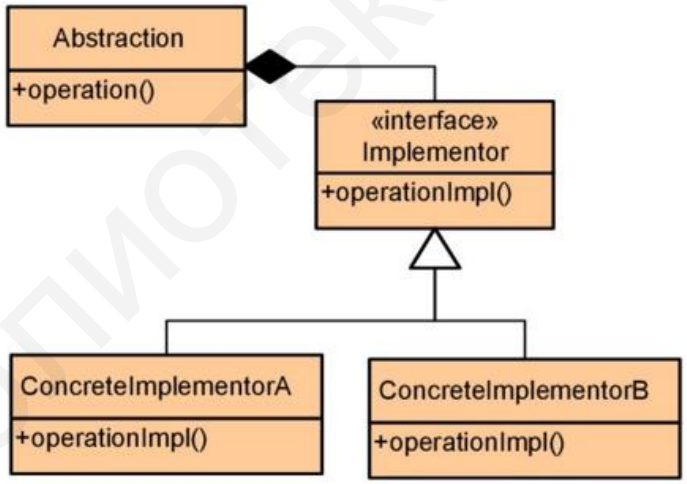



Рисунок 3.12 – UML-диаграмма паттерна Bridge

Так, пусть интерфейс или абстрактный класс Implementor содержит объявление абстрактного метода Operation:

```
public abstract class Implementor
 {
   public abstract void Operation(String word);
 }
```

Очевидно, любой класс, который наследует класс Implementor, может реализовать метод Operation различным способом. Так, можно привести следующий вариант:

```
public class ConcreteImplementorA : Implementor
  {
    public override void Operation(String word)
    {
      String answer = "???";
      switch (word)
      {
        case "cat":
          answer = "кот";
          break;
        case "dog":
          answer = "собака";
          break;
        default: break;
      }
      Console.WriteLine("Translation "+word +" :"+answer);
    }
  }

public class ConcreteImplementorB : Implementor
  {
    public override void Operation(string word)
    {
      String answer = "???";
      switch (word)
      {
        case "cat":
          answer = "кот";
          break;
        case "dog":
          answer = "собака";
          break;

        case "кот":
          answer = "cat";
          break;
        case "собака":
          answer = "dog";
          break;
        default: break;
      }

      Console.WriteLine("Translation " + word + " :" + answer);
    }
  }
```

Здесь два класса ConcreteImplementorA, ConcreteImplementorB представляют различные реализации метода Operation. Один из них выполняет перевод английских слов в русские, второй – в обратном порядке. Смысл паттерна в том, что мы вводим еще один класс – class Abstraction, который содержит свойство Implementor. Это свойство может получить значение любого объекта: либо класса ConcreteImplementorA, либо класса ConcreteImplementorB. Делается это следующим образом:

```
Abstraction abstr = new Abstraction();
abstr.Implementor = new ConcreteImplementorA();
abstr.Operation("cat");

abstr.Implementor = new ConcreteImplementorB();
abstr.Operation("кот");
Console.ReadLine();
```

Итак, объект abstr попеременно играет роль либо объекта ConcreteImplementorA, либо класса ConcreteImplementorB. Интегрируя все это в единый код, получим следующую программу:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace Lab_Bridge
{
   public class Abstraction
   {
      protected Implementor implementor;

// конструктор
      public Implementor Implementor
      {
         set { implementor = value; }
      }
      public virtual void Operation(String word)
      {
         implementor.Operation(word);
      }

   }

   public abstract class Implementor
     {
        public abstract void Operation(String word);
     }
   public class ConcreteImplementorA : Implementor
     {
```

```
public override void Operation(String word)
      {
         String answer = "???";
         switch (word)
         {
            case "cat":
               answer = "кот";
               break;
            case "dog":
               answer = "собака";
               break;
            default: break;
         }
         Console.WriteLine("Translation "+word +" :"+answer);
      }
   }

public class ConcreteImplementorB : Implementor
   {
      public override void Operation(string word)
      {
         String answer = "???";
         switch (word)
         {
            case "cat":
               answer = "кот";
               break;
            case "dog":
               answer = "собака";
               break;

            case "кот":
               answer = "cat";
               break;
            case "собака":
               answer = "dog";
               break;
            default: break;
         }
         Console.WriteLine("Translation " + word + " :" + answer);
      }
   }

class Program
   {
      static void Main(string[] args)
      {
         Abstraction abstr = new Abstraction();
         abstr.Implementor = new ConcreteImplementorA();
         abstr.Operation("cat");
```

```
            abstr.Implementor = new ConcreteImplementorB();
            abstr.Operation("кот");
            Console.ReadLine();

        }
    }
}
```

### 3.10 Компоновщик (Compositor)

Паттерн Компоновщик позволяет собирать древовидные структуры композиций. Например, автомобиль состоит из кузова, шасси и двигателя. В свою очередь, двигатель состоит из цилиндров, системы теплоотвода, системы зажигания и т. д. Нетрудно представить себе соответствующую иерархическую структуру, узлы которой соответствуют блокам или отдельным элементам всей конструкции. UML-диаграмма паттерна Компоновщик (Compositor) представлена на рисунке 3.13.

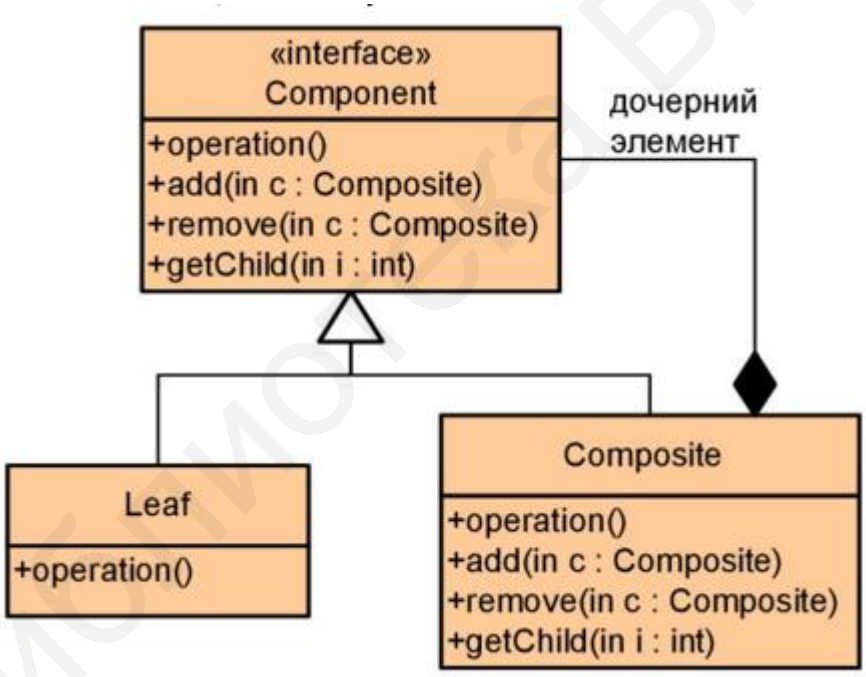



Рисунок 3.13 – UML-диаграмма паттерна Compositor

Видим, что интерфейс паттерна Компоновщик содержит объявления методов для добавления и удаления дочерних узлов, получения дочернего узла и произвольной операции над узлом. Рассмотрим следующий код в качестве примера:

```
using System;
using System.Collections.Generic;
using System.Linq;
```

```
using System.Text;

namespace Composit_Lab
{
   abstract class Component
   {
      protected String name;
      public Component(String name)
      {
         this.name = name;
      }

      public String Operation()
      {
         return name;
      }

      public abstract void Add(Component c);
      public abstract void Operation2();
   }


class Composite : Component
   {
      private List<Component> _items = new List<Component>();
      public Composite(String name):base(name)
      {

      }

      public  override void Add(Component cm)
      {
         _items.Add(cm);

      }

      public override void Operation2()
      {
         if ((this.Operation().IndexOf("rom") < 0) && (this.Operation().IndexOf("root") < 0))
         Console.WriteLine("*** {0}", this.name);
         foreach (Component cmp in _items)
         {
           //
            cmp.Operation2();
         }
      }
   }

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

```
    {
      Composite root = new Composite("root");
      Composite promroot = new Composite("PromRoot");
      Composite first = new Composite("First");
      Composite second = new Composite("Second");
      first.Add(new Leaf("Holodec"));
      first.Add(new Leaf("Ham"));
      second.Add(new Leaf("Gamburger"));
      second.Add(new Leaf("friedpotato"));

      Composite comp1 = new Composite("Adult Meal");
      promroot.Add(first);
      promroot.Add(second);
      comp1.Add(promroot);


      root.Add(comp1);

      Composite comp2 = new Composite("Child Meal");

      comp2.Add(new Leaf("Ice-cream"));
      comp2.Add(new Leaf("Cracket"));

      root.Add(comp2);
      root.Operation2();

      Console.ReadKey();

    }
  }

class Leaf : Component
  {
    //private String name;
    public Leaf(String name)
      : base(name)
    { this.name = name; }

    public override void Add(Component c)
    { Console.WriteLine("Cannot add item to leaf"); }

    public override void Operation2()
    { Console.WriteLine(name); }

  }

}
```

Данный код выводит окно, представленное на рисунке 3.14.

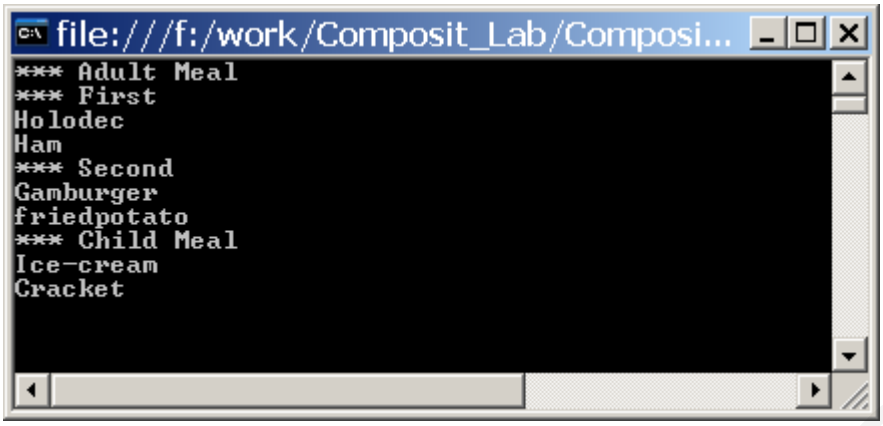



Рисунок 3.14 – Окно паттерна Compositor

Узлы дерева собираются из элементов с базовым классом Component:

```
abstract class Component
   {
      protected String name;
      public Component(String name)
      {
         this.name = name;
      }

      public String Operation()
      {
         return name;
      }

      public abstract void Add(Component c);
      public abstract void Operation2();
   }
```

Базовый класс содержит методы для добавления дочернего узла (Add) и две операции (Operation, Operation2). Собственно Компоновщик представлен классом Composit со следующей реализацией:

```
class Composite : Component
   {
      private List<Component> _items = new List<Component>();
      public Composite(String name):base(name)
      {

      }

      public  override void Add(Component cm)
      {
         _items.Add(cm);
      }
```

```
    public override void Operation2()
    {
      if ((this.Operation().IndexOf("rom") < 0) && (this.Operation().IndexOf("root") < 0))
      Console.WriteLine("*** {0}", this.name);
      foreach (Component cmp in _items)
      {
        //
          cmp.Operation2();
      }
    }
}
```

Видим, что состав всей конструкции определен в коллекции

```
private List<Component> _items = new List<Component>();
```

Добавление нового узла в коллекцию выполняется командой

```
_items.Add(cm);
```

Конечные узлы соответствуют листьям дерева (класс Leaf). Для каждого узла используется свой Компоновщик, формирующий состав этого узла.

### 3.11 Итератор (Iterator)

Перейдем к рассмотрению поведенческих паттернов. Итератор – это механизм перемещения по неупорядоченной коллекции. Мы знаем, что элементы массивов доступны по индексам. В коллекции индексов нет. Поэтому рассматриваемый паттерн обеспечивает две важные функции: создает механизм индексации (он называется индексатор) и реализует навигацию (перемещение) по коллекции. UML-диаграмма паттерна Итератор представлена на рисунке 3.15.

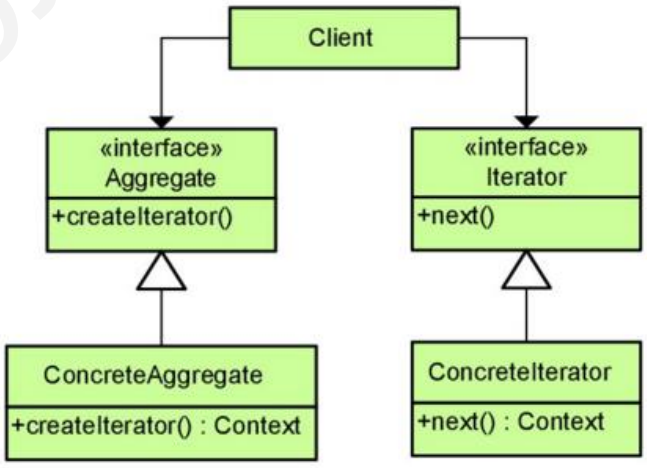



Рисунок 3.15 – UML-диаграмма паттерна Iterator

Коллекция представлена классом ConcreteAggregate. Ее базовый метод – CreateIterator объявлен в интерфейсе или абстрактном классе. Сама программа Итератор реализована интерфейсом и классом ConcreteIterator.

Далее представлен соответствующий программный код:

```
using System;
using System.Collections.Generic;
using System.Collections;

namespace Iterator2
{

   class Elements
   {
      private string _name;

      // Constructor
      public Elements(string name)
      {
         this._name = name;
      }

      // Gets name
      public string Name
      {
         get { return _name; }
      }
   }

interface IAggregate
   {
      Iterator CreateIterator();
   }
   class ConcreteAggegate : IAggregate
   {
      private ArrayList _items = new ArrayList();
      public Iterator CreateIterator()
      {
         return new Iterator(this);
      }

      public int Number
      {
         get { return _items.Count; }
      }

      // Indexer
      public object this[int index]
      {
         get { return _items[index]; }
```

```
            set { _items.Add(value); }
        }
    }

interface IAbstractIterator
    {
        Elements First();
        Elements Next();
        bool IsEnded { get; }
        Elements CurrentItem { get; }
    }

class Iterator : IAbstractIterator
    {
        private ConcreteAggegate _collection;
        private int _current = 0;
        private int _step = 1;

        public Iterator(ConcreteAggegate collection)
        {
            this._collection = collection;
        }

        // Gets first
        public Elements First()
        {
            _current = 0;
            return _collection[_current] as Elements;
        }

        // Gets next
        public Elements Next()
        {
            _current += _step;
            if (!IsEnded)
                return _collection[_current] as Elements;
            else
                return null;
        }

        // Gets-sets
        public int Step
        {
            get { return _step; }
            set { _step = value; }
        }

        // Get current
        public Elements CurrentItem
        {
            get { return _collection[_current] as Elements; }
```

```
        }

        public bool IsEnded
        {
            get { return _current >= _collection.Number; }
        }
    }

class Program
  {

      static ConcreteAggegate coll = new ConcreteAggegate();
      static  Iterator iter = new Iterator(coll);

       static void Main()
       {
          // Build a collection

          coll[0] = new Elements("One");
          coll[1] = new Elements("Two");
          coll[2] = new Elements("Three");
          coll[3] = new Elements("Four");
          coll[4] = new Elements("Five");
          coll[5] = new Elements("Six");
          coll[6] = new Elements("Seven");

          // Create iterator
          // Skip every other item
          iter.Step = 2;

          Console.WriteLine("Iterating over collection:");

          for (Elements item = iter.First();
              !iter.IsEnded; item = iter.Next())
          {
              Console.WriteLine(item.Name);
          }

          // Wait for user
          Console.ReadKey();
      }
  }
}
```

Окно программы Iterator представлено на рисунке 3.16.

