МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное автономное образовательное учреждение
высшего образования
«СЕВЕРО-КАВКАЗСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ

УЧЕБНОЕ ПОСОБИЕ

Направление подготовки:

09.03.02 «Информационные системы и технологии»

Квалификация выпускника: бакалавр

Ставрополь, 2018

УДК 004.41 (075.8)
ББК 22.18я73
Н 63

Печатается по решению
учебно-методического совета
Северо-Кавказского федерального
университета

Н 63 **Объектно-ориентированное программирование:** учебное пособие для студентов направления 09.03.02 «Информационные системы и технологии» / Николаев Е.И. – Ставрополь: Изд-во СКФУ, 2018. – 250 с.

Учебное пособие по дисциплине «Объектно-ориентированное программирование» для студентов направления 09.03.02 «Информационные системы и технологии». Пособие охватывает теоретические аспекты проектирования и разработки приложений с использованием объектного подхода. Изложение материала ведется с использованием примеров на языках Java и C#. Предполагается, что студенты, изучающие дисциплину «Объектно-ориентированное программирование» обладают базовыми знаниями в области программирования, владеют хотя бы одним языком программирования высокого уровня. Цель учебного пособия: сформировать у студентов целостный взгляд на современные тенденции в области объектно-ориентированного программирования; обеспечить студентов обширным теоретическим материалом, достаточным для освоения методик разработки приложений; сформировать систему профессиональных компетенций.

УДК 004.41 (075.8)
ББК 22.18я73

Автор:

канд. техн. наук, доцент **Е.И. Николаев**

Рецензенты

канд. техн. наук **Ю. В, Рокотов,**

доктор техн. наук, профессор **А. В. Маликов**

© Издательство Северо-Кавказского
федерального университета, 2018

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ..... 5
РАЗДЕЛ 1. ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ МОДЕЛИРОВАНИЕ 7
1.1 Объектная модель проектирования 8
1.2 Объектно-ориентированное программирование 9
1.3 Проектирование и анализ в рамках объектно-ориентированной модели 11
1.4 Основные концепции объектного подхода 12
1.5 Преимущества объектной модели 33
Контрольные вопросы 34
РАЗДЕЛ 2. КЛАССЫ И ОБЪЕКТЫ 36
2.1 Объекты 36
2.2 Отношения между объектами 46
2.3 Классы 49
2.4 Отношения между классами 52
2.5 Взаимосвязь классов и объектов 58
Контрольные вопросы 61
РАЗДЕЛ 3. ОБЪЕКТНОЕ МОДЕЛИРОВАНИЕ 62
3.1 Инструментальные средства моделирования 63
3.2 Типология диаграмм 64
3.3 Диаграмма вариантов использования 68
3.4 Диаграмма классов 70
3.5 Диаграмма деятельности 76
3.6 Диаграммы последовательностей 80
3.7 Проектирование программных систем 82
Контрольные вопросы 96
РАЗДЕЛ 4. ПАТТЕРНЫ ПРОЕКТИРОВАНИЯ 98

4.1 Основы терминологии. Классификация паттернов 99
4.2 Основы использования паттернов 103
4.3 Паттерн «Одиночка» (Singleton) 110
4.4 Порождающие паттерны 114
4.5 Структурные паттерны 133
4.6 Поведенческие паттерны 156
Контрольные вопросы 169
РАЗДЕЛ 5. ПРАКТИЧЕСКИЕ ПРИЕМЫ РАЗРАБОТКИ ПРИЛОЖЕНИЙ 171
5.1 Построение графического интерфейса средствами Java 172
5.2 Построение графического интерфейса средствами .NET Framework 191
5.3 Обработка событий 204
5.4 Шаблон Model-View-Controller (MVC) 228
5.5 Многопоточность 240
Контрольные вопросы 246
ЗАКЛЮЧЕНИЕ 248
СПИСОК ЛИТЕРАТУРЫ 250

ВВЕДЕНИЕ

Популярность объектного подхода обусловлена объективными факторами усложнения программных систем и неуклонным повышением требований к интеллектуальности, производительности, эргономичности, доступности и адаптивности программного обеспечения и средств разработки.

Объектно-ориентированное программирование, как самостоятельное направление, на текущем этапе развития информационных технологий занимает существенную долю всех доступных средств разработки, анализа, проектирования и моделирования.

Особая роль объектно-ориентированных технологий приводит к необходимости детального изучения принципов построения программных компонент информационных систем на базе объектных технологий. При этом объектно-ориентированное программирование (ООП) – это только одно из нескольких самостоятельных направлений изучения и использования теории, в основе которой лежат термины «объект» и «класс». При употреблении термина «объектно-ориентированное программирование» подразумевается вся совокупность всех языков программирования, которые по совокупным признакам можно отнести к данной группе. Такие языки относят к объектно-ориентированным языкам программирования (ООЯП). При этом ООП включает также различные технологии программирования, которые используются на практике при разработке приложений на ООЯП.

Современные технологии объектно-ориентированного программирования интенсивно развиваются – на данный момент программисту недостаточно понимать простейшие принципы ООП (инкапсуляция, полиморфизм, наследование). ООП, как технология, должна реагировать на появление новых требований современного высокотехнологичного мира: параллельный характер процессов в

информационных системах; распределенный характер информационных систем; повышение требований к защищенности программного обеспечения: слияние различных технологий разработки приложений и востребованность унифицированного подхода к проектированию и разработке веб-приложений, сервисов, интерфейсов.

Высококвалифицированный программист должен понимать основные ошибки, которые приводят к созданию неэффективных приложений. Программирование приложений должно быть критическим: необходимо использовать различные системы принципов (например, S.O.L.I.D.) и активно применять проверенные паттерны проектирования.

Целью изучения дисциплины «Объектно-ориентированное программирование» является освоение студентами теоретических знаний о современных принципах

Данное пособие ориентировано на студентов, освоивших синтаксические правила и базовые технологии языка программирования высоко уровня (Java или C#). Изложение материала ведется с использованием примеров на этих языках. Пособие ориентировано на формирование следующих компетенций:

ПК-11 – способность к проектированию базовых и прикладных информационных технологий;

ПК-12 – способность разрабатывать средства реализации информационных технологий (методические, информационные, математические, алгоритмические, технические и программные);

ПК-30 – способность проводить сборку информационной системы из готовых компонентов.

РАЗДЕЛ 1. ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ МОДЕЛИРОВАНИЕ

Развитие информационных технологий привело к созданию методов объектно-ориентированного анализа, проектирования и программирования, позволяющих создавать крупномасштабное программное обеспечение. Существует несколько парадигм программирования: процедурно-ориентированная, объектно-ориентированная, логико-ориентированная, ориентированная на правила и ориентированная на ограничения. Абстракция выделяет существенные характеристики объекта, отличающие его от всех других видов объектов и, таким образом, проводит четкие концептуальные границы объекта с точки зрения наблюдателя.

Инкапсуляция – это процесс выделения элементов абстракции, образующих ее структуру и поведение; инкапсуляция позволяет отделить контрактный интерфейс абстракции от ее реализации.

Модульность – это свойство системы, разложенной на целостные и слабо связанные между собой модули.

Иерархия – это ранжирование или упорядочение абстракций.

Контроль типов – это система правил, предотвращающих или сильно ограничивающих взаимную замену объектов разных классов.

Параллелизм – это свойство, отличающее активные объекты от пассивных.

Персистентность – это способность объекта преодолевать временные или пространственные пределы.

1.1 Объектная модель проектирования

Объектно-ориентированная технология основана на строгой теоретической базе. Элементы теории объектно-ориентированной технологии образуют объектную модель проектирования (object model of development) или объектную модель. Основные принципы объектной модели:

– иерархия;

– контроль типов;

– инкапсуляция;

– параллелизм;

– абстракция;

– модульность;

– персистентность.

Данные принципы существуют независимо от объектной содели и не являются ее внутренними свойствами. В рамках объектно-ориентированного подхода указанные принципы образуют синергетическое единство: они дополняют друг друга.

Объектно-ориентированный анализ и проектирование существенно превосходят структурное проектирование по своим потенциальным возможностям и предоставляют разработчику возможности для построения программных систем сложной архитектуры. Подобная сложность недостижима при использовании структурных подходов [1].

Методы структурного проектирования разрабатывались для оптимизации процессов разработки сложных систем на основе алгоритмов. Аналогично, методы объектно-ориентированного проектирования создавались, для оптимизации на основе внедрения выразительных средств языков объектного и объектно-ориентированного программирования. В ООП основными компонентами программ являются классы и объекты.

Объектная модель проектирования не ограничена рамками объектно-ориентированных языков программирования. Данная модель может применяться для проектирования, моделирования и анализа в таких сферах как пользовательские интерфейсы, базы данных, компьютерная архитектура реинжиниринг информационных систем.

Очевидно, что центральным понятием объектно-ориентированной технологии является объект.

Объект – относительно самостоятельная сущность, обладающая свойствами процедур и данных, что отражает способность сущности производить вычисления и обладать собственным состоянием.

1.2 Объектно-ориентированное программирование

В основе любого языка программирования лежит определенная модель (виртуальная вычислительная машина), позволяющая максимально эффективно использовать возможности вычислительных средств. Программист пишет программу не для ЭВМ, а для определенной модели. Таким образом, сложность задач, которые можно решать с использованием языка программирования, определяется уровнем абстракции при постановке задачи и возможностями модели, на которой основан язык программирования.

Развитие языков программирования неразрывно связано с развитием абстрактных моделей, облегчающих и систематизирующих проектирование.

Объектно-ориентированное программирование – это программирование, основанное на представлении программы в виде совокупности объектов, которые взаимодействуют между собой; каждый из объектов является экземпляром определенного класса, а классы являются членами иерархии наследования.

Данное определение показывает важные аспекты:

1) в структурном программировании основными элементами выступают алгоритмы; в объектно-ориентированном программировании основным элементом структурирования программ является объект;

2) объект представляет собой конкретное воплощение определенного класса – его экземпляр (instance of class);

3) классы не существуют в программе обособленно – они объединены в иерархические структуры.

Выполнение этих трех требований позволяет идентифицировать язык как объектно-ориентированный. Существуют языки, в которых может не соблюдаться одно из требований. Например, если в языке не реализовано наследование, то программирование называется не объектно-ориентированным, а программированием с помощью абстрактных типов данных (programming with abstract data types). К таким языкам относится объектно-осведомленный язык Visual Basic 6.0 (VB6).

Принципы, которым удовлетворяет объектно-ориентированный язык:

1) использует объекты, которые являются представлением абстракций предметной области; объекты реализуют различные операции, совокупность которых образует интерфейс объекта; закрытые переменные образуют состояние объекта;

2) каждому объекту поставлен в соответствие класс; существует множество объектов одного класса;

3) классы могут реализовывать механизмы наследования: наследование реализации и наследование интерфейсов.

К объектно-ориентированным языкам относят: Java, C++, C#, Ruby, Smalltalk.

1.3 Проектирование и анализ в рамках объектно-ориентированной модели

Программирование предполагает построение программных средств с безошибочным и эффективным использованием конкретного языка программирования.

При проектировании основное внимание уделяется точному и эффективному построению сложных систем.

Объектно-ориентированное проектирование представляет собой один из методов проектирования. Он сочетает в себе процесс объектной декомпозиции и систему обозначения. Система обозначения призвана представить логическую и физическую, а также статическую и динамическую модели для проектируемой системы.

Основные аспекты, вытекающие из данного определения:

1) ООП основано на представлении системы в виде совокупности объектов – объектная декомпозиция;

2) использует разные системы обозначений для представления логических (объекты и классы) и физических (модули и процессы) моделей системы, а также ее статические и динамические аспекты.

Объектно-ориентированная декомпозиция – основное отличие объектного подхода от структурного. В ООП для выражения логической структуры системы используются классы и объекты, а в случае структурного подхода – алгоритмы и алгоритмические абстракции. Под объектно-ориентированным проектированием иногда понимают различные методы в результате которых получается представление системы в виде объектов.

Методы структурного анализа основаны на исследовании потоков данных внутри программных систем. Существует также объектно-ориентированный анализ (object-oriented analysis). Объектно-

ориентированный анализ (ООА) позволяет создавать модели действительности с применением объектно-ориентированного подхода.

Объектно-ориентированный анализ – это метод анализа, исследующий требования к системе с точки зрения классов и объектов, относящихся к словарю предметной области.

Результатами объектно-ориентированного анализа являются модели, лежащие в основе объектно-ориентированного проектирования, которое в свою очередь позволяет разработать схему полной реализации системы с использованием объектно-ориентированного программирования.

1.4 Основные концепции объектного подхода

Основные парадигмы программирования и ключевые абстракции, используемые ими, представлены в таблице 1.1.

Таблица 1.1 – Абстракции, лежащие в основе стилей программирования.

Стиль программирования	Ключевая абстракция
Процедурно-ориентированный	Алгоритмы
Объектно-ориентированный	Классы и объекты
Логико-ориентированный	Цели, выражаемые в логике исчисления предикатов
Продукционный	Правила «если, то»
Ориентированный на ограничения	Инвариантные соотношения

Ни один стиль разработки программных систем не является наилучшим. Эффективность парадигмы заключена в задаче, решаемой с использованием языка программирования. Каждая проблема (построение базы данных, базы знаний; разработка вычислительного скрипта; программирование бизнес-приложения) требует применения самостоятельной технологии.

Основой объектно-ориентированной парадигмы является объектная модель. Эта модель состоит из следующих четырех главных элементов (при отсутствии одного из них модель не является объектно-ориентированной): абстракция, инкапсуляция, модульность, иерархия.

Кроме главных, существуют дополнительные элементы: контроль типов, параллелизм, персистентность.

1.4.1 Абстрагирование

Абстракция выделяет существенные характеристики некоторого объекта, отличающие его от всех других видов объектов и, таким образом, четко описывает его концептуальные границы с точки зрения наблюдателя. Абстракция концентрирует внимание на внешнем представлении объекта и позволяет отделить существенные особенности поведения от их реализации. Упрощенно можно утверждать, что абстракция – совокупность значимых свойств объекта.

Выбор правильной совокупности свойств объекта для заданной предметной области представляет собой главную задачу объектно-ориентированного проектирования. Многообразие процессов, явлений, объектов, субъектов и предметных областей порождает различные типы абстракций:

1) Абстракция сущности. Объект, представляющий собой полезную модель некой сущности в предметной области.

2) Абстракция действия. Объект, состоящий из обобщенного множества операций, каждая из которых выполняет однотипные функции.

3) Абстракция виртуальной машины. Объект, группирующий операции, которые либо вместе используются на более высоком уровне управления, либо сами используют некоторый набор операций более низкого уровня.

4) Произвольная абстракция. Объект, включающий в себя набор операций, не имеющих друг с другом ничего общего.

Наиболее части применяется абстракция сущности. Рассмотрим пример абстрагирования. Предметная область: система управления совокупностью автономных роботов (дронов), имеющих одинаковый функционал и предназначенных для выполнения однотипной работы. В заданной постановке уже существует значительный уровень абстракции – не указана цель функционирования роботов. В реальных проектах такое встречается достаточно часто: например, разрабатывается система управления ботами в игре, система контроля роботов широкого спектра применения и т.п.

Выделим основные объекты, которые очевидны из поставленной задачи:

1) Робот. Абстрактное представление автономного робототехнического устройства.

2) Система управления. Абстрактное представление объекта, собирающего информацию о всех роботах, генерирующего управляющие сигналы. Конкретная реализация данной сущности с использованием языка программирования может быть выполнена различными способами (на основе различных технологий), но данный модуль должен быть реализован – он важен для данной предметной области.

3) Сообщение. Абстрактное представление порции информации, передаваемой между роботами и системой управления.

Рассмотрим графическое представление выявленных абстракций (рис. 1.1).

Рисунок 1.1 – Основные абстракции предметной области.

На рисунке показаны выделенные сущности, необходимые для описания предметной области в создаваемом программном продукте. Возможно, при дальнейшем развитии проекта состав сущностей будет изменен, но на данном этапе этого достаточно.

Выделенные сущности на данном этапе проектирования объектной модели не содержат какой-либо детализации.

1.4.2 Инкапсуляция

Абстракция и инкапсуляция дополняют друг друга. В центре внимания абстракции находится наблюдаемое поведение объекта, а инкапсуляция отвечает за внутреннюю реализацию, которая и определяет внешнее поведение. Инкапсуляция реализуется за счет сокрытия информации о деталях функционирования объекта. Объект представляется своего рода «черным ящиком», предоставляя во внешнюю среду инструменты для ввода информации и определенным образом реагируя на изменения внешней среды. Очевидно, что в языках высокого уровня подобное поведение обеспечивается за счет открытых методов объекта.

Инкапсуляция определяет четкие границы между различными абстракциями. Для того, чтобы абстрактное представление сущности предметной области эффективно выполняло свои функции, необходимо, чтобы реализация абстракции была инкапсулирована. На практике это означает, что объект состоит из двух частей: реализация и интерфейс.

Интерфейс класса определяет только внешнее представление объекта, описывая абстракцию поведения всех объектов данного класса. Реализация класса содержит как представление абстракции, так и механизмы, определяющие требуемое поведение объекта.

Инкапсуляцию можно определить следующим образом:

Инкапсуляция – это процесс разделения элементов абстракции, определяющих структуру и поведение; инкапсуляция предназначена для изоляции контрактных обязательств абстракции от реализации.

Вернемся к предметной области: управление множеством роботов. Рассмотрим сущность предметной области «Робот». Поведение объекта, представляющего данную сущность может быть достаточно сложным, но на начальном этапе объектного проектирования достаточно определить, что относится к внешнему поведению (интерфейс) и какие аспекты функционирования сущности скрыты и недоступны из внешней среды. На рис. 1.2 представлен вариант такого разделения.

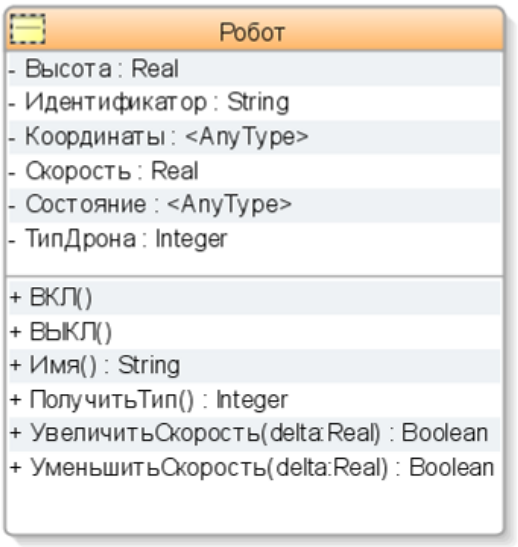


Рисунок 1.2 – Детализация сущности «Робот».

Для конкретизации абстракции «Робот» в результате анализа предметной области были введены закрытые поля данных (на рисунке отмечены знаком «–»). Закрытые поля описывают состояние сущности – они недоступны вне абстракции. Для взаимодействия с внешней средой при

детализации сущности определены открытые операции (помечены символом «+»). Совокупность открытых операций образует интерфейс.

1.4.3 Модульность

Разделение программы на отдельные модули позволяет существенно снизить сложность программ. Но, гораздо важнее, что при модульной декомпозиции возникает множество детально спроектированных, определенных и хорошо документированных интерфейсов. Данное особенности, присущие модульным программам, реализуется неявно.

Чем класс отличается от модуля? В некоторых языках программирования высокого уровня понятия модуль и класс тесно связаны, но эта связь очевидна только в простых проектах. Сложные программные системы неизбежно требуют разделения этих понятий. Модуль – это фрагмент программы, который компилируется отдельно, независимо от всех остальных. Но при этом модули в рамках одно проекта связаны между собой. По отдельности они представляют лишь ограниченное абстрактное представление подмножества сущностей предметной области и особенностей их взаимодействия.

Определение оптимальной стратегии модульной декомпозиции представляет собой такую же сложную задачу, как и выделение оптимального набора абстракций. При этом, следует понимать, что существуют хорошо изученные, распространенные задачи, в которых разделение на модули можно реализовать с использованием устоявшихся шаблонов. К таким задачам относятся корпоративные веб-приложения; приложения для работы с базами данных, содержащие стандартные наборы операций для реализации бизнес-логики. Но при интенсивном развитии информационных технологий все больше областей деятельности охватываются автоматизацией, поэтому возникают нетривиальные задачи, модульная декомпозиция в которых может быть представлена как сложный

процесс с обратной связью. К таким относят большинство задач с повышенными требованиями к устойчивости, безотказности: системы военного назначения, финансовые системы; системы управления физическими объектами, способными причинить вред жизни и здоровью людей.

Модули служат физическими контейнерами, содержащими классы и объекты. В структурном программировании модульность проявляется в группировке подпрограмм по логическим группам на основе критериев связности и целостности. В ООП модульность имеет другой характер, так как предполагает физическую упаковку классов, имеющих большое отличие от подпрограмм.

При практической реализации модульной декомпозиции можно руководствоваться правилами:

1. Конечной целью декомпозиции программы на модули является снижение затрат на программирование. Разделение на модули не носит механический характер: каждому классу – отдельный модуль.

2. Структура каждого модуля должна быть простой для понимания, допускать независимую реализацию других модулей и не влиять на их поведение. Изменение проекта, состоящего из множества модулей, должно существенно упроститься по сравнению аналогичной реализацией проекта в одномодульном варианте.

3. Решение проблемы повторной компиляции модулей. Модули могут компилироваться независимо, но если при компиляции будут выявлены зависимости (зависимости могут быть множественные), то процесс компиляции может занять существенное время. Поэтому при программировании модулей необходимо ориентироваться на использование интерфейсов.

4. Необходимо как можно больше скрывать детали реализации модуля. Данное требование связано с предыдущим, так как позволяет реализовывать интерфейс, а затем постепенно наполнять модуль конкретной реализацией.

На основе анализа представленной информации о модульной декомпозиции можно сформулировать следующее определение:

Модульность – это свойство системы, представленной отдельными, цельными и слабосвязанными фрагментами (модулями).

Принципы абстракции, инкапсуляции и модульности являются синергетическими. Класс определяется для задания границ абстракций предметной области; именно инкапсуляция и модульность позволяют реализовать различия между абстракциями.

Разделение системы на модули имеет важные последствия:

1. Модули являются цельными и неделимыми структурными элементами программного обеспечения. Они характеризуются тем, сто позволяют многократно использовать разработанный код. Модуль не только обеспечивает изоляцию кода, но и приводит к возможности повторного использования кода.

2. При компиляции исходного кода модуля создаются отдельные сборки. При этом программист может столкнуться с ограничением на физический размер модуля. Динамика вызовов подпрограмм и размещение объявлений внутри модулей могут существенно влиять на локальность ссылок и на управление страницами виртуальной памяти.

При модульной декомпозиции проекта необходимо учитывать факторы, не являющиеся следствием технических или программных ограничений. На практике при разработке модулей сложной системы применяются принципы коллективной разработки, следовательно логично, что разработку модуля отводят отдельной группе разработчиков. При таком разделении проекта необходимо сократить уровень взаимодействия между командами разработчиков, проектирующих и реализующих модули системы.

Подобная ситуация может возникнуть при проектировании сложных систем в отношениях субподрядчиков. Для решения подобных проблем необходимо использовать интерфейсы абстракций, которые должны быть согласованы.

1.4.4 Иерархия

Абстракция – важный механизм, но количество абстракций (классов) в реальных приложениях может оказаться очень большим. Механизм инкапсуляции существенно снижает сложность программных систем, так как скрывает внутренние особенности реализации классов, позволяет разработчику сконцентрироваться на абстракции. Модульность позволяет организовать классы абстракций в виде согласованных групп.

Иерархия – это механизм для упорядочивания классов абстракций. При реализации и проектировании программных систем следует учитывать следующие виды иерархий: отношения между классами (иерархия «общее/частное») и отношения объектов (иерархия «целое/часть»).

Одиночное наследование. Данный вид наследования представляет собой вид иерархии, которая наиболее распространена. Данное отношение между классами основано на принципе «является» («is a»). Именно данный вид наследования лежит в основе большинства объектных систем. Наследование может быть одиночным и множественным. Также выделяют наследование интерфейсов и наследование реализации.

При наследовании программист определяет иерархию классов. Каждый класс должен переопределять или расширять функционал базового класса. Каждый класс является абстракцией, поэтому перед реализацией иерархии требуется провести анализ отношений между сущностями предметной области.

В результате анализа и проектировании программной системы программист может выявить необходимость переноса функционала группы сущностей в один класс (суперкласс), который будет представлять все общее,

характерное для всей группы. Поэтому часто указывается, что наследование – это отношение типа обобщение/специализация Суперклассы предоставляют абстрактные интерфейсы (совокупность операций), а производные классы – специальные реализации, в которые добавляются, изменяются и даже скрываются поля и методы из базовых классов. Наследование позволяет существенно сократить исходный код приложения, сделать программы более читабельными. Пренебрежение иерархиями «общее/частное» может привести к «разбуханию» кода.

Без наследования все классы независимы. Каждый тип класса должен разрабатываться отдельно. В результате классы теряют общность, поскольку каждый программист реализует их методы по-своему. В этом случае согласованность классов системы может быть обеспечена только за счет скоординированной работы программистов.

Наследование позволяет создавать новые программы точно так же, как вводятся новые понятия – сравнивая новое с уже известным. Абстракция данных создает механизм, скрывающий методы и состояние объекта. Принцип наследования предполагает предоставление доступа и к состоянию, и к методам объекта, не прибегая к абстракции.

У любого класса обычно существуют два вида клиентов: объекты, выполняющие операции над экземплярами данного класса, и подклассы, наследующие свойства класса.

Разные языки программирования по-разному согласовывают наследование с инкапсуляцией. Во многих ООП-языках интерфейс класса может состоять из трех частей: закрытой (private), доступной только самому классу; защищенной (protected), в которой объявляются члены, доступные только для класса и его подклассов; открытой (public), доступной всем клиентам.

Пример иерархии множественного наследования:

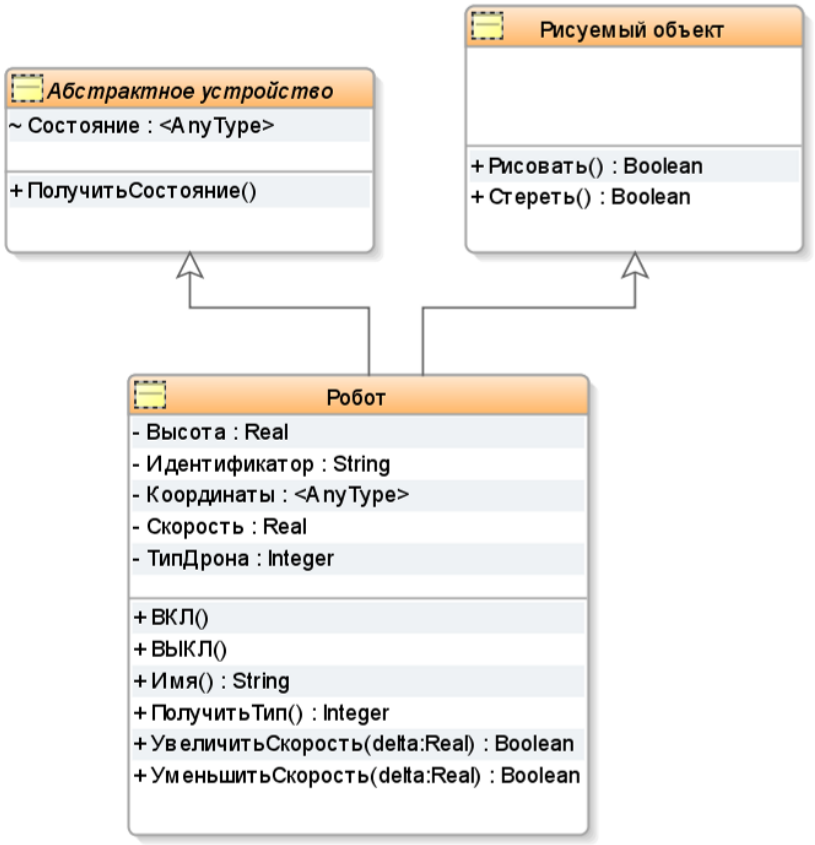


Рисунок 1.3 – Вариант реализации множественного наследования.

На рис. 1.3 представлен вариант множественного наследования: один класс «Робот» наследует реализацию классов «Абстрактное устройство» и «Рисуемый объект». Такие связи образовались в результате следующих соображений. В предметной области был выделен обширный класс устройств, обладающих состоянием. Роботы (дроны) являются частным случаем подобных устройств. Для всех абстрактных устройств характерно одно общее свойство – наличие состояния. Состояние в классе «Абстрактное устройство» решено реализовать как защищенное свойство (protected), на диаграмме это отражается с использованием символа «~». Следует обратить внимание, что на данном этапе проектирования тип поля «Состояние» не определен. Таким образом, все классы, наследуемые от «Абстрактное устройство» получат свойство «Состояние» и открытый интерфейсный метод «Получить состояние». В дополнение к возможностям абстрактного

устройства в классе «Робот» добавлены дополнительные поля, конкретизирующие данную абстракцию.

Наличие класса «Рисуемый объект» продиктовано необходимостью визуализировать роботов (например, на карте местности). Каждый подобный объект должен содержать два метода: «Рисовать» и «Стереть». Эти методы являются открытыми и образуют интерфейс объекта. Таким образом, класс «Робот» получает возможность использования методов рисования, наследуя их от «Рисуемый объект».

В языках программирования Java и C# множественное наследование реализации отсутствует, но проблема наследования функциональности от нескольких классов решается с использованием интерфейсов. Отсюда в этих языках присутствуют два вида наследования: наследование реализации и наследование интерфейсов.

Несмотря на то, что, теоретически, множественное наследование представляет собой простую концепцию, при практической реализации подобного отношения между классами возникает ряд проблем. Одна из основных проблем – конфликты имен между различными суперклассами и повторное наследование. Конфликты имен возникают, когда в двух или большем числе базовых классов существуют поля или методы с одинаковыми сигнатурами. Повторное наследование возникает, когда несколько суперклассов наследуют свойства общего суперкласса. В таких ситуациях возникает ромбическая структура наследования, и проектировщик должен решить, сколько копий полей должен унаследовать класс самого нижнего уровня у суперкласса самого верхнего уровня – одну или две. В большинстве языков программирования отсутствует возможность реализовать повторное наследование.

На рис. 1.4 представлен пример одиночного наследования.

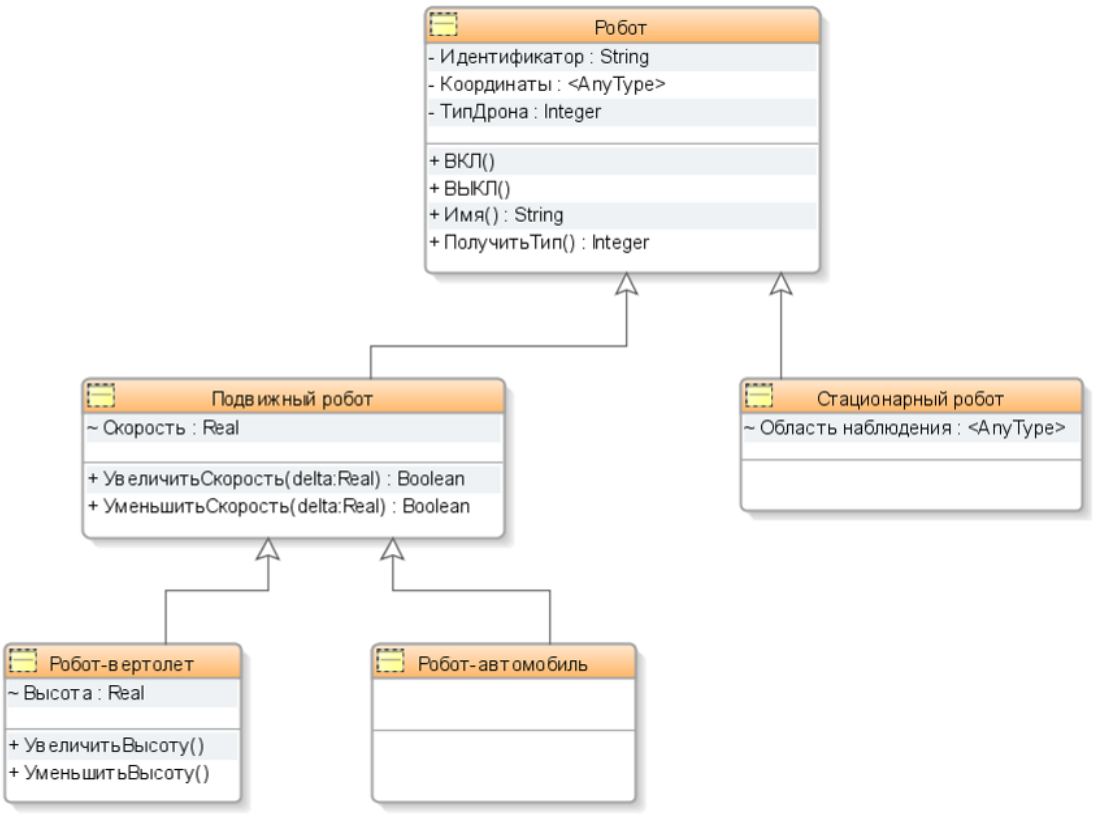


Рисунок 1.4 – Иерархия класса «Робот».

На рис. 1.4. представлен пример одиночного наследования. Полученная иерархия отражает представление исследователя о предметной области. В результате анализа было выявлено, что одного класса «Робот» для эффективного представления сущностей недостаточно. Были выявлены различные подтипы дронов, которые обладают связями типа «общее/частное». В соответствии с потребностями задачи была построена данная иерархия. На рис. 1.4 видно, что на данном этапе различия между классами «Подвижный робот» и «Робот-автомобиль» не выявлены, но исследователь закладывает в объектную модель именно такое разделение. Это общепринятая практика в объектном моделировании и в объектно-ориентированном программировании в частности. Разделение предметной области на отдельные сущности и представление классами позволяет гибко изменять состав и структуру классов на любом этапе проектирования программного обеспечения.

Если иерархии «общее/частное» определяют отношение «обобщение/специализация», то иерархии «целое/часть» описывают отношения агрегации. Рассмотрим, например, абстракцию «Робот». Можно предположить, что конкретный робот – это совокупность узлов и схема их соединения. То есть, детали и схема соединения – это части робота. Это отношение, которое можно описать фразой «является частью», называется агрегацией (aggregation).

Рассмотрим пример:

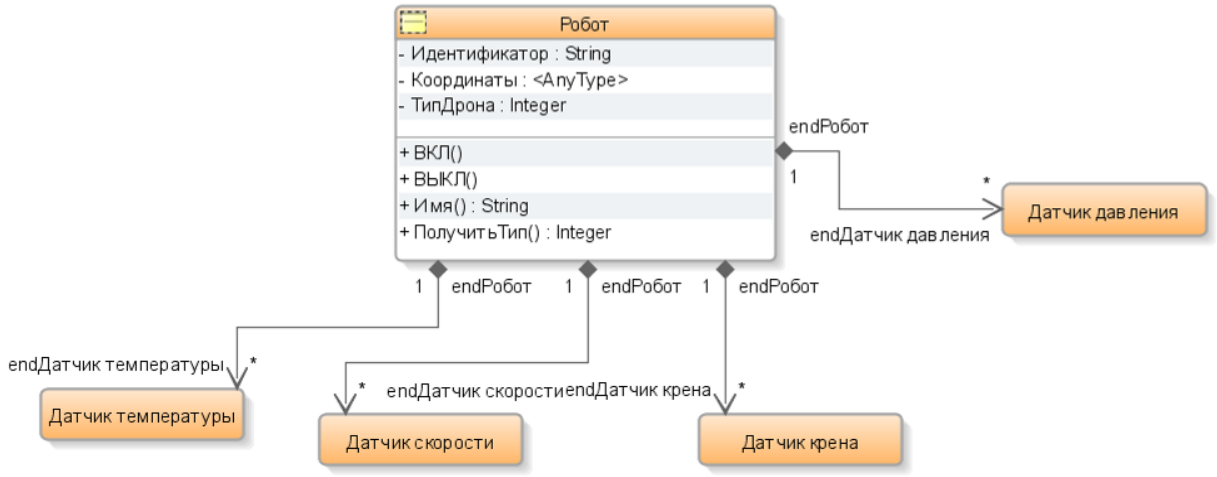


Рисунок 1.5 – Пример реализации агрегации.

В данном случае предполагается, что работ снабжен определенным количеством датчиков, которые являются частью его конструкции. На данном этапе проектирования не указывается существуют ли датчики отдельно от робота или нет, важно, что классы датчиков являются его частями.

Агрегация есть во всех объектно-ориентированных проектах и языках программирования. Агрегация и наследование сами по себе являются эффективными механизмами построения сложных систем, но в случае совместного использования они предоставляют программисту мощный инструмент: агрегация позволяет физически объединить логически связанные структуры, а наследование – повторно использовать эти общие группы в разных абстракциях.

Анализируя иерархии, подобные рассмотренным выше, часто говорят об уровнях абстракции, концепцию которых впервые предложил Дейкстра. В терминах иерархии «общее/частное» абстракция верхнего уровня является обобщенной (например, робот), а абстракция нижнего уровня – специализированной (например, деталь или схема соединения).

Агрегация поднимает проблему владения. Абстракция робота допускает, что робот состоит из множества деталей, но вследствие замены одной детали другой робот не становится другим роботом, а уничтожение робота не обязательно означает уничтожение всех его деталей (ведь их можно использовать для сборки другого робота). Иначе говоря, жизненные циклы робота и деталей не зависят друг от друга. Справедливы и обратные рассуждения, проектное решение предусматривает, что объект «Схема соединения» неразрывно связан с объектом «Робот» и не существует независимо от него. Следовательно, схема соединения может храниться в каждом экземпляре класса Robot и уничтожается вместе с ним.

1.4.5 Контроль типов

Понятие типа происходит из теории абстрактных типов данных. Тип – это точная характеристика структуры и поведения, присущих некоторой совокупности объектов.

В рамках данного пособия будем считать, что понятия «тип» и «класс» являются синонимами. Для большинства современных объектно–ориентированных языков это также справедливо. Но в Java существуют примитивные типы, которые классами не являются. В C# существуют типы по значению и ссылочные типы (классы C# – ссылочные типы, определенные программистом).

Контроль типов – это механизм, контролирующий правила замены объектов различных типов.

Контроль типов ограничивает разработчика – программист должен учитывать данный механизм при проектировании системы, но, одновременно, контроль типов позволяет разработчику избежать существенных проблем еще на этапе проектирования системы. Конкретный язык программирования может иметь сильный или слабый контроль типов, и даже совсем не иметь такого свойства, оставаясь объектно-ориентированным.

Строгий контроль позволяет использовать язык программирования для поддержки определенных проектных решений и позволяет справиться с нарастающей сложностью систем. Однако у строгого контроля типов есть обратная сторона. На практике она порождает семантические зависимости, при которых небольшое изменение интерфейса в базовом классе вынуждает повторную компиляцию всех его подклассов.

Преимущества, обеспечиваемые механизмом контроля типов::

1. Часто при отсутствии контроля типов программа может завершиться непредсказуемо, сто связано с отсутствием механизма выявления ошибок на этапе разработки.

2. Чаще всего, цикл разработки, включающий этапы редактирования, компиляции и отладки, является очень трудоемким. Автоматическое выявление ошибок на раннем этапе крайне желательно.

3. Системы с контролем типов оптимально подходят для создания библиотек программного кода, так как позволяют в таких системах можно просто реализовать процесс документирования.

4. Контроль типов позволяет оптимизировать работу компилятора.

Языки, в которых контроль типов отсутствует, обладают большей гибкостью, программисты должны знать и контролировать, какие объекты ожидаются в качестве аргументов и какие подлежат возвращению.

На практике безопасность, обеспечиваемая языками программирования со строгим контролем типов, с лихвой компенсирует потерю гибкости,

присущей языкам без контроля типов, особенно при проектировании крупномасштабных систем.

1.4.6 Параллелизм

На практике программист сталкивается с проблемами, в которых исходный код должен выполняться параллельно. Параллельное выполнение программ обусловлено также повышенными требованию к скорости решения задачи. В таких системах используют различные аппаратные м программные решения: кластеры, многопроцессорные системы, многоядерные системы. Разбиение логики приложения на отдельные процессы позволяет представить приложение как совокупность динамических независимых операций.

Каждая программа должна иметь один поток управления, но в параллельной системе существует возможность создавать множество потоков управления с разным приоритетом.

Выделяют различные виды параллелизма:

– «тяжеловесный» (heavyweight);

– «легковесный» (lightweight).

Тяжеловесные процессы, обычно, функционируют в среде операционной системы и предполагают значительный расход ресурсов. Легковесные процессы позволяют выполнять ограниченные операции, но не так требовательны к ресурсам. Основным методов взаимодействия тяжеловесных процессов является межпроцессорное взаимодействие, которое является слишком затратным. При программировании с использованием легковесных процессов используются конфигурации процессов, обрабатывающие единый массив данных.

Программирование приложений, ориентированных на параллельное выполнение сопряжено с большим набором трудностей: согласование потоков управления, согласование данных, проблемы критической секции, решение проблем согласования и безопасности типов.

Существенно отличаются уровни абстракции в объектно-ориентированном и параллельном программировании: для ООП основной интерес представляет сама абстракция, инкапсуляция, иерархия; для параллельной разработки интерес представляет абстракция и синхронизация процессов.

Объект – это понятие, объединяющее две точки зрения: каждый объект (построенный на основе абстракции реального мира) может представлять собой отдельный поток управления (абстракцию процесса). Такой объект называется активным (active). Действительно, каждый объект – не просто реализация абстракции, как описания реального процесса (явления) предметной области. Объект имеет свой жизненный цикл, свой период существования.

Система, построенная на основе объектно-ориентированного проектирования, может быть представлена как совокупность взаимодействующих объектов, часть из которых является активной и независимой от других. Следуя этому подходу, можно сформулировать такое определение параллелизма.

Параллелизм – это свойство, отличающее активные объекты от пассивных.

Рассмотрим пример параллелизма, который очевиден в рамках предметной области – системы управления роботами. В роботах установлено множество сенсоров. Например, навигационный датчик для определения координат, датчик высоты для определения высота летающего дрона; датчик крена и т.д. Очевидно, что все эти детали должны работать параллельно, регулярно и независимо передавать данные в управляющий блок робота, который, в свою очередь, должен реагировать на каждую полученную порцию данных.

Объектно-ориентированное программирование также рассматривает параллелизм. Во-первых, в современных языках программирования (Java, C#) параллелизм является неотъемлемой часть реализуемой технологии.

Экземпляры классов представляют потоки. Это значит, что существование нескольких объектов в программе означает существование нескольких параллельных потоков.

Во-вторых, часто используются библиотеки, реализующие легковесные потоки. Подобные библиотеки являются зависимыми от платформы, но программист может использовать универсальный интерфейс, который предоставляет сама библиотека.

В-третьих, параллелизм можно имитировать с помощью прерываний. Разумеется, для этого необходимы определенные знания низкоуровневого аппаратного обеспечения. Например, в реализации класса «Управляющий блок» можно предусмотреть аппаратный таймер, периодически прерывающий выполнение потока управления для того, чтобы все датчики измерили текущие параметры и активизировали свои функции обратного вызова.

Независимо от выбранного подхода включение параллелизма порождает проблему синхронизации работы как активных, так и последовательных объектов. Например, если два датчика пытаются послать сообщения блоку управления, необходимо использовать определенный механизм взаимной блокировки, гарантирующий, что блок управления не будет разрушен при одновременной попытке двух активных объектов изменить его состояние. Для решения этой проблемы необходимо использовать идеи абстракции, инкапсуляции и параллелизма.

В параллельных системах недостаточно определить методы объекта – необходимо также принять меры для сбережения семантики этих методов в присутствии нескольких потоков управления.

1.4.7 Персистентность

Любой программный объект занимает определенный объем памяти и существует определенное время. Множество способов существования объектов велико: от промежуточных объектов, возникающих лишь во время вычисления выражений, до постоянных баз данных, существующих даже после завершения работы программы.

Персистентность объектов может проявляться в разном виде:

1. Промежуточные результаты вычисления выражений.

2. Локальные переменные в вызове процедур.

3. Собственные переменные, глобальные переменные и динамические данные.

4. Данные, сохраняющиеся между разными сеансами выполнения программы.

5. Данные, сохраняющиеся при переходе от одной версии программы к другой.

6. Данные, существующие после исчезновения программы.

Первые три вида персистентности характерны для традиционных языков программирования, а последние – для баз данных.

Объекты, которые назвали «данными, существующими дольше программы», нашли интересное применение при создании Web-приложений, которые могут не иметь связей с источником данных, используемых на протяжении всего периода выполнения транзакции.

Данные, которые были предоставлены клиентскому приложению или Web-сервису, могут измениться. При этом связь с клиентом к этому времени, может быть разорвана. Каким образом будет вести себя вся система в этом случае необходимо решить программисту.

Для реализации таких распределенных и несвязных сценариев были разработаны особые технологии, такие как ADO.NET; различные механизмы ASP.NET и ASP.NET MVC, механизмы серверов сообщений в Java и другие.

Сочетание параллелизма и объектов привело к созданию параллельных объектно-ориентированных языков программирования. Точно также внедрение принципа персистентности в объектную модель привело к появлению объектно-ориентированных баз данных. На практике подобные базы данных создаются с помощью проверенных временем технологий – последовательных, индексированных, иерархических, сетевых или реляционных моделей, но наряду с этим программисты могут использовать абстракцию объектно-ориентированного интерфейса, позволяющего выполнять запросы к базе данных и другие операции над объектами объектов, время жизни которых превосходит время жизни отдельной программы.

Такое сочетание разных подходов значительно упрощает разработку отдельных видов приложений. В частности, это позволяет применить одни и те же методы проектирования к сегментам программы, одни из которых связаны с базами данных, а другие – нет.

Некоторые объектно-ориентированные языки программирования непосредственно поддерживают персистентность. Например, язык Java поддерживает технологии Enterprise Java Beans (EJBs) и Java Data Objects.

Иногда персистентность обеспечивается за счет коммерческих объектно-ориентированных баз данных. Но намного чаще для этого используются объектно-ориентированные оболочки реляционных баз данных (системы Object Relationsl Model, ORM): ADO.NET, Hibernate, Entity Frmework.

Персистентность часто рассматривается как способ существования объектов во времени. Это объясняется тем, что в большинстве систем однажды созданный объект занимает постоянный объем памяти, пока не будет уничтожен. Однако в распределенных системах персистентность можно рассматривать как способ существования объектов в пространстве. В таких системах полезно рассматривать объекты, которые могут

перемещаться с одной машины на другую, изменяя свое представление. Современные технологии Java EE и .NET подразумевают активное использование распределенных решений.

Персистентность – это способность объекта преодолевать временные рамки (т.е. продолжать свое существование после исчезновения своего создателя) или пространственные пределы (т.е. выходить за пределы своего первоначального адресного пространства).

1.5 Преимущества объектной модели

Объектная модель отличается от других моделей проектирования, анализа и программирования систем, но данная технология не является отдельной, изолированной и не связанной с прочими подходами.

Наиболее важно то, что объектная модель позволяет создавать системы, у которых есть пять признаков хорошо структурированных сложных систем: иерархическая структура, относительность выбора элементарных компонентов (например, несколько уровней абстракций), разделение функций, общая структура и устойчивые промежуточные формы.

Объектный подход предоставляет разработчику ряд практических преимуществ:

1. Объектная модель позволяет эффективно использовать выразительные возможности объектных и объектно-ориентированных языков программирования.

2. Использование объектного подхода стимулирует повторное использование не только кода, но и проектных решений. Объектно-ориентированные системы компактнее, чем их традиционные эквиваленты. Компактность означает не только уменьшение размера программ, но и удешевление и ускорение проекта за счет повторного использования проектных решений. Однако повторное использование кода, как правило, не

возникает само по себе, если оно не было основной целью проекта. Кроме того, первоначальная разработка кода, предназначенного для повторного использования, может оказаться более дорогой, чем обычно. Однако эти первоначальные затраты впоследствии будут компенсированы за счет экономии средств и времени при использовании в новых проектах.

3. Использование объектной модели приводит к созданию систем с устойчивыми промежуточными формами, что упрощает их изменение. Это позволяет улучшать систему со временем, не прибегая к ее полной переделке при первой же серьезной модификации.

4. Объектная модель уменьшает риски, связанные с проектированием сложных систем. Прежде всего, этому способствует то, что процесс объединения компонентов системы в одно целое растягивается на весь период ее разработки, а не является единовременным событием. Объектный подход к проектированию подразумевает разумное разделение функций, что также уменьшает степень риска и повышает уверенность в правильности принимаемых решений.

5. Объектная модель учитывает особенности процесса познания.

Контрольные вопросы

1. Перечислите основные принципы объектной модели проектирования.
2. Дайте определение понятий «объект», «класс», «абстракция».
3. Поясните сходства и различия в терминах «объектно-ориентированное проектирование» и «объектно-ориентированное программирование».
4. Дайте определение объектно-ориентированного программирования и охарактеризуйте основные аспекты, вытекающие из этого определения.

5. Перечислите и охарактеризуйте требования, предъявляемые к объектно-ориентированным языкам программирования.

6. Что такое объектно-ориентированный анализ? Что является результатом данного процесса?

7. Какие парадигмы программирования вы знаете? Какие абстракции лежат в основе каждой из этих парадигм?

8. Что такое абстрагирование? Какие можно выделить типы абстракций?

9. Что такое инкапсуляция? Приведите примеры с использованием конкретного языка программирования.

10. Что такое модульность? Какие принципы используются при модульной декомпозиции программ?

11. Что такое иерархия? Какие типы иерархий вы знаете?

12. Поясните назначение механизма контроля типов.

13. Что такое «параллелизм», как принцип объектной модели проектирования? Приведите примеры реализации этого принципа в языках программирования Java и C#.

14. Что такое персистентность? В чем проявляется персистентность? Приведите примеры проявления персистентности в языках программирования.

РАЗДЕЛ 2. КЛАССЫ И ОБЪЕКТЫ

Объект характеризуется состоянием, поведением и индивидуальностью. Структура и поведение одинаковых объектов описываются в общем для них классе. Состояние объекта определяет его статические и динамические свойства.

Поведение объекта – это действия и реакции объекта, выраженные в терминах изменения состояний и передачи сообщений. Индивидуальность объекта – это свойство объекта, отличающие его от всех других объектов.

Класс – это множество объектов с одинаковой структурой и поведением. Между классами существует три вида отношений: ассоциация, наследование и агрегация.

Основными абстракциями являются классы и объекты, образующие словарь предметной области. Механизм – это структура, с помощью которой множество взаимодействующих объектов обеспечивает требуемое поведение.

Качество абстракций измеряется степенью их связанности, связности, достаточности, полноты и элементарности.

2.1 Объекты

Упрощенно объект можно определить как материальная сущность, демонстрирующая четко определенное поведение. С точки зрения познавательных способностей человека, объектом может быть одна из следующих сущностей:

– материальный и (или) видимый предмет;

– сущность, поддающаяся осмыслению;

– сущность, на которую направлена мысль или действие.

Таким образом, неформальное определение объекта дополняется идеей о том, что объект является частью реального мира и, следовательно, существует во времени и пространстве. В программировании термин объект впервые был применен в языке Simula и применялся для моделирования определенных аспектов реальности.

Объекты реального мира – не единственная разновидность объектов, представляющих интерес при проектировании систем программного обеспечения. На этапе проектирования возникают и другие важные типы объектов, взаимодействие между которыми лежит в основе механизмов, реализующих более сложное поведение. Например, управляющие объекты, объединяющие в одно целое потоки событий и обеспечивающие связь между другими объектами.

Объект представляет собой конкретный идентифицируемый предмет, элемент или сущность, реальную или абстрактную, играющий четко определенную роль в данной предметной области.

Объект обладает состоянием, поведением и индивидуальностью. Структура и поведение схожих объектов определяется в общем для них классе. Термины экземпляр и объект являются синонимами.

2.1.1 Состояние объектов

Состояние объекта характеризуется перечнем всех (как правило, статических) свойств данного объекта и текущими (как правило, динамическими) значениями каждого из этих свойств.

Свойствами объекта называются отличительные характеристики, черты, качества или особенности, обеспечивающие его уникальность.

Тот факт, что каждый объект имеет состояние, означает, что любой объект занимает определенный объем пространства (физического или в памяти компьютера). Все объекты в системе инкапсулируют некое состояние, и все состояния в системе инкапсулируются объектами.

Инкапсуляция состояния объекта – это необходимое, но не достаточное условие полного описания предназначения абстракции, выявленной на этапе проектирования.

Рассмотрим пример: разработка многопользовательской компьютерной игры в жанре фэнтези. Выделим сущность «Игровой персонаж». Для сущности характерны некоторые общие свойства (имя, раса, идентификатор, тип, здоровье, урон) (рис. 2.1).

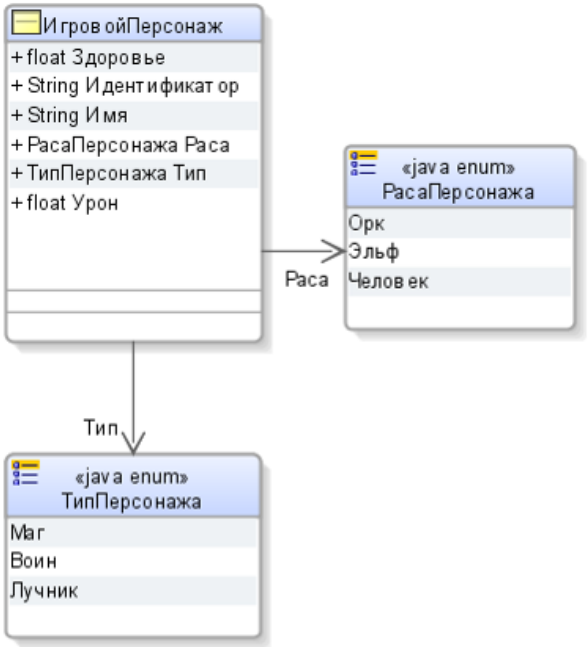


Рисунок 2.1 – Пример класса.

На рис. 2.1 введены два новых типа, отличных от класса – это перечисления. На диаграмме построена диаграмма в нотации Java. То есть задаются спецификаторы типов и модификаторы доступности, принятые в этом языке. Стрелка, указывающая на типы перечислений, сигнализирует о том, что тип «ИгровойПерсонаж» использует типы «ТипПерсонажа» и «РасаПерсонажа». В языках программирования Java и C# действуют правила построения идентификаторов, в соответствии с которыми имена типов и переменных не могут содержать пробелов – отсюда именно такое именование типов.

На рис. 2.1 описана абстракция в виде класса. Каждый компонент абстракции обозначает конкретное свойство игрового персонажа. Эта абстракция не является объектом, поскольку она не представляет собой конкретный экземпляр персонажа. Конкретизируя эту абстракцию, можно создать два разных объекта, например, «Орк» и «Эльф», занимающих определенный объем памяти. На данном этапе сущность «Игровой персонаж» достаточно конкретизирована для того, чтобы перейти к генерации кода на языке ООП. На рис. 2.2 показан код Java, соответствующий диаграмме, представленной на рис. 2.1.

```
3  public class ИгровойПерсонаж {
4      public String Имя;
5      public РасаПерсонажа Раса;
6      public String Идентификатор;
7      public ТипПерсонажа Тип;
8      public float Здоровье;
9      public float Урон;
10 }
```

Рисунок 2.2 – Описание класса на языке Java.

На практике принято инкапсулировать состояние объекта, а не открывать его для всех сущностей. С учетом сказанного необходимо изменить абстракцию (класс) следующим образом (см. рис. 2.3).

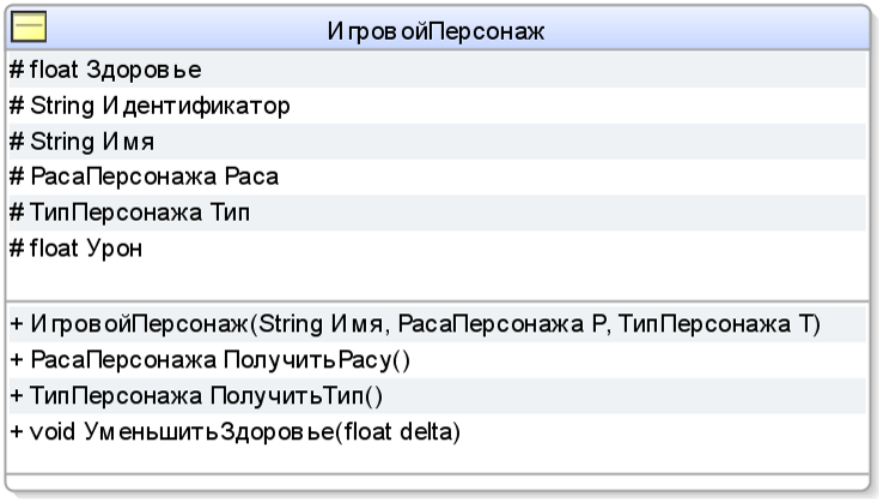


Рисунок 2.3 – Корректировка описания класса: реализация принципа инкапсуляции.

Эта абстракция ненамного сложнее предыдущей (рис. 2.1), но по ряду соображений гораздо предпочтительнее. В частности, ее внутренняя реализация скрыта от внешних клиентов (т.е. защищена, что обозначается символом #). Если представление абстракции будет изменено, то придется повторно скомпилировать часть кода, но, с семантической точки зрения, на внешних клиентов это изменение никак не повлияет (иначе говоря, существующий код останется неизменным), так как внешние клиенты будут использовать только интерфейсные методы класса (помеченные как «+»).

В данном представлении решена проблема занимаемой объектом памяти за счет явного определения операций (обязанностей), которые другие объекты могут выполнять над объектами данного класса. В частности, все объекты окружения имеют право узнать имя персонажа, его тип, а также игровую расу персонажа (открытые операции обозначены символом +).

2.1.2 Поведение объектов

Ни один объект не существует в изоляции от других. Все объекты либо подвергаются воздействию, либо сами воздействуют на другие объекты.

Поведение – это действия и реакции объекта, выраженные через изменения состояния объекта и передачу сообщений. Поведение объекта – это его видимая извне деятельность.

Операцией называется определенное воздействие одного объекта на другой с целью вызвать некую реакцию. Например, в примере компьютерной игры один персонаж может нанести крон другому персонажу, то есть уменьшить численное значение его свойства «здоровье». При этом менять имя другого персонажа или его урон воздействующий объект не может.

В языке Java операции, которые может выполнить клиент, называются методами. Используя терминологию языков, являющихся наследниками процедурно-ориентированных языков программирования, например, языка C++, говорят, что один объект вызывает функцию-член другого.

Но существует другой механизм взаимодействия между объектами. Во многих объектно-ориентированных языках, таких как Smalltalk, Java, C#, это явление описывается как передача сообщений между объектами. Сообщение – это операция, которую один объект выполняет над другим, хотя механизмы диспетчеризации сообщений и операции отличаются друг от друга. В дальнейшем будем считать термины операция и сообщение синонимами.

Передача сообщений – это лишь часть процесса, описывающего поведение объекта, поскольку оно зависит и от его состояния. Например, один персонаж наносит урон другому, но реакция, но реакция объекта влияния будет зависеть от его состояния. Если объект обладает высокой степенью защиты, то величина урона будет уменьшена на величину защиты. Если же урон превышает защиту существенно, то здоровье объекта влияния уменьшится до 0 и объект влияния считается уничтоженным (опять же изменится состояние объекта).

Таким образом, поведение объекта зависит как от выполняемых над ним операций, так и от его состояния, причем некоторые операции косвенно изменяют состояние объекта. Концепция побочного действия позволяет сформулировать следующее определение.

Состояние объекта представляет собой суммарный результат его поведения.

Наиболее интересными являются объекты, не имеющие статичного поведения: их состояние описывается значениями, доступными для модификаций и запросов.

2.1.3 Операции

Операция – это методы, которые класс предоставляет для вызова другим классам. Операции можно рассматривать как услуги, предоставляемые классом-сервером для классов-клиентов. На практике клиент обычно совершает над объектами операции пяти видов:

1. Модификатор: операция, изменяющая состояние объекта.

2. Селектор: операция, имеющая доступ к состоянию объекта, но не изменяющая его.

3. Итератор: операция, обеспечивающая доступ ко всем частям объекта в строго определенном порядке.

4. Конструктор: операция, создающая объект и/или инициализирующая его состояние.

5. Деструктор: операция, стирающая состояние объекта и/или уничтожающая сам объект.

Существует альтернативная классификация операций:

1. Функции управления.

2. Функции реализации.

3. Вспомогательные функции (все виды модификаторов).

4. Функции доступа (эквиваленты селекторов).

2.1.4 Роли и обязанности объектов

Совокупность всех методов, связанных с конкретным объектом, образует его протокол. Таким образом, протокол объекта описывает поведение объекта и содержит все его статические и динамические свойства. Работая со сложными абстракциями, полезно разделять протокол на логические группы свойств. Эти свойства описывают роли, которые может исполнять объект. Роль определяет контракт абстракции с клиентами.

Обязательства – это совокупность всех услуг, предусмотренных во всех контрактах. Состояние и поведение объекта совместно определяют исполняемые им роли, а те, в свою очередь, обеспечивают выполнение обязанностей данной абстракции.

Роли, которые играет объект в приложении, могут динамически изменяться. Например, игровой персонаж может атаковать, но в следующую

секунду он может подвергнуться воздействию другого объекта и вынужден играть другую роль – реагировать на воздействие.

Объектно-ориентированный анализ часто начинается с изучения разных ролей, которые играет объект. Во время проектирования они уточняются с помощью конкретных операций, реализующих обязанности, связанные с каждой ролью.

2.1.5 Объекты как конечные автоматы

Наличие внутреннего состояния объектов означает, что порядок выполнения операций имеет большое значение. Это обстоятельство позволяет представить объект в качестве маленького независимого автомата.

Целесообразно разделить объекты на активные и пассивные. Активный объект имеет собственный поток управления, в отличие от пассивного. Активные объекты обычно автономны, т.е. могут демонстрировать определенное поведение без воздействия со стороны других объектов. Пассивные объекты, наоборот, могут изменять свое состояние только под воздействием других объектов. Таким образом, активные объекты системы – это источники управляющих воздействий (roots of control). Если в системе существуют несколько потоков управления, то в ней обычно действуют несколько активных объектов.

Возвращаясь к примеру компьютерной игры: активными могут выступать все объекты – игровые персонажи (при этом одни могут управляться компьютером, другие – пользователем). Пассивными объектами могут являться статические объекты ландшафта: их здоровье будет уменьшаться только при воздействии на них со стороны активных объектов; они не выполняют никаких самостоятельных действий.

С другой стороны, в последовательных системах в каждый момент времени существует, как правило, только один активный объект, например, диспетчер, распределяющий сообщения. В таких архитектурах все остальные

объекты являются пассивными, а их поведение изменяется только тогда, когда они получают сообщение от активного объекта. В последовательных архитектурах другого типа (например, в системах обработки транзакций) нет явного центра активности, и управление распределено среди пассивных объектов системы.

2.1.6 Различия между объектами

Индивидуальность – это свойство объекта, отличающее его от всех других объектов. Данное явление проще всего объяснить на примере с использованием ссылочных типов Java или C#. На рис. 2.4 представлены различные стадии выполнения приложения. На первоначальном этапе выполнения приложения все переменные указывают на отдельные объекты.

На втором этапе две из трех переменных указывают на объект «Игровой персонаж 1», при этом адрес размещения объекта «Игровой персонаж 2» утерян. На заключительном этапе работы программы, все три переменные указывают на один и тот же объект «Игровой персонаж 3»; при этом оставшиеся два объекта недоступны.

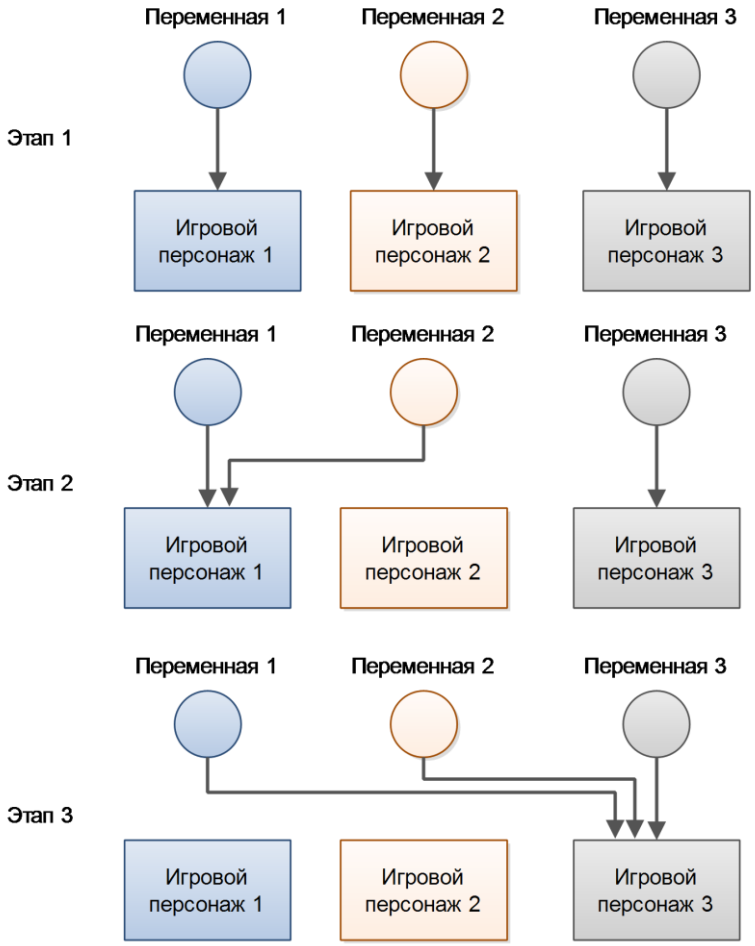


Рисунок 2.4 – Индивидуальность объекта.

Эта ситуация называется структурным разделением (structural sharing) и и означает, что объект может именоваться по-разному. Иначе говоря, объект может иметь псевдонимы. Структурное разделение порождает много проблем в объектно-ориентированном программировании. Трудность распознавания побочных эффектов при обращении к объекту через псевдонимы часто приводит к «утечкам памяти», нарушению правил доступа к памяти, и, что еще хуже, непредсказуемому изменению состояния.

В таких языках программирования, как Smalltalk, C# и Java, память, выделенная для подобных объектов, уничтожается механизмом «сборки мусора». В языках типа C++ утерянная память не освобождается, пока не завершится программа, создавшая объект.

Такие утечки памяти могут вызвать и простое неудобство, и крах, особенно если программа должна непрерывно работать длительное время.

2.2 Отношения между объектами

Объекты являются строительными блоками программы в рамках ООП парадигмы, но использование отдельных объектов не является эффективной стратегией. Поведение системы определяется только взаимодействием между объектами.

Отношения между двумя объектами основываются на взаимных предположениях о допустимых операциях и ожидаемом поведении. Особый интерес для объектно-ориентированного анализа и проектирования представляют два типа отношений между объектами: связи и агрегация.

2.2.1 Связи объектов

Связь (link) – физическое или концептуальное соединение между объектами. Объект взаимодействует с другими объектами, используя связи с ними.

Связь – это конкретное соединение, через которое объект (клиент) запрашивает услугу у другого объекта (сервера) или управляет им.

На рис. 2.5 продемонстрированы несколько связей. Линия, соединяющая две пиктограммы, которые соответствуют объектам, обозначает существование связи между ними и передачу сообщений от одного к другому. Сообщение изображается в виде стрелки, указывающей направление передачи, с меткой, представляющей собой его содержание.

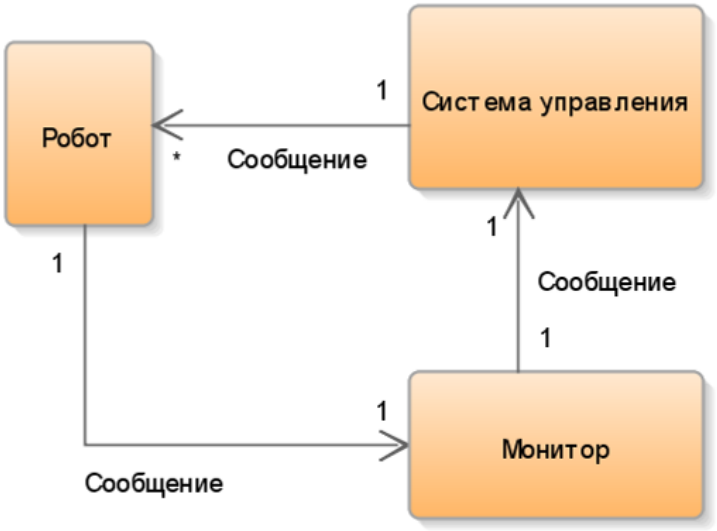


Рисунок 2.5 – Передача сообщений между объектами системы.