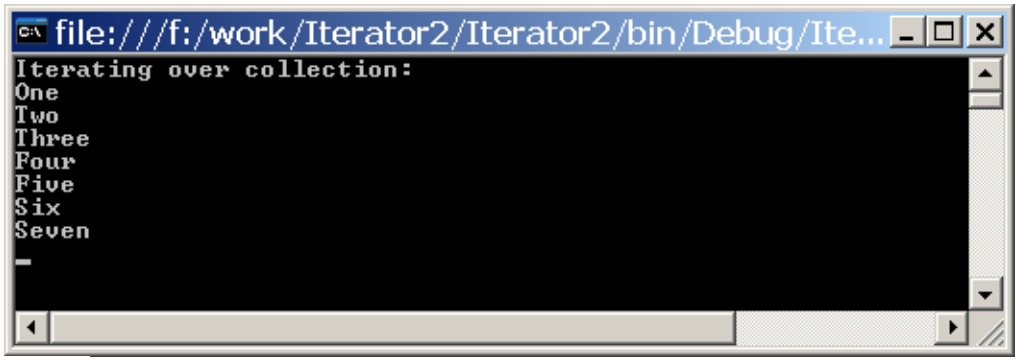



Рисунок 3.16 – Окно программы Iterator

Если изменить значение шага (step) просмотра на iter.step = 2, то в выводе получим только нечетные элементы (рисунок 3.17).

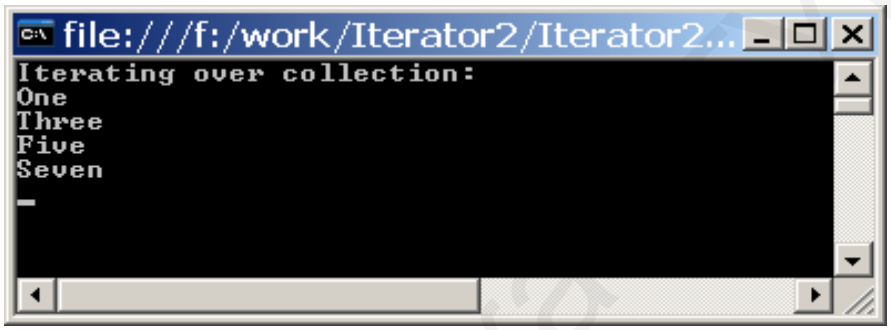



Рисунок 3.17 – Изменения шага просмотра

Прежде всего мы видим, что коллекция фактически создается на базе объекта типа ArrayList класса ConcreteAggregate:

```
private ArrayList _items = new ArrayList();
```

В этом классе реализован индексатор:

```
public object this[int index]
{
    get { return _items[index]; }
    set { _items.Add(value); }
}
```

Эту конструкцию полезно запомнить. Индексатор позволяет обратиться к коллекции по индексу: {return _items[index]. Далее в тексте программы мы видим использование индексатора, например:

```
return _collection[_current] as Elements;
```

Обращение выполняется по номеру (индексу) _current. Навигация по коллекции выполняется с помощью методов First и Next класса Iterator.

Также заметим, что конкретный тип элементов в коллекции заранее не указывается. Роль конкретной коллекции играет класс ConcreteAggregate.

### 3.12 Команда (Command)

Другим поведенческим паттерном является Команда, ее UML-диаграмма показана на рисунке 3.18.

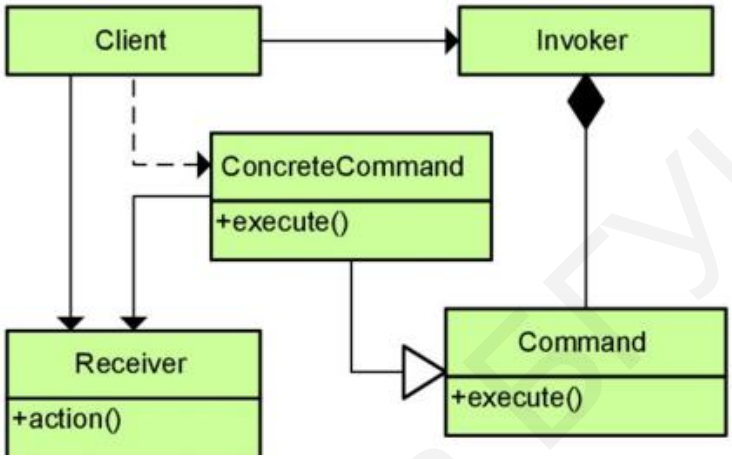



Рисунок 3.18 – UML-диаграмма паттерна Command

Цель этого паттерна – передать команду как объект классу Receiver (получатель). Команда (класс ConcreteCommand) строится от абстрактного класса или интерфейса Command. Эти действия выполняются по запросу клиента через класс Invoker. Команда передается на выполнение в Receiver (метод Action). Данный паттерн позволяет «упаковывать» в объект команды с различными параметрами. В примере ниже объявлен абстрактный класс Command, на основе которого реализован конкретный класс ConcreteCommand. В классе Command объявлен абстрактный метод RunCommand с текстовым параметром. Параметр можно при каждом вызове изменять. Для выполнения команды используется получатель ActionReceiver. Это выполнение инициируется в классе ConcreteCommand:

```
public override void RunCommand(String text)
    {
      aireceiver.Action(text);
    }
```

Для создания Команды и передачи ей на выполнение текстовой строки в нашем примере используется класс ActionManager (играет роль Invoker в диаграмме UML на рисунке 3.18). Далее размещен законченный текст программы:

```
using System;
namespace CommandPattern2
{
   abstract class Command
   {
      protected ActionReceiver aireceiver;

      public Command(ActionReceiver receiver, String text)
      {
         this.aireceiver = receiver;
      }

      public abstract void RunCommand(String text);
   }

   class ConcreteCommand : Command
   {
         public ConcreteCommand(ActionReceiver aireceiver,String txt) :
         base(aireceiver,txt)
      {       }

      public override void RunCommand(String text)
      {
         aireceiver.Action(text);
      }
   }

   class ActionReceiver
   {
      public void Action(String text)
      {
         Console.WriteLine(text);
      }
   }
   class ActionManager
   {
     private Command _command;

      public void SetCommand(Command command)
      {
         this._command = command;
      }

      public void RunCommand(String text)
      {
         _command.RunCommand(text);
      }
   }
   class Program
   {
    static void Main()
```

```
    {

      ActionManager am = new ActionManager();
      ActionReceiver aireceiver = new ActionReceiver();
      Command command = new ConcreteCommand(aireceiver,"");

      am.SetCommand(command);
      am.RunCommand("Hello, new pattern");
      Console.ReadLine();
    }
  }
}
```

Результат работы программы приведен на рисунке 3.19.

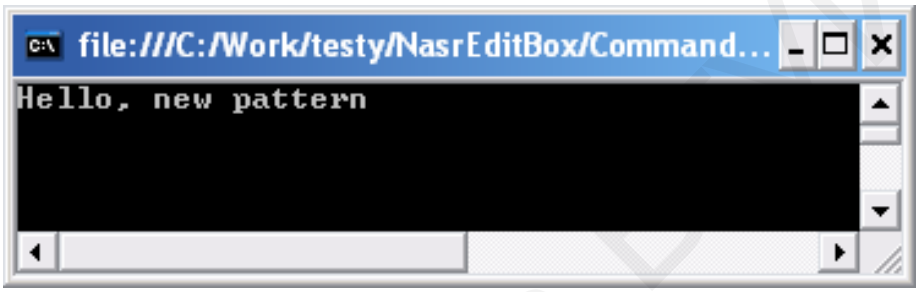



Рисунок 3.19 – Работа паттерна Command

# 4 СПЕЦИФИКАЦИИ НА ОСНОВЕ РЕКУРСИЙ И ПРАВИЛ

## 4.1 Краткие теоретические сведения

Рекурсивные спецификации представляют способ описания функционала программы, занимая «промежуточную» позицию между «чистыми» языками спецификаций (типа Z, B) и обычными программами. Уровень сложности рекурсивных спецификаций не столь высок, как у абстрактных языков спецификаций, что делает их более практичными. Универсальность рекурсивных спецификаций базируется на известном тезисе американского логика А. Черча, согласно которому класс алгоритмов и класс рекурсивных функций суть одно и то же (т. е. каждый алгоритм вычисляет некоторую рекурсивную функцию и каждая рекурсивная функция вычисляется некоторым алгоритмом).

Кроме спецификаций на основе рекурсии можно применять спецификации на основе правили «*если ... то...*».

## 4.2 Базовая концепция

Некоторые функции являются первичными, они формируют базис рекурсивных спецификаций. Другие функции определяются через первичные на основе оператора рекурсии.

Первичную функцию нельзя заменить вычислением более простых функций. Из первичных функций строится с помощью определенных операций все множество известных нам вычислимых функций. Таких первичных функций над целыми положительными числами всего три:

а) функция константы $f(x_1,x_2,...,x_n) = 0;$

б) функция непосредственного следования $f(x) = x + 1;$

в) тождественная функция $f(x_1,x_2,...,x_n) = x_i.$.

Для построения более сложных функций из первичных используют следующие операции:

- Операция подстановки. Заключается в том, что вместо каких-то переменных функции подставляют другие функции от тех же или иных переменных.

Пример. Пусть $f(x) = x + 1$ и $g(y) = y + 1$. Тогда подставив вместо переменной x функцию g(y), получим новую функцию: $f(y) = y + 2$.

- Операция примитивной рекурсии. Можно представить с помощью следующих примеров.

Пример 1. Пусть $h(x)$ – известная вычислимая функция, т. е. известен алгоритм вычисления этой функции для произвольного x. Определим новую функцию f(x) следующим образом:

$$\begin{cases} f(0) = a, \\ f(x+1) = f(x, h(x)). \end{cases}$$

Это и есть определение функции с помощью операции примитивной рекурсии.

Например, пусть $\begin{cases} f(x, 1) = x, \\ f(x, y + 1) = f(x, y) + x. \end{cases}$

Это есть определение целочисленного умножения. Так,

$$f(3,3) = f(3,2) + 3 = f(3,1) + 3 + 3 = 3 + 3 + 3 = 9.$$

Пример 2. Дано $\begin{cases} f(x, 1) = x, \\ f(x, y + 1) = h(x, f(x, y)) \end{cases}$ и $\begin{cases} h(1, x) = x, \\ h(y+1, x) = h(y, x) + x. \end{cases}$

Это определение позволяет находить значение $x^y$.

Рассмотрим следующий пример вычисления $3^2$:

$$f(3,1+1) = h(3,f(3,1)) = h(3,3) = h(2+1,3) = h(2,3)+3 = h(1+1,3)+3 = \\ = h(1,3)+3+3 = 3+3+3 = 9.$$

Определение. Функция, определяемая из простейших с помощью операций подстановки и/или примитивной рекурсии, называется *примитивно рекурсивной*.

Имеется еще одна операция для построения рекурсивных функций. Однако она не всегда дает результат или, как говорят, не является полностью определенной (тотальной). Она называется операцией минимизации (взятия минимального корня). Ее определение следующее:

$f(x_1,x_2,\ldots, x_n,y)$ = минимальному значению y, при котором $f(x_1,x_2,\ldots, x_n,y) = 0$.

Эту операцию можно для простоты обозначить как $\mu_y f(x, y)$.

Пример 3. Пусть $f(x, y) = x^y - 2\cdot x - 3$. Тогда $\mu_y f(3, y) = 2$.

Определение. Функция, определяемая из простейших с помощью операций подстановки примитивной рекурсии и минимизации, называется *частично рекурсивной*.

В дальнейшем мы будем строить функции над списками целых чисел. В качестве первичных будем использовать функции head – получение первого

элемента (головы) списка; tail – получение списка, из которого удален первый элемент; add – присоединение к списку нового элемента.

### 4.3 Функции head, tail, add

Реализация первичных функций представлена далее кодом и скриншотом. Функции объявлены как статические. Функция head имеет такую реализацию:

```
static object head(int[] arr)
   {
      if (arr.Length > 0)
          return arr[0];
      return null;
    }
```

Функция возвращает значение типа Object (в случае пустого списка возвращается null). Поэтому необходимо приведение к типу int.

Функция tail имеет следующую реализацию:

```
static object[] tail(int[] arr)
   {
        if (arr.Length > 0)
        {
           int k = arr.Length;
           object[] ans = new object[k];
           for (k = 1; k < ans.Length; k++)
           {
              ans[k-1] = arr[k];
           }
           return (object[])ans;
     }
        else
        return null;
      }
```

Также возвращается массив типа Object. Для конвертации в массив типа int используем функцию convert:

```
static int[] convert(object[] arr)
  {
     int[] arr2 = new int[arr.Length-1];

     int k = 0;
        foreach (var z in arr)
        {

           if (z != null)
```

```
            {
                arr2[k++] = (int)z;
            }
        }
    return arr2;
}
```

Логика функции add легко читается из общего кода, приведенного ниже:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace PrimaryFunctions
{
   class Program
   {
      static object head(int[] arr)
      {
         if (arr.Length > 0)
            return arr[0];
         return null;
      }

      static object[] tail(int[] arr)
      {
         if (arr.Length > 0)
         {
            int k = arr.Length;
            object[] ans = new object[k];
            for (k = 1; k < ans.Length; k++)
            {
               ans[k-1] = arr[k];
            }
            return (object[])ans;
         }
         else
         return null;
      }

      static int[] add(int[] arr, int z)
      {
         int[] ans = new int[arr.Length + 1];

         for (int k = 0; k < arr.Length; k++)
         {
            ans[k] = arr[k];
         }

         ans[ans.Length - 1] = z;
```

```
        return ans;
    }

    static int[] convert(object[] arr)
    {
        int[] arr2 = new int[arr.Length-1];

        int k = 0;
        foreach (var z in arr)
        {

            if (z != null)
            {
                arr2[k++] = (int)z;
            }
         }
        return arr2;
    }

    static void Main(string[] args)
    {
        int[] ans = { 1, 3, 0, 8 };
        Console.WriteLine((int)head(ans));
        int[] arr2 =new int[ans.Length-1];

        int[] objarr = convert(tail(ans));
        Console.WriteLine("\n\n");
        foreach(int z in objarr)
        {
            Console.WriteLine(z);
        }

        objarr = add(objarr, 25);

        Console.WriteLine("\n\n");
        foreach (int z in objarr)
        {
            Console.WriteLine(z);
        }
        Console.ReadKey();
    }
  }
}
```

В методе Main создали контрольный массив и реализовали вызов функций head, tail, add со следующим выходным скриншотом (рисунок 4.1).

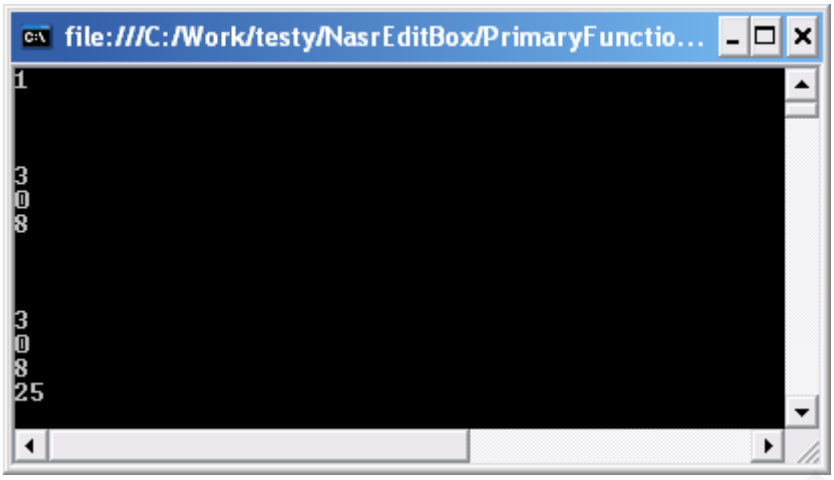



Рисунок 4.1 – Проверка первичных функций

### 4.4 Простые рекурсивные спецификации

Используя первичные рекурсивные функции из предыдущего подраздела, построим рекурсивные спецификации следующих функций: **sum** (сумма всех элементов списка), **sum_inv** (подсчет числа инверсий), **getItem** (выдать элемент, стоящий в заданной позиции), **intersect** (выдать список общих элементов двух списков), **min** (найти минимальный элемент). Функция sum вместе с ее вызовом реализуется следующим образом:

```
static int sum(int[] list, int res)
   {
        if (list.Length == 0)
           return res;

        int newres = res+(int)head(list);
           return sum(convert(tail(list)),newres);
    }
 static void Main(string[] args)
    {
        int[] ans = { 1, 3, 0, 8 };
        Console.WriteLine(sum(ans,0));
        Console.ReadKey();
     }
```

В рекурсивном вызове не забываем конвертировать объектный массив в массив целых чисел: sum(convert(tail(list)),newres). В тексте рекурсивной спецификации сначала проверяем длину массива, затем к накопленной сумме элементов res добавляем первый элемент массива:

```
int newres = res+(int)head(list);
```

а хвост (tail) передаем в рекурсивный вызов.

Вычисление числа инверсий реализуется похожим способом. Под инверсией понимается ситуация, когда следующий элемент списка меньше предыдущего. Соответствующая рекурсивная спецификация функции имеет следующий вид:

```
static int sum_inv(int[] list, int res)
   {
       if (list.Length <= 1)
          return res;
       if((int)head(list)>(int)head(convert(tail(list))))
          res++;
       return sum_inv(convert(tail(list)), res);
    }
```

Проверку наличия инверсии выполняет оператор

```
if((int)head(list)>(int)head(convert(tail(list))))
```

Функция getItem получения элемента, стоящего на указанной позиции, рекурсивно реализуется следующим образом:

```
static object getItem(int[] list, int ind)
     {
       if (list.Length <= 0)
          return null;
       if (ind==0)
          return (int)head(list);
          return getItem(convert(tail(list)), ind-1);
     }
```

Позиция элемента определяется параметром ind. При каждом рекурсивном вызове уменьшаем значение позиции на единицу и одновременно удаляем из списка первый элемент. Значение возвращаем, когда ind обнулится.

Обратимся теперь к функции пересечения двух списков. Нам потребуется сначала определить функцию member (проверяет, является ли элемент членом списка). Рекурсивное определение функции member имеет следующий вид:

```
static bool member(int[] list, int item)
     {
       if (list.Length <= 0)
          return false;
       int z=(int) head(list);
       if (z==item)
          return true;
       return member(convert(tail(list)),item);
     }
```

Теперь рекурсивная спецификации функции для отыскания пересечения двух списков такова:

```
static int[] intersect(int[] list1, int[] list2)
      {
         if (list1.Length <= 0)
            return list1;

         if (list2.Length <= 0)
            return list1;

         if (member(list2,(int)head(list1)))
            return add(intersect(convert(tail(list1)),list2),(int)head(list1));

            return intersect(convert(tail(list1)), list2);
      }
```

Проверку этой функции выполним таким образом:

```
static void Main(string[] args)
      {
         int[] list1 = { 1, 3, 0, 8 };
         int[] list2 = { 5, 3, 0, 9, 7 };

         int[] objarr = intersect(list1,list2);

         foreach(int z in objarr)
         {
          Console.WriteLine(z);
         }
              Console.ReadKey();
      }
```

Получаем выходную консоль, представленную на рисунке 4.2.

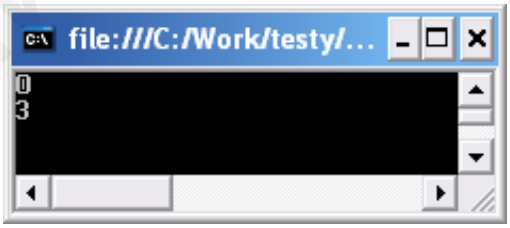



Рисунок 4.2 – Выходная консоль после инвертирования списка

Элементы в результирующем списке расположились в обратном порядке. Поэтому реализуем функцию для инверсии списка – inversion:

```
static int[] inversion(int[] list1)
      {
         if (list1.Length <= 0)
            return list1;
         return add(inversion(convert(tail(list1))),(int)head(list1));
```

```
}
```

С учетом этого результата предыдущую функцию перепишем таким образом:

```
static void Main(string[] args)
    {
      int[] list1 = { 1, 3, 0, 8 };
      int[] list2 = { 5, 3, 0, 9, 7 };
      int[] objarr = inversion(intersect(list1,list2));

      foreach(int z in objarr)
      {
       Console.WriteLine(z);
      }
         Console.ReadKey();
    }
```

Результат работы программы приведен на рисунке 4.3.

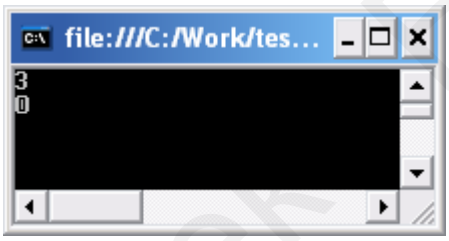



Рисунок 4.3 – Корректный вывод списка

Функция нахождения минимального элемента в списке реализуется следующим образом:

```
static object min(int[] list)
    {
      if (list.Length <= 0)
         return null;
      if (list.Length == 1)
         return head(list);

      if((int)head(list)< (int)min(convert(tail(list))))
         return head(list);
      return min(convert(tail(list)));
    }
```

Более сложные рекурсивные спецификации значительно труднее читать (соответственно определять).

### 4.5 Рекурсивные спецификации для работы с текстом

По аналогии с предыдущим материалом реализуем первичные функции для работы с текстом и словами, а именно:

- headS – получить первое слово предложения;
- headC – получить первую букву слова;
- tailS (tailC) – получить часть предложения (слова) без первого слова (первой буквы);
- addS (addC) – добавить слово (букву).

Программная реализация на C# имеет следующий вид:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace PrimarySentence
{
   class Program
   {
      static String headS(String arr)
      {
         if (arr.Length > 0)
         {
            int k = arr.IndexOf(" ");
            if (k == 0)
               return arr;
            return arr.Substring(0,k).Trim();

         }
         return "";
      }

      static String tailS(String arr)
      {
         if (arr.Length > 0)
         {
            int k = arr.IndexOf(" ");
            if (k == 0)
               return arr;
            return arr.Substring(k, arr.Length - k).Trim();
         }
         return "";

      }
      static String addS(String arr, String z)
      {
         return arr+" "+z;
      }
```

```
        static String headC(String arr)
        {
            if (arr.Length > 0)
            {
                return arr.Substring(0, 1).Trim();
            }
            return "";
        }
        static String tailC(String arr)
        {
            if (arr.Length > 0)
            {
                return arr.Substring(1, arr.Length - 1).Trim();
            }
            return "";
        }

        static String addC(String arr, String z)
        {
            return arr + z;
        }

        static void Main(string[] args)
        {
            String z = "Hello, fellow-programmer";
            Console.WriteLine(headS(z));
            Console.WriteLine(headC(z));
            Console.WriteLine(tailS(z));
            Console.WriteLine(tailC(z));
            Console.WriteLine(addS(z,". How are you?"));
            Console.ReadKey();
        }
    }
}
```

Окно программы для проверки первичных функций для работы с текстом и словами показано на рисунке 4.4.

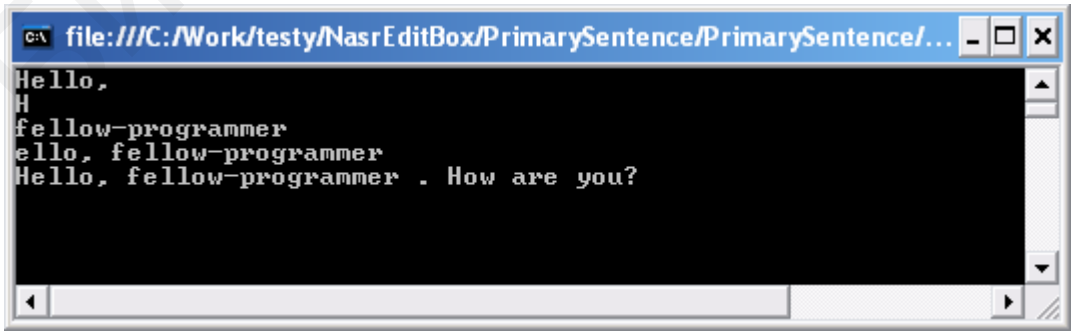



Рисунок 4.4 – Выходное окно программы для проверки первичных текстовых функций

Теперь получим рекурсивные спецификации следующих функций:

- возврата части текста, заключенного между двумя заданными позициями – getSubstring;
- инвертирования строки (слова записываем в обратном порядке) – strReverse;
- вставка строки в текст в указанной позиции – insertStr;
- удаление части строки с указанной позиции для заданной длины удаляемой строки – removeStr;

Функция getSubstring рекурсивно определяется следующим образом:

```
static String getSubstring(String str, int k1, int k2)
    {
      if (str.Length <= 0)
                    return "";

     if (k1>0)
        {
            return getSubstring(str.Substring(1,str.Length - 1), --k1,  k2);
         }

        if (k2>str.Length)
            k2= str.Length;
        return str.Substring(0, k2);
    }
```

Функция инвертирования реализуется следующим образом:

```
static String strReverse (String str1, String str2)
    {
      if (str1.Length <= 0)

            return str2;

      int k2=str1.IndexOf(" ");
            String str3=str1;
            if (k2 > 0)
            {
              str3 = str1.Substring(0, k2);
              return strReverse(str1.Substring(k2, str1.Length - k2), str3 + " " + str2);
            }
            if (k2<=0)
            k2=0;
            return str3 +" " + str2;
    }
```

Функция insertStr реализуется таким образом:

```
static String insertStr(String str1, String str2, int k1)
    {
```

```
        if (str1.Length <= 0)
              return str2;

        if (k1>str1.Length)
            {
              return str2;
            }

        if(k1>0)
        return  str1.Substring(0,1)+ insertStr (str1.Substring(1,str1.Length - 1), str2,--k1);

       return str2 + " "+str1;
}
```

Функция removeStr рекурсивно определяется следующим образом:

```
static String removeStr(String str1,  int k1, int k2)
  {
        if (str1.Length <= 0)
                    return "";

        if (k1>str1.Length)
            {
              return str1;
            }
    if((k1>0)&&(k2> 0))
      return  str1.Substring(0,1) +removeStr (str1.Substring(1,str1.Length-1),--k1,k2);
     if((k1==0)&&(k2==0))
            return str1;

     if((k1==0)&&(k2>0))
      return removeStr(str1.Substring(1,str1.Length-1),k1,--k2);
      return str1;
}
```

Объединим все функции в одно приложение:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace RecurSpecText
{
   class Program
   {
            static String removeStr(String str1,  int k1, int k2)
      {
        if (str1.Length <= 0)
                    return "";
```

```
        if (k1>str1.Length)
            {
                return str1;
            }

    if((k1>0)&&(k2> 0))
      return  str1.Substring(0,1) + removeStr (str1.Substring(1,str1.Length-1),--k1,k2);

     if((k1==0)&&(k2==0))
            return str1;

     if((k1==0)&&(k2>0))
            return removeStr(str1.Substring(1,str1.Length-1),k1,--k2);
     return str1;
      }

static String insertStr(String str1, String str2, int k1)
       {
              if (str1.Length <= 0)
                 return str2;

              if (k1 > str1.Length)
              {
                 return str2;
              }

              if (k1 > 0)
                 return str1.Substring(0, 1) + insertStr(str1.Substring(1, str1.Length - 1), str2, --k1);

              return str2 + " " + str1;
       }
static String strReverse (String str1, String str2)
      {
         if (str1.Length <= 0)
              return str2;

         int k2=str1.IndexOf(" ");
              String str3=str1;
              if (k2 > 0)
              {
                 str3 = str1.Substring(0, k2);
                 return strReverse(str1.Substring(k2, str1.Length - k2), str3 + " " + str2);
              }
              if (k2<=0)
              k2=0;
              return str3 +" " + str2;
      }


static String getSubstring(String str, int k1, int k2)
      {
```

```
    if (str.Length <= 0)
                    return "";

   if (k1>0)
        {
            return getSubstring(str.Substring(1,str.Length - 1), --k1,  k2);
         }

        if (k2>str.Length)
            k2= str.Length;
        return str.Substring(0, k2);
}

static String headS(String arr)
{
    if (arr.Length > 0)
    {
        int k = arr.IndexOf(" ");
        if (k == 0)
            return arr;
        return arr.Substring(0, k).Trim();
    }
    return "";
}

static String tailS(String arr)
{
    if (arr.Length > 0)
    {
        int k = arr.IndexOf(" ");
        if (k == 0)
            return arr;
        return arr.Substring(k, arr.Length - k).Trim();
    }
    return "";
}

static String addS(String arr, String z)
{
    return arr + " " + z;
}

static String headC(String arr)
{
    if (arr.Length > 0)
    {
        return arr.Substring(0, 1).Trim();
    }
    return "";
}
```

```
        static String tailC(String arr)
        {
            if (arr.Length > 0)
            {
                return arr.Substring(1, arr.Length - 1).Trim();
            }
            return "";
        }

        static String addC(String arr, String z)
        {
            return arr + z;
        }

        static void Main(string[] args)
        {
            String z = "Hello, fellow-programmer";
            Console.WriteLine(insertStr(z,"dear",7));
           // Console.WriteLine(removeStr(z,6,7));
           // Console.WriteLine(headC(z));
           // Console.WriteLine(tailS(z));
           // Console.WriteLine(tailC(z));
          //  Console.WriteLine(addS(z, ". How are you?"));
              Console.ReadKey();
        }
    }
}
```

### 4.6 Спецификации на основе правил

Достоинство этого типа спецификаций состоит в возможности сравнительно простого изменения, добавления или удаления правил. Это важно при необходимости реинжиниринга системы, поскольку базовые механизмы, используемые при обработке правил, остаются неизменными.

В качестве примера рассмотрим систему управления лифтом. Для создания правил введем переменные:

x0 – вызов есть;
x1 – лифт ниже точки вызова;
x2 – лифт выше точки вызова;
x3 – кабина пуста;
x4 – двери открыты.

Все переменные имеют логический тип. Нам понадобятся переменные управления:

c1 – открыть двери;

c2 – закрыть двери;
c3 – поднять лифт вверх на один этаж;
c4 – опустить лифт вниз на один этаж;
c5 – остановить лифт.

Используя эти обозначения, составим правила управления лифтом:

```
// Закрываем двери пустой кабины и поднимаем лифт на этаж вверх при вызове сверху

if (x0==1)&( x3==1) &( x4==1)&( x1==1)
{
  c2();
  c3();
}

// Закрываем двери пустой кабины и опускаем лифт на этаж вниз при вызове снизу

if (x0==1)&( x3==1) &( x4==1)&( x2==1)
{
  c2();
  c4();
}

// Поднимаем лифт на этаж вверх при вызове сверху

if (x0==1)& (x4==0)&( x3==1) &( x1==1)
{
  c3();
}

// Опускаем лифт на этаж вниз при вызове снизу

if (x0==1)& (x4==0)&( x3==1) &( x2==1)
{
  c4();
}

// Останавливаем лифт и открываем двери при попадании в точку вызова

if (x0==1)&( x1==0) &( x2==0)&( x3==0)
{
```

```
c5();
c1();
}
```

Приведем программную реализацию:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace Lift
{
   /*

x0 - вызов есть
x1 - лифт ниже точки вызова
x2 - лифт выше точки вызова
x3 - кабина пуста
x4 - двери открыты
curlift - точка лифта;
curfloor - этаж;

c1 - открыть двери
c2 - закрыть двери
c3 - поднять лифт вверх на один этаж
c4 - опустить лифт вниз на один этаж
c5 - остановить лифт

// Закрываем двери пустой кабины и поднимаем лифт на этаж вверх при вызове сверху

if (x0==1)&( x3==1) &( x4==1)&( x1==1)
{
  c2();
  c3();
}

// Закрываем двери пустой кабины и опускаем лифт на этаж вниз при вызове снизу

if (x0==1)&( x3==1) &( x4==1)&( x2==1)
{

    c2();
    c4();
}

// Поднимаем лифт на этаж вверх при вызове сверху

if (x0==1)& (x4==0)&( x3==1) &( x1==1)
{
```

```
  c3();
}

// Опускаем лифт на этаж вниз при вызове снизу

if (x0==1)& (x4==0)&( x3==1) &( x2==1)
{
  c4();
}

// Останавливаем лифт и открываем двери при попадании в точку вызова

if (x0==1)&( x1==0) &( x2==0)&( x3==0)
{
  c5();
  c1();
}
   */

class Program
  {
      static int x0 = 1;
      static int x1 = 0;
      static int x2 = 1;
      static int x3 = 1;
      static int x4 = 0;
      static int curlift = 4;
      static int callfloor = 1;
      static void c1()
      {
         if (x4 == 0)
         {
            x4 = 1;
            Console.WriteLine("Двери открыты");
         }
      }

      static void c2()
      {
         if (x4 == 1)
         {
            x4 = 0;
            Console.WriteLine("Двери закрыты");
         }
      }

      static void c3()
      {
         curlift++;
           if (curlift == callfloor)
         {
```

```
            x1 = 0;
            x2 = 0;
            Console.WriteLine("Лифт в точке вызова. Этаж {0}",curlift);
        }
    }

    static void c4()
    {
        curlift--;
        if (curlift == callfloor)
        {
            x1 = 0;
            x2 = 0;
            Console.WriteLine("Лифт в точке вызова. Этаж {0}", curlift);
        }
    }

    static void c5()
    {
        curlift=callfloor;
            x1 = 0;
            x2 = 0;
    }

    static void Main(string[] args)
    {
        while (true)
        {
            if ((x0 == 1) && (x3 == 1) && (x4 == 1) && (x1 == 1))
            {
                c2();
                c3();
                continue;
            }

 // Закрываем двери пустой кабины и опускаем лифт на этаж вниз при вызове снизу

            if ((x0 == 1) && (x3 == 1) && (x4 == 1) && (x2 == 1))
            {
                c2();
                c4();
                continue;
            }

// Поднимаем лифт на этаж вверх при вызове сверху
            if ((x0 == 1) && (x4 == 0) && (x3 == 1) && (x1 == 1))
            {
                Console.WriteLine("Двигаем вверх. Этаж: {0}", curlift);
                c3();
                continue;
            }
```

```
    // Опускаем лифт на этаж вниз при вызове снизу

            if ((x0 == 1) && (x4 == 0) && (x3 == 1) && (x2 == 1))
            {
               Console.WriteLine("Двигаем вниз. Этаж: {0}", curlift);
               c4();
               continue;
            }

    // Останавливаем лифт и открываем двери при попадании в точку вызова

            if ((x0 == 1) && (x1 == 0) && (x2 == 0) && (x3 == 1))
            {
               Console.WriteLine("Открываем двери");
               c5();
               c1();
               break;
            }

            Console.WriteLine("Движение завершено");
            break;
         }

         Console.ReadLine();
      }
   }
}
```

Лифт вызывается на первый этаж. Исходная позиция лифта – четвертый этаж. Скриншот с иллюстрацией движения приведен на рисунке 4.5.