Например, на рис. 2.5 показан фрагмент упрощенной схемы управления роботом. Робот передает сообщение блоку управления. Но передача осуществляется не напрямую, а через специальный класс «Монитор». Класс «Монитор» собирает информацию с датчиков роботов и передает все данные, со всех роботов централизовано классу «Блок управления».

В свою очередь, класс «Блок управления» воздействует на классы «Робот», передавая сообщения с управляющими сигналами.

Участвуя в связи, объект может выполнять одну из ролей:

1. Контроллер (controller). Этот объект может выполнять операции с другими объектами, но сам никогда не подвергается воздействию других объектов. В некоторых предметных областях термины активный объект (active object) и контроллер являются синонимами.

2. Сервер (server). Этот объект не выполняет операций с другими объектами, а лишь подвергается воздействию со стороны других объектов.

3. Агент (proxy). Такой объект может выполнять операции над другими объектами и подвергаться воздействию с их стороны. Как правило, агенты создаются для представления реальных объектов, существующих в предметной области конкретного приложения.

Рассмотрим два объекта A и B, между которыми существует связь. Для того чтобы объект A мог послать сообщение объекту B, необходимо, чтобы объект A имел доступ к нему. В ходе анализа задачи видимость объектов, чаще всего, можно игнорировать, но при разработке конкретной реализации системы необходимо обеспечить доступ связанных объектов друг к другу.

Когда один объект посылает сообщение другому по линии связи, говорят, что эти объекты синхронизируются. В последовательном приложении синхронизация объектов представляет собой обычный вызов метода. Однако в многопоточных системах объекты нуждаются в более изощренных схемах передачи сообщений, чтобы разрешить проблемы взаимного исключения, типичные для параллельных систем. Как уже отмечалось, активные объекты имеют свои собственные потоки управления, поэтому другие активные объекты на их семантику, как правило, не влияют. Однако если активный объект имеет связь с пассивным, возможны три варианта синхронизации:

1. Последовательная (sequential). Семантика пассивного объекта в каждый момент времени обеспечивается в присутствии только одного активного процесса.

2. Защищенная (guarded). Семантика пассивного объекта обеспечивается в присутствии многих потоков управления, но активные клиенты должны взаимодействовать, реализуя взаимное исключение.

3. Параллельная (concurrent). Семантика пассивного объекта обеспечивается в присутствии многих потоков управления, причем взаимное исключение гарантируется сервером.

2.2.2 Агрегация

В отличие от связей между объектами, которые обозначают равноправные отношения между объектами («клиент-серверные»), агрегация

описывает связь типа «целое-часть» и позволяет переходить от целого (агрегата) к его компонентам. Например, объект «Робот» представляет собой целое, а объект «Датчик» является его частью.

Очевидно, что объект «Робот» может пересылать сообщения объекту «Датчик». Но для реализации обратной связи (от части к контейнеру) необходимо, чтобы информация о контейнере («Робот») была частью состояния объекта «Датчик».

При проектировании объектных систем необходимо обеспечить компромисс между связями или агрегацией. Агрегация иногда предпочтительнее, поскольку инкапсулирует части целого, скрывая их от внешних клиентов. В других случаях, наоборот, предпочтительнее связи, поскольку они позволяют ослабить взаимоотношения между объектами. Разумное проектное решение требует тщательного анализа этих двух факторов.

2.3 Классы

Понятия класса и объекта настолько тесно связаны, что невозможно говорить об объекте, не упоминая его класс. Однако между этими понятиями существуют важные различия.

В то время как объект обозначает конкретную сущность, определенную во времени и в пространстве, класс представляет собой лишь абстракцию существенных свойств объекта. Таким образом, можно говорить о классе «Робот», описывающем характеристики, общие для всех роботов.

Можно сформулировать следующее определение класса:

Класс – группа, множество или разновидность сущностей, обладающих общими свойствами или отличающихся от других уровнем качества, степенью компетентности или состоянием.

В контексте объектно-ориентированного анализа используется следующее определение класса:

Класс – это множество объектов, обладающих общей структурой, поведением и семантикой.

Отдельный объект является экземпляром класса. Объекты, не имеющие общей структуры и поведения, нельзя объединить в класс, поскольку по определению они ничем не связаны между собой.

Объект является конкретной сущностью, играющей некую роль в общей системе, класс описывает структуру и поведение, общую для всех родственных объектов. Таким образом, класс – это контракт, связывающий между собой абстракцию и ее клиентов. Реализуя эти решения в интерфейсе класса, языки программирования со строгим контролем типов позволяют выявить нарушения этого контракта во время компиляции.

Концепция контракта порождает различия между внешним и внутренним представлениями класса. Внешний вид класса реализуется интерфейсом, который подчеркивает абстрактный характер класса, скрывая структуру и особенности его поведения. Интерфейс, как правило, состоит из объявлений операций, применимых к его объектам, а также объявлений других классов, констант, переменных и исключительных ситуаций, необходимых для полного описания абстракции. Внутреннее представление обеспечивает реализация класса, скрывающая секретные особенности его поведения. Реализация состоит, в основном, из описания всех операций, объявленных в интерфейсе класса.

Интерфейс класса можно разделить на четыре части:

1. Открытый раздел (public) – объявления, доступные всем клиентам.

2. Защищенный раздел (protected) – объявления, доступные самому классу и его подклассам.

3. Закрытый раздел (private) – объявления, доступные только самому классу.

4. Пакет (package) – объявления, доступные только классам, входящим в один и тот же пакет.

Константы и переменные, образующие представление класса, называются по-разному. Например, в языке Smalltalk используется термин переменная экземпляра (instance variable), в языке Java – поле (field), а в языке C++, C# – данные-члены (data member).

Состояние объекта должно иметь определенное представление в соответствующем классе. Как правило, оно выражается с помощью объявлений констант и переменных, помещенных в защищенном или закрытом разделе интерфейса класса. Таким образом, представление, общее для всех экземпляров класса, инкапсулировано в нем, а его изменения не влияют на функциональные свойства всех внешних клиентов.

Разные языки программирования по-разному используют открытые, защищенные и закрытые разделы, а также пакеты, устанавливая разные права доступа к интерфейсу класса и осуществляя контроль над тем, что клиент может видеть, а что нет (т.е. реализуя концепцию видимости). Язык C++ позволяет программисту провести четкое различие между всеми четырьмя перечисленными уровнями доступа. Механизм дружественных функций и классов в языке C++ позволяет классу объявлять определенные классы привилегированными, открывая им доступ к своим защищенным и закрытым разделам. Концепция дружественных классов нарушает принцип инкапсуляции. В языке Java этого механизма нет. Вместо этого язык Java реализует аналогичный тип видимости, называемый пакетным доступом (package access). При этом все классы, входящие в один и тот же пакет, имеют доступ друг к другу. Помимо этого все остальные уровни доступа в языке Java (открытый, защищенный и закрытый) реализуются точно так же, как и в языке C++. В противоположность этому, язык Ada допускает лишь открытые и закрытые разделы, но не предусматривает защищенных. В языке

Smalltalk все переменные экземпляров являются закрытыми, а все методы остаются открытыми.

2.4 Отношения между классами

Рассмотрим пример анализа предметной области (на примере компьютерной игры в жанре фэнтези). Выделим классы: «Орк», «Эльф», «Гном», «Воин», «Маг», «Оружие», «Броня», «Дерево», «Замок». В данном наборе классов можно выявить следующие закономерности:

1. «Орк», «Эльф», «Гном» – игровые расы.

2. «Воин», «Маг» – игровые специализации.

3. «Оружие», «Броня» – характеристики всех игровых рас и специализаций (в принципе, они являются частью контейнеров).

4. «Дерево», «Замок» – прочие игровые классы, которые взаимодействуют с классами, отражающими активные сущности предметной области. «Дерево» может быть разрушено игровым персонажем. В «Замке» игровые персонажи могут находиться.

Из этого примера следует, что классы, как и объекты, не существуют изолированно. В конкретной проблемной области, как правило, ключевые абстракции взаимодействуют друг с другом, образуя структуру класса.

Отношения между классами устанавливаются по двум причинам. Во-первых, классы могут обладать общими свойствами. Например, «Орк», «Эльф», «Гном» – игровые расы. Во-вторых, между классами может существовать семантическая связь. Например, у классов «Маг» и «Воин» больше общего, чем у классов «Маг» и «Оружие».

Существует три основных типа отношений между классами:

1.Отношение «Обобщение/специализация» (generalization/ specialization), или «is a». Например, «Орк» – это игровая раса. То есть «Орк» являются специализированным подклассом более общего класса «Раса».

2. Отношение «целое/часть» (whole/ part), или «part of». Так, «Оружие» являются частью игрового персонажа.

3. Отношение ассоциации, т.е. семантическая зависимость между классами, которые больше никак не связаны друг с другом, например, между строениями в игре и игровыми персонажами. Очевидно, что определенные статические объекты могут быть разрушены игровыми персонажами, а также существуют строения, которые уникальны для данной игровой расы, оформленные в определенном дизайнерском стиле.

2.4.1 Ассоциация

Среди типов отношений между классами ассоциация – наиболее абстрактное и наиболее слабое отношение. Идентификация ассоциаций между классами часто проводится на этапе анализа и ранних стадиях проектирования, когда необходимо обнаружить общие зависимости между абстракциями. На последующих этапах проектирования и реализации системы эти слабые ассоциации часто уточняются и воплощаются в виде одного из более конкретных отношений между классами.

На рис. 2.6 показана связь между классами «Игровой персонаж» и «Оружие». Ассоциация лишь обозначает семантическую зависимость, но не указывает ее направления (по умолчанию подразумевается двухсторонняя связь) и не объясняет, как классы связаны друг с другом (их семантику можно только подразумевать, указав роли классов в этих отношениях). Однако на этапе анализа проблемы, когда требуется лишь выявить такие зависимости, этой семантики вполне достаточно. Создавая ассоциации, проектировщик фиксирует участников семантических отношений, а также их роли и количество. Естественно, на данном этапе не идет никакой речи о практической реализации данного отношения (конкретных структурах данных).

Рисунок 2.6 – Отношение ассоциации между классами.

На рис. 2.6 показана простая ассоциация (более точно это отношение описывается как агрегация). Ассоциация является двусторонней: экземпляр класса «Игровой персонаж» должен однозначно идентифицировать соответствующий объект «Оружие», и наоборот.

Это пример ассоциации «один-множество» (one-to-many). Каждый экземпляр «Игровой персонаж» может содержать несколько экземпляров класса «Оружие» или не содержать ни одного. В то же время каждый экземпляр «Оружие» связан только с одним экземпляром «Игровой персонаж». Это свойство называется множественностью ассоциации. Существуют следующие разновидности множественности:

1. Взаимно-однозначная зависимость (one-to-one).
2. Зависимость «один-множество» (one-to-many).
3. Зависимость «множество-множество» (many-to-many).

Взаимно-однозначное отношение обозначает очень узкую ассоциацию. Например, существует однозначное соответствие между классом «Оружие» и классом «Графическое представление» – каждому игровому предмету соответствует свое графическое представление. Отношение «множество-множество» распространено широко. Например, каждый игровой персонаж может быть связан с несколькими экземплярами «Замок». В то же время каждое строение может взаимодействовать с различнвми игровыми персонажами.

2.4.2 Наследование

С семантической точки зрения, наследование является наиболее интересным среди отношений типа «обобщение/специализация». Но одного наследования недостаточно для того, чтобы выразить все возможные

отношения, существующие между основными абстракциями в исследуемой предметной области. Альтернативой наследованию является делегирование (delegation), означающее, что объект поручает реализацию своего поведения другим объектам. Например, в примере с блоком управления роботами (рис. 2.5) каждый робот пересылает определенную информацию со своих датчиков. Характер этой информации и формат представления этих данных может существенно отличаться в зависимости от периодичности получения результатов измерений, назначения датчика, требований к получаемым показаниям со стороны пользователей. Изменив формат одного сообщения (например, о скорости полета робота) потребуется проверить и модифицировать функции для каждого робота. Оптимальным вариантом является выделение отельного класса «Сообщение», который будет реализовывать общие принципы форматирования данных, применимые ко всем типам параметров. Затем необходимо конкретизировать и расширить поведение классов сообщений. Для этого необходимо реализовать подклассы сообщений: «Сообщение о высоте», «Сообщение о скорости», и т.д.

Наследование – это отношение между классами, в котором один класс повторяет структуру и поведение одного (одиночное наследование) или нескольких (множественное наследование) классов.

Класс, структура и поведение которого наследуются, называется суперклассом (superclass). Класс, производный от суперкласса, называется подклассом (subclass). Подкласс обычно либо расширяет, либо ограничивает существующую структуру и поведение своего суперкласса. На рис. 1.4 представлена иерархия классов. Класс «Робот» является суперклассом иерархии. Совокупность всех классов, производных от одного суперкласса («Робот»), называется иерархией наследования. Путь от конкретного класса к его потомкам в иерархии называется цепочкой наследования.

Некоторые классы, представленные на рис. 1.4. Могут образовывать экземпляры, другие – нет. Классы, экземпляры которых не создаются,

называются абстрактными (abstract classes). Абстрактные классы создаются с расчетом на то, что подклассы дополнят их структуру и поведение с помощью конкретной реализации не полностью описанных методов.

Между наследованием и инкапсуляцией существуют серьезные противоречия. Использование механизма наследования в значительной мере открывает подклассу секреты реализации суперкласса. На практике это означает, что понять, как работает конкретный класс, можно лишь разобравшись, как устроены его суперклассы.

Такое поведение обеспечивается полиморфизмом, а операции называются полиморфными (polymorphic). Полиморфизм – это концепция теории типов, согласно которому одно и то же имя может обозначать экземпляры разных классов, связанных с общим суперклассом. Таким образом, любой объект с этим именем может по-разному выполнять общий набор операций. Используя полиморфизм, одну и ту же операцию можно по-разному реализовывать в классах, образующих иерархию. В результате подкласс может расширять возможности суперкласса или замещать базовые операции.

Код программы, не использующей полиморфизм, содержит множество операторов многовариантного выбора. Без полиморфизма невозможно создать иерархию классов для разных типов игровых персонажей или разных типов передаваемых сообщений. В этом случае проектировщику пришлось бы создавать единую вариантную запись, содержащую все свойства, связанные со всеми типами данных. Для того чтобы отличать один вариант от другого, пришлось бы проверять дескриптор, связанный с записью.

Наследование без полиморфизма возможно, но не эффективно.

Полиморфизм тесно связан с механизмом позднего связывания. При полиморфизме связь метода и имени определяется только в процессе выполнения программы. В языках C++, C# программист имеет возможность выбирать между ранним и поздним связыванием имени с членом-функцией.

В частности, если метод объявлен как виртуальный, то реализуется позднее связывание, а функция считается полиморфной. Если объявление виртуальной функции отсутствует, то используется раннее связывание и метод уточняется на этапе компиляции. В языке Java позднее связывание не требует использования ключевого слова virtual.

Кроме одиночного наследования, в некоторых языках используется механизм множественного наследования, который в рамках данного пособия не рассматривается.

2.4.3 Агрегация

Отношения целое/часть между экземплярами классов выражаются отношениями агрегации. Отношения агрегации между классами абсолютно аналогичны отношениям агрегации между объектами этих классов.

На рис. 2.7а показано отношение агрегации между классами «Робот», «Датчик высоты», «Датчик крена», «Датчик скорости».

а) б)

Рисунок 2.7 – Отношение агрегации: а – включение по значению; б – композиция.

Класс «Робот» демонстрирует агрегацию как включение по значению (containment by value). Это значит, что объекты класса «Датчик» не существуют отдельно от содержащего его экземпляра класса «Робот». Периоды существования этих двух объектов тесно связаны между собой.

Второй вид агрегации – композиция (composition) – является включением по ссылке (containment by reference). В этом случае (рис. 2.7б) класс «Игровой персонаж» по-прежнему означает целое, а экземпляр класса «Оружие» по-прежнему представляет собой его часть, но клиент имеет косвенный доступ к нему. Следовательно, продолжительности

существования этих объектов слабо связаны друг с другом. Экземпляры классов теперь можно создавать и уничтожать независимо.

Агрегация придает направленность отношению «целое/часть». Разумеется, агрегация не всегда означает физическое включение. Например, несмотря на то что игровой персонаж владеет оружием, оно не являются его физической частью. Кроме того, продолжительность существования каждого из этих объектов может быть совершенно разной, хотя концептуально отношение целого и части сохраняется.

Необходимый признак агрегации таков: классы находятся в отношении агрегации друг с другом, только если между любыми двумя их объектами существует отношение «целое/часть».

2.4.4 Зависимости между классами

Кроме наследования, агрегации и ассоциации между объектами существует другой тип отношений – зависимости (dependencies). Зависимость означает, что один из элементов каким-то образом зависит от другого. Изменение одного из элементов зависимости влияте на другой. При этом, существуют различные варианты зависимостей.

2.5 Взаимосвязь классов и объектов

Классы и объекты – это отдельные, хотя и тесно связанные понятия. В частности, каждый объект является экземпляром какого-либо класса, а любой класс может порождать произвольное число объектов или не порождать ни одного.

В большинстве практических приложений классы статичны, т.е. все их периоды существования, семантика и зависимости фиксируются до выполнения программы. Аналогично, класс любого созданного объекта

фиксируется заранее. В противоположность этому, объекты интенсивно создаются и уничтожаются в процессе выполнения программы.

На этапе разработки реалиных систем перед разработчиками возникают следующие задачи:

1. Идентификация классов, образующих словарь предметной области.

2. Изобретение структур, в которых объекты взаимодействуют друг с другом, обеспечивая поведение, соответствующее требованиям задачи.

Классы и объекты называются основными абстракциями (key abstraction), а взаимодействующие структуры – механизмами реализации (mechanisms of the implementation).

На ранних стадиях проектирования внимание разработчиков должно быть сосредоточено на внешних проявлениях основных абстракций и механизмов. Это представление образует логическую основу системы и должно учитывать структуру ее классов и объектов.

На последующих фазах проекта и на этапе реализации задача проектировщика изменяется и в центре внимания оказывается внутреннее представление основных абстракций и механизмов, включая их физическое воплощение. Принимаемые в процессе проектирования решения задают архитектуру системы: и архитектуру процессов, и архитектуру модулей.

Проектирование классов и объектов является итеративным процессом. За исключением самых простых задач сразу дать точное описание класса практически невозможно.

Для оценки качества классов и объектов, выделяемых в системе, можно предложить следующие пять показателей:

1. Связанность (coupling).

2. Связность (cohesion).

3. Достаточность (sufficiency).

4. Полнота (completeness).

5. Элементарность (primitiveness).

Термин связанность заимствован из теории структурного проектирования, но в более вольном толковании его можно использовать и в объектно-ориентированном проектировании. Сильная связанность усложняет систему, поскольку тесно связанные модули гораздо сложнее воспринимать и модифицировать. Сложность можно уменьшить, разрабатывая системы с как можно более слабыми связями между модулями.

Понятие связности также заимствовано из структурного проектирования. Связность – это глубина взаимодействия между элементами отдельного модуля (а в объектно-ориентированном проектировании – между элементами отдельного класса или объекта). Наименее желательной является случайная связность, когда в одном классе или модуле собираются несвязанные друг с другом абстракции.

Достаточность означает, что класс или модуль содержат достаточно много характеристик абстракции, чтобы обеспечить целенаправленное и эффективное взаимодействие. В противном случае компоненты системы становятся бессмысленными.

Полнота означает, что интерфейс класса или модуля описывает все существенные свойства абстракции. В то время как достаточность подразумевает минимальный интерфейс, полный интерфейс по определению охватывает все аспекты абстракции. Полным называется класс или модуль, интерфейс которого гарантирует все необходимые средства для взаимодействия с клиентами.

Многие высокоуровневые операции являются избыточными, поскольку их можно реализовать с помощью операций более низкого уровня. Элементарными называются операции, для эффективной реализации которых необходимо иметь доступ к внутреннему представлению абстракции.

Выбирая отношения, целесообразно пользоваться законом Деметера: методы класса не должны зависеть от структуры других классов, за исключением ближайшей структуры (верхнего уровня) самого класса.

Каждый метод должен посылать сообщения только объектам, принадлежащим предельно ограниченному множеству классов. Следование этому закону позволяет создавать слабо связанные классы, реализация которых инкапсулирована. Такие классы автономны, т.е. чтобы разобраться в их работе, нет необходимости знать строение других классов.

Контрольные вопросы

1. Что такое объект, Чем определяется состояние объекта?
2. Что такое операция класса? Какие виды операций вы знаете?
3. Охарактеризуйте различные типы отношений между объектами.
4. Приведите примеры отношения связи между объектами.
5. Приведите примеры отношения агрегации между объектами.
6. Чем классы отличаются от объектов? Что такое класс?
7. Назовите типы отношений между классами. Дайте их характеристику и укажите примеры каждого типа отношений.
8. Назовите показатели, которые используются для оценки качества объектной абстракции. Дайте характеристику каждого показателя.

РАЗДЕЛ 3. ОБЪЕКТНОЕ МОДЕЛИРОВАНИЕ

В рамках данной главы подробно рассмотрены только некоторые диаграммы, которые необходимо использовать в рамках данного курса. Наиболее часто будут использоваться диаграммы классов.

Проектирование – это не процесс создания диаграмм; диаграммы просто отражают результаты проектирования. При проектировании сложной системы важно рассмотреть ее с разных точек зрения – понятийной, логической и физической структуры, а также структурной и динамической семантики.

Язык UML содержит тринадцать видов диаграмм: диаграммы пакетов, диаграммы компонентов, диаграммы развертывания, диаграммы прецедентов использования, диаграммы деятельности, диаграммы классов, диаграммы последовательностей, диаграммы обзора взаимодействий, диаграммы композитных структур, диаграммы конечных автоматов, диаграммы синхронизации, диаграммы объектов и диаграммы коммуникации.

Диаграмма пакетов является инструментом организации артефактов проектирования для анализа предметной области и представления проектных решений. Диаграмма пакетов позволяет сосредоточить внимание на физической структуре визуальной модели или представлении модели с разных точек зрения.

Диаграмма компонентов показывает внутреннюю структуру компонентов и их зависимости от других компонентов. Эта диаграмма позволяет представить компоненты, сотрудничающие с помощью точно определенных интерфейсов с целью обеспечения функционирования системы.

Диаграмма развертывания показывает расположение артефактов в узлах физического проекта системы. Отдельная диаграмма развертывания отражает структуру артефактов системы. На этапе проектирования

диаграммы развертывания используются для указания физической совокупности узлов, являющихся платформой для реализации системы.

Диаграмма прецедентов использования отражает контекст, в котором создается система и обеспечивается ее функционирование. Они представляют элементы, взаимодействующие в системе, и показывают, что внешняя среда ожидает от системы.

Диаграмма деятельности позволяет изобразить потоки операций в рамках системных, производственных, технологических и других процессов. Эти диаграммы сосредоточивают внимание на выполняемых операциях и распределении ответственности за их выполнение.

Диаграмма классов показывает существующие классы и связи между ними в логической структуре системы. На этапе анализа диаграмма классов отражает общие роли и обязанности сущностей, реализующих функциональные свойства системы. На этапе проектирования диаграмма классов отражает структуру классов, образующих архитектуру системы.

Диаграмма последовательностей позволяет отследить выполнение сценария в том же контексте, что и диаграмма объектов. В большой степени диаграмма последовательностей просто является еще одним способом представления диаграммы объектов.

Следование принципам SOLID выводит объектно-ориентированное программирование на новый качественный уровень.

3.1 Инструментальные средства моделирования

Перед непосредственным программированием системы (кодированием) с использованием языка высокого уровня необходимо спроектировать объектную систему.

Проектировщик может даже не устанавливать на компьютере средства разработки приложений и компиляторы, но должен обладать средствами,

позволяющими спроектировать объектную систему. Модель передается программисту (соответственно основные компоненты модели должны быть понятны разработчику); программист на основе спроектированной объектной модели осуществляет разработку программного продукта.

В связи с описанным упрощенным процессом создания программного продукта возникает вопрос: каковы технические и методологические средства разработки объектных моделей? Для этих целей используется язык UML (Unified Modeling Language). UML – это язык моделирования, используемый для анализа, описания и разработки систем программного обеспечения.

Несмотря на то, что язык UML является самостоятельной технологией, в рамках данной дисциплины учить его не нужно (данная технология требует отдельного рассмотрения). Инструментарий UML необходим для понимания материала данного пособия, так как позволяет заменить словесное описание приложений более эффективным инструментом представления объектно-ориентированных систем.

Как и во многих других научных дисциплинах, имеющих собственные системы обозначений для описания создаваемых продуктов, язык UML используется в объектно-ориентированном проектировании для моделирования (т.е. представления) разрабатываемой системы. В целом, построенная модель UML с определенной точностью описывает реальный программный продукт, который будет создан в результате реализации проекта.

3.2 Типология диаграмм

Существует большое количество различных типов диаграмм, но в рамках данного курса интересуют те типы, использование которых принесет

существенную помощь программисту при разработке программных систем на языках высокого уровня.

Диаграмма – в некотором смысле одна из проекции системы. Как правило, за исключением наиболее тривиальных случаев, диаграммы дают свернутое представление элементов, из которых составлена система. Один и тот же элемент может присутствовать во всех диаграммах, или только в нескольких (самый распространенный вариант), или не присутствовать ни в одной (очень редко). Теоретически диаграммы могут содержать любые комбинации сущностей и отношений. На практике, однако, применяется сравнительно небольшое количество типовых комбинаций, соответствующих пяти наиболее употребительным видам.

Диаграмма в UML – это графическое представление набора элементов, изображаемое чаще всего в виде связанного графа с вершинами (сущностями) и ребрами (отношениями).

Диаграммы делят на две группы: структурные диаграммы и диаграммы поведения.

Структурные диаграммы используются для демонстрации статической структуры элементов в системе. Они могут изображать архитектурную организацию системы, ее физические элементы, текущую конфигурацию, а также специфические элементы предметной области.

Структурные диаграммы:

1. Диаграммы пакетов (package diagram).

2. Диаграммы классов (class diagram). На диаграмме классов показывают классы, интерфейсы, объекты и кооперации, а также их отношения. При моделировании объектно-ориентированных систем этот тип диаграмм используют чаще всего. Диаграммы классов соответствуют статическому виду системы с точки зрения проектирования. Диаграммы классов, которые включают активные классы, соответствуют статическому виду системы с точки зрения процессов.

3. Диаграммы компонентов (component diagram). На диаграмме компонентов представлена организация совокупности компонентов и существующие между ними зависимости. Диаграммы компонентов относятся к статическому виду системы с точки зрения реализации. Они могут быть соотнесены с диаграммами классов, так как компонент обычно отображается на один или несколько классов, интерфейсов или кооперацій.

4. Диаграммы развертывания (deployment diagram). На диаграмме развертывания представлена конфигурация обрабатывающих узлов системы и размещенных в них компонентов. Диаграммы развертывания относятся к статическому виду архитектуры системы с точки зрения развертывания. Они связаны с диаграммами компонентов, поскольку в узле обычно размещаются один или несколько компонентов.

5. Диаграммы объектов (object diagram). На диаграмме объектов представлены объекты и отношения между ними. Они являются статическими «фотографиями» экземпляров сущностей, показанных на диаграммах классов. Диаграммы объектов, как и диаграммы классов, относятся к статическому виду системы с точки зрения проектирования или процессов, но с расчетом на настоящую или макетную реализацию.

6. Диаграммы композитных структур (composite structure diagram).

Структурные диаграммы часто используются в сочетании с диаграммами поведения для описания определенных аспектов системы. Каждый класс может иметь соответствующую диаграмму конечных автоматов (state machine diagram), описывающую поведение его экземпляров. Аналогично, совместно с диаграммами объектов, представляющих определенный сценарий, диаграммы взаимодействия (interaction diagram) демонстрируют моменты времени или порядок передачи сообщений по мере их вычисления. В объектно-ориентированном проектировании семантика динамического поведения и способы ее реализации описываются с помощью диаграмм поведения:

1. Диаграмма прецедентов использования (use case diagram). На диаграмме прецедентов представлены прецеденты и актеры (частный случай классов), а также отношения между ними. Диаграммы прецедентов относятся к статическому виду системы с точки зрения прецедентов использования. Они особенно важны при организации и моделировании поведения системы.

2. Диаграмма деятельности (activity diagram). Диаграмма действий (деятельности) – это частный случай диаграммы состояний; на ней представлены переходы потока управления от одной деятельности к другой внутри системы. Диаграммы деятельности относятся к динамическому виду системы; они наиболее важны при моделировании ее функционирования и отражают поток управления между объектами.

3. Диаграмма конечных автоматов (state machine diagram);

4. Диаграммы взаимодействий (interaction diagram):

5. Диаграмма последовательностей (sequence diagram). **Диаграммы последовательностей** и **кооперации** являются частными случаями диаграмм взаимодействия. На диаграммах взаимодействия представлены связи между объектами; показаны, в частности, сообщения, которыми объекты могут обмениваться. Диаграммы взаимодействия относятся к динамическому виду системы. При этом диаграммы последовательности отражают временную упорядоченность сообщений, а диаграммы кооперации – структурную организацию обменивающихся сообщениями объектов. Эти диаграммы являются изоморфными, то есть могут быть преобразованы друг в друга.

6. Диаграмма коммуникации (communication diagram).

7. Диаграмма обзора взаимодействий (interaction overview diagram).

8. Диаграмма синхронизации (timing diagram).

3.3 Диаграмма вариантов использования

Опыт показывает, что одной из основных причин неудач коммерческих программных проектов является недостаточное взаимопонимание между его основными участниками или полное отсутствие контактов между ними. Это особенно важно в ситуациях, когда проектировщики и заказчики работают в разных областях знаний. Бизнесмены могут хорошо знать определенную задачу, которую необходимо решить, но, тем не менее, способны дать организаторам проекта лишь самое общее описание своих пожеланий, уточнив некоторые детали.

Проектировщики получают подробное задание, но не имеют ни малейшего представления о целях бизнеса, т.е. для чего они разрабатывают новую систему.

Основное противоречие этого процесса: заказчики, постановщики задач, разработчики, специалисты по тестированию – все участники процесса используют различные терминологические словари и обладают глубокими знаниями в различных областях.

Для преодоления данного противоречия предназначены диаграммы вариантов использования (use case diagram). Другое название – диаграмма прецедентов. Они изображают элементы системы, взаимодействующие друг с другом.

Действующие лица (actors) – это сущности, контактирующие с системой. Действующими лицами могут быть как люди, так и другие системы. Они являются внешними сущностями по отношению к системе и изображаются в виде стилизованных фигурок.

Действующие лица описываются с помощью их ролей. В реальной предметной области люди (и системы) могут исполнять несколько ролей. Например, один и тот же человек может быть менеджером, покупателем, пользователем информационной системы, разработчиком и т.д.

Прецеденты использования демонстрируют, что именно действующие лица хотят от системы. Некоторые прецеденты использования, связанные с системой управления роботами, изображены на рис. 3.1.

Рисунок 3.1 – Прецеденты диаграммы вариантов использования.

Прецеденты использования – это не просто возможности, предоставляемые системой. Прецедент использования должен быть полным потокам целесообразных поступков действующего лица.

Прецедент использования – это особая последовательность связанных между собой операций, выполняемых действующим лицом и системой в диалоге. Каждый прецедент использования представляет собой полный перечень событий, происходящих в системе, с точки зрения пользователя.

Описать систему с использованием диаграммы прецедентов – это значит создать диаграмму прецедентов использования, соединив действующих лиц и соответствующие прецеденты с помощью основных ассоциаций, изображаемых линиями. На рис. 3.2 представлена диаграмма, описывающая систему управления роботами.

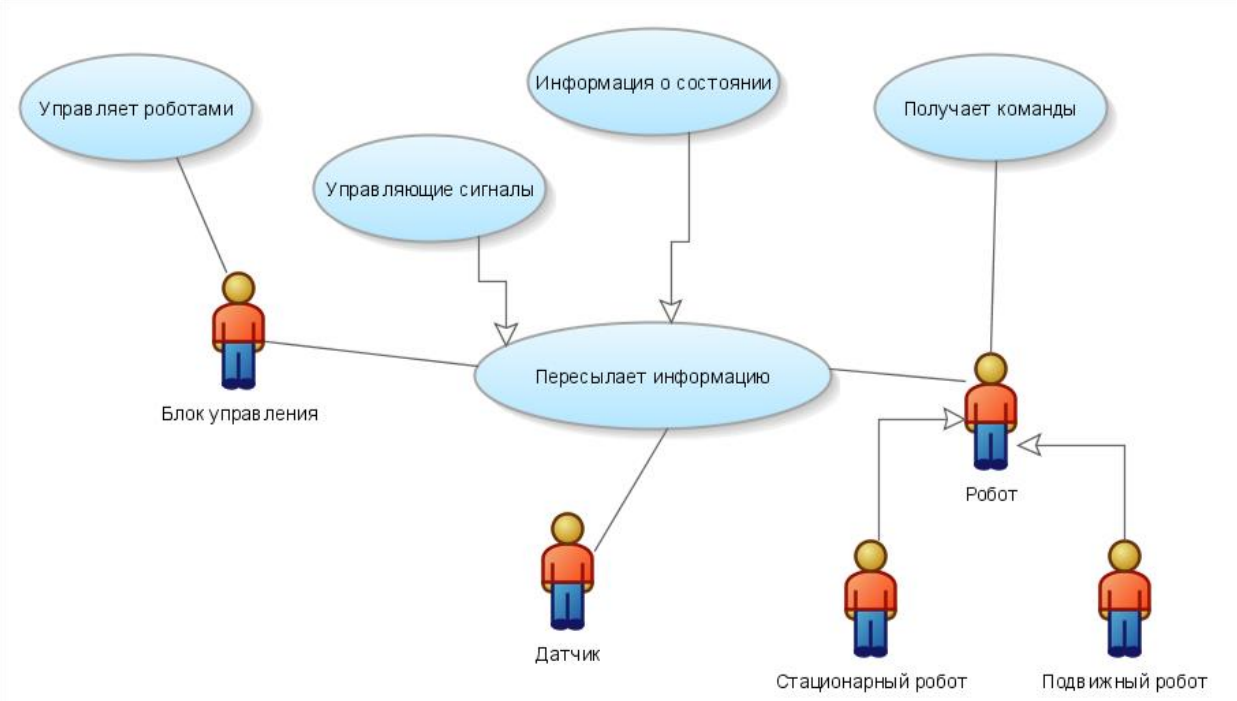


Рисунок 3.2 Диаграмма вариантов использования (реализована средствами Oracle JDeveloper Studio).