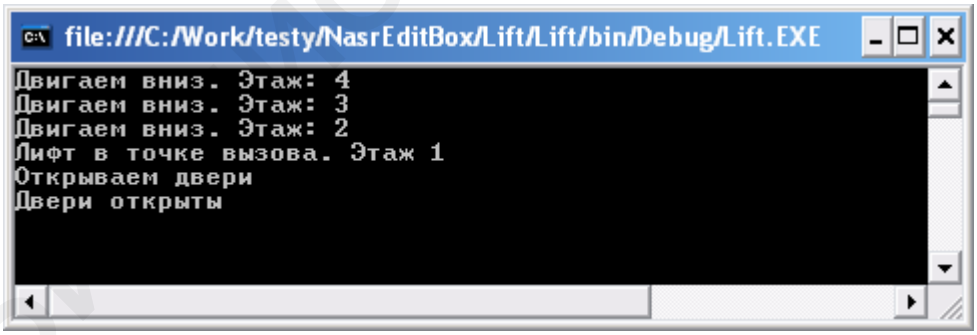



Рисунок 4.5 – Программа движения лифта вниз на первый этаж

Поменяем исходную диспозицию: лифт вызывается на четвертый этаж с первого (рисунок 4.6).

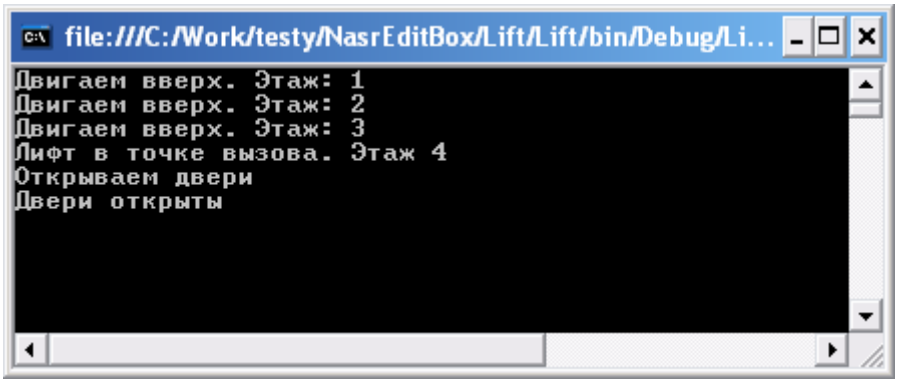



Рисунок 4.6 – Программа движения лифта вверх на четвертый этаж

Теперь изменим программу: организуем доставку пассажира в лифте. Для этой цели просто добавим в систему два новых правила:

```
if ((x0 == 1) && (x4 == 0) && (x3 == 0) && (callfloor>curlift))
            {
               Console.WriteLine("Двигаем вверх. Этаж: {0}", curlift);
               c3();
               continue;
            }
 if ((x4 == 1) && (x3 == 1) && (x1 == 0) && (x2 == 0))
            {
               Console.WriteLine("Условно вводим этаж 5 для доставки пассажира");
               x4 = 0;
               x3 = 0;
               callfloor = 5;
               Console.WriteLine("Двери закрыты - двигаемся на этаж 5");
               continue;   }
```

Первое из этих правил определяет условия перемещения лифта вверх с пассажиром в кабине. Второе правило закрывает двери в кабине после входа в кабину пассажира, изменяя соответствующие системные переменные. Программа теперь получает такой итоговый вид:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace Lift
{
   /*
x0 - вызов есть
x1 - лифт ниже точки вызова
x2 - лифт выше точки вызова
x3 - кабина пуста
x4 - двери открыты
```

curlift - точка лифта;
curfloor - этаж;

c1 - открыть двери
c2 - закрыть двери
c3 - поднять лифт вверх на один этаж
c4 - опустить лифт вниз на один этаж
c5 - остановить лифт

Используя эти обозначения, составим правила управления лифтом:

```
// Закрываем двери пустой кабины и поднимаем лифт на этаж вверх при вызове сверху

if (x0==1)&( x3==1) &( x4==1)&( x1==1)
{
  c2();
  c3();
}

// Закрываем двери пустой кабины и опускаем лифт на этаж вниз при вызове снизу

if (x0==1)&( x3==1) &( x4==1)&( x2==1)
{
  c2();
  c4();
}
// Поднимаем лифт на этаж вверх при вызове сверху

if (x0==1)& (x4==0)&( x3==1) &( x1==1)
{
  c3();
}
// Опускаем лифт на этаж вниз при вызове снизу

if (x0==1)& (x4==0)&( x3==1) &( x2==1)
{
  c4();
}

// Останавливаем лифт и открываем двери при попадании в точку вызова
if (x0==1)&( x1==0) &( x2==0)&( x3==0)
{
  c5();
  c1();
}
   */

   class Program
   {
      static int x0 = 1;
      static int x1 = 0;
```

```
static int x2 = 1;
static int x3 = 1;
static int x4 = 0;
static int curlift = 4;
static int callfloor = 1;

static void c1()
{
   if (x4 == 0)
   {
      x4 = 1;
      Console.WriteLine("Двери открыты");
   }
}
static void c2()
{
   if (x4 == 1)
   {
      x4 = 0;
      Console.WriteLine("Двери закрыты");
   }
}
static void c3()
{
   curlift++;
   if (curlift == callfloor)
   {
      x1 = 0;
      x2 = 0;
      Console.WriteLine("Лифт в точке вызова. Этаж {0}", curlift);
   }
}
static void c4()
{
   curlift--;
   if (curlift == callfloor)
   {
      x1 = 0;
      x2 = 0;
      Console.WriteLine("Лифт в точке вызова. Этаж {0}", curlift);
   }
}

static void c5()
{
   curlift = callfloor;
   x1 = 0;
   x2 = 0;
}

static void Main(string[] args)
```

```
{
   while (true)
   {
      if ((x0 == 1) && (x4 == 0) && (x3 == 0) && (callfloor>curlift))
      {
         Console.WriteLine("Двигаем вверх. Этаж: {0}", curlift);
         c3();
         continue;
      }
```

Выходной скриншот этой программы представлен на рисунке 4.7.

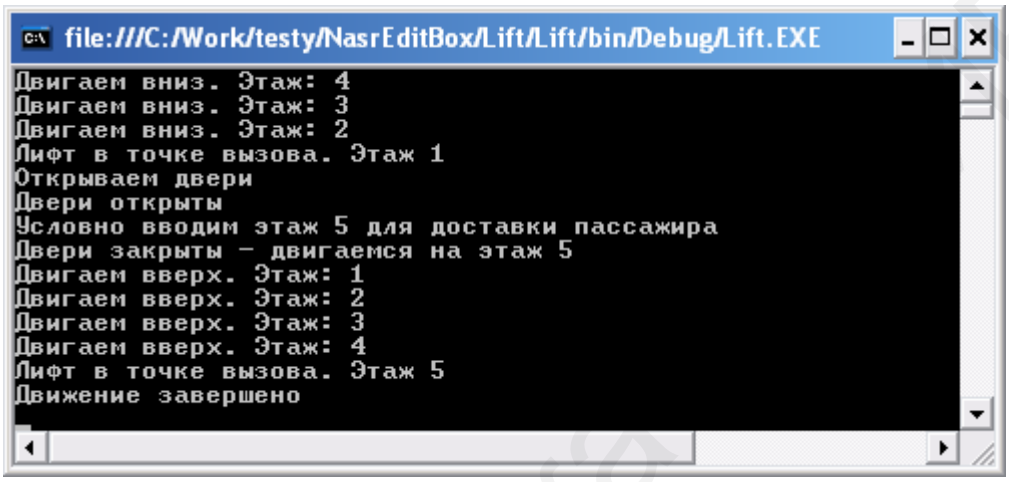



Рисунок 4.7 – Добавление правила для доставки пассажира вверх

Итак, использование правил упрощает задачу «наращивания» системы и учета новых условий. В общем случае правила можно записывать в произвольном порядке (за некоторыми исключениями). Упрощается также отладка, поскольку каждое правило можно проверять независимо от других правил.

## 4.7 Linq-выражения

С помощью Linq-спецификаций можно производить выбор одного и более объектов (элементов) из множества объектов по условию. В качестве примера приведем следующее выражение:

```
var elements = arr.Where(x => test_(x));
```

Здесь формируется выборка из массива arr тех элементов, которые удовлетворяют условию (функции) test_(x). Пример такой функции:

```
 static bool test_(int x)
    {
```

```
        return ((x > 2) && (x < 12));
    }
```

Законченное приложение может иметь следующий вид:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace Linq_Lab1
{
   class Program
   {
      static bool test_(int x)
      {
         return ((x > 2) && (x < 12));
      }
      static void Main(string[] args)
      {
         int[] arr = { 1, 2, 5, 9, 12 };
         var elements = arr.Where(x => test_(x));

         foreach (var z in elements)
         {
            Console.Write(z + "\n");
         }
         Console.ReadLine();
      }
   }
}
```

Выходной результат представлен на рисунке 4.8.

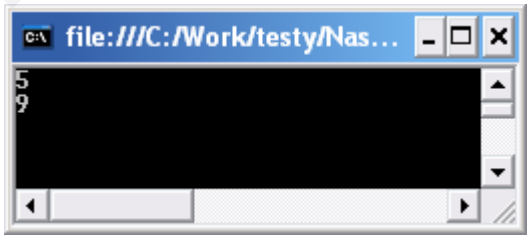



Рисунок 4.8 – Выбор элементов из множества по Linq-выражению

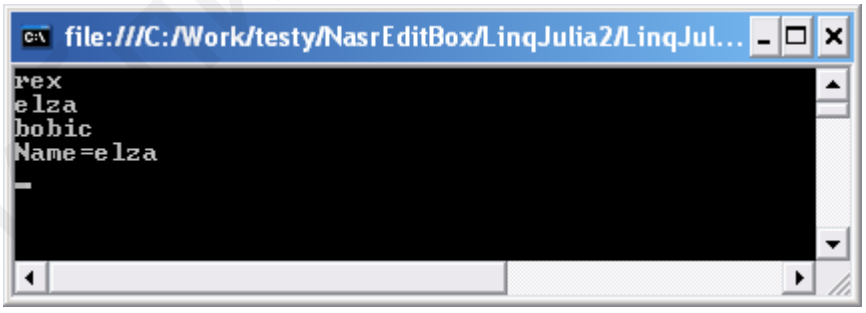
Рассмотрим второй пример подобного типа:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
```

```
namespace LinqJulia2
{
   class Dog
   {
      public string Name { set; get; }
      public string Breed { set; get; }
   }
   class Program
   {
      static void Main(string[] args)
      {
         List<Dog> dl = new List<Dog>() {
            new Dog { Name = "rex", Breed = "taxa" },
            new Dog { Name = "elza", Breed = "bolonka"},
            new Dog { Name = "bobic", Breed = "foxterrier"}};

         var names = dl.Select(x => x.Name);
         foreach (var t in names)
            Console.WriteLine("" + t);
         IEnumerable<Dog> quer = from ditem in dl
                                 where ditem.Breed.Equals("bolonka")
                                 select ditem;
         foreach (var z in quer)
         {
            Console.WriteLine("Name=" + z.Name);
         }
         Console.ReadKey();
      }
   }
}
```

Работа программы представлена на рисунке 4.9.




Рисунке 4.9 – Окно программы для второго примера

Во втором примере использованы два Linq-выражения. Первое выражение:

```
var names = dl.Select(x => x.Name);
```

Из списка dl выбираем все имена объектов. Обозначение x играет формальную роль. Можно использовать любое допустимое имя. Но выбираем не объекты x, а имена (поля Name).

Вторая спецификация имеет следующий вид:

```
IEnumerable<Dog> quer = from ditem in dl
                        where ditem.Breed.Equals("bolonka")
                        select ditem;
```

Здесь формируем перечисление quer. Дословно: quer состоит из элементов ditem (ditem можно заменить на любое другое имя) из dl таких, что

```
ditem.Breed.Equals("bolonka")
```

Следующий пример закрепляет сказанное:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace JuliaLinq3
{
   class Program
   {
      static void Main()
      {

         string[] words = { "hello", "wonderful", "LINQ", "beautiful", "world" };

              var shortWords = from word in words where word.Length <= 5 select word;

         //Print
         foreach (var word in shortWords)
         {
            Console.WriteLine(word);
         }

         Console.ReadLine();
      }
   }
}
```

Здесь Linq-выражение имеет следующий вид:

```
var shortWords = from word in words where word.Length <= 5 select word;
```

Оно позволяет выбрать из списка слов те, длина которых не превосходит пять символов. Скриншот работы программы представлен на рисунке 4.10.