На данной диаграмме присутствует вид связи – обобщение (generalization), которая позволяет описать отношение типа «общее/частное» как между прецедентами, так и между действующими лицами.

Описание системы с использованием диаграммы прецедентов – начальный шаг, который применяется в объектном моделировании.

3.4 Диаграмма классов

Рассмотрим вид диаграмм, который используется в ООП наиболее часто – диаграмма классов. Диаграммы данного типа уже фигурировали в пособии – при описании объектных моделей невозможно было лаконично отразить семантику предметной области без их применения. (рис. 1.4) Диаграмма классов показывает классы и их отношения в логической схеме проекта. Отдельная диаграмма классов представляет определенный аспект структуры системы. На стадии анализа диаграммы классов используются для того, чтобы указать общие роли и обязанности сущностей, обеспечивающих

функционирование системы. На стадии проектирования диаграммы классов позволяют представить архитектуру системы.

На рис. 3.3а представлена диаграмма классов, построенная с использованием инструментария интегрированной среды разработки JDeveloper Studio. На рис. 3.3б представлена аналогичная иерархия классов, выполненная в среде MS VisualStudio 2013. Визуально, диаграммы, выполненные с использованием различных инструментов, отличаются, но отношения между классами в современных программах проектирования отображаются одинаково в рамках нотаций UML.

Каждый класс должен иметь уникальное имя в соответствующем пространстве имен. По общепринятому соглашению имя начинается с прописной буквы, а пробелы между именами пропускаются.

В нотации UML первая буква атрибута и операции является строчной, а последующие слова начинаются с прописных букв. Выбор имен операций зависит от правил, принятых в конкретном языке программирования.

Формат спецификации атрибута:

видимость имяАтрибута : Тип [кратность] =

ЗначениеПоУмолчанию {строка свойств}

Фримат спецификации операции:

видимость имяОперации (имяПараметра : Тип) =

ТипВозвращаемогоЗначения{строка свойств}

Существуют различные инструменты для визуализации классов и отношений между ними, которые отходят от данной нотации (рис. 3.3).

В некоторых диаграммах классов полезно указывать лишь некоторые атрибуты и операции. Это объясняется тем, что, за исключением простейших случаев, перечислять все члены класса в диаграмме нецелесообразно. Таким образом, представленные атрибуты и операции лишь конспективно представляют спецификацию класса, который является единственным

местом объявления всех своих членов. Если необходимо показать больше членов класса, следует увеличить размер его пиктограммы.

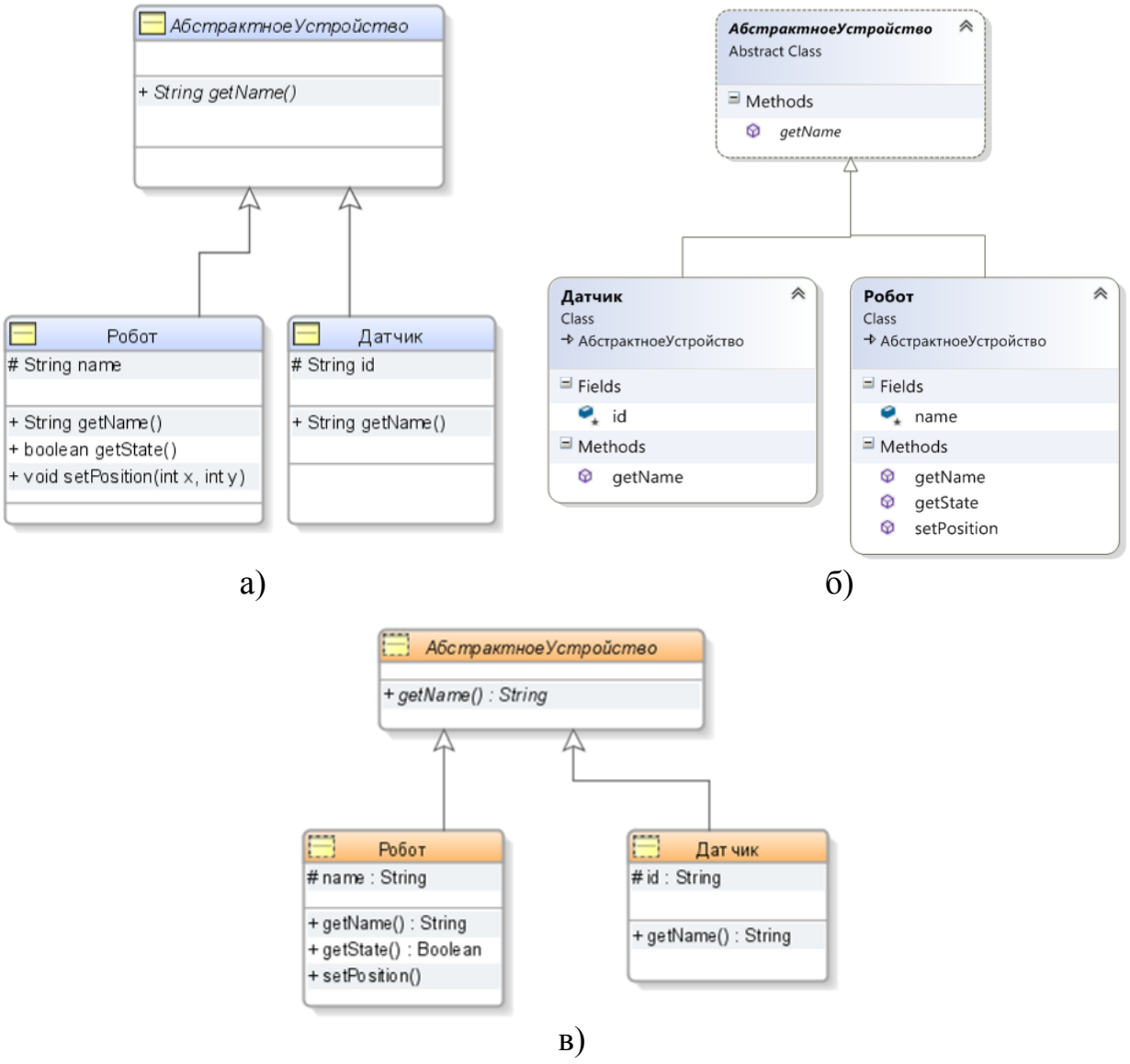


Рисунок 3.3 – Диаграмма классов:
а – реализация в среде Oracle JDeveloper 12c; б – реализация в среде разработки Microsoft Visual Studio 2013; в – диаграмма классов в нотации UML (не привязывается к конкретному языку).

Если проектировщик решил не показывать ни один член класса, можно указать лишь имя класса. Абстрактные классы на диаграммах выделяются курсивом.

Классы редко бывают изолированными. Они взаимодействуют друг с другом, используя разные методы. К основным отношениям между классами

относятся ассоциация (association), обобщение (generalization), агрегация (aggregation) и объединение (composition).

На рис. 3.4 представлены диаграммы классов, демонстрирующие различные отношения.

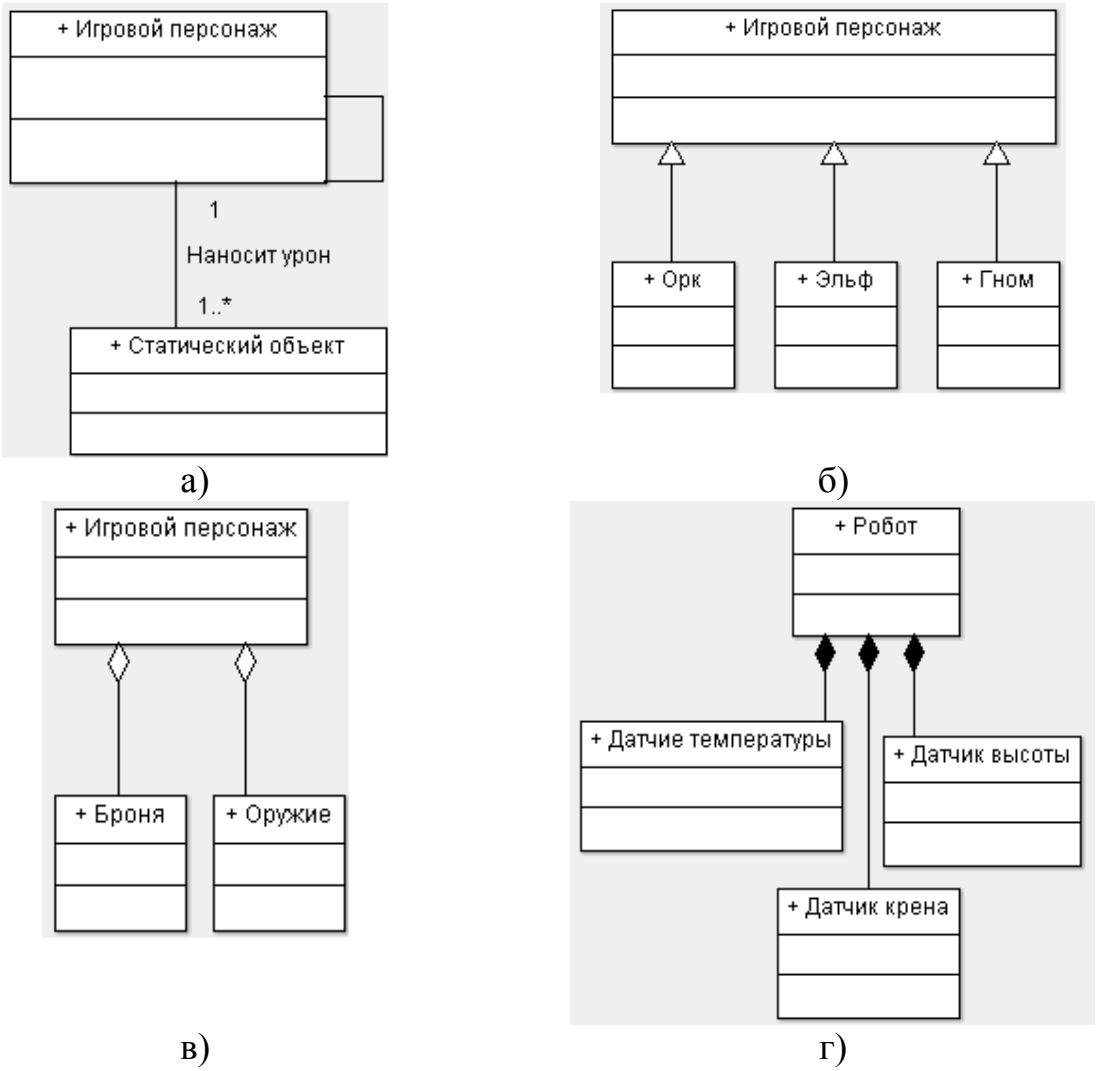


Рисунок 3.4 – Отношения между классами: а – ассоциация; б – отношение «обобщение/специализация»; в – агрегация без физического включения; г – агрегация с физическим включением.

Каждое отношение сопровождается текстуальной меткой, в которой указывается имя отношения или его предназначение.

Пиктограмма ассоциации соединяет два класса и означает семантическую связь между ними. Ассоциации часто обозначаются именными группами, например «Наносит урон», описывающими природу связи (см. рис. 3.4а). Класс может иметь ассоциацию с самим собой (так

называемая рефлексивная ассоциация (reflexive association)). Одна пара классов может иметь несколько ассоциативных связей. Возле пиктограммы ассоциации можно указать ее кратность. Синтаксис кратности:

– одна связь: 1;

– неограниченное количество связей: *;

– ноль или более: 0..*;

– один или более: 1..*;

– ноль или один: 0..1;

– конкретный диапазон: 3..7.

Обозначение кратности пишется возле целевого полюса ассоциации (target end). Оно указывает количество связей между каждым экземпляром класса с экземплярами целевого класса. Если кратность отношения явно не указана, то следует считать, что она не определена.

Обозначения оставшихся трех типов связи уточняют пиктограмму ассоциации. Это позволяет уточнять отношения на этапе разработки проекта. Сначала констатируется существование семантической связи между двумя классами, а затем, после принятия тактических решений об истинной природе этих отношений, эта связь уточняется как обобщение, агрегация или объединение.

Пиктограмма обобщения означает отношение обобщения/специализации (отношение «является») и изображается в виде закрытой стрелки (рис. 3.4б). Конец стрелки указывает на суперкласс, а ее противоположный конец обозначает подкласс.

Агрегация, будучи отношением «является частью», является частным вариантом отношения ассоциации. Пиктограмма агрегации обозначает иерархию «целое/часть» и подразумевает возможность перемещаться от объединения к его частям. Агрегация изображается в виде линии с незакрашенным ромбом на конце (рис. 3.4в), указывающим на агрегат (целое). Класс на другом конце стрелки – это класс, экземпляры которого

являются частями объекта агрегата. Допускаются рефлексивная и циклическая агрегации. Кратность * (т.е. нуль или больше), указанная возле линии, изображающей отношение агрегации, подчеркивает отсутствие физического включения.

Выбор отношения агрегации обычно осуществляется на этапе анализа или архитектурного проектирования. Выбор отношения объединения (физического включения) представляет собой пример тактического решения. Отношение включения имеет большое значение, поскольку оно играет важную роль в процессе конструирования и уничтожения частей агрегата. Пиктограмма объединения изображается в виде линии с закрашенным ромбом на конце (рис. 3.4г), соединенном с агрегатом. Кратность этой линии равна 1, поскольку часть агрегата не имеет смысла за его пределами. Таким образом, продолжительность существования части совпадает с продолжительностью существования целого.

Рассмотренные пиктограммы являются основными элементами диаграмм классов. В совокупности они позволяют разработчику достаточно полно описать структуру классов системы.

Многие стратегические и тактические решения требуют расширения системы обозначений. Как правило, разработчики стараются придерживаться основных обозначений и применять лишь те дополнительные понятия, без которых нельзя обойтись при анализе и проектировании деталей, играющих важную роль в визуализации или анализе системы.

Некоторые языки программирования, например, C++, C#, Java предусматривают шаблонные (параметризованные) классы. Шаблонный класс (template class) – это семейство классов, структура и функциональные свойства которых определяются формальными параметрами независимо друг от друга. Для того чтобы создать конкретный класс из этого семейства, формальные параметры должны быть заменены фактическими (процесс связывания). Конкретные классы могут иметь экземпляры.

Шаблонные классы сильно отличаются от обычных классов. Это является основанием для того, чтобы использовать особые обозначения в рамках диаграмм. Рассмотрение шаблонных классов в рамках диаграмм выходит за рамки данного пособия.

Видимость можно задать, указав на диаграмме один из следующих символов:

1) открытая видимость (+) – доступен для всех элементов в области видимости класса;

2) защищенная видимость (#) – доступен только для членов класса и подклассов;

3) закрытая видимость (-) – доступен только для членов класса;

4) пакетная видимость (~) – доступен только для элементов пакета.

На диаграммах классов присутствуют м другие обозначения, позволяющие отразить следующие аспекты объектных моделей: имена полюсов ассоциации; квалификаторы; ограничения; классы ассоциаций и примечания. Но в рамках данного курса необходимо уделить внимание именно программированию объектно-ориентированных приложений, а не проектированию объектных систем и их анализу. Поэтому рассмотренного инструментария диаграмм классов достаточно для дальнейшего усвоения материала.

3.5 Диаграмма деятельности

В подразделе 3.3 описана спецификация прецедента использования в виде текста. Текстовое описание прецедента имеет ряд преимуществ: их легко создавать и изменять Но сложные прецеденты использования, деловые операции и алгоритмы иногда трудно описать в виде текста. Сложные потоки информации намного удобнее представлять с помощью визуальных средств. Визуальное представление процессов иногда позволяет выявить

потенциальные проблемы, которые невозможно обнаружить в текстуальном описании прецедента использования.

Диаграммы деятельности позволяют визуально изобразить поток операций в системном, производственном или технологическом процессе. Эти диаграммы концентрируют внимание на том, какие действия выполняются и кто (или что) несет ответственность за их выполнение.

Элементами диаграмм деятельности являются узлы действий (action node), узлы управления (control nodes) и узлы объектов (object nodes). Узлы управления разделяются на три категории: начальные и конечные (причем существует два варианта конечных узлов: конец действия или конец потока), принятия решений и слияния (decision and merge), а также ветвления и объединения (fork and join)

Действие (action) – это элементарный компонент поведения, представленный в диаграмме деятельности. Деятельность может состоять из многих действий, которые изображаются на соответствующих диаграммах.

На рис. 3.5 представлена диаграмма деятельности, на которой показано действие «Послать показания высоты».

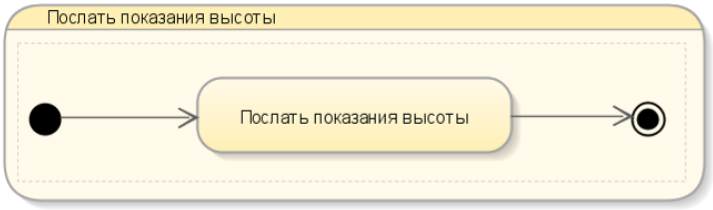


Рисунок 3.5 – Простейшее действие.

Поскольку диаграмма деятельности демонстрирует последовательность технологических операций, она должна иметь начало и конец. Стартовая точка (начальный узел) диаграммы деятельности изображается в виде черного кружка, а точка остановки (конечный узел процесса) – в виде зружка с меткой.

Кроме конечного узла действий, в диаграммах существует конечный узел потока (flow final node), обозначаемый вложенным символом «X».

Конечный узел потока используется для остановки отдельного потока без остановки всего процесса (действия).

Узлы принятия решений и слияний управляют потоком в диаграмме деятельности. Каждый такой узел изображается в виде ромба с входящими и исходящими стрелками.

Узел принятия решений имеет один входящий поток и несколько исходящих. Он используется для направления входящего потока по одному (и только одному) каналу, исходящему их этого узла. Входящие потоки обычно сопровождаются предохранительными условиями, определяющими выбор их дальнейшего направления. На рис. 3.6 показано предохранительное условие [есть связь с блоком управления] и альтернатива [блок управления недоступен] для действия «Передача данных блоку управления». В узле принятия решений не предусмотрены ни задержки, ни синхронизация потоков.

Рисунок 3.6 – Использование предохранительных условий.

Обратите внимание на символ в правом нижнем углу пиктограммы действия. Он означает, что данная операция относится к типу callBehavior, который является одним из предопределенных типов операций в языке UML 2 и представляет собой элементарные действия, моделирующие манипуляции с объектами и связи между ними, а также вычисления и обмены сообщениями между объектами. В данном случае это означает, что узел «Передача данных блоку управления» (с трезубцем) создан на основе действия «Послать показания высоты» (рис. 3.5).

На рис. 3.7 представлена диаграмма действий, содержащая большее количество различных элементов.

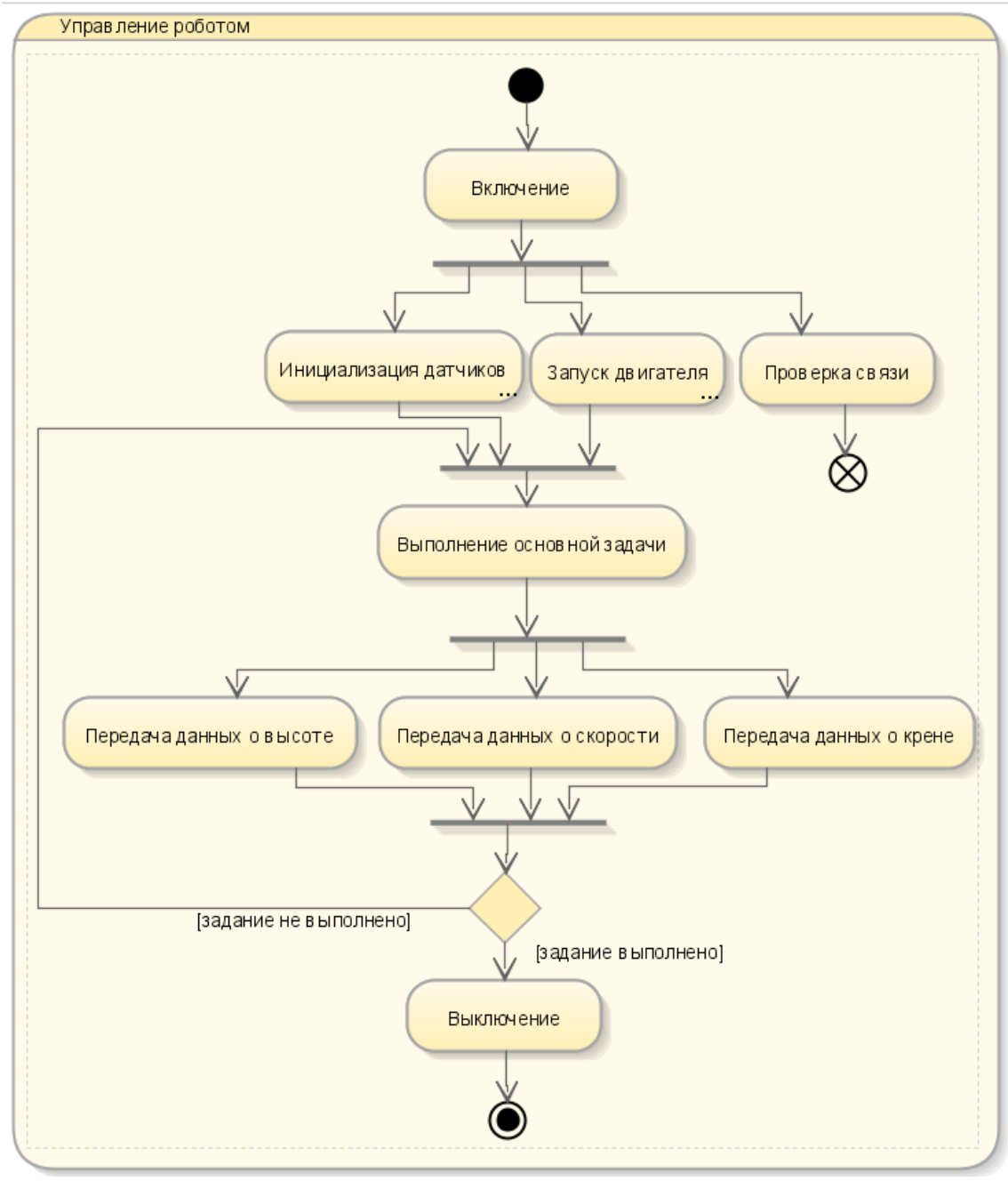


Рисунок 3.7 – Диаграмма деятельности, описывающая процесс управления роботом.

На рисунке присутствуют узлы разветвления (fork node) и объединения (join node), которые аналогичны узлам принятия решения (decision node) и слияния (merge node) соответственно.

Основным отличием между ними является параллельность. Как и узел принятия решений, узел разветвления имеет один входящий поток и несколько исходящие Разница заключается в том, что в узле принятия решений происходит выбор только одного исходящего потока, а в узле разветвления один поток расщепляется на несколько.

Таким образом, с использованием диаграмм активности можно детализировать операции объектов в графическом виде с учетом асинхронного характера их выполнения.

3.6 Диаграммы последовательностей

Диаграммы последовательностей (sequence diagram) используются для отслеживания хода выполнения сценария в том же контексте, что и диаграммы взаимодействий. Рассмотрение диаграмм взаимодействий выходит за рамки данного пособия.

Преимущество диаграммы последовательностей заключается в простоте отслеживания сообщений, передаваемых последовательно. Диаграммы последовательностей часто оказываются удобнее, чем диаграммы объектов (которые будут рассмотрены далее), и позволяют выразить семантику сценария на ранних стадиях проектирования, еще до создания протоколов отдельных классов. Кроме того, диаграммы последовательностей уже на ранних этапах проектирования позволяют очертить контуры разрабатываемой системы.

В диаграмме последовательностей основные сущности изображаются горизонтально вдоль верхней границы. Из пиктограммы каждого объекта выходит вертикальная пунктирная линия, называемая «линией жизни» (lifeline). Эта линия показывает пределы существования объекта.

Сообщения (которые часто означают события или инициирование операции) изображаются в виде горизонтальных линий. Концы этих линий

расположены на вертикальных линиях, выходящих из пиктограмм, соответствующих объектам на верхней границе диаграммы. Сообщения направляются от отправителя к адресату. Порядок сообщений указывается в вертикальном направлении: первое сообщение расположено выше остальных, а последнее – ниже. В результате нумерация сообщений становится излишней. Тип сообщения отражается с помощью разнообразных линий и стрелок, представленных на рис. 3.8.

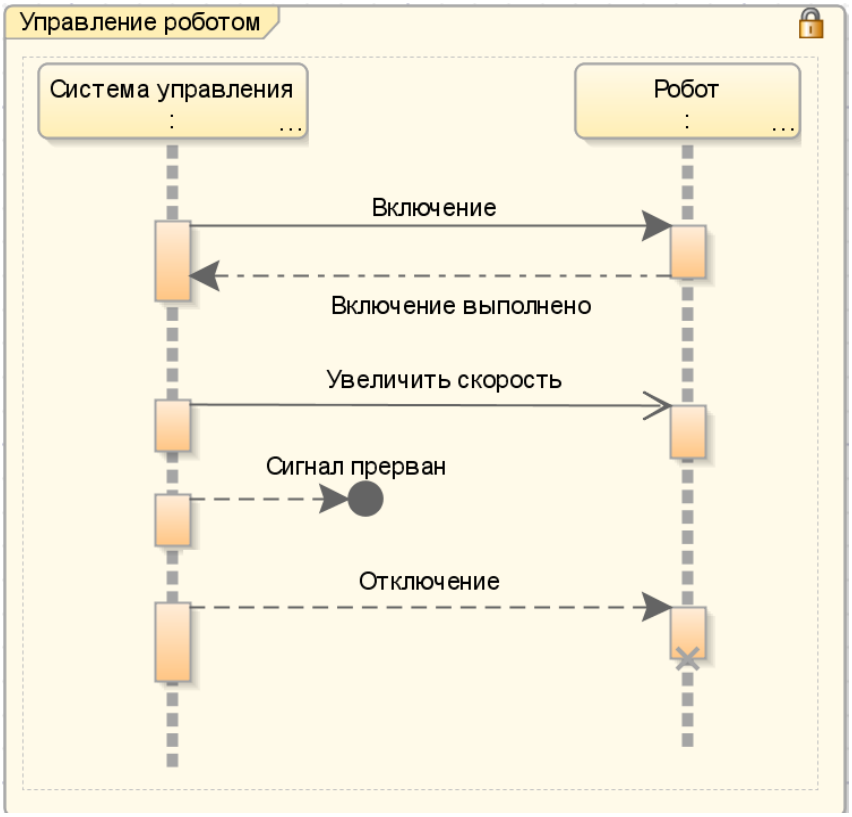


Рисунок 3.8 – Диаграмма последовательности.

Из представленной диаграммы становится очевидной последовательность передачи сообщений. Синхронное сообщение (synchronous message), как правило, представляющее собой вызов операции, изображается в виде сплошной линии с закрытой стрелкой. Асинхронное сообщение (asynchronous message) изображается в виде сплошной линии с открытой стрелкой. Потерянное сообщение (lost message), т.е. сообщение, не дошедшее до адресата, представляет собой синхронное сообщение, линия которого упирается в конечную точку (закрашенный кружок). Найденное

сообщение (found message) – сообщение с неизвестным отправителем – представляется как синхронное сообщение, исходящее из конечной точки.

Уничтожение (destruction event) означает, что объект разрушен. Этот факт изображается символом X, расположенным к конце «линии жизни».

3.7 Проектирование программных систем

Проектирование системы не заканчивается на получении готовых диаграмм. Сколько бы ни получилось диаграмм (и каких бы они ни были типов) этого недостаточно для получения гарантированного высококачественного программного продукта.

Диаграммы UML не существуют отдельно от кода; например интегрированные среды разработки приложений позволяют осуществлять рефакторинг кода и проектирование классов с использование визуальных средств построения диаграмм. Набор диаграмм описывает лишь отдельные части проекта, а не весь проект целиком. Окончательное воплощение проекта – исходный код.

Некорректное понимание сущности проектирования, анализа и разработки может привести к проекту, который обладает существенными недостатками:

1. Закрепощенность. Система практически не предполагает внесение изменений. Любое изменение приводит к большому объему работы по переписыванию существующего кода. Возникает эффект «снежного кома».

2. Неустойчивость. В результате незначительных изменений компонента в системе появляются ошибки, которые не имеют прямого отношения к модифицируемому блоку.

3. Неподвижность. Невозможность разделения системы на блоки, допускающие повторное использование.

4. Вязкость. Ошибки, накопленные на предыдущих этапах проектирования и разработки, приводят к ситуации, когда дальнейшая реализация ошибочных действий по проектированию и разработке является наиболее легкой стратегией.

5. Неоправданная сложность. В проекте используются технологии, которые не гарантируют выгоды, но предполагают увеличение слжности.

6. Неоправданное дублирование. Проект содержит повторяющиеся структуры, которых можно избежать за счет применения простой абстракции.

7. Непредсказуемость. Сложность понимания назначения функций и методов, классов и интерфейсов, модулей.

Накопленный опыт практической разработки сложных систем позволяет сформулировать ряд принципов, следование которым позволяет избежать возникновения вышеуказанных проблем:

1. Принцип персональной ответственности (SRP, Single Responsibility Principle).

2. Принцип закрытости-открытости (OCP, Open-Closed Principle).

3. Принцип подстановки Лискоу (LSP, Liskov Substitution Principle).

4. Принцип отделения интерфейса (ISP, Interface Segregation Principle).

5. Принцип инверсии зависимостей (DIP, Dependency Injection Principle).

Данные принципы иногда объединяют в систему принципов SOLID (или S.O.L.I.D.).

3.7.1 Принцип персональной ответственности (SRP)

Упрощенно принцип SRP можно выразить так: существует единственная причина, приводящая к изменению класса. Этот принцип часто перефразируют: каждый объект должен иметь одну обязанность и эта обязанность должна быть полностью инкапсулирована в классе. Все методы

класса должны быть направлены исключительно на обеспечение этой обязанности.

Рассмотрим пример: на рис. 3.9 представлен класс «Сотрудник», внутри которого инкапсулированы поля и методы, определяющие роль данного класса.

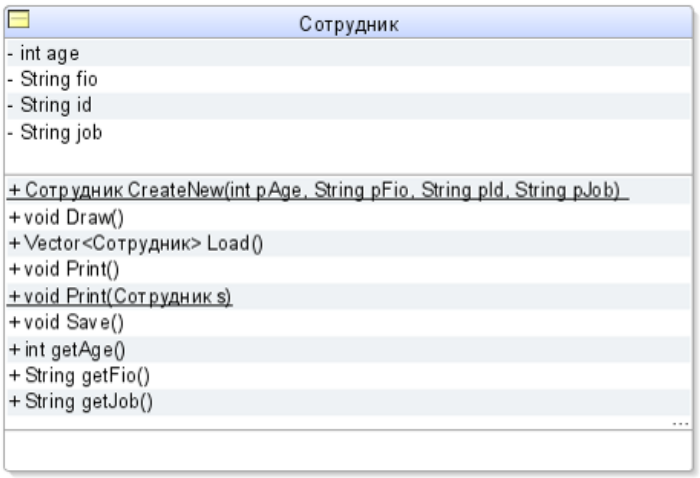


Рисунок 3.9 – Класс, представляющий сущность «Сотрудник».

Из представленной диаграммы класса видно, что однозначно определить назначение данного класса не представляется возможным. Модно выделить следующие функциональные области, которые реализуют методы класса:

1. Обслуживание сущности «Сотрудник». Поддержание семантики сущности:

– закрытые поля класса;

– метод public static Сотрудник CreateNew(int pAge, String pFio, String pId, String pJob);

– метод public int getAge();

– метод public String getFio();

– метод public String getJob().

2. Интерфейсная часть класса, предназначенная для сериализации объектов типа «Сотрудник»:

– метод public Vector<Сотрудник> Load();

– метод public void Save().

3. Интерфейсная часть класса, предназначенная для визуализации сущности, печати, рисования:

– метод public void Draw();

– метод public void Print();

– public static void Print(Сотрудник s).

Из представленного примера очевидны следующие выводы:

1. Класс, рассмотренный на рис. 3.9 достаточно компактный, содержит разумное количество методов, но при его проектировании были допущены ошибки, которые повлекут проблемы на этапе программирования и сопровождения.

2. В классе присутствуют методы, реализующие следующие роли: поддержание семантики сущности «Сотрудник»; интерфейс для сохранения/восстановления (сериализации) объектов; интерфейс для визуализации сущности различными способами.

В соответствии с принципом единственной обязанности (персональной ответственности) данный класс должен быть модифицирован.

Второй распространенный пример нарушения принципа единственной обязанности – валидация данных в классе, ответственном за хранение данных. В подобном случае необходимо реализовать отдельный класс валидатора, который будет использовать класс хранения данных; при этом реализация класса с данным инее должна зависеть от методов валидации.

3.7.2. Принцип закрытости-открытости (OCP)

Программные сущности (классы, модули, функции) должны быть открыты для расширения, но в то же время закрыты для модификации.

Если единственное незначительное изменение в программе приводит к каскаду изменений в других классах, модулях, то используя принцип

закрытия-открытия необходимо выполнить рефакторинг системы, чтобы дальнейшие изменения не вызывали кардинальной переработки кода.

Модули, соответствующие принципу открытости-закрытости обладают двумя признаками:

1. «Открыт для расширения». То есть поведение модуля может быть расширено. В результате развития системы, может понадобиться добавить определенный функционал для модкля за счет инкапсуляции новых типов поведения.

2. «Закрыт для модификации». В результате расширения поведения модуля изменения в двоичном коде модуля не производятся. Исполняемая версия модуля (например *.jar для Java или *.dll для C#) остается неизменной.

Рассмотрим пример. На рис. 3.10 представлена диаграмма классов: класс «Client» использует класс «Server». Оба класса определены в различных модулях. Класс Client использует класс Server через метод и через закрытое поле. Если клиенту потребуется использовать другой объект сервера, то модуль клиента придется перекомпилировать.

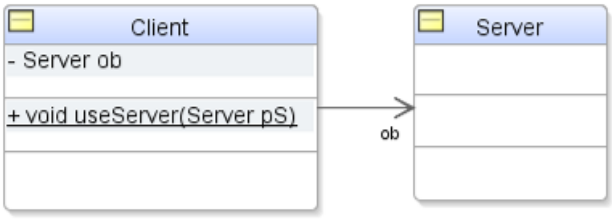


Рисунок 3.10 – Класс, представляющий сущность «Сотрудник».