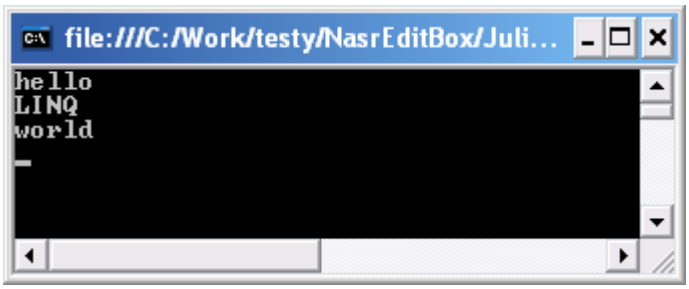



Рисунок 4.10 – Отбор слов по длине

Linq-конструкции можно применить к классам. Рассмотрим такой пример:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace JuliaLinq4
{
   class User
   {
      public string Name { get; set; }
      public int Age { get; set; }
      public List<string> Languages { get; set; }
      public User()
      {
         Languages = new List<string>();
      }
   }
   class Program
   {
      static void Main(string[] args)
      {
         List<User> users = new List<User>
{
   new User {Name="Иван", Age=21, Languages = new List<string> {"английский", "немецкий"
}},
   new User {Name="Борис", Age=22, Languages = new List<string> {"английский",
"французский" }},
   new User {Name="Илья", Age=21, Languages = new List<string> {"английский",
"норвежский" }},
   new User {Name="Франек", Age=24, Languages = new List<string> {"польский", "немецкий"
}}
};

var selectedUsers = from user in users
                    where user.Age > 21
```

```
                select user;
foreach (User user in selectedUsers)
   Console.WriteLine("{0}   {1}",user.Name, user.Age);
Console.ReadLine();
      }
   }
}
```

Здесь создается перечисляемый список объектов класса User:

```
List<User> users = new List<User>
{
   new User {Name="Иван", Age=21, Languages = new List<string> {"английский", "немецкий"
}},
   new User {Name="Борис", Age=22, Languages = new List<string> {"английский",
"французский" }},
   new User {Name="Илья", Age=21, Languages = new List<string> {"английский",
"норвежский" }},
   new User {Name="Франек", Age=24, Languages = new List<string> {"польский", "немецкий"
}}
};
```

Полезно запомнить эту общую конструкцию. Затем применяем Linq-запрос:

```
var selectedUsers = from user in users
                where user.Age > 21
                select user;
```

По форме этот запрос повторяет предыдущие конструкции. Результат представлен на рисунке 4.11.


Рисунок 4.11 – Linq-запрос к коллекции объектов

### 4.8 Технологии линейки IDEF

Технология **IDEF** включает линейку различных диаграмм от IDEF0 до IDEF11. В отличие от языка UML, IDEF использует функцию (процесс) в качестве основного элемента диаграммы.

Диаграмма IDEF0 связана с построением иерархической системы функциональных схем. Каждая последующая диаграмма (схема) раскрывает реализацию того или иного функционального блока. Сначала строится самая общая функциональная диаграмма, которая далее последовательно раскрывается через другие диаграммы. Диаграммы содержат блоки и стрелки. Блоки представляют выполняемые функции. Стрелки связывают блоки и показывают их взаимодействие.

Функциональные блоки (работы) изображаются в форме прямоугольников, представляющих процессы, функции или задачи. Стрелки, исходящие с правой стороны блока, представляют результаты выполнения данного блока, стрелки, входящие в блок слева, – входные данные блока. Стрелки, входящие в блок снизу, соответствуют управляющим воздействиям. Наконец, стрелки, входящие в блок сверху, представляют параметры среды, режимы работы, уровни ошибок и т. п.

**IDEF0** требует, чтобы в диаграмме было не менее трех и не более шести блоков. Эти ограничения поддерживают сложность диаграмм и модели на уровне, доступном для чтения, понимания и использования.

Таким образом, IDEF реализует модель нисходящего проектирования программ на основе принципа пошаговой детализации.

Для создания IDEF-диаграмм можно использовать инструментальную систему BPWIN, MS Visio и другие подобные средства. В качестве примера рассмотрим диаграмму, представленную на рисунке 4.12.


Рисунок 4.12 – Пример диаграммы IDEF0

На диаграмме изображены три функции: определение категории пользователей, определение полномочий и открытие доступа в базу данных системы. Функция определения полномочий использует выход первого блока – определение категорий пользователей – как управляющий вход и имя клиента как информационный вход. Управляющие воздействия в каждом блоке связаны с функциями мониторинга системы. Категория пользователя выступает как информация, регулирующая определение полномочий.

Технология **IDEF3** расширяет набор средств описания модели, например, за счет логических связок (И, ИЛИ, НЕ) или использования отношения предшествования на множестве работ. Примером служит диаграмма, изображенная на рисунке 4.13.

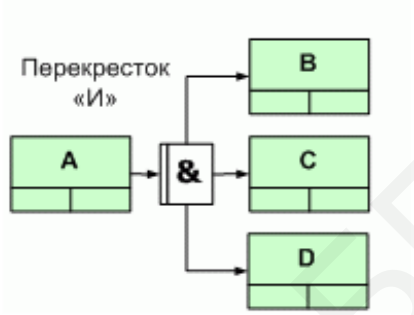



Рисунок 4.13 – Пример диаграммы IDEF3

Здесь работы B, C, D стартуют все вместе по завершении работы A. Временные отношения можно также передавать операторами ИЛИ, «не раньше, чем», «при условии, что» и др. Заметим, что диаграммы IDEF не определяют сам функционал каждого блока, а только его связи с другими блоками.

## 5 ЛАБОРАТОРНЫЕ РАБОТЫ

### 5.1 Изучение паттернов Factory и Abstract Factory

**Цель работы**: изучить назначение и принципы реализации паттернов Factory и Abstract Factory.

Паттерн Factory (Фабрика классов) предназначен для простого порождения объектов различных классов. Как правило, последние имеют общий прототип – абстрактный класс или интерфейс. Рассмотрим следующий код:

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

namespace JuLab1
{

   abstract class Product
{
      public abstract String getName();
}
// "ProductA"
class ProductA : Product
{
   String name = "AUDI";
   public override String getName()
   {
      return name;
   }
}
// "ProductB"
class ProductB : Product
{
   String name="TOYOTA";
  public override String getName(){
      return name;}
}
// "Manager"
abstract class Manager
{
public abstract Product Method();
}
// "ConcreteManagerA"
class ManagerA : Manager
{
```

```
public override Product Method()
{
return new ProductA();
}
}

// "ConcreteManagerB"
class ManagerB : Manager
{
public override Product Method()
{
return new ProductB();
}

   class Program
   {
      public static void Main()
      {

Manager[] managers = { new ManagerA(), new ManagerB() };

// iterate over creators and create products
foreach (Manager z in managers)
{
Product product = z.Method();
Console.WriteLine("Name {0}", product.getName());
}

Console.ReadLine();
}}}}
```

Здесь создаются два типа объектов – ProductA и ProductB. Их прототипом является общий класс Product. Для создания объектов используется менеджер конкретного класса. У всех менеджеров общий прототип (класс Manager). Выходное окно показано на рисунке 5.1.


Рисунок 5.1 – Выходное окно паттерна Factory

Паттерн Абстрактная фабрика классов использует несколько Фабрик классов. Они наследуются от одного абстрактного класса, который является Абстрактной фабрикой классов.

Предыдущий пример с некоторыми вариациями можно заменить следующим:

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

namespace JuLab1
{
   public abstract class AbstractProduct
   {
      public abstract String getName();

   }

   public class ProductA : AbstractProduct
   {
      public override String getName()
      {
        return "This is product A";
      }
   }

   public class ProductB : AbstractProduct
   {
      public override String getName()
      {
         return "This is product B";
      }
   }

   public interface AbstractFactory
   {
   }

   public class FactoryA : AbstractFactory
   {
      public static  AbstractProduct CreateProduct()
      {
         return new ProductA();
      }
   }

   public class FactoryB : AbstractFactory
   {
      public static AbstractProduct CreateProduct()
      {
         return new ProductB();
```

```
    }
  }

  class Program
  {
    public static void Main()
     {

     AbstractFactory factoryA =new FactoryA();
     AbstractFactory factoryB =new FactoryB();

     AbstractProduct _productA = FactoryA.CreateProduct();
     AbstractProduct _productB = FactoryB.CreateProduct();
       Console.WriteLine("Product says {0}", _productA.getName());
       Console.WriteLine("Product says {0}", _productB.getName());
       Console.ReadLine();

    }
  }
}
```

Используя данные примеры, выполните следующее задание.

**Задание**:

1. Измените примеры классов, представленных в описании лабораторной работы, на свои собственные.
2. Добавьте в классы методы, причем в различных классах использовать разные методы.
3. Выбор создаваемого объекта класса реализуйте передачей в Фабрику номера (например, 1 – это первый класс, 2 – второй класс).
4. Ограничьте число создаваемых объектов каждого класса (см. паттерн Одиночка).
5. Замените абстрактные классы интерфейсами.
6. Реализуйте паттерн Фабрика классов без использования родового абстрактного класса.
7. Защитите работу.

**Вопросы для контроля:**

1. Для чего используется паттерн Фабрика классов?
2. Для чего используется паттерн Абстрактная фабрика классов?
3. Приведите ваш собственный пример использования паттерна Фабрика классов.
4. В чем принципиальная разница между паттерном Фабрика классов и паттерном Абстрактная фабрика классов?

## 5.2 Изучение паттерна Builder

**Цель работы**: изучить и закрепить на практике основные положения паттерна Builder.

Данный паттерн создает сложный объект из составных частей. Составные части определяются условиями задачи. В подразделе 3.6 данного учебно-методического пособия составляли меню для взрослого и ребенка. Рассмотрим пример такого же рода.

Необходимо собрать мобильный телефон, включающий экран, батарею и операционную систему. Для этого задаем интерфейс с объявлением методов для выбора составных частей:

```
interface IPhoneBuilder
{
   void BuildScreen();
   void BuildBattery();
   void BuildOS();
    MobilePhone Phone { get;}
}
```

Пишем класс реализации интерфейса:

```
class PhoneBuilder : IPhoneBuilder
{
   MobilePhone phone;
   public PhoneBuilder()
   {
      phone = new MobilePhone("Android Phone");
   }

   public void BuildScreen()
   {
      phone.PhoneScreen = ScreenType.ScreenType_TOUCH_RESISTIVE;
   }

   public void BuildBattery()
   {
      phone.PhoneBattery = Battery.MAH_1500;
   }

   public void BuildOS()
   {
      phone.PhoneOS = OperatingSystem.iOS;
   }
```

Пишем класс, который возвращает собранный телефон:

```
    // GetResult Method which will return the actual phone
    public MobilePhone Phone
    {
        get { return phone; }
    }
}

class Manufacturer
{
    public void Construct(IPhoneBuilder phoneBuilder)
    {
        phoneBuilder.BuildBattery();
        phoneBuilder.BuildOS();
        phoneBuilder.BuildScreen();
    }
}

// various parts
public enum ScreenType
{
    ScreenType_TOUCH_CAPACITIVE,
    ScreenType_TOUCH_RESISTIVE,
    ScreenType_NON_TOUCH
};

public enum Battery
{
    MAH_1000,
    MAH_1500,
    MAH_2000
};

public enum OperatingSystem
{
    ANDROID,
    WINDOWS_MOBILE,
    WINDOWS_PHONE,
    iOS
};

class MobilePhone
{
    // fields to hold the part type
    string phoneName;
    ScreenType phoneScreen;
    Battery phoneBattery;
    OperatingSystem phoneOS;

    public MobilePhone(string name)
    {
```

```
        phoneName = name;
    }

    // Public properties to access phone parts

    public string PhoneName
    {
        get { return phoneName; }
    }

    public ScreenType PhoneScreen
    {
        get { return phoneScreen; }
        set { phoneScreen = value; }
    }

    public Battery PhoneBattery
    {
        get { return phoneBattery; }
        set { phoneBattery = value; }
    }

    public OperatingSystem PhoneOS
    {
        get { return phoneOS; }
        set { phoneOS = value; }
    }

    // Method to display phone details in our own representation
    public override string ToString()
    {
        return string.Format("Name: {0}\nScreen: {1}\nBattery {2}\nOS: {3}",
            PhoneName, PhoneScreen, PhoneBattery, PhoneOS);
    }
}

class Program
{
    static void Main(string[] args)
    {
        Manufacturer newManufacturer = new Manufacturer();
        IPhoneBuilder phoneBuilder = null;
        phoneBuilder = new PhoneBuilder();
        newManufacturer.Construct(phoneBuilder);
        Console.WriteLine("A new Phone built:\n\n{0}", phoneBuilder.Phone.ToString());
        Console.ReadLine();
    }
}
```

**Задание**:

1. Используя приведенные «заготовки», получите готовый код для сборки телефона. Модель должна собираться по номеру версии. В конструкторе new PhoneBuilder(Number) параметр Number должен указывать вариант сборки, например:

1– ScreenType_TOUCH_CAPACITIVE, MAH_1000, ANDROID;
2– ScreenType_TOUCH_CAPACITIVE, MAH_1500, iOS;
3– ScreenType_TOUCH_RESISTIVE, MAH_1500, WINDOWS_MOBILE;

2. Предусмотрите формирование цены на собираемую модель как сумму цен составных частей.

3. Предусмотрите программное добавление новой составной части (например, операционной системы).

4. Ограничьте число создаваемых объектов каждого класса (см. паттерн Одиночка).

5. Продемонстрируйте работающее приложение.

6. Защитите работу.

**Вопросы для контроля:**

1. Для чего используется паттерн Builder?

2. Приведите ваши примеры использования паттерна Builder.

3. Можно ли ввести ограничения на включение объектов в коллекцию, собираемую с помощью паттерна Builder (например, если выбрана Mac OS, то рабочим языком не может быть язык Java Android). Как это реализовать программно?

4. Можно ли совместно использовать паттерн Фабрика классов и паттерн Builder?

### 5.3 Изучение паттерна Decorator

**Цель работы**: изучить назначение и принципы реализации паттерна Decorator.

Паттерн Decorator предназначен для добавления или изменения (улучшения) функциональности базового (абстрактного) класса или интерфейса. В подразделе 3.7 был рассмотрен пример с выпечкой, где декорацией служило добавление новых ингредиентов и соответствующее изменение цены. Рассмотрим другой пример. Пусть имеется абстрактный класс Item и реализующий его конкретный класс ConcreteItem:

```
abstract class Item
   {
      public abstract String Operation();
   }

   class ConcreteItem : Item
   {
      public override String Operation()
      {
         return "Basic operation in action";
      }
   }
```

Объявлена одна абстрактная операция и ее конкретная реализация. Необходимо улучшить эту операцию – декорировать ее. Соответственно, объявим абстрактный декоратор и два конкретных декоратора на его основе. Абстрактный декоратор имеет следующий вид:

```
abstract class Decorator : Item
   {
      protected Item item;
      public void SetItem(Item item)
      {
         this.item = item;
      }
      public override String Operation()
      {
         if (item != null)
         {
            return "Hello, from basic decorator";
         }
         return "Something incorrect in code. Nethertheless, hello!";
      }
   }
```

Метод Operation переписан. В конкретных декораторах он реализуется по-разному. Далее приведены полная версия приложения и окно программы (рисунок 5.2).

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

namespace Decor
{
   class Program
   {
      static void Main()
```

```
    {
        // Create ConcreteComponent and two Decorators
        ConcreteItem c = new ConcreteItem();
        ConcreteDecoratorA d1 = new ConcreteDecoratorA();
        ConcreteDecoratorB d2 = new ConcreteDecoratorB();

        d1.SetItem(c);
        d2.SetItem(d1);
        Console.WriteLine(d1.Operation());
         Console.WriteLine(d2.Operation());
        Console.Read();
    }
}
/// <summary>

abstract class Item
{
    public abstract String Operation();
}

class ConcreteItem : Item
{
    public override String Operation()
    {
        return "Basic operation in action";
    }
}

abstract class Decorator : Item
{
    protected Item item;
    public void SetItem(Item item)
    {
        this.item = item;
    }
    public override String Operation()
    {
        if (item != null)
        {
            return "Hello, from basic decorator";
        }
        return "Something incorrect in code. Nethertheless, hello!";
    }
}

class ConcreteDecoratorA : Decorator
{
    private string addedState;
    public override String Operation()
    {
      // base.Operation();
```

```
            addedState = "New State";
            return "Hello from decorA";
        }
    }
    // "ConcreteDecoratorB"
    class ConcreteDecoratorB : Decorator
    {
        public override String Operation()
        {
          //  base.Operation();
            AddedBehavior();
            return "Hello from decor B. Now is :{0}"+ DateTime.Now;
        }
        void AddedBehavior()
        {
        }
    }
}
```

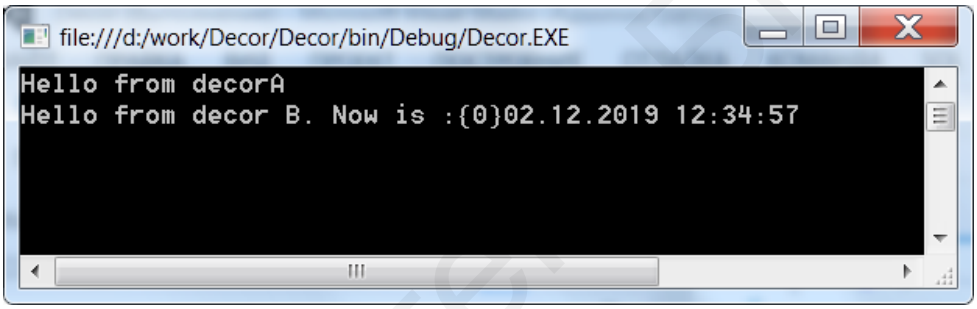



Рисунок 5.2 – Выходное окно программы Decorator

**Задание**:

1. Изучите программу и выполните ее.