Решение представлено на рис. 3.11.

Рисунок 3.11 – Класс, представляющий сущность «Сотрудник».

Принцип закрытости-открытости реализуется через дополнительный интерфейс, через который класс Client, связфвается с классом Server. Следует обратить на имя класса интерфейса – именно «ClientInterface». Это объясняется тем, что связь между интерфейсом и классов Client намного сильнее, чем связь ClientInterface и Server.

В общем, принцип закрытости-открытости реализуется через абстракции. То есть модуль (Client) может манипулировать абстракцией (ClientInterface) вместо использования неабстрактного объекта (Server).

Еще один метод модификации проекта для удовлетворения принципа OCP показан на рис. 3.12 и рис. 3.13. Рисунок 3.12 демонстрирует неэффективный прием реализации приложения. Класс Drawer должен реализовывать отдельные алгоритмы для рисования конкретных геометрических объектов: Triangle, Rectangle и Circle.

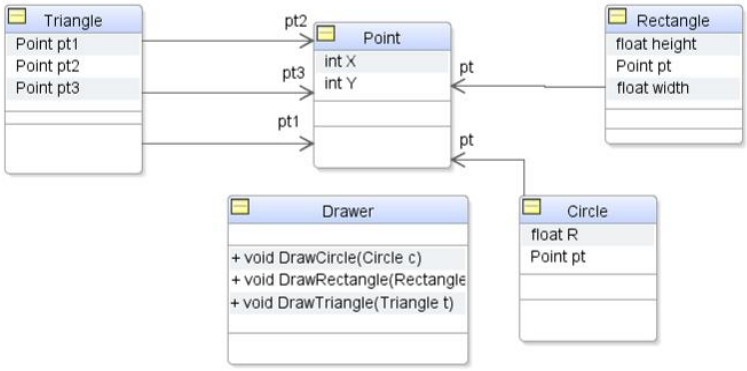


Рисунок 3.12 – Система, не удовлетворяющая принципу закрытости-открытости.

Если необходимо добавить новую фигуру или изменить алгоритм рисования существующей сущности, то модуль с классом Drawer придется перекомпилировать. Решение проблемы показано на рис. 3.13: выделяется абстрактный класс, реализующий абстрактные методы (базовые операции).

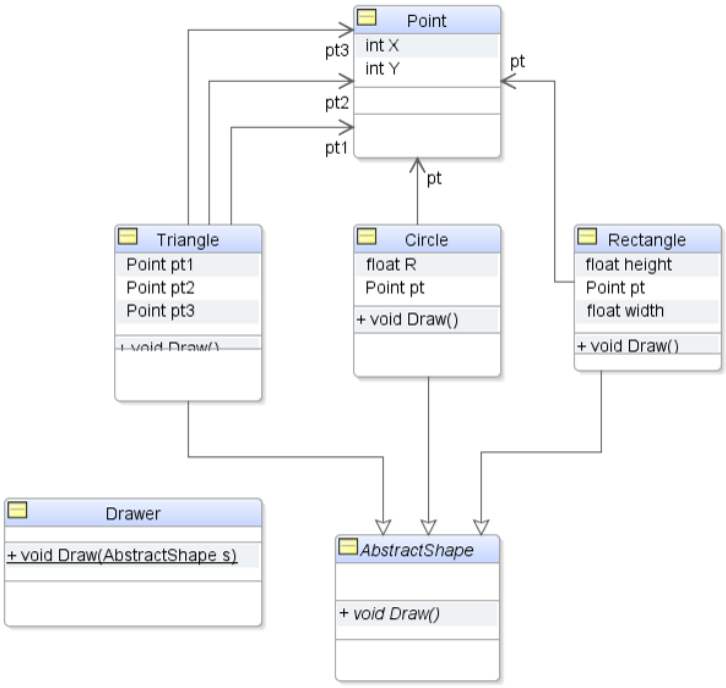


Рисунок 3.13 – Система, не удовлетворяющая принципу закрытости-открытости.

Таким образом, модуль, содержащий Drawer, может быть скомпилирован один раз, добавление потомков AbstractShape (или подклассов) не повлечет его повторной перекомпиляции.

3.7.3. Принцип подстановки Лискоу (LSP)

Принцип LSP упрощенно можно сформулировать следующим образом: подтипы должны быть заменяемы их исходными типами.

Тип S будет подтипом T тогда и только тогда, когда каждому объекту oS типа S соответствует некий объект oT типа T таким образом, что для всех программ P, реализованных в терминах T, поведение P не будет меняться, если oT заменить на oS.

Рассмотрим пример. На рис. 3.14 представлена иерархия классов, которая на первый взгляд, пригодна к использованию в проекте (система управления учебным заведением).

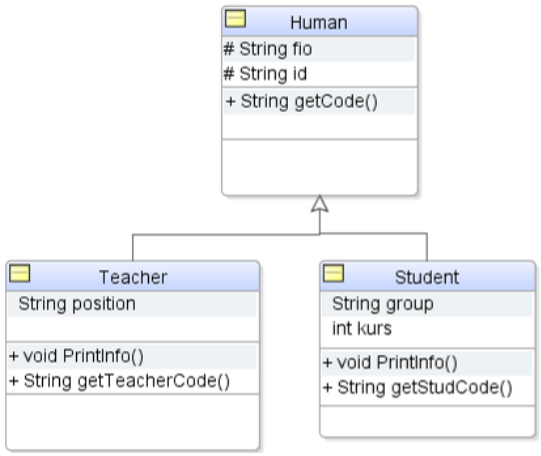


Рисунок 3.14 – Система, не удовлетворяющая принципу LSP.

На рис. 3.15 – 3.17 представлены листинги классов.

```
4   public class Human {
5       // ФИО
6       protected String fio;
7       // Идентификатор
8       protected String id;
9
10      public String getCode(){
11          return fio + "@" + id;
12      }
13  }
```

Рисунок 3.15 – Определение класса Human.

```
4   public class Student extends Human {
5       // группа
6       String group;
7       // курс
8       int kurs;
9
10      public String getStudCode(){
11          return fio + "@" + id +"@"
12                  + group + "@" + kurs;
13      }
14
15      public void PrintInfo(){
16          System.out.println(fio + " " + group);
17      }
18  }
```

Рисунок 3.16 – Определение класса Student.

```
public class Teacher extends Human {
    // должность
    String position;

    public String getTeacherCode(){
        return fio + "@" + id +"@"
                + position;
    }

    public void PrintInfo(){
        System.out.println(fio + " " + position);
    }
}
```

Рисунок 3.17 – Определение класса Teacher.

Но на рис. 3.18 представлен листинг, выявляющий нарушение принципа LSP: очевидно, что объекты базового класса полностью не могут быть заменены объектами производных.

```
4  public class Program {
5      public static void main(String[] args){
6          Vector<Human> all = getHumans();
7
8          for (Human h : all) {
9              System.out.println(h.getClass().getSimpleName());
10             if(h.getClass().getSimpleName().equals("Teacher")){
11                 System.out.println(((Teacher)h).getTeacherCode());
12                 ((Teacher)h).PrintInfo();
13             }
14             if(h.getClass().getSimpleName().equals("Student")){
15                 System.out.println(((Student)h).getStudCode());
16                 ((Student)h).PrintInfo();
17             }
18         }
19     }
20
21     public static Vector<Human> getHumans(){
22         Vector<Human> res = new Vector<Human>();
23         Student stud = new Student();
24         stud.fio = "Новиков Е.И."; stud.id = "s123";
25         stud.group = "ИСТ-151"; stud.kurs = 3;
26         res.add(stud);
27         Teacher t = new Teacher();
28         t.fio = "Николаев Е.И."; t.id = "t1254";
29         t.position = "доцент"; res.add(t);
30         return res;
31     }
32 }
```

Рисунок 3.18 – Использование классов Human, Student и Teacher.

Программист вынужден опрашивать типы, использовать конструкции if или switch, хотя, казалось бы, программист выполнил наследование реализации синтаксически корректно.

Ошибка проявляется в том, что объекты подклассов не способны заместить объекты суперкласса. Для исправления данной ошибки и приведения иерархии классов в соответствие с принципом LSP необходимо выполнить рефакторинг кода. Новые определения классов (рис. 3.19 – 3.20) и новый механизм получения информации об участниках образовательного процесса (рис. 3.21) позволяют устранить допущенные ранее ошибки.

```
public abstract class Human {
    // ФИО
    protected String fio;
    // Идентификатор
    protected String id;
    public abstract String getCode();
    public abstract void PrintInfo();
}
```

Рисунок 3.19 – Определение класса Human.

```
public class Student extends Human {
    // группа
    String group;
    // курс
    int kurs;
    @Override
    public void PrintInfo(){
        System.out.println(fio + " " + group);
    }
    @Override
    public String getCode() {
        return fio + "@" + id +"@"
                + group + "@" + kurs;
    }
}
```

Рисунок 3.20 – Определение класса Student.

```
public class Teacher extends Human {
    // должность
    String position;
    @Override
    public void PrintInfo(){
        System.out.println(fio + " " + position);
    }
    @Override
    public String getCode() {
        return fio + "@" + id +"@"
                + position;
    }
}
```

Рисунок 3.21 – Определение класса Teacher.

```
    public static void main(String[] args){
        Vector<Human> all = getHumans();
        for (Human h : all) {
                System.out.println(h.getCode());
                h.PrintInfo();
        }
    }
```

Рисунок 3.22 – Функция main() после рефакторинга в соответствии с

принципом LSP.

На рис. 3.23 показан окончательный вариант диаграммы классов.

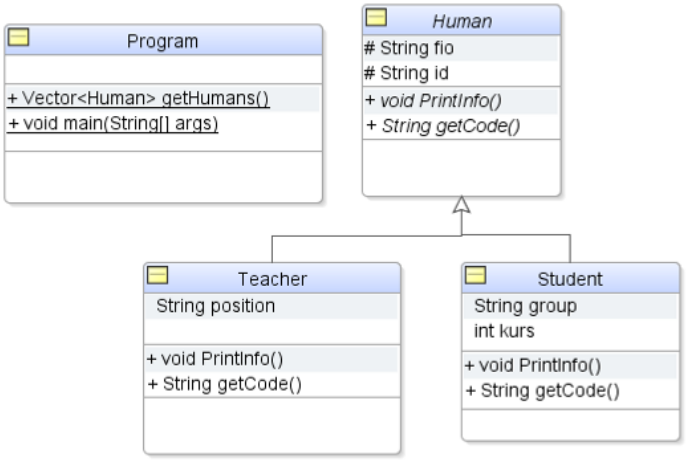


Рисунок 3.23 – Диаграмма классов проекта, удовлетворяющего принципу LSP.

Таким образом, класс Human реализован как абстрактный. Модуль с этим классом может быть скомпилирован один раз. Абстрактность Human принуждает программистов разрабатывать все последующие подклассы с обязательной реализацией методов PrintInfo() и getCode(). Модуль, содержащий класс Program также не потребует модификации при добавлении новых подклассов Human.

3.7.4. Принцип инверсии зависимостей (DIP)

Данный принцип базируется на том, что зависимости внутри системы строятся на основе абстракций и выражается в виде следующих правил:

1. Модули верхних уровней не должны зависеть от модулей нижних уровней. Оба типа модулей должны зависеть от абстракций.

2. Абстракции не должны зависеть от деталей. Детали должны зависеть от абстракций.

В объектно-ориентированном программировании общепринятой практикой является разбиение приложения на слои, каждый из которых

реализует свой собственный функционал. Естественно, что слои должны взаимодействовать.

В структурном анализе и проектировании модули верхнего уровня часто основаны (и следовательно зависят) на модулях нижнего уровня. На рис. 3.24.

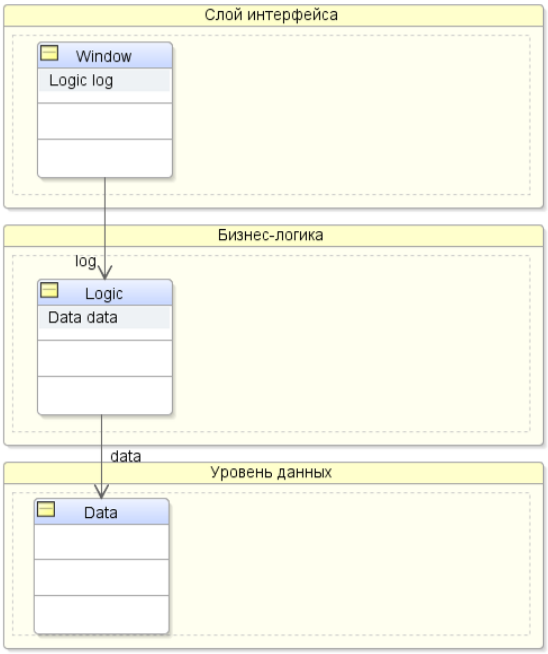


Рисунок 3.24 – Слои объектно-ориентированного приложения, спроектированного с нарушением принципа DIP.

Очевидно, что изменение в нижележащем слое повлечет модификацию классов вышележащих модулей. Из диаграммы на рис. 3.24 также явно следует транзитивная зависимость между модулями. В объектно-ориентированном программировании структура программ инвертирована по отношению к общепринятой логике взаимодействия в структурном программировании. Для устранения недостатков необходимо выполнить рефакторинг с целью избавления от подобного рода зависимостей.

На рис. 3.25 показан более совершенный вариант архитектуры программной системы; связи между слоями существенно ослаблены. Изменения в классах каждого слоя не отражаются на соседних слоях.

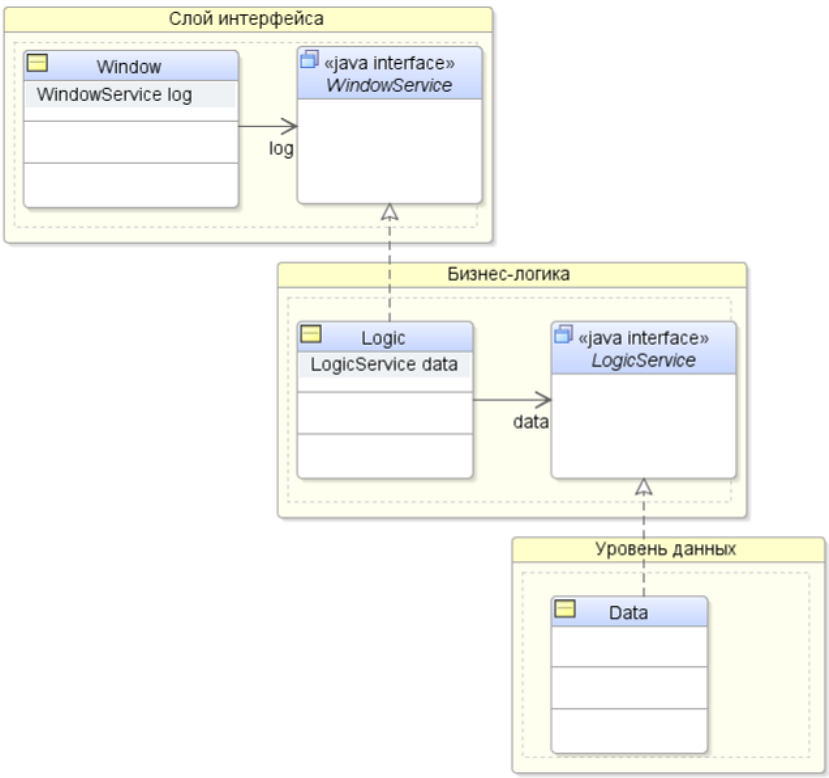


Рисунок 3.25 – Слои объектно-ориентированного приложения,

Упрощенно, суть DIP можно интерпретировать как установление зависимости только от абстракций. То есть все отношения зависимости нужно строить на абстрактном классе или интерфейсе. Можно пользоваться следующими правилами при построении приложений с учетом принципа DIP:

– ни одно поле не должно указывать на конкретные классы;

– ни один класс не должен порождаться другим классом;

– ни один метод не должен отвергать реализацию любого из базовых классов.

3.7.5. Принцип разделения интерфейса (ISP)

Суть принципа проста: интерфейсы, содержащие большое количество различных методов, необходимо разделять на более специфические интерфейсы. Клиенты специализированных интерфейсов (классы, которые их реализуют) должны знать только о методах, которые необходимы им в

работе. В итоге, при изменении метода одного интерфейса не должны меняться клиенты, которые этот метод не используют.

Контрольные вопросы

1. Какая диаграмма является основной для проектирования объектно-ориентированных приложений?
2. Опишите основные типы диаграмм.
3. Для чего используется диаграмма вариантов использования? Назовите основные элементы диаграммы прецедентов и дайте их характеристику.
4. Для чего используется диаграмма последовательностей? Опишите структуру диаграммы данного типа.
5. Опишите состав элементов и назначение диаграммы деятельности.
6. Для чего применяется диаграмма классов? Перечислите основные отношения между классами, которые могут быть представлены на диаграмме классов.
7. Какая диаграмма должна разрабатываться в первую очередь при проектировании сложной системы?
8. Какие программные продукты можно использовать при построении диаграмм? В чем заключается отличие этих средств?
9. Для чего предназначена технология UML?
10. Поясните область применения и назначение понятий: «линия жизни», «прецедент», «действие», «действующее лицо», «асинхронное сообщение».
11. Сформулируйте и объясните смысл принципа персональной ответственности (SRP). Приведите примеры.
12. Сформулируйте и объясните смысл принципа закрытости-открытости (OCP). Приведите примеры.

13. Сформулируйте и объясните смысл принципа подстановки Лискоу (LSP). Приведите примеры.

14. Сформулируйте и объясните смысл принципа инверсии зависимостей (DIP). Приведите примеры.

15. Сформулируйте и объясните смысл принципа разделения интерфейса (ISP). Приведите примеры.

РАЗДЕЛ 4. ПАТТЕРНЫ ПРОЕКТИРОВАНИЯ

Паттерны относятся к одним из самых сложных аспектов объектно-ориентированного программирования. Знание принципов ООП (инкапсуляция, полиморфизм, наследование) и умение ими пользоваться отдельно, без использования паттернов, приведет к неэффективным программным системам, наследующим большую часть недостатков структурного подхода.

Хорошо продуманные объектно-ориентированные системы внешне похожи на собрание многочисленных паттернов, но вовсе не потому, что их проектировщики мыслили именно такими категориями. Композиция на уровне паттернов, а не классов или объектов, позволяет добиться той же синергии, но с меньшими усилиями.

При практической разработке приложений у программиста появляется опыт, проявляющийся в повышении качества кода, моделей и документации. Программисты, основываясь на эвристических методах, способны разрабатывать приложения, представляющие синергетическое единство паттернов.

Для начинающих программистов важно также уяснить, что использование того или иного паттерна не гарантирует корректной работы программы. Также не существует конкретных рекомендаций какой паттерн следует применять при проектировании графического редактор, программы для работы с базами данных или веб-приложения.

На начальных этапах профессиональной деятельности программиста оптимальной стратегией является использование возможностей, предоставляемых библиотеками программного кода. Особенно данное правило актуально при разработке подсистем безопасности в сложных программных системах. Реализация своих собственных модулей,

основываясь на изученных паттернах, может привести к критическим ошибкам, приводящим к утечке данных.

В то время как паттерны проектирования развиваются из успешных практик, антипаттерны можно представлять себе как наборы неудавшихся методик. Существуют убедительные документальные доказательства того, что большинство программных проектов в конечном счете признаются неудачными.

Термин «антишаблон» («антипаттерн») стал следствием того факта, что паттерны проектирования создаются с целью заранее решать проблемы определенного типа. Антипаттерн, с другой стороны, является реакцией на проблему и происходит из плохого опыта.

Одним словом, в то время как шаблоны проектирования базируются на надежных методиках проектирования, антишаблоны можно рассматривать как методики, которых следует избегать.

4.1 Основы терминологии. Классификация паттернов

Развитие технологий объектно-ориентированного программирования привело к созданию устойчивых, широко распространенных принципов взаимодействия классов и объектов. На основе этих принципов стали формироваться определенные шаблоны структуры, взаимодействия и поведения. Данные шаблоны стали применяться многими программистами; для их описания используют слово-заимствование «паттерн» (pattern). Наименование шаблон проектирования не совсем корректное. Термин шаблон имеет устоявшийся перевод «template», который отражает иную технологию – шаблонизированные типы.

Под паттерном понимается описание взаимодействия объектов и классов, адаптированных для решения общей задачи проектирования в конкретном контексте.

Паттерн проектирования именует, абстрагирует и идентифицирует ключевые аспекты структуры общего решения, которые и позволяют применить его для создания повторно используемого дизайна. Он вычленяет участвующие классы и экземпляры, их роль и отношения, а также функции. При описании каждого паттерна внимание акцентируется на конкретной задаче объектно-ориентированного проектирования.

Хотя, строго говоря, паттерны используются в проектировании, они основаны на практических решениях, реализованных на основных языках объектно-ориентированного программирования типа Smalltalk, C++, # и Java.

При описании паттернов проектировании придерживаются единого принципа. Описание каждого паттерна разбито на разделы:

1. Название и классификация паттерна. Название паттерна должно четко отражать его назначение.

2. Назначение. Лаконичный ответ на следующие вопросы: каковы функции паттерна, его обоснование и назначение, какую конкретную задачу проектирования можно решить с его помощью.

3. Синонимы паттерна. Другие распространенные названия паттерна, если таковые имеются.

4. Мотивация. Сценарий, иллюстрирующий задачу проектирования и то, как она решается данной структурой класса или объекта. Благодаря мотивации можно лучше понять последующее, более абстрактное описание паттерна.

5. Применимость. Описание ситуаций, в которых можно применять данный паттерн. Примеры проектирования, которые можно улучшить с его помощью. Распознавание таких ситуаций.

6. Структура. Графическое представление классов в паттерне с использованием диаграмм UML.

7. Участники. Классы или объекты, задействованные в данном паттерне проектирования, и их функции.

8. Отношения. Взаимодействие участников для выполнения своих функций.

9. Результаты. Насколько паттерн удовлетворяет поставленным требованиям? Результаты применения, компромиссы, на которые приходится идти. Какие аспекты поведения системы можно независимо изменять, используя данный паттерн?

10. Реализация. Сложности и так называемые подводные камни при реализации паттерна. Советы и рекомендуемые приемы. Есть ли у данного паттерна зависимость от языка программирования?

11. Пример кода. Фрагмент кода, иллюстрирующий вероятную реализацию на языках C# или Java.

12. Известные применения. Возможности применения паттерна в реальных системах.

13. Родственные паттерны. Связь других паттернов проектирования с данным. Важные различия. Использование данного паттерна в сочетании с другими.

Паттерны проектирования различаются степенью детализации и уровнем абстракции и должны быть каким-то образом организованы. В данном разделе описывается классификация, позволяющая ссылаться на семейства взаимосвязанных паттернов. Она поможет быстрее освоить паттерны, описываемые в каталоге, и укажет направление поиска новых.

Паттерны удобно классифицировать по двум критериям (табл. 4.1). Первый – цель – отражает назначение паттерна. В связи с этим выделяются порождающие паттерны, структурные паттерны и паттерны поведения. Первые связаны с процессом создания объектов. Вторые имеют отношение к композиции объектов и классов. Паттерны поведения характеризуют то, как классы или объекты взаимодействуют между собой.

Таблица 4.1 – Классификация паттернов проектирования.

Уровень/ Цель	Порождающие паттерны	Структурные паттерны	Паттерны поведения
Класс	Фабричный метод	Адаптер класса	Интерпретатор Шаблонный метод
Объект	Абстрактная фабрика Одиночка Прототип Строитель	Адаптер объекта Декоратор Заместитель Компоновщик Мост Приспособленец Фасад	Итератор Команда Наблюдатель Посетитель Посредник Состояние Стратегия Хранитель Цепочка обязанностей

Второй критерий – уровень – говорит о том, к чему обычно применяется паттерн: к объектам или классам. Паттерны уровня классов описывают отношения между классами и их подклассами. Такие отношения выражаются с помощью наследования, поэтому они статичны, то есть зафиксированы на этапе компиляции. Паттерны уровня объектов описывают отношения между объектами, которые могут изменяться во время выполнения, то есть предполагают динамические связи. Почти все паттерны в какой-то мере используют наследование. Поэтому к категории «паттерны классов» отнесены только те, что сфокусированы лишь на отношениях между классами. Большинство паттернов действуют на уровне объектов.

Порождающие паттерны классов частично делегируют ответственность за создание объектов своим подклассам, тогда как порождающие паттерны объектов передают ответственность другому объекту. Структурные паттерны классов используют наследование для составления классов, в то время как структурные паттерны объектов описывают способы сборки объектов из частей. Поведенческие паттерны классов используют наследование для описания алгоритмов и потока управления, а поведенческие паттерны объектов описывают, как объекты, принадлежащие некоторой группе,

совместно функционируют и выполняют задачу, которая ни одному отдельному объекту не под силу.

4.2 Основы использования паттернов

Паттерны проектирования позволяют разными способами решать многие задачи, с которыми постоянно сталкиваются проектировщики объектно-ориентированных приложений.

Самая трудная задача в объектно-ориентированном проектировании - разложить систему на объекты. При решении приходится учитывать множество факторов: инкапсуляцию, глубину детализации, наличие зависимостей, гибкость, производительность, развитие, повторное использование. Все это влияет на декомпозицию, причем часто противоречивым образом.

Паттерны проектирования помогают выявить не вполне очевидные абстракции и объекты, которые могут их использовать. Например, объектов, представляющих процесс или алгоритм, в действительности нет, но они являются неотъемлемыми составляющими гибкого дизайна.

В объектно-ориентированных системах интерфейсы фундаментальны. Об объектах известно только то, что они сообщают о себе через свои интерфейсы. Никакого способа получить информацию об объекте или заставить его что-то сделать в обход интерфейса не существует. Интерфейс объекта ничего не говорит о его реализации; разные объекты вправе реализовывать сходные запросы совершенно по-разному. Это означает, что два объекта с различными реализациями могут иметь одинаковые интерфейсы.

У манипулирования объектами строго через интерфейс абстрактного класса есть два преимущества:

– клиенту не нужно иметь информации о конкретных типах объектов, которыми он пользуется, при условии, что все они имеют ожидаемый клиентом интерфейс;

– клиенту необязательно «знать» о классах, с помощью которых реализованы объекты. Клиенту известно только об абстрактном классе (или классах), определяющих интерфейс.

Данные положения фактически заключены в принципах SOLID.

Системы необходимо проектировать с учетом их дальнейшего развития. Благодаря паттернам систему всегда можно модифицировать определенным образом. Каждый паттерн позволяет изменять некоторый аспект системы независимо от всех прочих, таким образом, она менее подвержена влиянию изменений конкретного вида.

Вот некоторые типичные причины, приводящие к повторному проектированию, а также паттерны, которые позволяют этого избежать:

1. При создании объекта явно указывается класс. Задание имени класса привязывает вас к конкретной реализации, а не к конкретному интерфейсу. Это может осложнить изменение объекта в будущем. Чтобы уйти от такой проблемы, создавайте объекты косвенно. Паттерны проектирования: абстрактная фабрика, фабричный метод, прототип;

2. Зависимость от конкретных операций. Задавая конкретную операцию, вы ограничиваете себя единственным способом выполнения запроса. Если же не включать запросы в код, то будет проще изменить способ удовлетворения запроса как на этапе компиляции, так и на этапе выполнения. Паттерны проектирования: цепочка обязанностей, команда.

3. Зависимость от аппаратной и программной платформ. Внешние интерфейсы операционной системы и интерфейсы прикладных программ (API) различны на разных программных и аппаратных платформах. Если программа зависит от конкретной платформы, ее будет труднее перенести на другие. Даже на «родной» платформе такую программу трудно

поддерживать. Поэтому при проектировании систем так важно ограничивать платформенные зависимости. Паттерны проектирования: абстрактная фабрика, мост.

4. Зависимость от представления или реализации объекта. Если клиент «знает», как объект представлен, хранится или реализован, то при изменении объекта может оказаться необходимым изменить и клиента. Сокрытие этой информации от клиентов поможет уберечься от каскада изменений. Паттерны проектирования: абстрактная фабрика, мост, хранитель, заместитель.

5. Зависимость от алгоритмов. Во время разработки и последующего использования алгоритмы часто расширяются, оптимизируются и заменяются. Зависящие от алгоритмов объекты придется переписывать при каждом изменении алгоритма. Поэтому алгоритмы, вероятность изменения которых высока, следует изолировать. Паттерны проектирования: мост, итератор, стратегия, шаблонный метод, посетитель.

6. Сильная связанность. Сильно связанные между собой классы трудно использовать отдельно, так как они зависят друг от друга. Сильная связанность приводит к появлению монолитных систем, в которых нельзя ни изменить, ни удалить класс без знания деталей и модификации других классов. Такую систему трудно изучать, переносить на другие платформы и сопровождать. Слабая связанность повышает вероятность того, что класс можно будет повторно использовать сам по себе. При этом изучение, перенос, модификация и сопровождение системы намного упрощаются. Для поддержки слабо связанных систем в паттернах проектирования применяются такие методы, как абстрактные связи и разбиение на слои. Паттерны проектирования: абстрактная фабрика, мост, цепочка обязанностей, команда, фасад, посредник, наблюдатель.

7. Расширение функциональности за счет порождения подклассов. Специализация объекта путем создания подкласса часто оказывается

непростым делом. С каждым новым подклассом связаны фиксированные издержки реализации (инициализация, очистка и т.д.). Для определения подкласса необходимо также ясно представлять себе устройство родительского класса. Например, для замещения одной операции может потребоваться заместить и другие. Замещение операции может оказаться необходимым для того, чтобы можно было вызвать унаследованную операцию. Кроме того, порождение подклассов ведет к комбинаторному росту числа классов, поскольку даже для реализации простого расширения может понадобиться много новых подклассов.

Композиция объектов и делегирование – гибкие альтернативы наследованию для комбинирования поведений. Приложению можно добавить новую функциональность, меняя способ композиции объектов, а не определяя новые подклассы уже имеющихся классов. С другой стороны, при интенсивном использовании композиции объектов проект может оказаться трудным для понимания. С помощью многих паттернов проектирования удается построить такое решение, где специализация достигается за счет определения одного подкласса и комбинировании его экземпляров с уже существующими. Паттерны проектирования: мост, цепочка обязанностей, компоновщик, декоратор, наблюдатель, стратегия.

8. Неудобства при изменении классов. Иногда нужно модифицировать класс, но делать это неудобно. Допустим, вам нужен исходный код, а его нет (так обстоит дело с коммерческими библиотеками классов). Или любое изменение тянет за собой модификации множества существующих подклассов. Благодаря паттернам проектирования можно модифицировать классы и при таких условиях. Паттерны проектирования: адаптер, декоратор, посетитель.

В таблице 4.2 представлены паттерны и указаны аспекты системы, которые можно изменить в результате их применения.

Таблица 4.2 – Влияние паттернов на функционирование программных систем.

Назначение паттерна	Паттерн проектирования	Изменяемые свойства системы
Порождающие паттерны	Абстрактная фабрика	Семейства порождаемых объектов
	Одиночка	Единственный экземпляр класса
	Прототип	Класс, из которого инстанцируется объект
	Строитель	Способ создания составного объекта
	Фабричный метод	Инстанцируемый подкласс объекта
Структурные паттерны	Адаптер	Интерфейс к объекту
	Декоратор	Обязанности объекта без порождения подкласса
	Заместитель	Способ доступа к объекту, его местоположение
	Компоновщик	Структура и состав объекта
	Мост	Реализация объекта
	Приспособленец	Накладные расходы на хранение объектов
	Фасад	Интерфейс к подсистеме
Паттерны поведения	Интерпретатор	Грамматика и интерпретация языка
	Итератор	Способ обхода элементов агрегата
	Команда	Время и способ выполнения запроса
	Наблюдатель	Множество объектов, зависящих от другого объекта; способ, которым зависимые объекты поддерживают себя в актуальном состоянии
	Посетитель	Операции, которые можно применить к объекту или объектам, не меняя класса
	Посредник	Объекты, взаимодействующие между собой, и способ их кооперацией
	Состояние	Состояние объекта
	Стратегия	Алгоритм
	Хранитель	Закрытая информация, хранящаяся вне объекта, и время ее сохранения
	Цепочка обязанностей	Объект, выполняющий запрос
	Шаблонный метод	Шаги алгоритма

В современном объектно-ориентированном проектировании можно выделить более 100 паттернов, которые являются устоявшимися и широко применимыми. В данном учебном пособии рассматриваются наиболее распространенные паттерны.

Каждый паттерн можно охарактеризовать следующим образом:

Abstract Factory (абстрактная фабрика). Предоставляет интерфейс для создания семейств, связанных между собой, или независимых объектов, конкретные классы которых неизвестны.

Adapter (адаптер). Преобразует интерфейс класса в некоторый другой интерфейс, ожидаемый клиентами. Обеспечивает совместную работу классов, которая была бы невозможна без данного паттерна из-за несовместимости интерфейсов.

Bridge (мост). Отделяет абстракцию от реализации, благодаря чему появляется возможность независимо изменять то и другое.

Builder (строитель). Отделяет конструирование сложного объекта от его представления, позволяя использовать один и тот же процесс конструирования для создания различных представлений.

Chain of Responsibility (цепочка обязанностей). Можно избежать жесткой зависимости отправителя запроса от его получателя, при этом запросом начинает обрабатываться один из нескольких объектов. Объекты-получатели связываются в цепочку, и запрос передается по цепочке, пока какой-то объект его не обработает.

Command (команда). Инкапсулирует запрос в виде объекта, позволяя тем самым параметризировать клиентов типом запроса, устанавливать очередность запросов, протоколировать их и поддерживать отмену выполнения операций.

Composite (компоновщик). Группирует объекты в древовидные структуры для представления иерархий типа «часть-целое». Позволяет

клиентам работать с единичными объектами так же, как с группами объектов.

Decorator (декоратор). Динамически возлагает на объект новые функции. Декораторы применяются для расширения имеющейся функциональности и являются гибкой альтернативой порождению подклассов.

Facade (фасад). Предоставляет унифицированный интерфейс к множеству интерфейсов в некоторой подсистеме. Определяет интерфейс более высокого уровня, облегчающий работу с подсистемой.

Factory Method (фабричный метод). Определяет интерфейс для создания объектов, при этом выбранный класс инстанцируется подклассами.

Flyweight (приспособленец). Использует разделение для эффективной поддержки большого числа мелких объектов.

Interpreter (интерпретатор). Для заданного языка определяет представление его грамматики, а также интерпретатор предложений языка, использующий это представление.

Iterator (итератор). Дает возможность последовательно обойти все элементы составного объекта, не раскрывая его внутреннего представления.