2. Измените базовый класс Item, добавьте в него метод GetStr для ввода строки.

3. В классе ConcreteItem реализуйте просто вывод строки, введенной по GetStr.

4. В декораторе реализуйте иную обработку строки: рассмотрите ее как простое арифметическое выражение, например 2+4·(1–5). Задача вашего декоратора такова: если строка, введенная по GetStr, содержит арифметическое выражение, то вычислите результат и отобразите его, иначе просто верните введенную строку.

5. Пункт 4 выполните в окне формы, для чего вызовите из консоли оконное приложение и передайте ему для отображения результат.

**Вопросы для контроля:**

1. Почему паттерн Decorator раньше называли Оберткой?
2. Укажите, в чем приведенная в лабораторной работе программа является избыточной (какой класс можно было бы удалить с незначительным изменением кода)?
3. Можно ли реализовать декорацию по принципу «матрешки» (второй класс декорирует первый, третий класс декорирует второй и т. д.)?
4. Можно ли декорировать метод? Приведите пример.

## 5.4 Создание диаграммы классов

**Цель работы**: изучить основы построения диаграмм классов, создания пакетов и группировки классов в пакеты.

Диаграмма классов представляет множество классов и интерфейсов, связанных отношениями. Классы соответствуют реальным классам, т. е. моделируют предметную область. Создадим новую диаграмму в Logical View с именем *Add New Order* (добавить новый заказ). Эта диаграмма будет соответствовать некоторому варианту использования со следующими акторами:

- заказа;
- клиента;
- комплектующих изделий, входящих в заказ.

Создадим соответствующие классы-сущности *Client*, *Order*, *Component Part* и один дополнительный класс *OrderItem (Состав заказа)*. Опишем классы следующим образом.

***Client***

| **Комментарий** | **Класс, представляющий клиента** |
|---|---|
| Атрибуты | name : String – имя клиента<br>address : String – адрес клиента<br>phone : String – телефон клиента |
| Методы | AddClient() – добавить клиента<br>RemoveClient() – удалить клиента<br>GetInfo() – получить информацию |

***Order***

| **Комментарий** | **Класс заказа клиента** |
|---|---|
| Атрибуты | orderNumber : Integer – номер заказа<br>orderDate : Date – дата оформления<br>orderComplete : Date – дата выполнения |
| Методы | Create() – создание нового заказа<br>SetInfo() – ввести данные о заказе<br>GetInfo() – получить данные |

***OrderItem***

| **Параметр** | **Значение** |
|---|---|
| **Комментарий** | **Класс элемента заказа** |
| Атрибуты | itemNumber : Integer – номер элемента<br>quantity : Integer – количество комплектующих<br>price : Double – цена за единицу |
| Методы | Create() – создание нового элемента (строки) заказа<br>SetInfo() – занести информацию об элементе заказа<br>GetInfo() – получить информацию об элементе заказа |

***ComponentPart***

| **Параметр** | **Значение** |
|---|---|
| Атрибуты | name : String – наименование<br>manufacturer : String – производитель<br>price : Double – цена за единицу<br>description – описание |
| Операции | AddComponent() – добавление  комплектующего изделия<br>RemoveComponent() – удаление комплектующего изделия<br>GetInfo() – получить информацию о комплектующем изделии |

На диаграмме классов получаем следующие спецификации (рисунок 5.3).

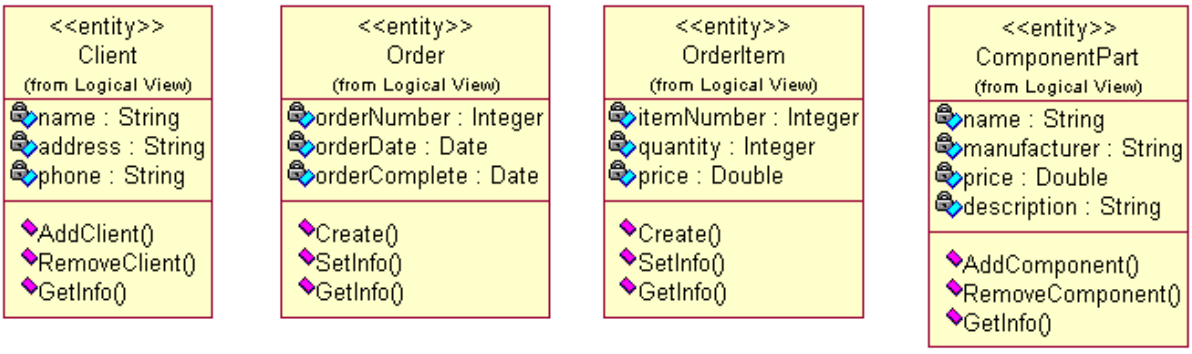



Рисунок 5.3 – Созданные классы

Теперь добавим отношения между классами (рисунок 5.4).

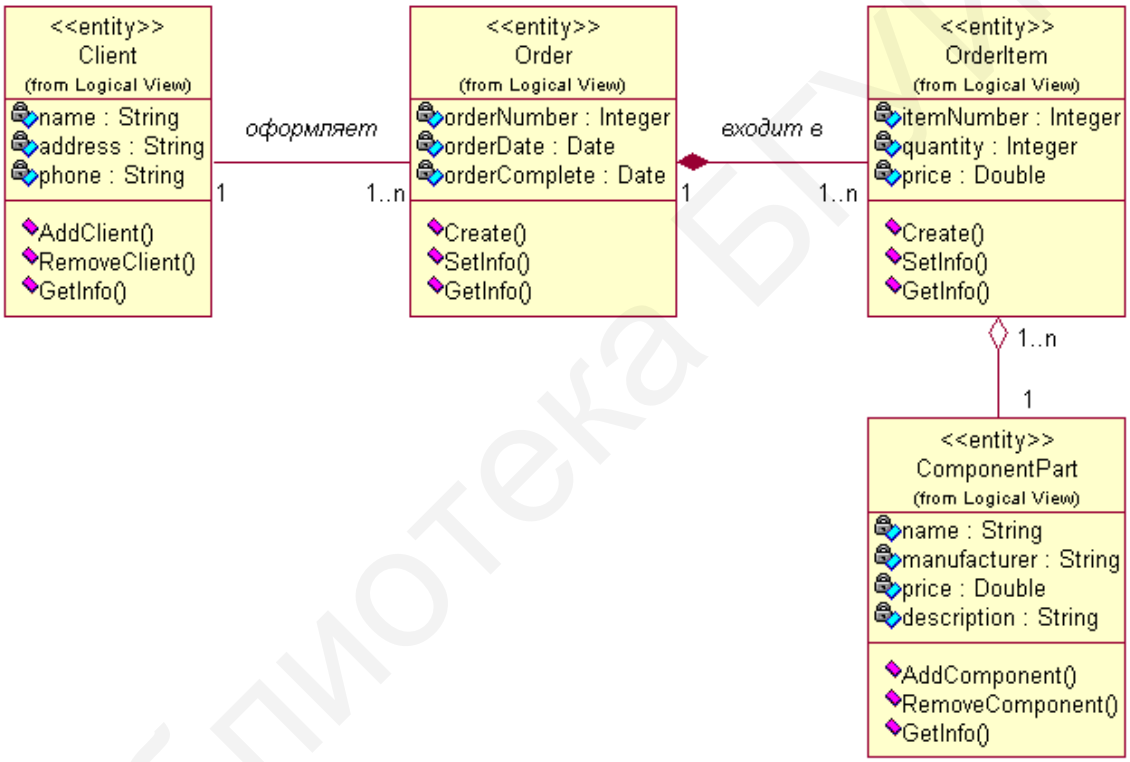



Рисунок 5.4 – Классы-сущности и отношения между ними

Классы-сущности связаны между собой следующими отношениями:

- классы *Client* и *Order* – отношением ассоциации: клиент может сделать несколько заказов, заказ поступает только от одного клиента;
- класс *Order* и *OrderItem* – отношением композиции, поскольку строка заказа является частью самого заказа;
- класс *OrderItem* и *ComponentPart* – отношением агрегации, т. к. комплектующие изделия являются частями элемента заказа.

В отличие от композиции комплектующее изделие может существовать без привязки к элементу заказа.

Добавим на диаграмму граничные и управляющие классы, как показано на рисунке 5.5. Управляющий класс OrderManager служит для управления заказами, пограничный класс OrderOptions – для формирования элементов заказа и его характеристик. Созданные классы следует разместить в одном пакете (package).

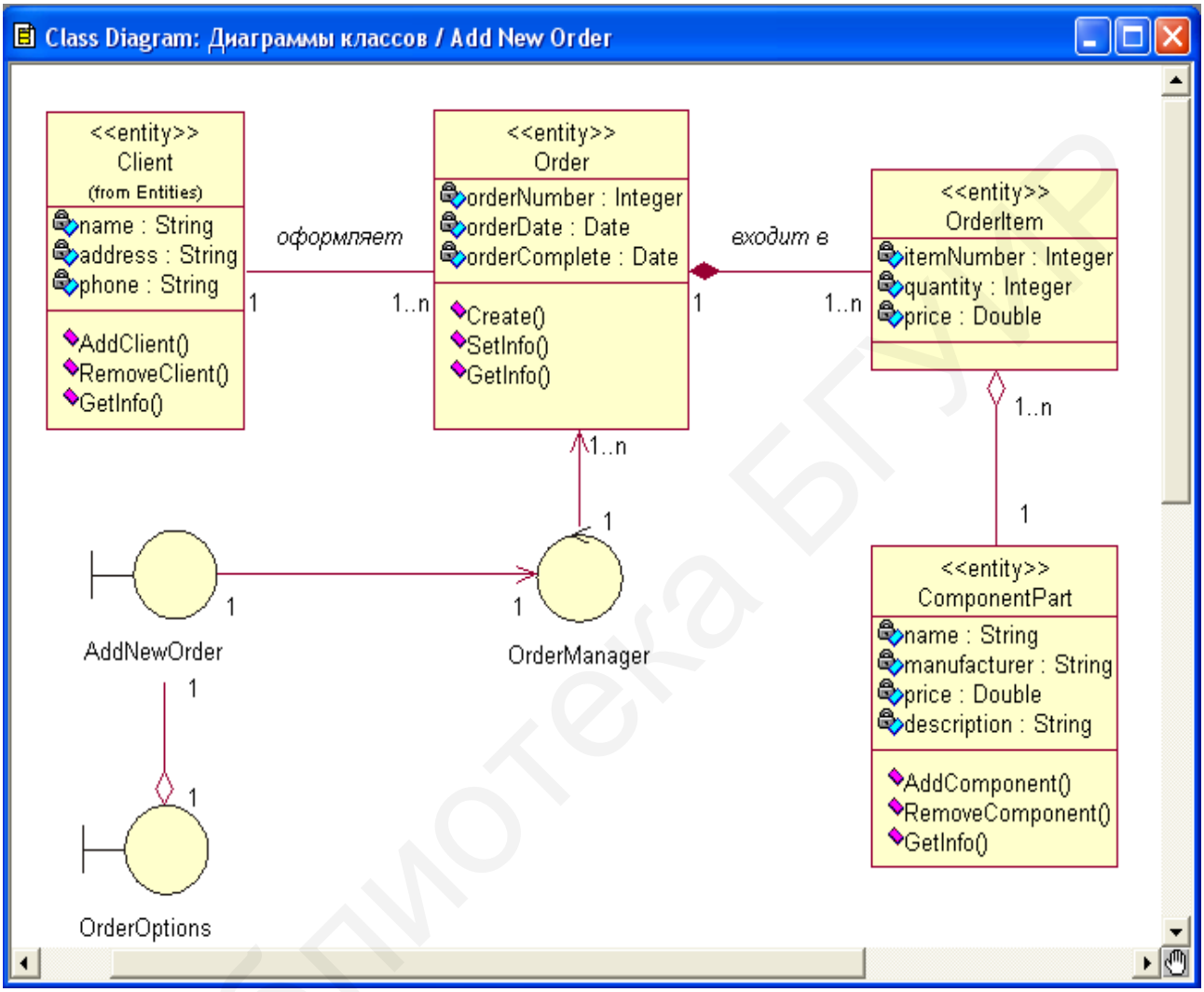



Рисунок 5.5 – Диаграмма классов

Создадим следующие пакеты:

- *Entities* (классы-сущности);
- *Boundaries* (граничные классы);
- *Control* (управляющие классы).

Щелкаем правой кнопкой мыши по узлу Logical View, в появившемся меню выбираем пункт New → Package, указываем имя пакета (рисунок 5.6).

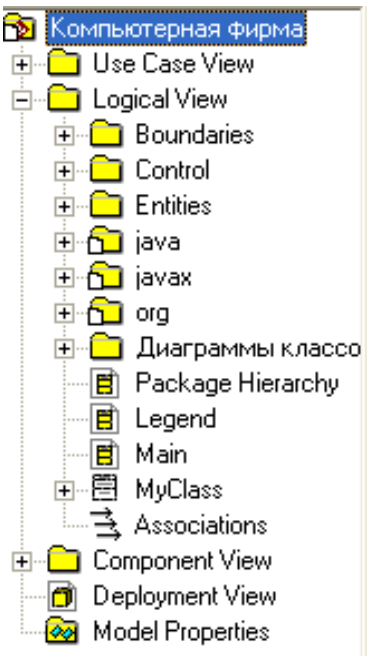



Рисунок 5.6 – Пакеты

Для каждого пакета зададим комментарий:

- для пакета Entities – *классы-сущности*;
- для пакета Boundaries – *граничные классы*;
- для пакета Control – *управляющие классы*.

Занесение классов в пакеты выполняется перетаскиванием каждого класса в соответствующий пакет с помощью кнопки мыши:

- классы *OrderOptions* и *AddNewOrder* – в пакет *Boundary*;
- классы *Client*, *Order*, *OrderItem* и *ComponentPart* – в пакет *Entities*;
- класс *OrderManager* – в пакет *Control* (рисунок 5.7).

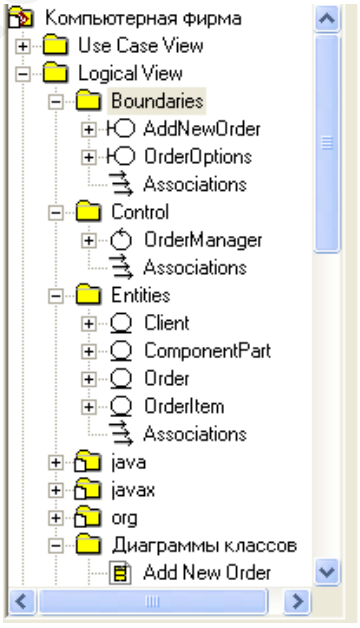



Рисунок 5.7 – Занесение классов в пакеты

Теперь в каждый пакет нужно добавить диаграмму классов и отобразить на ней соответствующие классы и отношения.

Активизируем контекстное меню щелчком правой кнопки мыши на пакете и выбираем в контекстном меню New → Class Diagram (Создать → Диаграмма классов), вводим имя класса *Main* (Главная). Затем открываем саму диаграмму двойным щелчком кнопки мыши и перетаскиваем на нее требуемые классы. При этом созданные ранее отношения между классами воспроизводятся на диаграммах (рисунки 5.8–5.10).

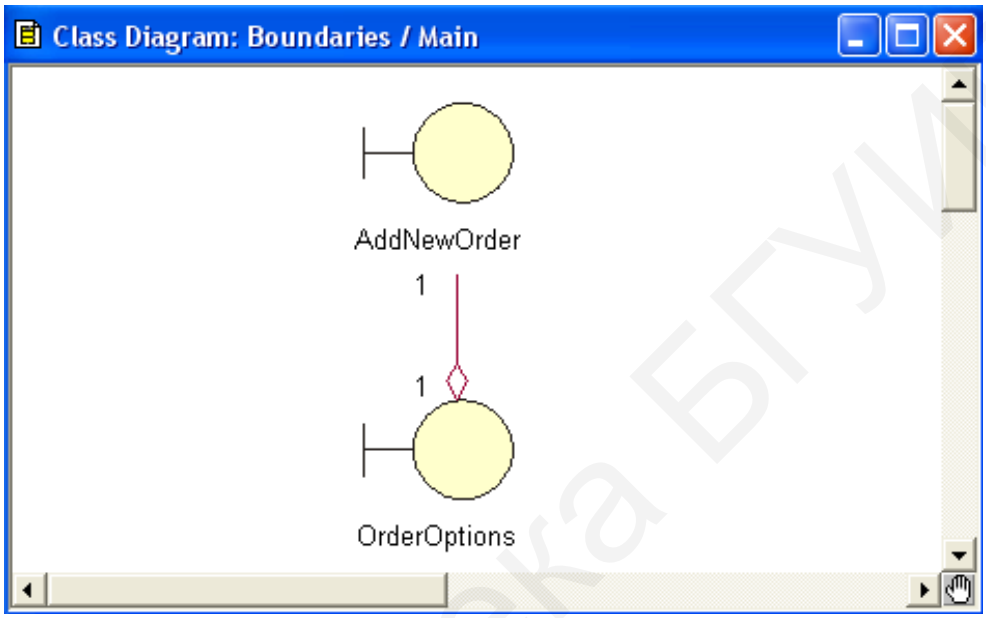



Рисунок 5.8 – Диаграмма классов пакета Boundaries

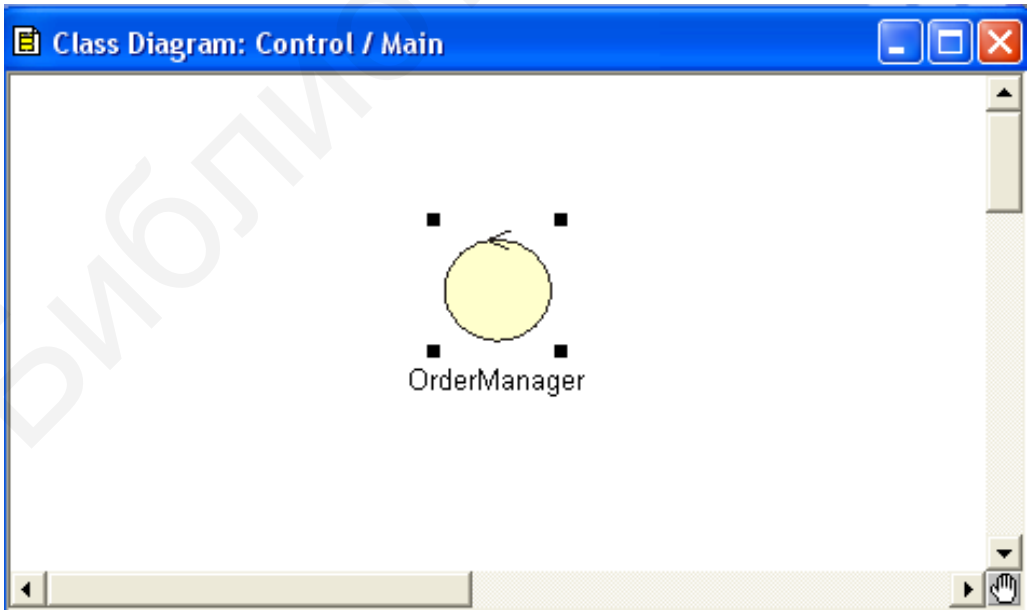



Рисунок 5.9 – Диаграмма классов пакета Control

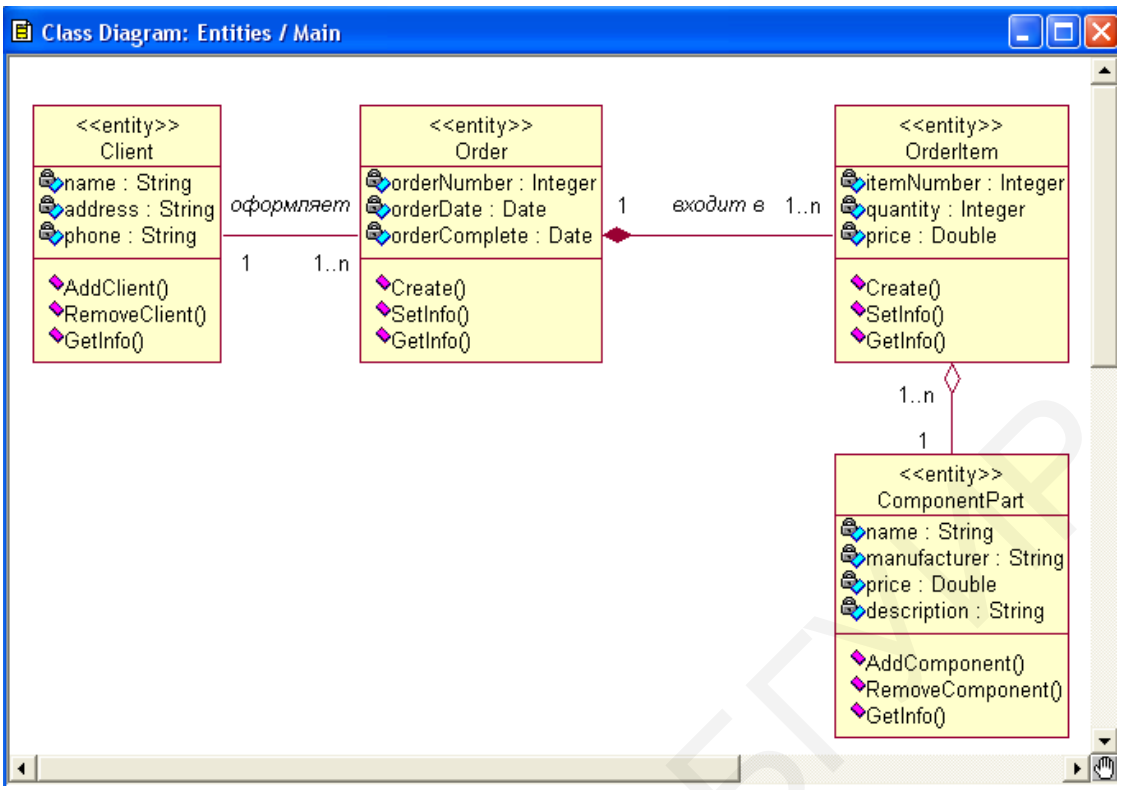



Рисунок 5.10 – Диаграмма классов пакета Entities

Заключительным шагом является построение главной диаграммы с размещенными на ней пакетами. По умолчанию в узле Logical View уже имеется главная диаграмма с именем Main. Просто перетаскиваем на нее пакеты Boundaries, Entities и Control (рисунок 5.11).

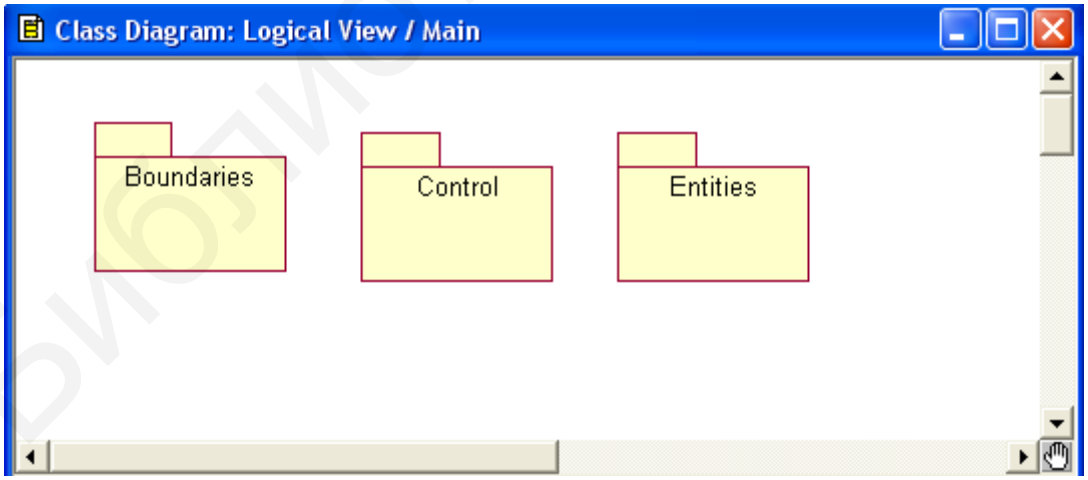



Рисунок 5.11 – Главная диаграмма классов

**Задание**:

1. Создайте диаграмму классов для сценария: студенты сдают экзамены по предметам (классы – Студент, Группа, Преподаватель, Предмет).

Продумайте и реализуйте атрибуты и методы классов, а также отношения между классами. Каждый класс должен быть подробно задокументирован.

2. Создайте пакеты для классов и разместите классы в пакетах.
3. Для каждого пакета создайте диаграмму классов.
4. Постройте главную диаграмму классов.

**Вопросы для контроля**:

1. Какие вы знаете диаграммы, кроме Диаграммы классов?
2. Что отображается на Диаграмме классов?
3. Для чего служат пакеты?
4. Использует ли пакет свою диаграмму?
5. Что такое агрегация (пример)?

## 5.5 Создание рекурсивной спецификации

**Цель работы**: освоить технику написания рекурсивных спецификаций.

В этой лабораторной работе необходимо создать рекурсивные спецификации для работы автосправки. Система будет обрабатывать запросы клиентов. Запросы представляют собой тексты. Пример запроса: *какой курс доллара*? Для ответа на вопросы имеется текстовая база ответов, хранящихся в текстовом файле. Ответы однострочные, например,

*курс доллара равен* 2 *руб*. 4 *коп*.

Проблема в том, что текстовые запросы пользователя не имеют строго заданной структуры и могут содержать ошибки, например,

*укожите курсс далара*.

Запросы вводятся строчными буквами. Необходимо найти ответ в текстовом файле. Для решения этой задачи нужно написать рекурсивные спецификации двух функций.

Первая функция – оценить сходство двух слов – similarity(w1, w2). Вторая функция – получить число слов, совпавших у запроса и предложения из текстового файла – countwords(s1,s2). Таким образом, ответ на запрос будет представлен тем предложением s2 из текстового файла, у которого наибольшее число совпадений со словами запроса. При этом запрос может иметь нарушенный синтаксис и орфографию.

Далее мы рассмотрим в общих чертах реализацию функции similarity(w1, w2). Нам понадобятся базовые простейшие функции для работы со строками: headC(String) – возвращает первую букву в слове String; tailC(String) –

возвращает остаток слова String без первой буквы или пустую строку, если возвращать нечего. Эти базовые функции реализуются следующим образом:

```
static String headC(String arr)
  {
     if (arr.Length > 0)
      {
         return arr.Substring(0, 1).Trim();
       }
         return "";
      }

static String tailC(String arr)
 {
     if (arr.Length > 0)
      {
         return arr.Substring(1, arr.Length - 1).Trim();
       }
         return "";
      }

//добавление символа к строке
static String addC(String arr, String z)
  {
      return arr + z;
   }
```

Для реализации функции similarity будем последовательно символ за символом сравнивать два слова (строки). Если суммарное число ошибок больше двух, то считаем слова различными. Рассмотрим следующие варианты:

- ☐ буква написана с ошибкой;
- ☐ буква пропущена;
- ☐ вставлена лишняя буква.

Для распознавания этих вариантов и принятия адекватных действий будем использовать следующие частные случаи (варианты):

```
similarity(z1,z)
{
if(z1.Length==0)
  if (z.Lenngth<=2)
    return 0;
. . . . . .
if(z.Length==0)
  if (z1.Lenngth<=2)
    return 0;
. . . . . .
```

```
if(headC(z1)==headC(z))
  return  similarity(tailC(z1),tailC(z))

. . . . . .


if(headC(z1)!=headC(z))
  if(headC(tailC(z1))==headC(tailC((z)))

  return 1+ similarity(tailC(z1),tailC(z))

. . . . . .

if(headC(z1)!=headC(z))
  if(headC(z1))==headC(tailC((z)))

  return 1+ similarity(z1,tailC(z))

. . . . . .

if(headC(z1)!=headC(z))
  if(headC(tail(z1))==headC(z))

  return 1+ similarity(tail(z1),z)

. . . . . .
```

Рассмотрим, например, последнее правило. Проверяем первые символы обеих строк z1 и z. Если они не равны, то проверяем на равенство второй символ первой строки и первый символ второй строки. Если они равны, то полагаем, что в первой строке пропущен символ и процесс рекурсивно повторяем. Самая первая проверка

```
if(z1.Length==0)
  if (z.Lenngth<=2)
    return 0;
```

считает, что строки совпадают, если при прочих равных условиях длина второй строки больше длины первой не более, чем на два символа. На основании этих сведений можно сформулировать задание на выполнение.

**Задание**:

1. Полностью реализуйте функцию similarity.
2. На основании функций headC, tailC, similarity реализуйте распознавание сходства/различия слов с ошибками.
3. Реализуйте выбор из файла той строки, где число совпадений со словами вопроса наибольшее.

**Вопросы для контроля:**

1. В чем недостаток рекурсивных спецификаций с точки зрения использования вычислительных ресурсов?
2. Является ли рекурсивная функция эквивалентной понятию алгоритма?
3. Почему рекурсивные спецификации не используют как универсальное средство для описания функций (алгоритмов)?

## 5.6 Использование Linq-выражений

**Цель работы**: освоить технику реализации Linq-выражений спецификаций.

В этой работе мы ограничимся функциями агрегирования в Linq. Начнем с простых примеров:

1. Пример суммы элементов списка:

```
var numbers = new List<int> { 8, 2, 6, 3 };
int sum = numbers.Sum();  // sum: 19
```

2. Пример получения максимального значения:

```
var numbers = new List<int> { 1, 8, 3, 2 };
int maxNumber = numbers.Max();  // maxNumber: 8
```

3. Число элементов в коллекции:

```
IEnumerable<string> items = new List<string> { "A", "B", "C" };
int count = items.Count();  // count: 3
```

4. Среднее значение элемента в коллекции:

```
var list = new List<int> { 1, 8, 3, 2 };
double result = list.Average();  // result: 3.5
```

Далее приведем более сложный пример:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Xml.Linq;
using System.ComponentModel;

namespace Linq1
```

```
{

   namespace AggregateOperators
   {
      public class Product
      {
         public int ProductID { get; set; }
         public string ProductName { get; set; }
         public string Category { get; set; }
         public decimal UnitPrice { get; set; }
         public int UnitsInStock { get; set; }
      }

      class Program
      {
         static List<Product> productList;

         public static List<Product> GetProductList()
         {
            if (productList == null)
               createLists();

            return productList;
         }

         public static void createLists()
         {
            productList =
               new List<Product> {
               new Product { ProductID = 1, ProductName = "Chai", Category = "Beverages",
UnitPrice = 18.0000M, UnitsInStock = 39 },
               new Product { ProductID = 2, ProductName = "Chang", Category = "Beverages",
UnitPrice = 19.0000M, UnitsInStock = 17 },
               new Product { ProductID = 3, ProductName = "Aniseed Syrup", Category =
"Condiments", UnitPrice = 10.0000M, UnitsInStock = 13 },
               new Product { ProductID = 4, ProductName = "Chef Anton's Cajun Seasoning",
Category = "Condiments", UnitPrice = 22.0000M, UnitsInStock = 53 },
               new Product { ProductID = 5, ProductName = "Chef Anton's Gumbo Mix", Category =
"Condiments", UnitPrice = 21.3500M, UnitsInStock = 0 },
               new Product { ProductID = 6, ProductName = "Grandma's Boysenberry Spread",
Category = "Condiments", UnitPrice = 25.0000M, UnitsInStock = 120 },
               new Product { ProductID = 7, ProductName = "Uncle Bob's Organic Dried Pears",
Category = "Produce", UnitPrice = 30.0000M, UnitsInStock = 15 },
               new Product { ProductID = 8, ProductName = "Northwoods Cranberry Sauce",
Category = "Condiments", UnitPrice = 40.0000M, UnitsInStock = 6 },
               new Product { ProductID = 9, ProductName = "Mishi Kobe Niku", Category =
"Meat/Poultry", UnitPrice = 97.0000M, UnitsInStock = 29 },
               new Product { ProductID = 10, ProductName = "Ikura", Category = "Seafood",
UnitPrice = 31.0000M, UnitsInStock = 31 },
               new Product { ProductID = 11, ProductName = "Queso Cabrales", Category = "Dairy
Products", UnitPrice = 21.0000M, UnitsInStock = 22 },
```