Mediator (посредник). Определяет объект, в котором инкапсулировано знание о том, как взаимодействуют объекты из некоторого множества. Способствует уменьшению числа связей между объектами, позволяя им работать без явных ссылок друг на друга. Это, в свою очередь, дает возможность независимо изменять схему взаимодействия.

Memento (хранитель). Позволяет, не нарушая инкапсуляции, получить и сохранить во внешней памяти внутреннее состояние объекта, чтобы позже объект можно было восстановить точно в таком же состоянии.

Observer (наблюдатель). Определяет между объектами зависимость типа один-ко-многим, так что при изменении состоянии одного объекта все зависящие от него получают извещение и автоматически обновляются.

Prototype (прототип). Описывает виды создаваемых объектов с помощью прототипа и создает новые объекты путем его копирования.

Proxy (заместитель). Подменяет другой объект для контроля доступа к нему.

Singleton (одиночка). Гарантирует, что некоторый класс может иметь только один экземпляр, и предоставляет глобальную точку доступа к нему.

State (состояние). Позволяет объекту варьировать свое поведение при изменении внутренне го состояния. При этом создается впечатление, что поменялся класс объекта.

Strategy (стратегия). Определяет семейство алгоритмов, инкапсулируя их все и позволяя подставлять один вместо другого. Можно менять алгоритм независимо от клиента, который им пользуется.

Template Method (шаблонный метод). Определяет скелет алгоритма, перекладывая ответственность за некоторые его шаги на подклассы. Позволяет подклассам переопределять шаги алгоритма, не меняя его общей структуры.

Visitor (посетитель). Представляет операцию, которую надо выполнить над элементами объекта. Позволяет определить новую операцию, не меняя классы элементов, к которым он применяется.

Перед рассмотрением особенностей реализации паттернов, рассмотрим самый простой – Singleton (Одиночка).

4.3 Паттерн «Одиночка» (Singleton)

Паттерн Singleton предназначен для создания объектов. Данный паттерн гарантирует, что у класса в программе существует только один экземпляр. При этом определение паттерна гарантирует, что к этому экземпляру будет предоставлена единственная точка доступа.

Паттерн «Одиночка» следует использовать в случаях:

– должен быть ровно один экземпляр некоторого класса, доступный всем клиентам;

– единственный экземпляр должен расширяться путем порождения подклассов, и клиентам нужно иметь возможность работать с расширенным экземпляром без модификации своего кода.

На рис. 4.1 представлена диаграмма классов, удовлетворяющая данным требованиям.

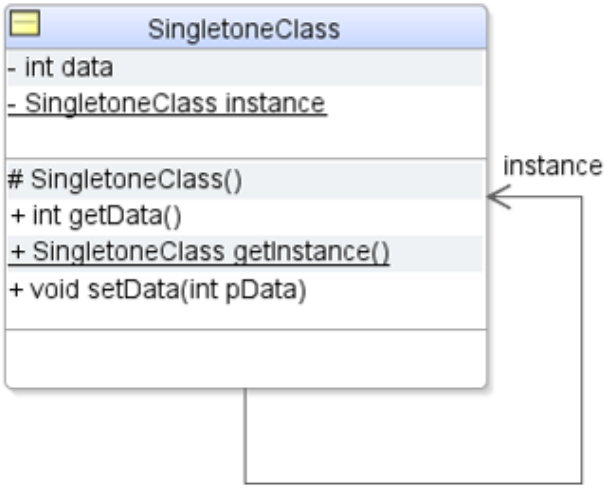


Рисунок 4.1 – Паттерн «Одиночка».

На рис. 4.2 представлен исходный код класса SingletoneClass.

```
public class SingletoneClass {
    private static SingletoneClass instance;
    // данные класа
    private int data;
    // защищенный конструктор
    protected SingletoneClass(){

    }
    // точка доступа к экземпляру
    public static SingletoneClass getInstance(){
        if(instance == null){
            instance = new SingletoneClass();
        }
        return instance;
    }
    // методы работы с данными
    public void setData(int pData){
        data = pData;
    }
    public int getData(){
        return data;
    }
}
```

Рисунок 4.2 – Описание класса SingletoneClass.

Ключевые особенности реализации шаблона:

1. Единственный экземпляр класса instance является статическим и закрыт внутри класса.

2. Доступ к экземпляру instance осуществляется только через открытый метод getInstance().

3. Для предотвращения повторного создания объекта типа SingletoneClass реализован единственный защищенный конструктор.

4. Класс, построенный по принципу «Одиночка» содержит необходимые поля и методы, представляющие данные.

Использование данного класса представлено на рис. 4.3. Сколько бы объектов ни создавалось в программе – это будет один и тот же объект.

```
3  public class Program {
4      public static void main(String[] args){
5          SingletoneClass o1 = SingletoneClass.getInstance();
6          o1.setData(100);
7          SingletoneClass o2 = SingletoneClass.getInstance();
8          System.out.print(o2.getData());
9      }
10 }
```

Рисунок 4.3 – Использование класса SingletoneClass.

Достоинства паттерна «Одиночка»:

1. Контролируемый доступ к единственному экземпляру. Поскольку класс SingletoneClass инкапсулирует свой единственный экземпляр, он полностью контролирует то, как и когда клиенты получают доступ к нему.

2. Уменьшение числа имен. Паттерн одиночка – шаг вперед по сравнению с глобальными переменными. Он позволяет избежать засорения пространства имен глобальными переменными, в которых хранятся уникальные экземпляры.

3. Допускает уточнение операций и представления. От класса SingletoneClass можно порождать подклассы, а приложение легко сконфигурировать экземпляром расширенного класса. Можно конкретизировать приложение экземпляром того класса, который необходим во время выполнения.

4. Допускает переменное число экземпляров. Паттерн позволяет легко изменить свое решение и разрешить появление более одного экземпляра класса SingletoneClass. Можно применять один и тот же подход для управления числом экземпляров, используемых в приложении. Изменить нужно будет лишь операцию, дающую доступ к экземпляру класса Singleton.

5. Большая гибкость, чем у операций класса. Еще один способ реализовать функциональность одиночки – использовать операции класса, то есть статические функции-члены. Но оба этих приема препятствуют изменению дизайна, если потребуется разрешить наличие нескольких

экземпляров класса. Кроме того, статические функции-члены не могут быть виртуальными, так что их нельзя полиморфно заместить в подклассах.

4.4 Порождающие паттерны

Порождающие паттерны проектирования абстрагируют процесс инстанцирования – создания объектов. Они помогут сделать систему независимой от способа создания, композиции и представления объектов. Паттерн, порождающий классы, использует наследование, чтобы варьировать инстанцируемый класс, а паттерн, порождающий объекты, делегирует инстанцирование другому объекту.

Эти паттерны оказываются важны, когда система больше зависит от композиции объектов, чем от наследования классов. Получается так, что основной упор делается не на жестком кодировании фиксированного набора поведений, а на определении небольшого набора фундаментальных поведений, с помощью композиции которых можно получать любое число более сложных. Таким образом, для создания объектов с конкретным поведением требуется нечто большее, чем простое инстанцирование класса.

При изучении порождающих паттернов необходимо акцентировать внимание на следующих аспектах:

1. Эти паттерны инкапсулируют знания о конкретных классах, которые применяются в системе.

2. Паттерны скрывают детали того, как эти классы создаются и стыкуются. Единственная информация об объектах, известная системе, – это их интерфейсы, определенные с помощью абстрактных классов.

Следовательно, порождающие паттерны обеспечивают большую гибкость при решении вопроса о том, что создается, кто это создает, как и когда. Можно собрать систему из «готовых» объектов с самой различной

структурой и функциональностью статически (на этапе компиляции) или динамически (во время выполнения).

4.4.1 Паттерн «Абстрактная фабрика» (Abstract Factory)

Абстрактная фабрика – паттерн, порождающий объекты. Предоставляет интерфейс для создания семейств взаимосвязанных или взаимозависимых объектов, не специфицируя их конкретных классов.

На рис. 4.4 представлена диаграмма классов, представляющая концепцию данного паттерна. На диаграмме скрыты отношения зависимости.

На диаграмме присутствуют три иерархии, порожденные абстрактными классами AbstractProductA, AbstractProductB, AbstractProductC. Допустим, что данные классы символизируют три различные категории продуктов; каждая порождаемая иерархия – конкретные продукты данной категории.

Иерархия, порожденная суперклассом AbstractFactory, представляет механизм для создания различных товаров: классы, производные от AbstractFactory, символизируют производителя. Таким образом, каждый производитель может предоставить товары трех категорий, причем товары разных производителей из одной товарной категории будут отличаться (будут принципиально различны и представлены в программе объектами различных классов).

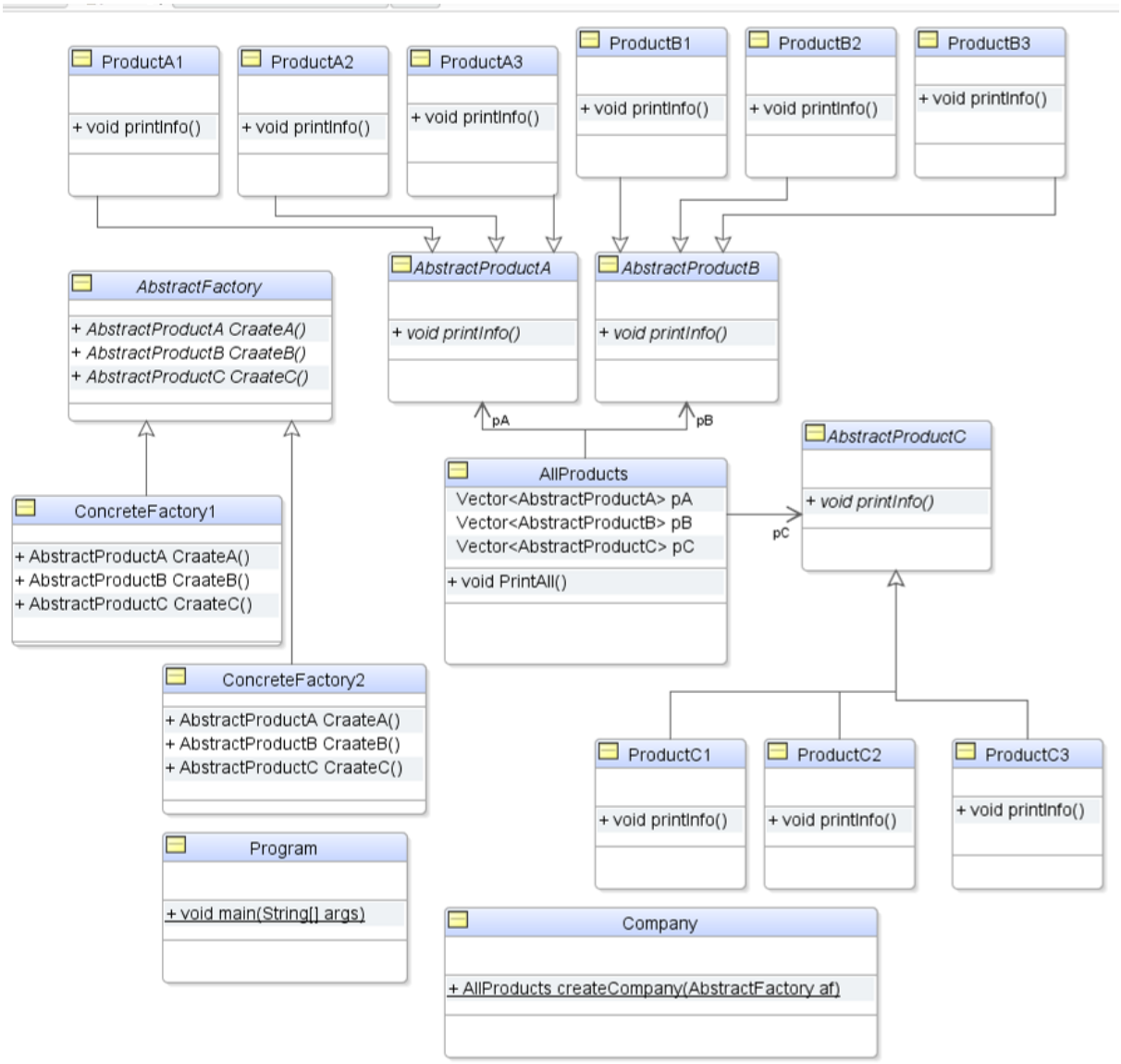


Рисунок 4.4 – Паттерн «Абстрактная фабрика».

На рис. 4.5 показан листинг кода класса AllProducts. Данный класс представляет собой сущность «Товарный ассортимент компании». Класс зависит только от абстрактных классов AbstractProductA, AbstractProductB, AbstractProductC.

На рис. 4.6 показан листинг класса Company. Данный класс непосредственно создает компанию и наполняет товарный ассортимент.

```
6  public class AllProducts {
7      Vector<AbstractProductA> pA = new Vector<>();
8      Vector<AbstractProductB> pB = new Vector<>();
9      Vector<AbstractProductC> pC = new Vector<>();
10
11     public void PrintAll(){
12         for(AbstractProductA a : pA)
13             a.printInfo();
14         for(AbstractProductB b : pB)
15             b.printInfo();
16         for(AbstractProductC c : pC)
17             c.printInfo();
18     }
19 }
```

Рисунок 4.5 – Листинг класса AllProducts.

```
3  public class Company {
4      public static AllProducts
5      createCompany(AbstractFactory af){
6          AllProducts ap = new AllProducts();
7          ap.pA.add(af.CreateA());
8          ap.pB.add(af.CreateB());
9          ap.pC.add(af.CreateC());
10         return ap;
11     }
12 }
```

Рисунок 4.6 – Класс, создающий компанию.

На рис. 4.7 показан пример кода, использующий данный паттерн.

```
3  public class Program {
4      public static void main(String[] args){
5          // Создание объектов фабрик
6          ConcreteFactory1 cfa1 = new ConcreteFactory1();
7          ConcreteFactory2 cfa2 = new ConcreteFactory2();
8          // Класс-клиент запрашивает возможности фабрик
9          AllProducts продуктоваяКомпания = Company.createCompany(cfa1);
10         AllProducts хозяйственныеТовары = Company.createCompany(cfa2);
11         // Вывод информации (проверка)
12         System.out.println("Товары продуктовой компании:");
13         продуктоваяКомпания.PrintAll();
14         System.out.println("Товары хозмагазина:");
15         хозяйственныеТовары.PrintAll();
16     }
17 }
```

Рисунок 4.7 – Класс, создающий компанию.

В результате выполнения простой программы, были созданы две компании, произведено наполнение их товарного ассортимента. Результат работы программы показан на рис. 4.8.

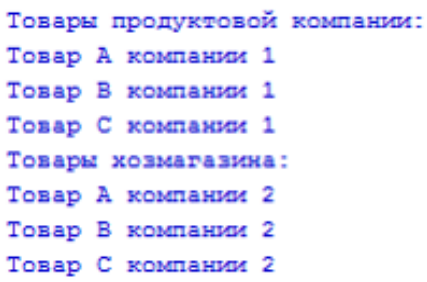


Рисунок 4.8 – Вывод программы.

В приведенном шаблоне можно выделить следующих участников:

1. Абстрактная фабрика (класс AbstractFactory) – это необходимый компонент паттерна. Он объявляет интерфейс (совокупность методов) для создания абстрактных объектов-продуктов (AbstractProductA, AbstractProductB, AbstractProductC).

2. Конкретные фабрики (классы ConcreteFactory1, ConcreteFactory2) – классы конкретных фабрик объектов. Если в классе AbstractFactory интерфейс был объявлен, то в этих классах данный интерфейс реализуется.

3. Классы абстрактных продуктов (AbstractProductA, AbstractProductB, AbstractProductC) – классы, объявляющие интерфейсы для каждого типа объекта-продукта.

4. Классы конкретных продуктов. На диаграмме (рис. 4.4) в этой категории находятся классы: ProductA1, ProductA2, ProductA3, ProductB1, ProductB2, ProductB3, ProductC1, ProductC2, ProductC3. Данные классы реализуют интерфейсы абстрактных типов объектов-продуктов.

5. Клиент. Классы Company и AllProduct. Особенность этих классов в том, что они используют только интерфейсы, объявленные в классах абстрактной фабрики и абстрактных продуктов.

Использование паттерна «Абстрактная фабрика» целесообразно в случае:

– система не должна зависеть от того, как создаются, компонуются и представляются входящие в нее объекты;

– входящие в семейство взаимосвязанные объекты должны использоваться вместе и необходимо обеспечить выполнение этого ограничения;

– система должна конфигурироваться одним из семейств составляющих ее объектов;

– необходимо предоставить библиотеку объектов, раскрывая только их интерфейсы, но не реализацию.

На практике обычно создается единственный экземпляр класса конкретной фабрики. Эта конкретная фабрика создает объекты-продукты, имеющие вполне определенную реализацию. Для создания других видов объектов клиент должен воспользоваться другой конкретной фабрикой; а абстрактная фабрика передоверяет создание объектов-продуктов своему подклассу.

Преимущества использования паттерна Abstract Factory:

1. Изолирует конкретные классы. Помогает контролировать классы объектов, создаваемых приложением. Поскольку фабрика инкапсулирует ответственность за создание классов и сам процесс их создания, то она изолирует клиента от деталей реализации классов. Клиенты манипулируют экземплярами через их абстрактные интерфейсы. Имена изготавливаемых классов известны только конкретной фабрике, в коде клиента они не упоминаются.

2. Упрощает замену семейств продуктов. Класс конкретной фабрики появляется в приложении только один раз: при инстанцировании. Это облегчает замену используемой приложением конкретной фабрики. Приложение может изменить конфигурацию продуктов, просто подставив новую конкретную фабрику. Поскольку абстрактная фабрика создает все семейство продуктов, то и заменяется сразу все семейство. В рассмотренном

примере с товарной номенклатурой компаний перейти от продуктов одной компании к ассортименту другой можно, просто переключившись на продукты соответствующей фабрики и заново создав интерфейс.

3. Гарантирует сочетаемость продуктов. Если продукты некоторого семейства спроектированы для совместного использования, то важно, чтобы приложение в каждый момент времени работало только с продуктами единственного семейства. Класс AbstractFactory позволяет легко соблюсти это ограничение.

Это преимущества шаблона, но есть и недостаток: поддержание нового вида продуктов представляет определенную сложность. Расширение абстрактной фабрики для изготовления новых видов продуктов – непростая задача. Интерфейс AbstractFactory фиксирует набор продуктов, которые можно создать. Для поддержки новых продуктов необходимо расширить интерфейс фабрики, то есть изменить класс AbstractFactory и все его подклассы.

Особенности применения паттерна «Абстрактная фабрика»:

1. Фабрики как объекты, существующие в единственном экземпляре. Как правило, приложению нужен только один экземпляр класса конкретной фабрики на каждое семейство продуктов. Поэтому для реализации лучше всего применить паттерн одиночка.

2. Создание продуктов. Класс AbstractFactory объявляет только интерфейс для создания продуктов. Фактическое их создание – дело подклассов конкретных продуктов. Чаще всего для этой цели определяется фабричный метод для каждого продукта (паттерн «Фабричный метод»). Конкретная фабрика специфицирует свои продукты путем замещения фабричного метода для каждого из них. Хотя такая реализация проста, она требует создавать новый подкласс конкретной фабрики для каждого семейства продуктов, даже если они почти ничем не отличаются.

3. Определение расширяемых фабрик. Класс AbstractFactory обычно определяет разные операции для каждого вида изготавливаемых продуктов. Виды продуктов кодируются в сигнатуре операции. Для добавления нового вида продуктов нужно изменить интерфейс класса AbstractFactory и всех зависящих от него классов.

4.4.2 Паттерн «Строитель» (Builder)

Данный паттерн также является механизмом создания новых объектов. Его применение позволяет отделить конструирование сложного объекта от его представления, так что в результате одного и того же процесса конструирования могут получаться разные представления.

На рис. 4.9 приведен простейший пример реализации паттерна Builder.

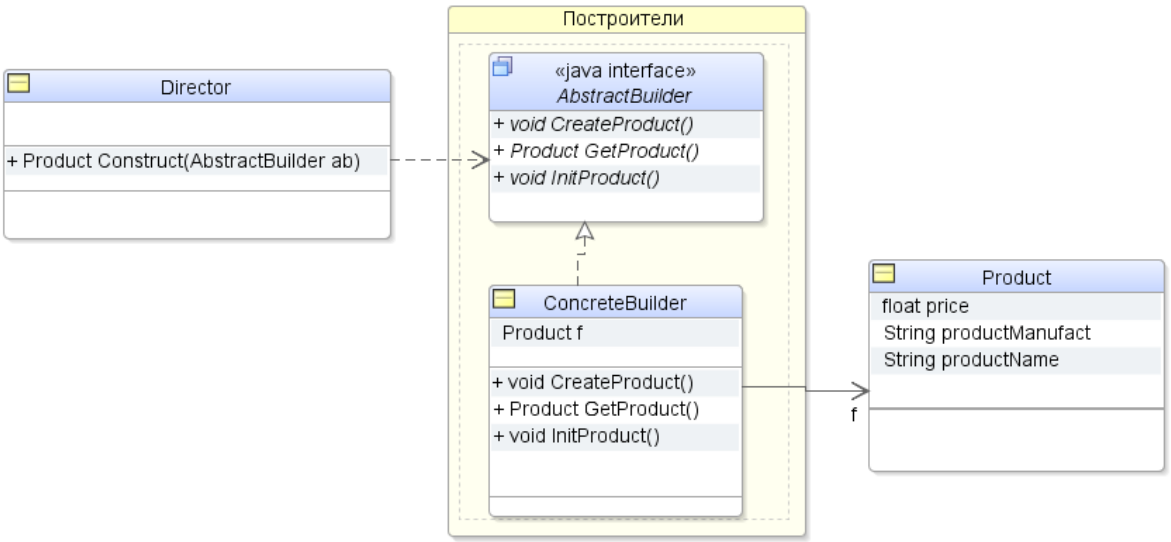


Рисунок 4.9 – Диаграмма классов, реализующих шаблон «Строитель».

Основные участники:

1. Класс AbstractBuilder – класс выполняющий роль абстрактного построителя. Задает абстрактный интерфейс для создания частей объекта Product. В данном примере реализован с использованием интерфейса, но вместо абстрактного построителя может использоваться абстрактный класс или конкретный класс, реализующий методы создания и инициализации объекта по умолчанию.

2. Класс ConcreteBuilder. Осуществляет реализацию абстрактного интерфейса, определенного в AbstractBuilder. Даный элемент конструирует и собирает вместе части продукта посредством реализации абстрактного интерфейса. Класс определяет создаваемое представление и следит за ним.

3. Класс Director – класс-распорядитель. Конструирует объект, пользуясь интерфейсом AbstractBuilder.

4. Класс Product – продукт. Класс, объект которого необходимо создать.

На рис. 4.10 представлен код, использующий данный шаблон.

```
3  public class Program {
4      public static void main(String[] args){
5          Product pr = Director.Construct(new ConcreteBuilder());
6          System.out.println(pr.productName);
7      }
8  }
```

Рисунок 4.10 – Исходный код, реализующих шаблон «Строитель».

В данном случае роль паттерна не очевидна, но на практике используют следующую реализацию данного паттерна (рис. 4.11). В данном случае в классе Decorator содержится единый код, который позволяет создавать объекты различных типов.

Для создания нового типа объектов необходимо добавить пару классов: конкретный построитель и новый класс продукта.

Данный паттерн используется в следующих случаях:

– алгоритм создания сложного объекта не должен зависеть от того, из каких частей состоит объект и как они стыкуются между собой;

– процесс конструирования должен обеспечивать различные представления конструируемого объекта.

Последовательность действий по реализацию данного шаблона:

1. Клиент создает объект-распорядитель Director и конфигурирует его нужным объектом-строителем Builder (конкретным классом построителя).

2. Распорядитель уведомляет строителя о том, что нужно построить очередную часть продукта.

3. Строитель обрабатывает запросы распорядителя и добавляет новые части к продукту, а клиент забирает продукт у строителя.

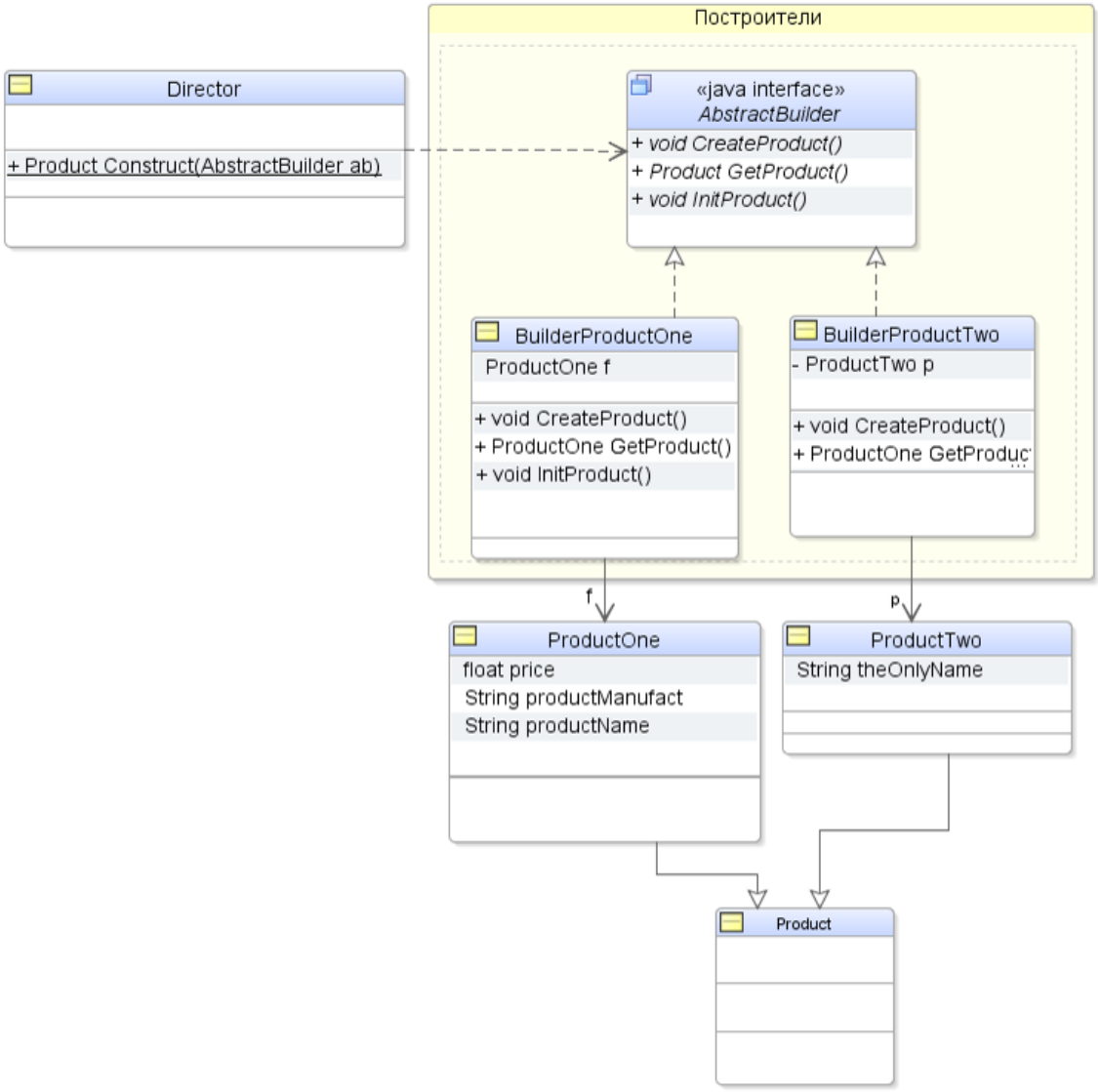


Рисунок 4.11 – Использование паттерна «Строитель» для создания различных сложных объектов.

Особенности, которые необходимо учитывать при использовании данного шаблона (см. рис. 4.9):

1. Позволяет изменять внутреннее представление продукта. Объект AbstractBuilder предоставляет распорядителю абстрактный интерфейс для конструирования продукта, за которым он может скрыть представление и

внутреннюю структуру продукта, а также процесс его сборки. Поскольку продукт конструируется через абстрактный интерфейс, то для изменения внутреннего представления достаточно всего лишь определить новый вид строителя ConcreteBuilder.

2. Изолирует код, реализующий конструирование и представление. Паттерн строитель улучшает модульность, инкапсулируя способ конструирования и представления сложного объекта. Клиентам ничего не надо знать о классах, определяющих внутреннюю структуру продукта, они отсутствуют в интерфейсе строителя.

3. Каждый конкретный строитель ConcreteBuilder содержит весь код, необходимый для создания и сборки конкретного вида продукта. Код пишется только один раз, после чего разные распорядители могут использовать его повторно для построения вариантов продукта из одних и тех же частей.

4. Дает более тонкий контроль над процессом конструирования. В отличие от порождающих паттернов, которые сразу конструируют весь объект целиком, строитель делает это шаг за шагом под управлением распорядителя. И лишь когда продукт завершен, распорядитель забирает его у строителя.

Поэтому интерфейс строителя в большей степени отражает процесс конструирования продукта, нежели другие порождающие паттерны. Это позволяет обеспечить более тонкий контроль над процессом конструирования, а значит, и над внутренней структурой готового продукта.

При практической реализации паттерна Builder обычно существует абстрактный класс AbstractBuilder, в котором определены операции для каждого компонента, который распорядитель может «попросить» создать.

По умолчанию эти операции ничего не делают. Но в классе конкретного строителя ConcreteBuilder они замещены для тех компонентов, в создании которых он принимает участие.

Строители конструируют свои продукты шаг за шагом. Поэтому интерфейс класса AbstractBuilder должен быть достаточно общим, чтобы обеспечить конструирование при любом виде конкретного строителя.

В типичном случае продукты, изготавливаемые различными строителями, имеют настолько разные представления, что изобретение для них общего родительского класса не представляется возможным. Поскольку клиент обычно конфигурирует распорядителя подходящим конкретным строителем, то, ему известно, какой именно подкласс класса AbstractBuilder используется и как нужно обращаться с произведенными продуктами.

4.4.3 Паттерн «Фабричный метод» (Method Factory)

Данный паттерн направлен на порождение классов. Определяет интерфейс для создания объекта, но оставляет подклассам решение о том, какой класс инстанцировать. Фабричный метод позволяет классу делегировать инстанцирование подклассам.

Каркасы пользуются абстрактными классами для определения и поддержания отношений между объектами. Кроме того, каркас часто отвечает за создание самих объектов.

В системе часто требуется создавать объекты самых разных типов. Паттерн Factory Method применяют при решении следующих задач:

1. Система должна оставаться расширяемой путем добавления объектов новых типов. Непосредственное использование выражения new является нежелательным, так как в этом случае код создания объектов с указанием конкретных типов может получиться разбросанным по всему приложению. Такие операции как добавление в систему объектов новых типов или замена объектов одного типа на другой будут затруднительными. Паттерн Factory Method позволяет системе оставаться независимой как от самого процесса порождения объектов, так и от их типов.

2. Заранее известно, когда нужно создавать объект, но неизвестен его тип.

При практической реализации паттерна, программист строит иерархию классов продуктов (рис. 4.12).

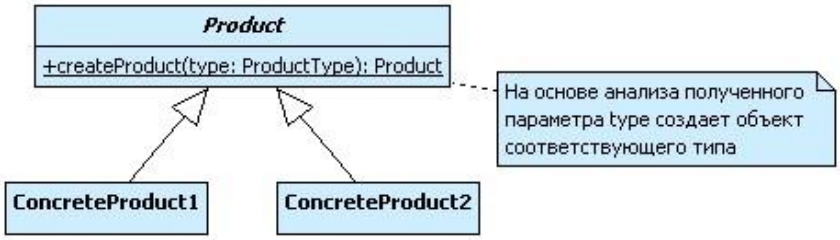


Рисунок 4.12 – Иерархия классов инстанцируемых объектов.

Классический вариант фабричного метода, когда интерфейс фабричных методов объявляется в независимом классе-фабрике, а их реализация определяется конкретными подклассами этого класса (рис. 4.13).

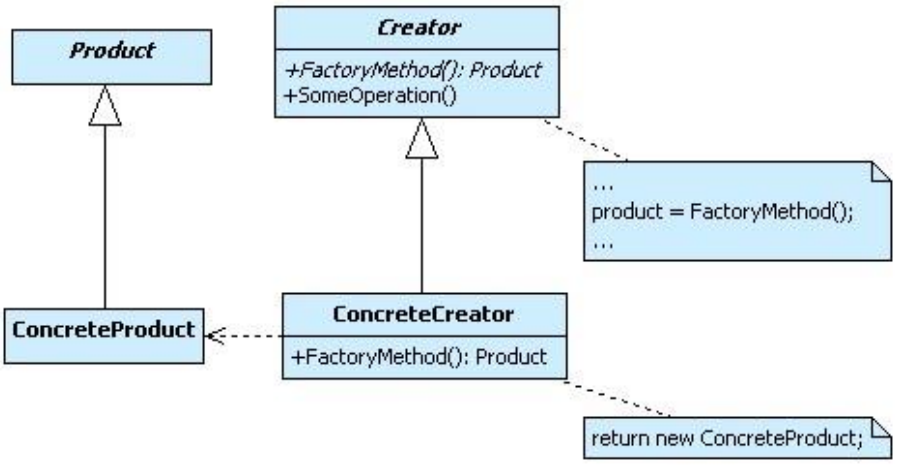


Рисунок 4.13 – Реализация паттерна «Фабричный метод».

Основные участники диаграммы:

1. Product – класс продукта, определяющий интерфейс объектов, создаваемых фабричным методом.

2. ConcreteProduct – конкретный продукт, реализующий интерфейс Product.

3. Creator – создатель. Класс объявляет фабричный метод, возвращающий объект типа Product. Creator может также определять реализацию по умолчанию фабричного метода, который возвращает объект ConcreteProduct. В классе может вызываться фабричный метод для создания объекта Product.

4. ConcreteCreator – конкретный создатель. Класс замещает фабричный метод, возвращающий объект ConcreteProduct.

Фабричные методы избавляют проектировщика от необходимости встраивать в код зависящие от приложения классы. Код имеет дело только с интерфейсом класса Product, поэтому он может работать с любыми определенными пользователями классами конкретных продуктов.

Недостаток фабричного метода состоит в том, что клиентам придется создавать подкласс класса Creator для создания лишь одного объекта ConcreteProduct. Порождение подклассов оправдано, если клиенту приходится создавать подклассы Creator, в противном случае клиенту придется иметь дело с дополнительным уровнем подклассов.

Применение паттерна «Фабричный метод» оправдано в следующих случаях:

1. Предоставляет подклассам операции-зацепки (hooks). Создание объектов внутри класса с помощью фабричного метода всегда оказывается более гибким решением, чем непосредственное создание. Фабричный метод создает в подклассах операции-зацепки для предоставления расширенной версии объекта.

2. Соединяет параллельные иерархии. Часто фабричные методы вызываюся только создателем. Но это необязательно: клиенты тоже могут применять фабричные методы, особенно при наличии параллельных иерархий классов. Параллельные иерархии возникают в случае, когда класс делегирует часть своих обязанностей другому классу, не являющемуся производным от него.

Рассмотрим, например, графические фигуры, которыми можно манипулировать интерактивно: растягивать, двигать или вращать с помощью мыши.

Реализация таких взаимодействий с пользователем – не всегда простое дело. Часто приходится сохранять и обновлять информацию о текущем состоянии манипуляций. Но это состояние нужно только во время самой манипуляции, поэтому помещать его в объект, представляющий фигуру, не следует. К тому же фигуры ведут себя по-разному, когда пользователь манипулирует ими. Например, растягивание отрезка может сводиться к изменению положения концевой точки, а растягивание текста – к изменению междустрочных интервалов. При таких ограничениях лучше использовать отдельный объект-манипулятор.

При практической реализации данного паттерна следует обращать внимание на следующие аспекты:

1. Две основные разновидности паттерна. Во-первых, это случай, когда класс Creator является абстрактным и не содержит реализации объявленного в нем фабричного метода. Вторая возможность: Creator – конкретный класс, в котором по умолчанию есть реализация фабричного метода. Редко, но встречается и абстрактный класс, имеющий реализацию по умолчанию. В первом случае для определения реализации необходимы подклассы, поскольку никакого поведения не существует. При этом обходится проблема, связанная с необходимостью инстанцировать заранее неизвестные классы. Во втором случае конкретный класс Creator использует фабричный метод, главным образом ради повышения гибкости. Выполняется правило: «Создавай объекты в отдельной операции, чтобы подклассы могли подменить способ их создания». Соблюдение этого правила гарантирует, что авторы подклассов смогут при необходимости изменить класс объектов, инстанцируемых их родителем.

2. Параметризованные фабричные методы. Это еще один вариант паттерна, который позволяет фабричному методу создавать разные виды продуктов. Фабричному методу передается параметр, который идентифицирует вид создаваемого объекта. Все объекты, получающиеся с помощью фабричного метода, разделяют общий интерфейс Product.

4.4.4 Паттерн «Прототип» (Prototype)

Паттерн «Прототип» также относится к механизмам, порождающим объекты. Он задает виды создаваемых объектов с помощью экземпляра-прототипа и создает новые объекты путем копирования этого прототипа.

На рис. 4.14 показана реализация данного паттерна. Обычно прототип используется, когда система не должна зависеть от того, как в ней создаются, компонуются и представляются продукты:

– инстанцируемые классы определяются во время выполнения, например с помощью динамической загрузки;

– для того чтобы избежать построения иерархий классов или фабрик, параллельных иерархии классов продуктов;

– экземпляры класса могут находиться в одном из не очень большого числа различных состояний. Может оказаться удобнее установить соответствующее число прототипов и клонировать их, а не инстанцировать каждый раз класс вручную в подходящем состоянии.

Следует обратить внимание, что создание объекта происходит только с использованием метода clone().

Основные участники механизма на основе паттерна «Прототип»:

1. Prototype (Graphic) – класс-прототип, предоставляющий интерфейс для клонирования самого себя.

2. Классы конкретных прототипов (Car, Helicopter) – конкретный прототип, реализующий операцию клонирования себя.

3. Client – класс-клиент, создающий новый объект, обращаясь к прототипу с запросом клонировать себя.

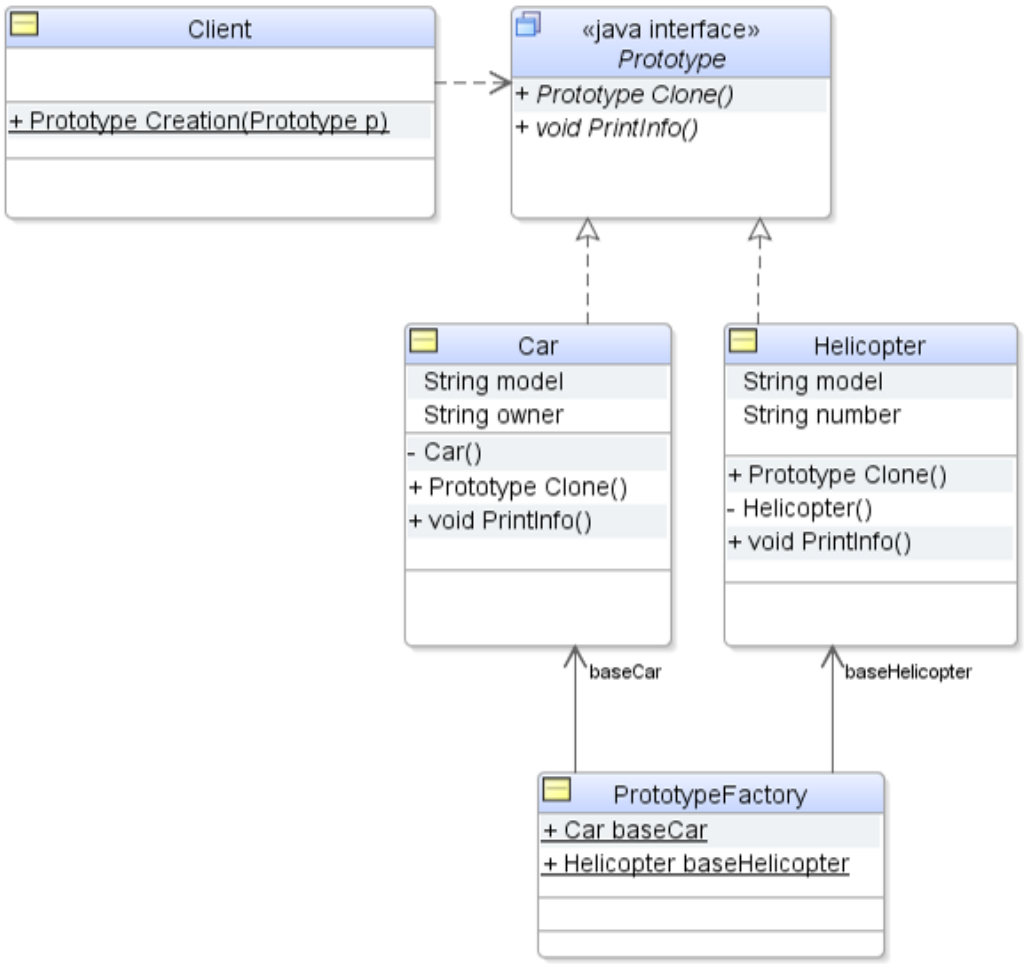


Рисунок 4.14 – Реализация паттерна «Прототип».

Преимущества паттерна:

1. Добавление и удаление продуктов во время выполнения. Прототип позволяет включать новый конкретный класс продуктов в систему, просто сообщив клиенту о новом экземпляре-прототипе. Это несколько более гибкое решение по сравнению с тем, что удастся сделать с помощью других порождающих паттернов, ибо клиент может устанавливать и удалять прототипы во время выполнения;

2. Спецификация новых объектов путем изменения значений. Динамичные системы позволяют определять поведение за счет композиции объектов – например, путем задания значений переменных объекта, – а не с

помощью определения новых классов. Фактически, определяются новые виды объектов, инстанцируя уже существующие классы и регистрируя их экземпляры как прототипы клиентских объектов. Клиент может изменить поведение, делегируя свои обязанности прототипу.

Такой дизайн позволяет пользователям определять новые классы без программирования. Фактически клонирование объекта аналогично инстанцированию класса. Паттерн прототип может резко уменьшить число необходимых системе классов.

3. Специфицирование новых объектов путем изменения структуры. Многие приложения строят объекты из крупных и мелких составляющих.

4. Уменьшение числа подклассов. Паттерн фабричный метод часто порождает иерархию классов Creator, параллельную иерархии классов продуктов. Прототип позволяет клонировать прототип, а не запрашивать фабричный метод создать новый объект. Поэтому иерархия класса Creator становится вообще ненужной.

Основные аспекты, на которые должен обращать внимание программист при реализации паттерна «Прототип»:

1. Использование диспетчера прототипов. Если число прототипов в системе не фиксировано (то есть они могут создаваться и уничтожаться динамически), необходимо вести реестр доступных прототипов. Клиенты не должны управлять прототипами самостоятельно, а сохранять и извлекать их из реестра. Клиент запрашивает прототип из реестра перед его клонированием. Диспетчер прототипов – это ассоциативное хранилище, которое возвращает прототип, соответствующий заданному ключу. В нем есть операции для регистрации прототипа с указанным ключом и отмены регистрации. Клиенты могут изменять и даже «просматривать» реестр во время выполнения, а значит, расширять систему и вести контроль над ее состоянием без написания кода.

2. Реализация операции Clone(). Самая трудная часть паттерна прототип – корректная реализация операции Clone(). Особенно сложно это в случае, когда в структуре объекта есть круговые ссылки. В большинстве языков имеется некоторая поддержка для клонирования объектов.

3. Инициализация клонов. Хотя некоторым клиентам вполне достаточно клона как такового, другим нужно инициализировать его внутреннее состояние полностью или частично. Обычно передать начальные значения операции Clone() невозможно, поскольку их число различно для разных классов прототипов. Для некоторых прототипов нужно много параметров инициализации, другие вообще ничего не требуют. Передача Clone параметров мешает построению единообразного интерфейса клонирования.

4.4.5. Применение порождающих паттернов

Есть два наиболее распространенных способа параметризовать систему классами создаваемых ей объектов. Первый способ – порождение подклассов от класса, создающего объекты. Он соответствует паттерну фабричный метод. Основной недостаток метода: требуется создавать новый подкласс лишь для того, чтобы изменить класс продукта. И таких изменений может быть очень много. Например, если создатель продукта сам создается фабричным методом, то придется замещать и создателя тоже.

Другой способ параметризации системы в большей степени основан на композиции объектов. Определяется объект, которому известно о классах объектах-продуктах, затем он делвется параметром системы. Это ключевой аспект таких паттернов, как абстрактная фабрика, строитель и прототип. Для всех трех характерно создание «фабричного объекта», который изготавливает продукты.

В абстрактной фабрике фабричный объект производит объекты разных классов. Фабричный объект строителя постепенно создает сложный продукт,

следуя специальному протоколу. Фабричный объект прототипа изготавливает продукт путем копирования объекта-прототипа. В последнем случае фабричный объект и прототип – это одно и то же, поскольку именно прототип отвечает за возврат продукта.

4.5 Структурные паттерны

В структурных паттернах рассматривается вопрос о том, как из классов и объектов образуются более крупные структуры. Структурные паттерны уровня класса используют наследование для составления композиций из интерфейсов и реализаций. Простой пример – использование множественного наследования для объединения нескольких классов в один. В результате получается класс, обладающий свойствами всех своих родителей. Особенно полезен этот паттерн, когда нужно организовать совместную работу нескольких независимо разработанных библиотек. Другой пример паттерна уровня класса – адаптер. В общем случае адаптер делает интерфейс одного класса (адаптируемого) совместимым с интерфейсом другого, обеспечивая тем самым унифицированную абстракцию разнородных интерфейсов. Это достигается за счет закрытого наследования адаптируемому классу. После этого адаптер выражает свой интерфейс в терминах операций адаптируемого класса.

4.5.1 Паттерн «Адаптер» (Adapter)

Паттерн предназначен для структурирования классов и объектов. Преобразует интерфейс одного класса в интерфейс другого, который ожидают клиенты. Адаптер обеспечивает совместную работу классов с несовместимыми интерфейсами, которая без него была бы невозможна.

Для данного паттерна используется также наименование «Обертка» (Wrapper).

На рис. 4.15 представлен состав классов, присутствующих в данном паттерне.

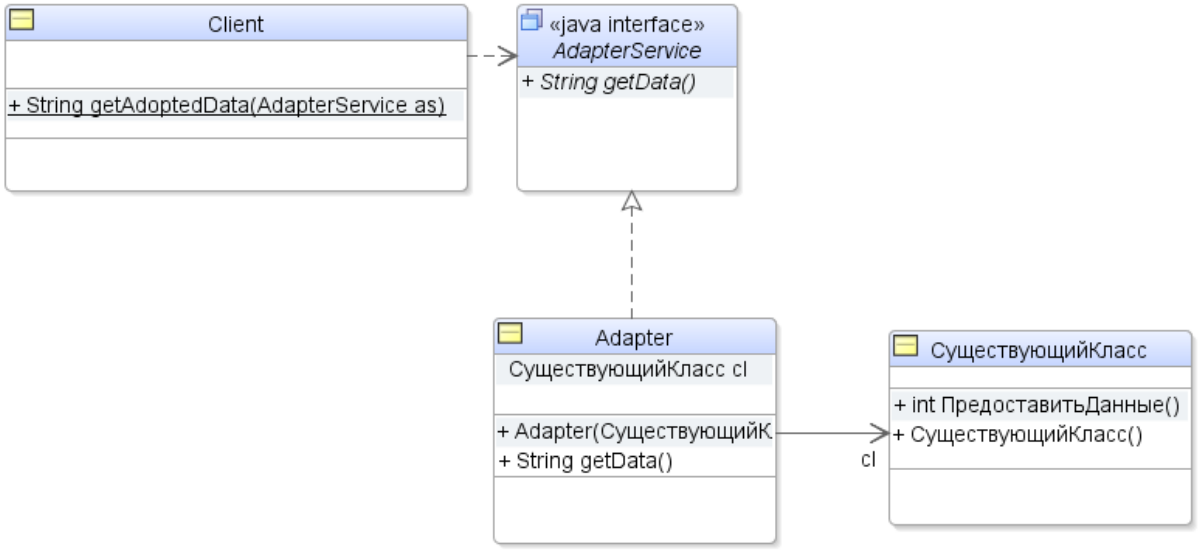


Рисунок 4.15 – Реализация паттерна «Адаптер».

Участники взаимодействия:

1. AdapterService – целевая абстракция. Оределяет зависящий от предметной области интерфейс, которым пользуется Client.

2. Client – класс-клиент. Вступает во взаимоотношения с объектами, удовлетворяющими интерфейсу AdapterService.

3. СуществующийКласс – адаптируемый класс, определяющий существующий интерфейс, который нуждается в адаптации.

4. Adapter – класс, адаптирующий интерфейс класса «СуществующийКласс» к интерфейсу AdapterService.

Результаты применения данного паттерна отличаются в зависимости от того, что именного адаптируется: класс или объект.

В случае реализации адаптера класса:

– адаптирует «СуществующийКласс» к AdapterService, перепоручая действия конкретному классу Adapter; поэтому данный паттерн не будет работать, если мы захотим одновременно адаптировать класс и его подклассы.

– позволяет адаптеру Adapter заместить некоторые операции адаптируемого класса, так как Adapter есть не что иное, как подкласс «СуществующийКласс»;

– вводит только один новый объект; Чтобы добраться до адаптируемого класса, не нужно никакого дополнительного обращения по указателю.

В случае реализации адаптера объектов:

– позволяет одному адаптеру Adapter работать со многими адаптируемыми объектами типа «СуществующийКласс», то есть с самим «СуществующийКласс» и его подклассами. Адаптер может добавить новую функциональность сразу всем адаптируемым объектам;

– затрудняет замещение операций класса «СуществующийКласс». Для этого потребуется породить от «СуществующийКласс» подкласс и заставить Adapter ссылаться на этот подкласс, а не на сам «СуществующийКласс».

При практической реализации паттерна «Адаптер» программист должен учитывать аспекты:

1. Объем работы по адаптации. Адаптеры сильно отличаются по тому объему работы, который необходим для адаптации интерфейса класса «СуществующийКласс» к интерфейсу AdapterService. Это может быть как простейшее преобразование, например изменение имен операций, так и поддержка совершенно другого набора операций. Объем работы зависит от того, насколько сильно отличаются друг от друга интерфейсы целевого и адаптируемого классов.

2. Сменные адаптеры. Степень повторной используемости класса тем выше, чем меньше предположений делается о тех классах, которые будут его применять. Встраивая адаптацию интерфейса в класс, программист отказывается от предположения, что другим классам станет доступен тот же самый интерфейс. Другими словами, адаптация интерфейса позволяет

включить класс в существующие системы, которые спроектированы для класса с другим интерфейсом.

3. Использование двусторонних адаптеров для обеспечения прозрачности. Адаптеры непрозрачны для всех клиентов. Адаптированный объект уже не обладает интерфейсом класса «СуществующийКласс», так что его нельзя использовать там, где «СуществующийКласс» был применим. Двусторонние адаптеры способны обеспечить такую прозрачность. Точнее, они полезны в тех случаях, когда клиент должен видеть объект по-разному.

4.5.2 Паттерн «Мост» (Bridge)

Данный паттерн предполагает структуризацию объектов. Основное его назначение – разделение абстракции от ее реализации, причем обе части можно изменять независимо.

Если для некоторой абстракции возможно несколько реализаций, то обычно применяют наследование. Абстрактный класс определяет интерфейс абстракции, а его конкретные подклассы по-разному реализуют его. Но такой подход не всегда обладает достаточной гибкостью. Наследование жестко привязывает реализацию к абстракции, что затрудняет независимую модификацию, расширение и повторное использование абстракции и ее реализации.

Рассмотрим пример. На рис. 4.16 представлена иерархия классов, разрабатываемого игрового приложения. Существуют классы Orc, Elf, Human (представляют игровые расы). Для каждого класса потребовалось определить по три подкласса, определяющих игровые навыки: Archer, Wizard, Warrior. На данном этапе проектирования система является устойчивой, но в какое состояние перейдет приложение, если понадобиться добавить еще несколько игровых рас и несколько игровых навыков ждя каждой расы? Очевидно, что количество классов начнет расти лавинообразно.

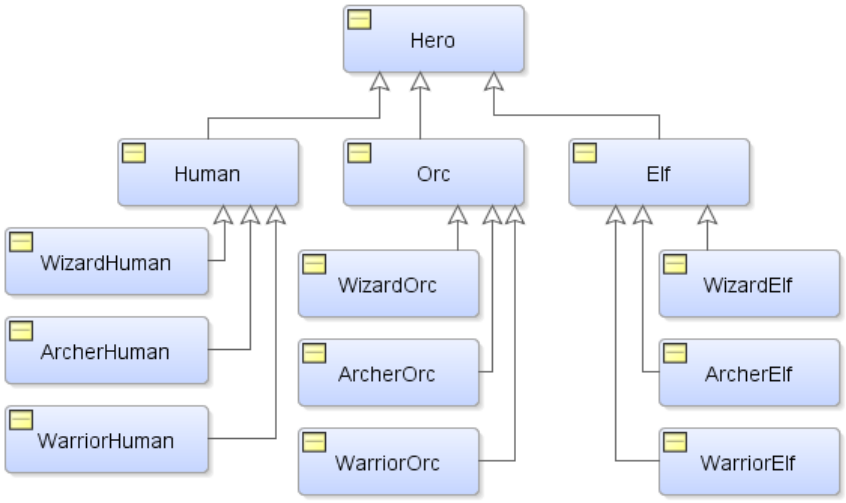


Рисунок 4.16 – Иерархия классов компьютерной игры.

Для преодоления недостатков подхода целесообразно применение паттерна «Мост» (Bridge).

Паттерн Bridge разделяет абстракцию и реализацию на две отдельные иерархии классов так, что их можно изменять независимо друг от друга.

Первая иерархия определяет интерфейс абстракции, доступный пользователю. Для случая проектируемого нами логгера абстрактный базовый класс Hero мог бы объявить интерфейс метода Draw() для отображения персонажа. Класс Hero также содержит указатель на реализацию BaseImplementation, который инициализируется должным образом при создании героя конкретной расы. Этот указатель используется для перенаправления пользовательских запросов в реализацию. Необходимо отметить, что в общем случае подклассы Orc, Elf и Human могут расширять интерфейс класса Hero.

Все детали реализации, связанные с особенностями поведения (навыками) скрываются во второй иерархии классов. Базовый класс BaseImplementation объявляет интерфейс операций, предназначенных для рисования сущностей типа Orc, Elf, Human, а подклассы WizardImplementation, WarriorInplementation и ArcherImplementation реализуют этот интерфейс для различных навыков. В общем случае,

интерфейс BaseImplementation необязательно должен в точности соответствовать интерфейсу абстракции. Иногда вместо интерфейса BaseImplementation используется класс, содержащий базовую реализацию.

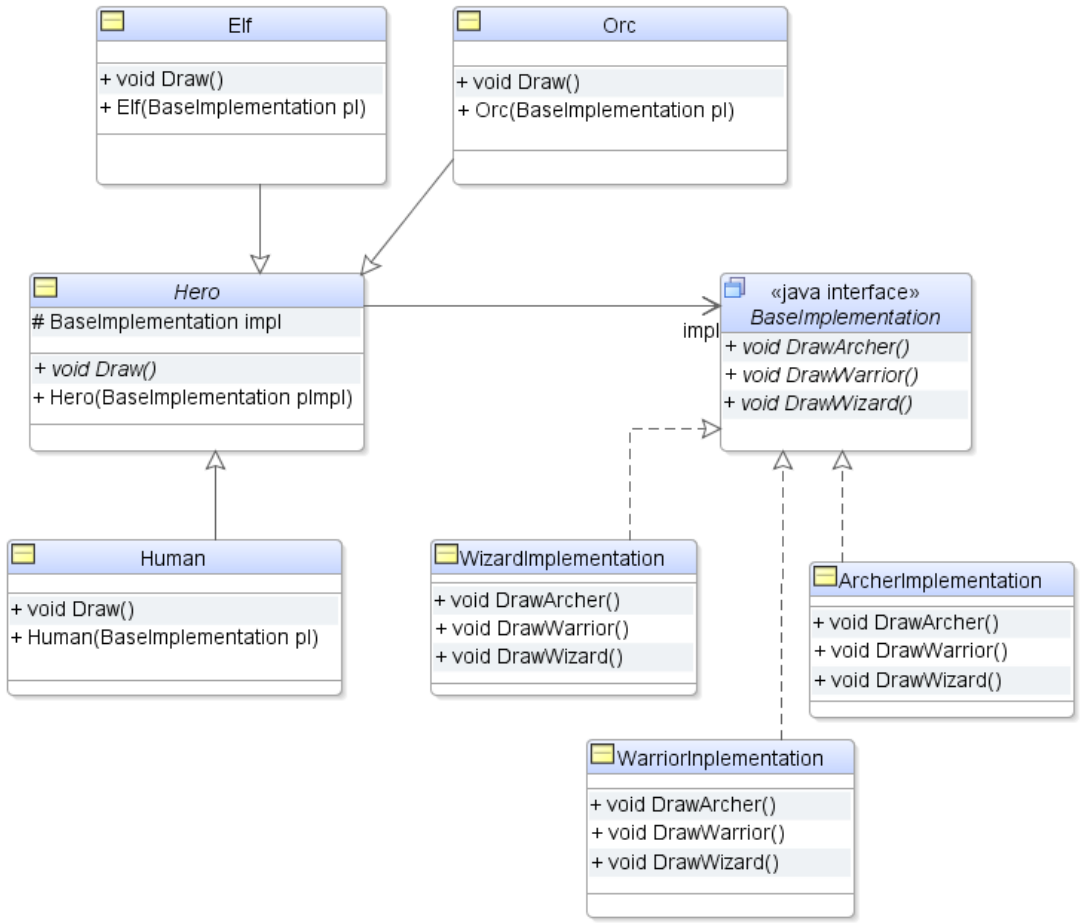


Рисунок 4.17 – Иерархия классов компьютерной игры, после рефакторинга с использованием шаблона «Мост».

Основные участники паттерна:

1. Класс абстракции (Hero) – определяет интерфейс абстракции; хранит ссылку на объект ркализации (BaseImplementation);

2. Уточненная абстракция (классы Human, Orc, Elf) – расширяет интерфейс, определенный абстракцией Hero.

3. Абстрактный реализатор (BaseImplementation) – определяет интерфейс для классов реализации. Он не обязан точно соответствовать интерфейсу класса Hero. На самом деле оба интерфейса могут быть совершенно различны. Обычно интерфейс класса BaseImplementation

предоставляет только примитивные операции, а класс Hero определяет операции более высокого уровня, базирующиеся на этих примитивах.

4. Конкретные реализации (WizardImplementation, WarriorInplementation и ArcherImplementation) – содержат конкретную реализацию интерфейса класса BaseImplementation.

Таким образом, объект Hero просто направляет своему объекту BaseImplementation запросы клиента. Клиент использует только интерфейс класса Hero.

Результаты применения паттерна Bridge:

1. Отделение реализации от интерфейса. Реализация больше не имеет постоянной привязки к интерфейсу. Реализацию абстракции можно конфигурировать во время выполнения. Объект может даже динамически изменять свою реализацию. Разделение классов Hero и BaseImplementation устраняет также зависимости от реализации, устанавливаемые на этапе компиляции. Чтобы изменить класс реализации, вовсе не обязательно перекомпилировать класс Hero и его клиентов. Это свойство особенно важно, если необходимо обеспечить двоичную совместимость между разными версиями библиотеки классов.

Кроме того, такое разделение облегчает разбиение системы на слои и тем самым позволяет улучшить ее структуру. Высокоуровневые части системы должны знать только о двух классах Hero и BaseImplementation.

2. Повышение степени расширяемости. Можно расширять независимо иерархии классов Hero и BaseImplementation (то есть добавить произвольное количество игровых рас или типологий поведения).

3. Сокрытие деталей реализации от клиентов. Клиентов можно изолировать от таких деталей реализации, как разделение объектов класса BaseImplementation и сопутствующего механизма подсчета ссылок.

Программисту необходимо учитывать следующие аспекты при практическом использовании паттерна Bridge:

1. Только один класс BaseImplementation. В ситуациях, когда есть только одна реализация, создавать абстрактный класс BaseImplementation необязательно. Это вырожденный случай паттерна мост – между классами Hero и BaseImplementation существует взаимно-однозначное соответствие. Тем не менее разделение все же полезно, если нужно, чтобы изменение реализации класса не отражалось на существующих клиентах (должно быть достаточно заново скомпоновать программу, не перекомпилируя клиентский код). Это позволяет полностью скрыть реализацию класса от клиентов.

2. Создание правильного объекта BaseImplementation. Как, когда и где принимается решение о том, какой из нескольких классов BaseImplementation инстанцировать? Если у класса Hero есть информация о конкретных классах-потомках BaseImplementation, то он может инстанцировать один из них в своем конструкторе; какой именно – зависит от переданных конструктору параметров.

4.5.3 Паттерн «Компоновщик» (Composite)

Паттерн предназначен для структурирования объектов. Использование паттерна позволяет скомпоновать объекты в древовидные структуры для представления иерархий часть-целое. Использование данного приема позволяет клиентам единообразно трактовать индивидуальные и составные объекты.

Диаграмма данного паттерна представлена на рис. 4.18

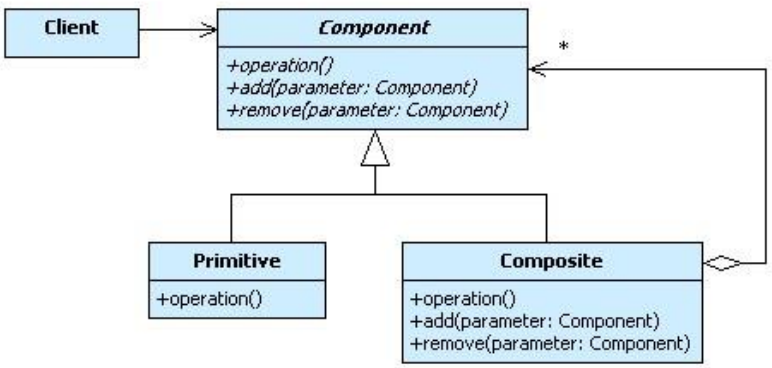


Рисунок 4.18 – Концептуальная схема реализации паттерна «Компоновщик».

Основные элементы, принимающие участие в построении паттерна:

1. Component – компонент. Объявляет интерфейс для компонуемых объектов; предоставляет подходящую реализацию операций по умолчанию, общую для всех классов; объявляет интерфейс для доступа к потомкам и управления ими; определяет интерфейс для доступа к родителю компонента в рекурсивной структуре и при необходимости реализует его (данная возможность необязательна).

2. Primitive – лист дерева. Представляет листовые узлы композиции и не имеет потомков; Класс определяет поведение примитивных объектов в композиции.

3. Composite – составной объект. Данный класс определяет поведение компонентов, у которых есть потомки. Он хранит компоненты-потомки и реализует относящиеся к управлению потомками операции в интерфейсе клдсса Component.

4. Client – класс-клиент. Манипулирует объектами композиции через интерфейс Component.

Рассмотрим пример использования данного паттерна.

На рис. 4.19 представлена диаграмма классов приложения Java, описывающая следующую предметную область: в игровом приложении существуют игровые единицы (класс Unit), которые характеризуются оружием (класс Weapon) и броней (класс Armor). Таким образом, класс Unit – агрегат, который заключает в себе дочерние объекты классов Weapon и Armor. Для упрощения модели будем считать, что классы Weapon, Armor и Unit не имеют подклассов. Отдельные игровые единицы могут объединяться в группы (класс Group).

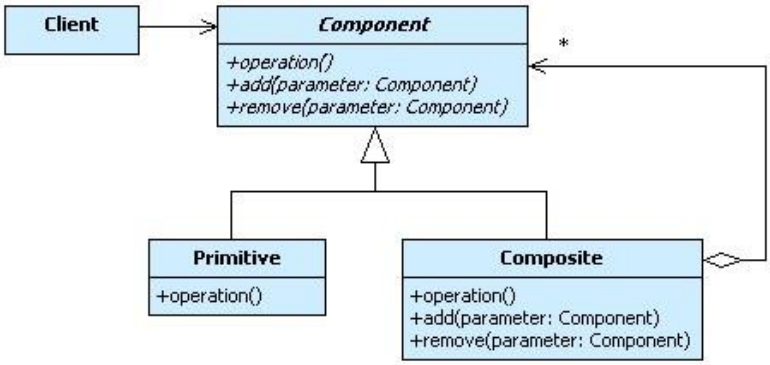


Рисунок 4.19 – Концептуальная схема реализации паттерна «Компоновщик».

Таким образом, в предметной области было выделено четыре различных класса, объекты которых реализуют иерархию. Необходимо эффективно управлять данной иерархией. Например, нужно унифицированным образом определять урон, наносимый отдельным предметом оружия, игровым персонажем или группой, а также определять показатели защиты для этих же иерархических сущностей.

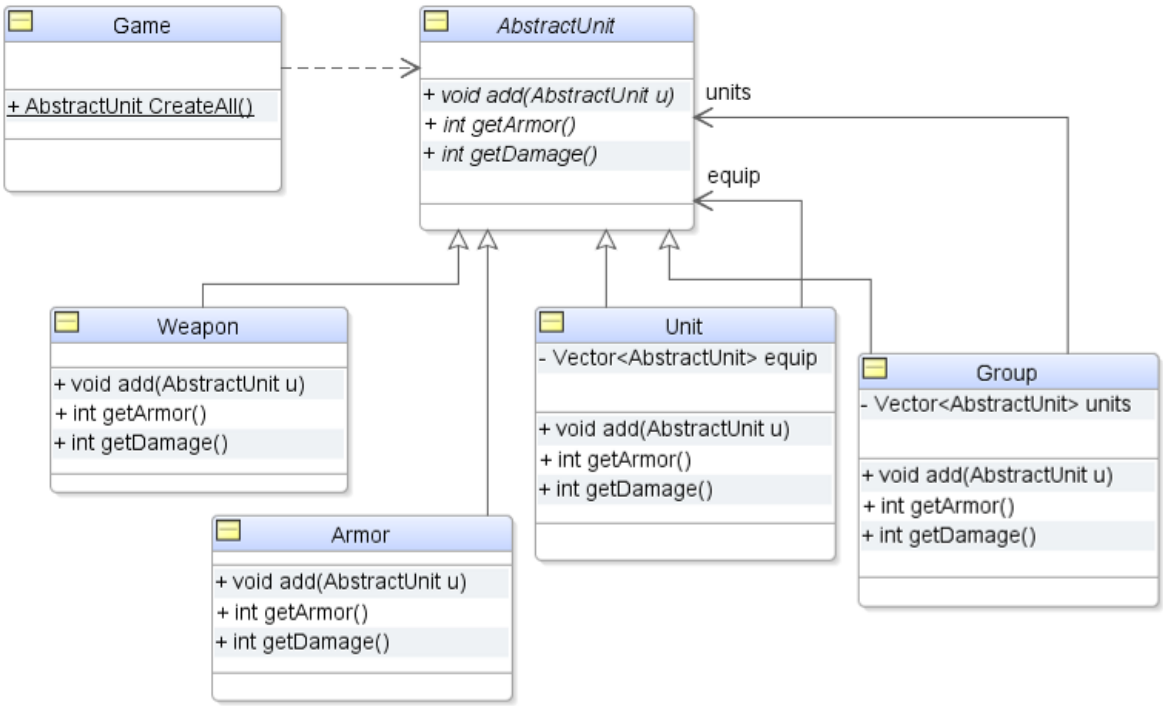


Рисунок 4.20 – Концептуальная схема реализации паттерна «Компоновщик».

На рис. 4.20 представлена диаграмма классов приложения, связывающая сущности предметной области. На рис. 4.21 класс Armor

реализует интерфейс абстракции AbstractUnit специфичным для него образом: так как сущность представляет защиту, то метод getDamage() возвращает 0, а метод getArmor() – значение 10.

```
public class Armor extends AbstractUnit {
    @Override
    public void add(AbstractUnit u) {
    }
    @Override
    public int getArmor() {
        return 10;
    }
    @Override
    public int getDamage() {
        return 0;
    }
}
```

Рисунок 4.21 – Класс Armor: представляет сущность «Броня» предметной области.

Метод add() – пустой, чтобы предотвратить добавление дочерних сущностей. На рис. 4.22 представлена реализация класса Weapon. Концепция реализация методов абстракции схожа с классом Armor.

```
public class Weapon extends AbstractUnit {
    @Override
    public void add(AbstractUnit u) {
    }
    @Override
    public int getArmor() {
        return 0;
    }
    @Override
    public int getDamage() {
        return 2;
    }
}
```

Рисунок 4.22 – Класс Weapon: представляет сущность «Оружие» предметной области.