```
            new Product { ProductID = 12, ProductName = "Queso Manchego La Pastora",
Category = "Dairy Products", UnitPrice = 38.0000M, UnitsInStock = 86 },
            new Product { ProductID = 13, ProductName = "Konbu", Category = "Seafood",
UnitPrice = 6.0000M, UnitsInStock = 24 },
            new Product { ProductID = 14, ProductName = "Tofu", Category = "Produce",
UnitPrice = 23.2500M, UnitsInStock = 35 },
            new Product { ProductID = 15, ProductName = "Genen Shouyu", Category =
"Condiments", UnitPrice = 15.5000M, UnitsInStock = 39 }
        };
    }
    static void Main(string[] args)
    {
        List<Product> products = GetProductList();

        var categoryCounts =
            from prod in products
            group prod by prod.Category into prodGroup
            select new { Category = prodGroup.Key, ProductCount = prodGroup.Count() };

        foreach (var z in categoryCounts)
        {
            Console.WriteLine(z.Category+" total:"+z.ProductCount);
        }
        Console.ReadLine();
    }
  }
}
```

Данный код выдает следующее окно (рисунок 5.12).

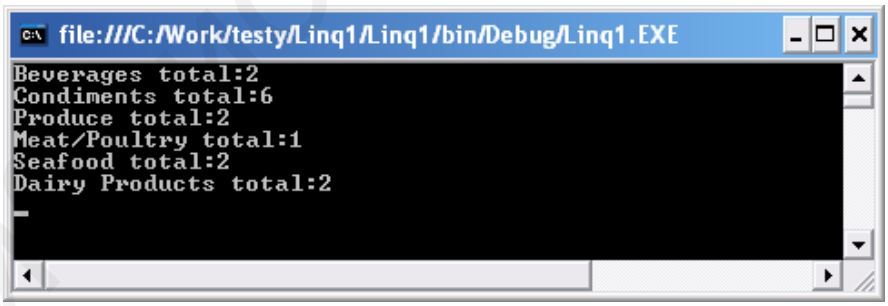



Рисунок 5.12 – Linq-агрегирование

Основной конструкцией здесь является

```
var categoryCounts =
from prod in products
group prod by prod.Category into prodGroup
select new { Category = prodGroup.Key, ProductCount = prodGroup.Count() };
```

Продукты группируются по полям prod.Category. Новая запись содержит всего два поля: Category и ProductCount. Остается выполнить вывод новой коллекции categoryCounts:

```
foreach (var z in categoryCounts)
    {
       Console.WriteLine(z.Category+" total:"+z.ProductCount);
     }
Console.ReadLine();
```

Второй пример того же рода выдает среднюю цену в каждой категории продукта:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace Linq1
{
   using System;
   using System.Collections.Generic;
   using System.Linq;
   using System.Text;
   using System.Xml.Linq;
   using System.ComponentModel;

   namespace AggregateOperators
   {
      public class Product
      {
         public int ProductID { get; set; }
         public string ProductName { get; set; }
         public string Category { get; set; }
         public decimal UnitPrice { get; set; }
         public int UnitsInStock { get; set; }
      }

      class Program
      {
         static List<Product> productList;
         public static List<Product> GetProductList()
         {
            if (productList == null)
               createLists();
            return productList;
         }
         public static void createLists()
         {
            productList =
```

```
            new List<Product> {
            new Product { ProductID = 1, ProductName = "Chai", Category = "Beverages",
UnitPrice = 18.0000M, UnitsInStock = 39 },
            new Product { ProductID = 2, ProductName = "Chang", Category = "Beverages",
UnitPrice = 19.0000M, UnitsInStock = 17 },
            new Product { ProductID = 3, ProductName = "Aniseed Syrup", Category =
"Condiments", UnitPrice = 10.0000M, UnitsInStock = 13 },
            new Product { ProductID = 4, ProductName = "Chef Anton's Cajun Seasoning",
Category = "Condiments", UnitPrice = 22.0000M, UnitsInStock = 53 },
            new Product { ProductID = 5, ProductName = "Chef Anton's Gumbo Mix", Category =
"Condiments", UnitPrice = 21.3500M, UnitsInStock = 0 },
            new Product { ProductID = 6, ProductName = "Grandma's Boysenberry Spread",
Category = "Condiments", UnitPrice = 25.0000M, UnitsInStock = 120 },
            new Product { ProductID = 7, ProductName = "Uncle Bob's Organic Dried Pears",
Category = "Produce", UnitPrice = 30.0000M, UnitsInStock = 15 },
            new Product { ProductID = 8, ProductName = "Northwoods Cranberry Sauce",
Category = "Condiments", UnitPrice = 40.0000M, UnitsInStock = 6 },
            new Product { ProductID = 9, ProductName = "Mishi Kobe Niku", Category =
"Meat/Poultry", UnitPrice = 97.0000M, UnitsInStock = 29 },
            new Product { ProductID = 10, ProductName = "Ikura", Category = "Seafood",
UnitPrice = 31.0000M, UnitsInStock = 31 },
            new Product { ProductID = 11, ProductName = "Queso Cabrales", Category = "Dairy
Products", UnitPrice = 21.0000M, UnitsInStock = 22 },
            new Product { ProductID = 12, ProductName = "Queso Manchego La Pastora",
Category = "Dairy Products", UnitPrice = 38.0000M, UnitsInStock = 86 },
            new Product { ProductID = 13, ProductName = "Konbu", Category = "Seafood",
UnitPrice = 6.0000M, UnitsInStock = 24 },
            new Product { ProductID = 14, ProductName = "Tofu", Category = "Produce",
UnitPrice = 23.2500M, UnitsInStock = 35 },
            new Product { ProductID = 15, ProductName = "Genen Shouyu", Category =
"Condiments", UnitPrice = 15.5000M, UnitsInStock = 39 }
        };
    }

    static void Main(string[] args)
    {
        List<Product> products = GetProductList();

        //var categoryCounts =
        //    from prod in products
        //    group prod by prod.Category into prodGroup
        //    select new { Category = prodGroup.Key, ProductCount = prodGroup.Count() };
        var categories =
         from prod in products
         group prod by prod.Category into prodGroup
         select new { Category = prodGroup.Key, AveragePrice = prodGroup.Average(p =>
p.UnitPrice) };

        foreach (var z in categories)
        {
            Console.WriteLine(z.Category+"  AveragePrice="+z.AveragePrice);
```

```
            }
            Console.ReadLine();
        }
    }
  }
}
```

Результат работы программы представлен на рисунке 5.13.

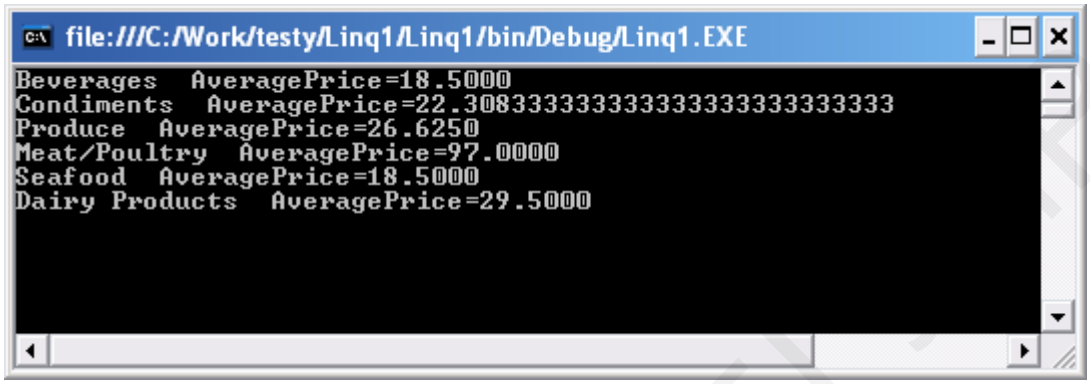



Рисунок 5.13 – Средняя цена в каждой категории продукта

**Задание**:

1. Руководствуясь представленными в данной лабораторной работе программами, создайте список товаров, число которых на складе (UnitsInStock) меньше 20.
2. Определите суммарное число товаров на складе.
3. Определите товар (категорию и название), число которого на складе наибольшее.
4. Определите фирму с самым дешевым товаром.
5. Определите список товаров, цена на которые ниже средней цены на все товары.
6. Выведите среднюю цену товара на складе.

**Вопросы для контроля**:

1. Какие агрегатные функции используются в Linq?
2. Запишите пример запроса на определение максимального элемента списка.
3. Запишите пример запроса на определение числа элементов списка, не превосходящих его среднее значение.
4. Объясните назначение кода

```
var categoryCounts =
from prod in products
group prod by prod.Category into prodGroup
```

```
select new { Category = prodGroup.Key, ProductCount =
prodGroup.Count() };
```

## 5.7 Спецификации на основе правил

**Цель работы**: освоить технику создания кода на основе правил.

В этой работе мы рассматриваем реализацию кода на основе правил (*if … then … else*). Такой подход обеспечивает лучшую структуру программы для последующего изменения. Систематизация данного подхода заключается в следующем. Прежде всего необходимо определить переменные состояния системы. Например, будем рассматривать задачу: автомобиль стоит перед закрытым шлагбаумом в пункте A, его нужно перевезти за шлагбаум в пункт B (рисунок 5.14).

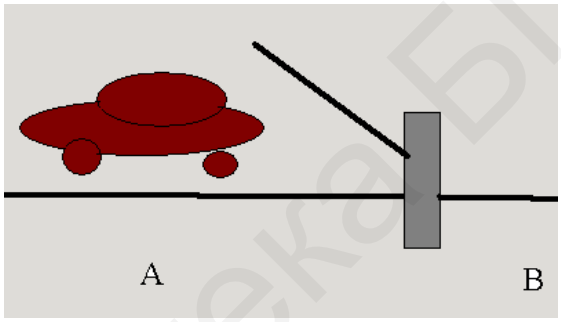



Рисунок 5.14 – Пояснение к задаче

Введем переменные состояния:
x1 – автомобиль в точке A;
x2 – автомобиль в точке B;
x3 – шлагбаум закрыт.
Далее следует определить управления:
move() – перевести автомобиль за шлагбаум;
up() – поднять шлагбаум;
down() – опустить шлагбаум.
Логику управления этой системой нетрудно передать следующими правилами для управлений, представленными в программе:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
```

```
/*
x1 – автомобиль в точке A
x2 – автомобиль в точке B
x3 – шлагбаум закрыт
```

Далее следует определить управления:

```
move() – перевести автомобиль за шлагбаум;
up() – поднять шлагбаум;
down() – опустить шлагбаум.
*/

namespace Auto
{
   class Program
   {
      static int x1 = 1;
      static int x2 = 0;
      static int x3 = 1;

      static bool move()
      {
         if ((x1 == 1) &&(x3==0))
         {
            x1 = 0;
            x2 = 1;
            return true;
         }

         return false;
      }

      static bool up()
      {
         if ((x1 == 1) && (x3 == 1))
         {
            x3 = 0;

            return true;
         }
         return false;
      }
      static bool down()
      {
         if ((x1 == 0) && (x3 == 0))
         {
            x3 = 1;
            return true;
         }
         return false;
      }
```

```
        static void Main(string[] args)
        {
            while (true)
            {
                if (move())
                { Console.WriteLine("Переехал в B");
                continue;
                }
                if (up())
                {
                    Console.WriteLine("Шлагбаум поднят");
                    continue;
                }
                if (down())
                {
                    Console.WriteLine("Шлагбаум закрыт");
                    continue;
                }
                Console.WriteLine("Движение завершено");
                break;
            }

            Console.ReadLine();
        }
    }
}
```

Каждое управление выполняет проверку необходимых условий. Так, для перемещения автомобиля производим следующую проверку:

```
static bool move()
  {
     if ((x1 == 1) &&(x3==0))
       {……………….}
```

В фигурных скобках помещаем результат применения управления:

```
x1 = 0;
x2 = 1;
return true;
```

Заметим, что код для управлений возвращает true, если управление реализовано, в противном случае возвращается false. Окно программы приведено на рисунке 5.15.

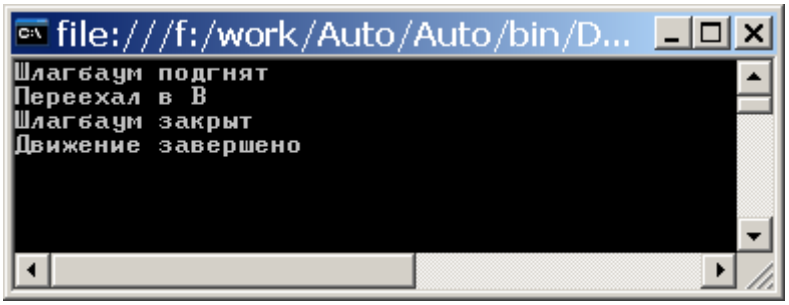



Рисунок 5.15 – Выполнение задачи об автомобиле

**Задание**:

1. Руководствуясь представленными в данной лабораторной работе программами, создайте программный код для управления турникетом в метро (введите переменные состояний, напишите правила).

2. На основе правил запрограммируйте задачу об обезьяне и банане: в системе в точке A находится обезьяна, в точке B (справа от A) – подвешен банан, в точке C (справа от B) находится ящик. Чтобы завладеть бананом обезьяна должна перетащить ящик из точки C в точку B и забраться на него. Введите системные переменные, запишите правила и разработайте код.

3. Напишите правила для упрощенной задачи о ханойской башне: имеется набор из трех разных по диаметру колец. Необходимо собрать из них пирамиду, причем на любом кольце можно располагать только кольцо меньшего диаметра. Введите систему переменных и правил. Разработайте код.

**Вопросы для контроля**:

1. В чем преимущество спецификаций на основе правил в сравнении, например, с рекурсивными спецификациями?

2. В каких языках программирования код пишется на основе правил?

3. Как соединить спецификации на основе правил с рекурсивными спецификациями? Приведите пример.

4. Напишите правила для отыскания наибольшего общего делителя для трех целых положительных чисел.

## ЛИТЕРАТУРА

1. Бабанов, А. М. Технология разработки программного обеспечения. Структурный подход / А. М. Бабанов. – Томск : Томс. гос. ун-т, 2006. – 157 с.

2. Калайда, В. Т. Технология разработки программного обеспечения / В. Т. Калайда, В. В. Романенко.– Томск : Томс. межвуз. центр дистанц. образ., 2007. – 257 с.

3. Дейкстра, Э. Дисциплина программирования / Э. Дейкстра. – М. : Мир, 1978. – 277 с.

4. Spivey, J. The Z notation: a reference manual / J. M. Spivey. – England : Prentice Hall Ltd., 1992. – 156 p.

5. Лисков, Б. Абстракции и спецификации в разработке программ / Б. Лисков, Дж. Гатэг. – М. : Мир, 1989. – 424 с.

6. Кларк, Э. М. Верификация моделей программ. Model Checking / Э. М. Кларк, О. Грамберг, Д. Пелед. – М. : МЦНМО, 2002. – 416 с.

7. Приемы объектно-ориентированного программирования. Паттерны проектирования / Э. Гамма [и др.]. – СПб. : Питер, 2001. – 368 с.

8. Трофимов, С. А. CASE-технологии: практическая работа в Rational Rose / С. А. Трофимов. – М. : Бином-Пресс, 2002. – 288 с.

Св. план 2020, поз. 31

*Учебное издание*

**Герман** Юлия Олеговна
**Герман** Олег Витольдович

# *ПРОЕКТИРОВАНИЕ И РАЗРАБОТКА ИНФОРМАЦИОННЫХ СИСТЕМ*

УЧЕБНО-МЕТОДИЧЕСКОЕ ПОСОБИЕ

Редактор *М. А. Зайцева*
Корректор *Е. Н. Батурчик*
Компьютерная правка, оригинал-макет *В. М. Задоля*

Подписано в печать 26.05.2020. Формат 60×84 1/16. Бумага офсетная. Гарнитура «Таймс».
Отпечатано на ризографе. Усл. печ. л. 7,56. Уч.-изд. л. 8,0. Тираж 50 экз. Заказ 47.

Издатель и полиграфическое исполнение: учреждение образования
«Белорусский государственный университет информатики и радиоэлектроники».
Свидетельство о государственной регистрации издателя, изготовителя,
распространителя печатных изданий №1/238 от 24.03.2014,
№2/113 от 07.04.2014, №3/615 от 07.04.2014.
Ул. П. Бровки, 6, 220013, г. Минск