Рис. 4.23 демонстрирует реализацию класса Unit.

```
4
public class Unit extends AbstractUnit {
6       private Vector<AbstractUnit> equip =
7           new Vector<AbstractUnit>();
8       @Override
        public void add(AbstractUnit u) {
10          equip.add(u);
11      }
12
13      @Override
        public int getArmor() {
15          int res = 0;
16          for(AbstractUnit a : equip){
17              res += a.getArmor();
18          }
19          return res;
20      }
21
22      @Override
        public int getDamage() {
24          int res = 0;
25          for(AbstractUnit w : equip){
26              res += w.getDamage();
27          }
28          return res;
29      }
30  }
```

Рисунок 4.23 – Класс Unit: представляет сущность «Игровая единица» предметной области.

Методы getArmor() и getWeapon(), реализующие интерфейс обстракции вычисляют суммарную защиту и суммарный урон игровой единицы соответственно. Расчет производится суммированием показателей дочерних элементов, которые хранятся в коллекции equip. В данном классе реализован метод add() – он позволяет добавлять игровому персонажу оружие и броню.

На рис. 4.24 рассматривается класс Group. Реализация данного класса практически не отличается от Unit. Обе сущности представляют собой контейнеры для содержания других сущностей.

```
public class Group extends AbstractUnit {
    private Vector<AbstractUnit> units =
        new Vector<AbstractUnit>();

    @Override
    public void add(AbstractUnit u) {
        units.add(u);
    }

    @Override
    public int getArmor() {
        int res = 0;
        for(AbstractUnit u : units) {
            res += u.getArmor();
        }
        return res;
    }

    @Override
    public int getDamage() {
        int res = 0;
        for(AbstractUnit u : units) {
            res += u.getDamage();
        }
        return res;
    }
}
```

Рисунок 4.24 – Класс Group: представляет сущность «Группа игровых персонажей» предметной области.

Класс, который играет роль клиента в паттерне Composite, представлен на рис. 4.25. Единственный метод данного класса создает множество игровых объектов. Реализацию данного метода можно улучшить, если применить совместно с данным паттерном паттерн «Метод фабрики», «Абстрактная фабрика» или другой паттерн, позволяющий создавать новые объекты.

```
5  public class Game {
6
7      public static AbstractUnit CreateAll(){
8          // Создаем корневую группу
9          AbstractUnit res = new Group();
10
11         // Создаем 100 персонажей с оружием и броней
12         for(int i = 0; i<100; i++){
13             Unit n = new Unit();
14             n.add(new Armor()); n.add(new Weapon());
15             res.add(n);
16         }
17
18         // Создаем 10 групп по 10 юнитов
19         for(int i = 0; i<10; i++){
20             Group g = new Group();
21             for(int o = 0; o<10; o++){
22                 Unit n = new Unit();
23                 n.add(new Armor()); n.add(new Weapon());
24                 g.add(n);
25             }
26             res.add(g);
27         }
28         return res;
29     }
30 }
```

Рисунок 4.25 – Класс Game: представляет класс-клиент для использования возможностей паттерна Composite.

На рис. 4.26 показан код функции main().

```
3  public class Program {
4      public static void main(String[] args){
5          AbstractUnit army = Game.CreateAll();
6          System.out.println("Суммарный урон: "
7                             + army.getDamage());
8          System.out.println("Суммарная защита: "
9                             + army.getArmor());
10     }
11 }
```

Рисунок 4.26 – Функция main().

Рисунок 4.27 демонстрирует результат работы программы.

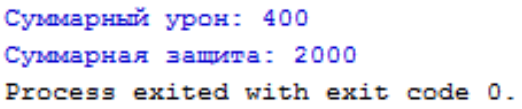


Рисунок 4.27 – Консольный вывод приложения.

Клиенты (класс Game) используют интерфейс класса Component (класс AbstractUnit) для взаимодействия с объектами в составной структуре. Если получателем запроса является листовый объект Primitive (в примере это классы Weapon и Armor), то он и обрабатывает запрос. Когда же получателем является составной объект Composite (классы Unit и Group), то обычно он делегирует запрос своим потомкам, возможно, выполняя некоторые дополнительные операции до или после перенаправления.

Результаты применения паттерна «Компоновщик»:

1. Определяет иерархии классов, состоящие из примитивных и составных объектов. Из примитивных объектов можно составлять более сложные, которые, в свою очередь, участвуют в более сложных композициях и так далее. Любой клиент работает как с примитивным, так и с составным объектом.

2. Упрощает архитектуру клиента. Клиенты могут единообразно работать с индивидуальными и объектами и с составными структурами. Обычно клиенту неизвестно, взаимодействует ли он с листовым или составным объектом. Это упрощает код клиента, поскольку нет необходимости писать функции, ветвящиеся в зависимости от того, с объектом какого класса они работают.

3. Облегчает добавление новых видов компонентов. Новые подклассы классов Composite или Primitive будут автоматически работать с уже существующими структурами и клиентским кодом. Изменять клиента при добавлении новых компонентов не нужно. Это означает, что в рассмотренном примере можно добавить различные виды игровых

персонажей, любое количество игровых предметов. Главное – они должны реализовать абстракцию AbstractUnit.

4. Способствует созданию общего дизайна. Однако такая простота добавления новых компонентов имеет и свои отрицательные стороны: становится трудно наложить ограничения на то, какие объекты могут входить в состав композиции. Иногда желательно, чтобы составной объект мог включать только определенные виды компонентов. Паттерн компоновщик не позволяет воспользоваться для реализации таких ограничений статической системой типов. Вместо этого следует проводить проверки во время выполнения.

Программист, использующий данный паттерн, должен обращать внимание на следующие аспекты реализации:

1. Явные ссылки на родителей. Хранение в компоненте ссылки на своего родителя может упростить обход структуры и управление ею. Наличие такой ссылки облегчает передвижение вверх по структуре и удаление компонента. Кроме того, ссылки на родителей помогают поддержать паттерн цепочка обязанностей.

Обычно ссылку на родителя определяют в классе Component. Классы Primitive и Composite могут унаследовать саму ссылку и операции с ней. При наличии ссылки на родителя важно поддерживать следующий инвариант: если некоторый объект в составной структуре ссылается на другой составной объект как на своего родителя, то для последнего первый является потомком. Простейший способ гарантировать соблюдение этого условия – изменять родителя компонента только тогда, когда он добавляется или удаляется из составного объекта. Если это удается один раз реализовать в операциях Add и Remove, то реализация будет унаследована всеми подклассами и, значит, инвариант будет поддерживаться автоматически.

2. Разделение компонентов. Часто бывает полезно разделять компоненты, например для уменьшения объема занимаемой памяти. Но если

у компонента может быть более одного родителя, то разделение становится проблемой.

Возможное решение – позволить компонентам хранить ссылки на нескольких родителей. Однако в таком случае при распространении запроса по структуре могут возникнуть неоднозначности. Паттерн «Приспособленец» показывает, как следует изменить дизайн, чтобы вовсе отказаться от хранения родителей. Работает он в тех случаях, когда потомки могут не посылать сообщений своим родителям, вынеся за свои границы часть внутреннего состояния.

3. Максимизация интерфейса класса Component. Одна из целей паттерна «Компоновщик» – избавить клиентов от необходимости знать, работают ли они с листовым или составным объектом. Для достижения этой цели класс Component должен сделать как можно больше операций общими для классов Composite и Primitive. Обычно класс Component предоставляет для этих операций реализации по умолчанию, а подклассы Composite и Primitive замещают их. Например, в рассматриваемом примере в абстрактном интерфейсе можно было бы реализовать метод CreateUnit() с параметром, указывающим, какой конкретно экземпляр требуется создать.

4. Объявление операций для управления потомками. Хотя в классе Composite обычно реализованы операции add() и remove() для добавления и удаления потомков, но для паттерна компоновщик важно, в каких классах эти операции объявлены. Надо ли объявлять их в классе Component и тем самым делать доступными в Primitive, или их следует объявить и определить только в классе Composite и его подклассах? В данном случае программист должен выбрать между прозрачностью и безопасностью приложения.

4.5.4 Паттерн «Декоратор» (Decorator)

Основным назначением данного паттерна также является структурирование объектов. Назначение паттерна: динамически добавлять

объекту новые обязанности. Фактически, паттерн предоставляет разработчику альтернативу порождению подклассов. Иногда этот паттрен называется «Обертка» (Wrapper).

На рис. 4.28 представлена диаграмма, демонстрирующая основных участников паттерна.

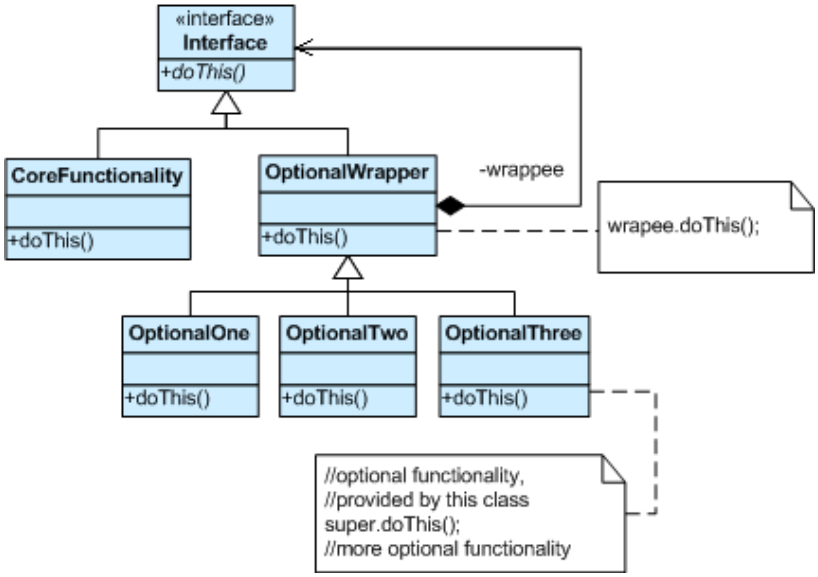


Рисунок 4.28 – Основные элементы, принимающие участие в реализации паттерна «Декоратор».

Иногда бывает нужно возложить дополнительные обязанности на отдельный объект, а не на класс в целом. Так, например, библиотека для построения графиков должна иметь возможность представлять данные разными способами: столбиковая, круговая диаграммы или график. Одновременно можно добавить возможность вывода исходных данных, задания различных линий, заливки.

Добавить новые обязанности допустимо с помощью наследования. При наследовании класса гистограммы, данные будут выводиться в виде столбиковой диаграммы. Однако это решение статическое, а значит, недостаточно гибкое. Клиент не может управлять оформлением компонента.

Более гибким является другой подход: поместить компонент в другой объект, называемый декоратором, который как раз и добавит оформление для

данных. Декоратор следует интерфейсу декорируемого объекта, поэтому его присутствие прозрачно для клиентов компонента.

Основные участники паттерна (рис. 4.28):

1. Interface – компонент, определяющий интерфейс для объектов, на которые могут быть динамически возложены дополнительные обязанности.

2. CoreFunctionality – конкретный компонент, определяющий объект, на который возлагаются дополнительные обязанности.

3. OptionalWrapper – класс декоратора. Хранит ссылку на объект Interface и определяет интерфейс, соответствующий интерфейсу Interface.

4. OptionalOne, OptionalTwo, OptionalThree – конкретные декораторы. Именно эти классы возлагают дополнительные обязанности на компонент.

Рассмотрим диаграмму классов, демонстрирующую использование паттерна «Декоратор» для предметной области: рисование графиков функций (рис. 4.29).

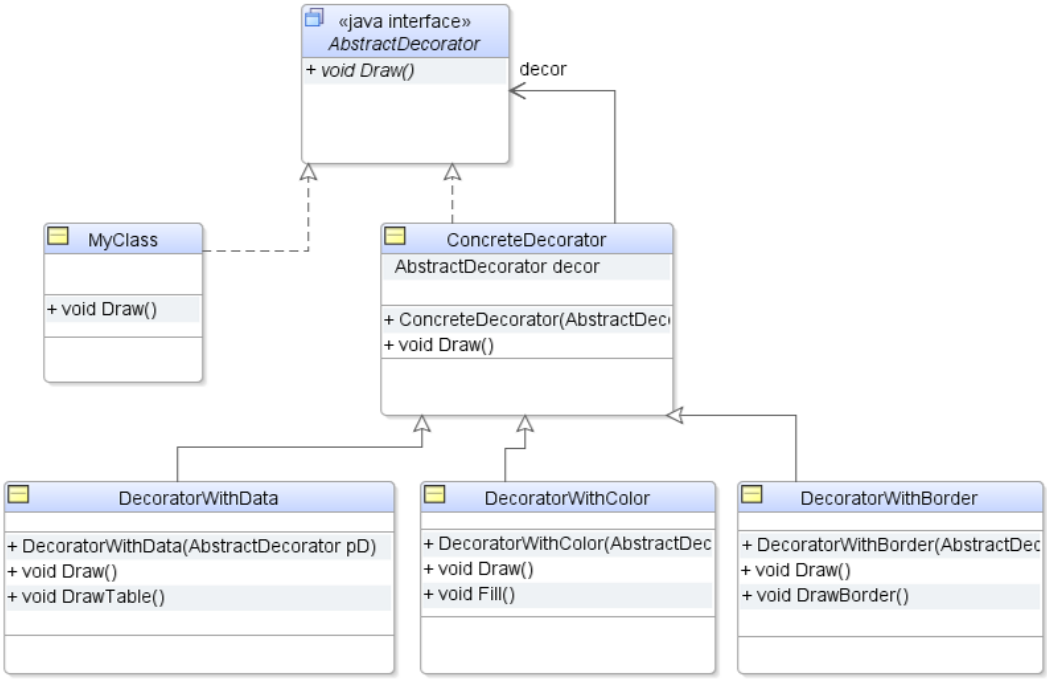


Рисунок 4.29 – Диаграмма классов для реализации паттерна «Декоратор».

На рис. 4.30 представлен листинг класса ConcreteDecorator.

```
public class ConcreteDecorator implements AbstractDecorator {

    AbstractDecorator decor;

    public ConcreteDecorator(AbstractDecorator pD){
        decor = pD;
    }

    @Override
    public void Draw() {
        decor.Draw();
        System.out.print(". Добавляем заголовок");
    }
}
```

Рисунок 4.30 – Класс ConcreteDecorator.

Класс добавляет объекту типа AbstractDecorator, передаваемому в качестве параметра, дополнительный функционал – вывод заголовка. Это расширение осуществляется за счет переопределения метода Draw().

На рис. 4.31 представлен класс DecoratorWithColor, который должен добавлять цветовое оформление для диаграммы.

```
public class DecoratorWithColor extends ConcreteDecorator {
    public DecoratorWithColor(AbstractDecorator pD){
        super(pD);
    }

    @Override
    public void Draw(){
        decor.Draw();
        this.Fill();
    }

    public void Fill(){
        System.out.print(". Раскрасили диаграмму");
    }
}
```

Рисунок 4.31 – Класс DecoratorWithColor.

В классах, производных от ConcreteDecorator могут быть реализованы любые методы, расширяющие возможности декоратора. В классе

DecoratorWithColor переопределен метод Draw(), что позволяет добавлять цветовое оформление диаграмме.

Классы DecoratorWithData, DecoratorWithBorder реализуются аналогично классу DecoratorWithColor. Наибольший интерес представляет использование полученного механизма. На рис. 4.32 представлен исходный код приложения.

```
3  public class Program {
4      public static void main(String[] args){
5          AbstractDecorator simple = new MyClass();
6          AbstractDecorator title = new ConcreteDecorator(new MyClass());
7          AbstractDecorator color = new DecoratorWithColor(new MyClass());
8          AbstractDecorator borderAndTitle = new ConcreteDecorator(
9                                                  new DecoratorWithBorder(
10                                                     new MyClass()));
11         AbstractDecorator all = new DecoratorWithData(
12                                     new DecoratorWithColor(
13                                         new DecoratorWithBorder(new MyClass())));
14         simple.Draw();
15         title.Draw();
16         color.Draw();
17         borderAndTitle.Draw();
18         all.Draw();
19         System.out.println();
20     }
21 }
```

Рисунок 4.32 – Метод main().

Из приведенного фрагмента видно, что паттерн «Декоратор» позволяет осуществлять добавление функционала целевому классу путем его вложения в классы-декораторы, которых может быть как угодно много.

На рис. 4.33 показан результат работы программы.

```
Некрасивая гистограмма
Некрасивая гистограмма. Добавляем заголовок
Некрасивая гистограмма. Раскрасили диаграмму
Некрасивая гистограмма. Дорисовали рамку. Добавляем заголовок
Некрасивая гистограмма. Дорисовали рамку. Раскрасили диаграмму. Дорисовали исходные данные
Process exited with exit code 0.
```

Рисунок 4.33 – Консольный вывод приложения.

В результате применения паттерна «Декоратор» обеспечиваются:

1. Большая гибкость, нежели у статического наследования. Паттерн декоратор позволяет более гибко добавлять объекту новые обязанности, чем

было бы возможно в случае статического (множественного) наследования. Декоратор может добавлять и удалять обязанности во время выполнения программы. При использовании же наследования требуется создавать новый класс для каждой дополнительной обязанности , что ведет к увеличению числа классов и, как следствие, к возрастанию сложности системы. Кроме того, применение нескольких декораторов к одному компоненту позволяет произвольным образом сочетать обязанности.

Декораторы позволяют легко добавить одно и то же свойство дважды. Например, чтобы окружить объект MuClass двойной рамкой, нужно просто добавить два декоратора DecoratorWithBorder. Двойное наследование класса DecoratorWithBorder невозможно.

2. Позволяет избежать перегруженных функциями классов на верхних уровнях иерархии. Декоратор разрешает добавлять новые обязанности по мере необходимости. Вместо того чтобы пытаться поддержать все мыслимые возможности в одном сложном, допускающем разностороннюю настройку классе, программист может определить простой класс и постепенно наращивать его функциональность с помощью декораторов. В результате приложение уже не платит за неиспользуемые функции. Нетрудно также определять новые виды декораторов независимо от классов, которые они расширяют, даже если первоначально такие расширения не планировались. При расширении же сложного класса обычно приходится вникать в детали, не имеющие отношения к добавляемой функции.

3. Декоратор и его компонент не идентичны. Декоратор действует как прозрачное обрамление. Но декорированный компонент все же не идентичен исходному объекту. При использовании декораторов это следует иметь в виду.

4. Множество мелких объектов. При использовании в проекте паттерна декоратор нередко получается система, составленная из большого числа мелких объектов, которые похожи друг на друга и различаются только

способом взаимосвязи, а не классом и не значениями своих внутренних переменных. Хотя проектировщик, разбирающийся в устройстве такой системы, может легко настроить ее, но изучать и отлаживать ее проблематично.

Программист, реализующий паттерн «Декоратор» должен учитывать следующие аспекты:

1. Соответствие интерфейсов. Интерфейс декоратора должен соответствовать интерфейсу декорируемого компонента.

2. Отсутствие абстрактного класса Interface. Нет необходимости определять абстрактный класс Interface, если планируется добавить всего одну обязанность. Так часто происходит, когда вы работаете с уже существующей иерархией классов, а не проектируете новую. В таком случае ответственность за переадресацию запросов, которую обычно несет класс Interface, можно возложить непосредственно на ConcreteDecorator;

3. Облегченные классы. Чтобы можно было гарантировать соответствие интерфейсов, компоненты и декораторы должны наследовать общему классу Interface. Важно, чтобы этот класс был настолько легким, насколько возможно. Иными словами, он должен определять интерфейс, а не хранить данные. В противном случае декораторы могут стать тяжеловесными, и применять их в большом количестве будет накладно. Включение большого числа функций в класс Interface также увеличивает вероятность, что конкретным подклассам придется платить за то, что им не нужно.

4. Изменение представления, а не внутреннего устройства объекта. Декоратор можно рассматривать как появившуюся у объекта оболочку, которая изменяет его поведение. Альтернатива – изменение внутреннего устройства объекта, хорошим примером чего может служить паттерн стратегия.

Стратегии лучше подходят в ситуациях, когда класс Interface уже достаточно тяжел, так что применение паттерна декоратор обходится слишком дорого. В паттерне стратегия компоненты передают часть своей функциональности отдельному объекту-стратегии, поэтому изменить или расширить поведение компонента допустимо, заменив этот объект.

4.5.5 Применение структурных паттернов

Рассмотреть все структурные паттерны и объяснить их применение на конкретных примерах не представляется возможным в рамках данного учебного пособия. Поэтому, перейдем к рассмотрению поведенческих паттернов. Те механизмы реализации структуры программы, которые были рассмотрены в данном подразделе схожи между собой. Действительно, структурные паттерны похожи между собой, особенно когда речь идет об их участниках и взаимодействиях: все структурные паттерны основаны на небольшом множестве языковых механизмов структурирования кода и объектов (одиночном и множественном наследовании для паттернов уровня класса и композиции для паттернов уровня объектов). Но с помощью разных паттернов можно решать совершенно разные задачи.

4.6 Поведенческие паттерны

Паттерны поведения связаны с алгоритмами и распределением обязанностей между объектами. Речь в них идет не только о самих объектах и классах, но и о типичных способах взаимодействия. Паттерны поведения характеризуют сложный поток управления, который трудно проследить во время выполнения программы. Внимание акцентировано не на потоке управления как таковом, а на связях между объектами.

В паттернах поведения уровня класса используется наследование – чтобы распределить поведение между разными классами. В паттернах поведения уровня объектов используется не наследование, а композиция.

4.6.1 Паттерн «Стратегия» (Strategy)

Данный паттерн определяет семейство алгоритмов, инкапсулирует каждый экземпляр алгоритма и позволяет оперировать отдельными алгоритмами как взаимозаменяемыми элементами.

Можно сказать проще: стратегия инкапсулирует определенное поведение, но предполагает возможность его замены.

Для данного паттерна существует альтернативное наименование «Политика» (Policy).

На рис. 4.34 представлен канонический вид данного паттерна. Канонический вид – это наиболее простой набор классов и их связей, отражающих суть паттерна.

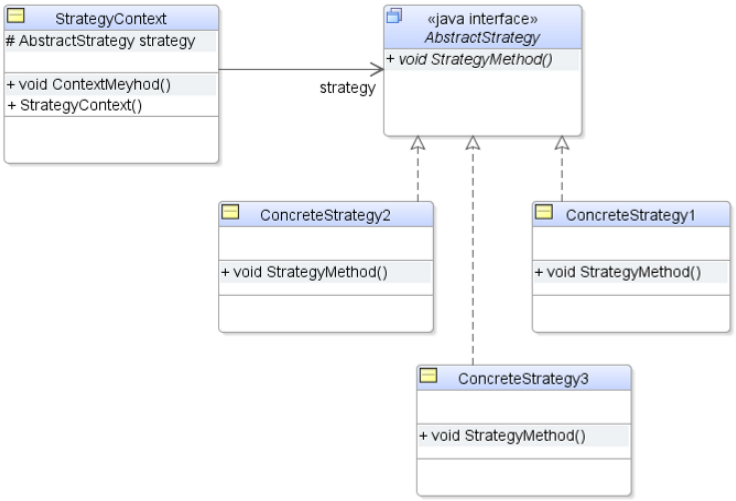


Рисунок 4.34 – Канонический вид паттерна «Стратегия».

В канонической реализации паттерна «Стратегия» присутствуют следующие участники:

1. Непосредственно стратегия (класс AbstractStrategy). Данный класс (интерфейс) определяет метод, который и реализует стратегию (метод StrategyMethod).

2. Конкретные стратегии (классы ConcretteStrategy1, …, ConcreteStrategyn). Классы, которые реализуют алгоритмы, объявленные в интерфейсе AbstractStrategy.

3. Контекст (класс StrategyContext). Это класс методы которого переадресуют запросы клиента объектам-стратегиям.

Следует обратить внимание, что в канонической реализации паттерна «Стратегия» не указывается как передаются данные в метод стратегии, как определяется какой вариант стратегии будет использоваться в контексте. Такие специфические подробности реализации определяются программистом.

При решении практических задач паттерн «Стратегия» может усложняться. Часто данный паттерн применяется в комплексе с другими паттернами.

Применение паттерна «Стратегия» оправдано в случаях:

1. Имеется множество родственных классов (ConcreteStrategy1, ConcreteStrategy2 и т.п.), отличающихся поведением (конкретной реализацией алгоритма). Данный паттерн позволяет сконфигурировать класс клиента (StrategyContext) для придания ему конкретного варианта поведения.

2. Данный паттерн применяется, когда имеется множество различных алгоритмов. Данная ситуация предполагает различные сценарии: алгоритмов много и какой будет использован заранее неизвестно; алгоритм один, но на следующих стадиях разработки могут понадобиться различные варианты алгоритма.

3. Алгоритм StrategyMethod() использует данные, которые не желательно открывать классу-клиенту StrategyContext.

Рассмотрим пример применения паттерна «Стратегия». Требуется реализовать библиотеку, которая обеспечит парсинг информации из источника данных в требуемый формат приложения. Особенность примера состоит в том, что источник данных – текстовый файл, но в дальнейшем

потребуется использовать и другие источники: XML, базы данных, csv и прочие.

Начнем с разработки иерархии классов. На рис. 4.35 представлена иерархия используемых классов.

Рисунок 4.35 – Иерархия классов парсера.

4.6.2 Паттерн «Цепочка обязанностей» (Chain of responsibility)

Данный паттерн позволяет избежать привязки отправителя запроса к его получателю, давая шанс обработать запрос нескольким объектам. Связывает объекты-получатели в цепочку и передает запрос вдоль этой цепочки, пока его не обработают.

На рис. 4.34 представлена диаграмма классов, реализующих подобный механизм для передачи сообщения от клиента и обработки этого сообщения.

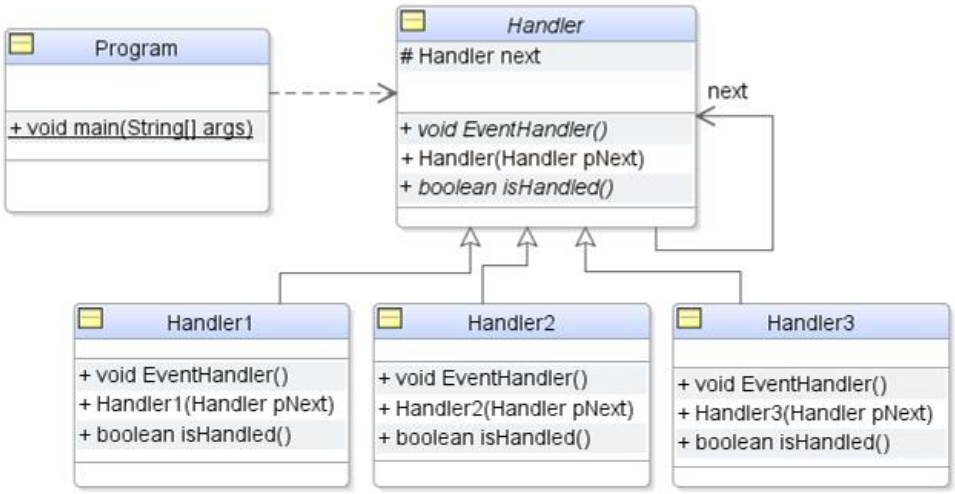


Рисунок 4.34 – Диаграмма, демонстрирующая взаимодействие классов в паттерне «Цепочка обязанностей».

Важным является тот факт, что клиент передает сообщение одному объекту. Паттерн реализует передачу этого сообщения, клиенту не нужно самостоятельно осуществлять поиск целевых объектов. Объекты-обработчики могут быть различных типов.

На рис. 4.35 представлен листинг класса Handler1. Классы Handler2 и Handler3 имеют схожую структуру и функционал. В реализации классов цепочки событий есть только одно отличие: в классе Handler2 метод isHandled() возвращает true и цепочка распространения события прерывается.

```
public class Handler1 extends Handler {
    public Handler1(Handler pNext){
        super(pNext);
    }
    @Override
    public boolean isHandled() {
        return false;
    }
    @Override
    public void EventHandler() {
        System.out.println("Сработал обработчик Handler1");
        if(!isHandled() && next != null)
            next.EventHandler();
    }
}
```

Рисунок 4.35 – Класс Handler1.

На рис. 4.36 показан пример использования данного механизма. В функции main() создается цепочка объектов, затем метод EventHandler(), вызванный для одного объекта запускает механизм распространения события.

```
public class Program {
    public static void main(String[] args){
        Handler ob1 = new Handler1(null);
        Handler ob2 = new Handler2(ob1);
        Handler ob3 = new Handler3(ob2);
        Handler ob4 = new Handler3(ob3);
        ob4.EventHandler();
    }
}
```

Рисунок 4.36 – Листинг функции main().

Вывод приложения представлен на рис 4.37.

```
Сработал обработчик Handler3
Сработал обработчик Handler3
Сработал обработчик Handler2
Process exited with exit code 0.
```

Рисунок 4.37 – Консольный вывод приложения.

На практике паттерн «Цепочка обязанностей» используется в случаях:

– есть более одного объекта, способного обработать запрос, причем настоящий обработчик заранее неизвестен и должен быть найден автоматически;

– программисту требуется отправить запрос одному из нескольких объектов, не указывая явно, какому именно;

– набор объектов, способных обработать запрос, должен задаваться динамически.

В качестве элементов, принимающих участие в данном паттерне, используются (рис. 4.34):

1. Handler – обработчик. Класс определяет интерфейс для обработки запросов; реализует связь с преемником (если это предусмотрено программистом).

2. Конкретные обработчики (Handler1, Handler2, Handler3) – классы которые могут обработать событие, реализуют интерфейс Handler. Данные классы обрабатывают запрос, за который отвечают. Имеют доступ к классу-преемнику. Если конкретный обработчик способен обработать запрос, то так и делает, если не может, то направляет его – его своему преемнику.

3. lient – класс-клиент (в данном примере класс Program), отправляющий запрос некоторому объекту конкретного обработчика в цепочке.

Результаты применения паттерна «Цепочка обязанности»:

1. Ослабление связанности. Этот паттерн освобождает объект от необходимости «знать», кто конкретно обработает его запрос. Отправителю и получателю ничего неизвестно друг о друге, а включенному в цепочку объекту – о структуре цепочки. Таким образом, цепочка обязанностей помогает упростить взаимосвязи между объектами. Вместо того чтобы хранить ссылки на все объекты, которые могут стать получателями запроса,

объект должен располагать информацией лишь о своем ближайшем преемнике.

2. Дополнительная гибкость при распределении обязанностей между объектами. Цепочка обязанностей позволяет повысить гибкость распределения обязанностей между объектами. Добавить или изменить обязанности по обработке запроса можно, включив в цепочку новых участников или изменив ее каким-то другим образом. Этот подход можно сочетать со статическим порождением подклассов для создания специализированных обработчиков.

3. Получение не гарантировано. Поскольку у запроса нет явного получателя, то нет и гарантий, что он вообще будет обработан: он может достичь конца цепочки и пропасть. Необработанным запрос может оказаться и в случае неправильной конфигурации цепочки.

Если программист реализует какой-либо механизм с использованием паттерна «Цепочка обязанностей» необходимо обращать внимание на следующие аспекты:

1. Особенности реализации связи цепочки преемников. Связи могут быть реализованы как динамически, так и заданы статически.

2. Представление запросов. В представленном примере запрос передается по цепочке жестко кодированным методом. В настоящее время широко применяется программирование на основе событий, поэтому использование событийной модели является предпочтительным вариантом.

4.6.2 Паттерн «Команда» (Command)

Данный паттерн также является паттерном поведения, но представляет собой достаточно простой механизм, смысл которого понятен из диаграммы классов (рис 4.38).

Данный паттерн используется в случаях, когда нужно:

1. Параметризировать объекты выполняемым действием, как в случае с пунктами меню и панели инструментов. В процедурном языке такую параметризацию можно выразить с помощью функции обратного вызова, то есть такой функции, которая регистрируется, чтобы быть вызванной позднее. Команды представляют собой объектно-ориентированную альтернативу функциям обратного вызова.

2. Определять, ставить в очередь и выполнять запросы в разное время. Время жизни объекта Command необязательно должно зависеть от времени жизни исходного запроса. Если получателя запроса удается реализовать так, чтобы он не зависел от адресного пространства, то объект-команду можно передать другому процессу, который займется его выполнением;

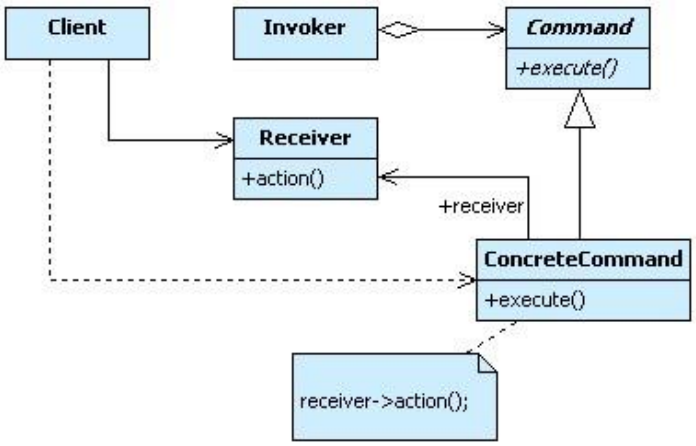


Рисунок 4.38 – Основные классы, участвующие в реализации паттерна «Команда».

3. Поддержать отмену операций. Операция Execute объекта Command может сохранить состояние, необходимое для отката действий, выполненных командой. В этом случае в интерфейсе класса Command должна быть дополнительная операция Undo(), которая отменяет действия, выполненные предшествующим обращением к Execute(). Выполненные команды хранятся в списке истории. Для реализации произвольного числа уровней отмены и повтора команд нужно обходить этот список соответственно в обратном и

прямом направлениях, вызывая при посещении каждого элемента команду Undo()или Execute().

4. Поддержать протоколирование изменений, чтобы их можно было выполнить повторно после аварийной остановки системы. Дополнив интерфейс класса Command операциями сохранения и загрузки, программист сможет вести протокол изменений во внешней памяти. Для восстановления после сбоя нужно будет загрузить сохраненные команды с диска и повторно выполнить их с помощью операции Execute().

5. Структурировать систему на основе высокоуровневых операций, построенных из примитивных. Такая структура типична для информационных систем, поддерживающих транзакции. Транзакция инкапсулирует набор изменений данных. Паттерн команда позволяет моделировать транзакции. У всех команд есть общий интерфейс, что дает возможность работать одинаково с любыми транзакциями. С помощью этого паттерна можно легко добавлять в систему новые виды транзакций.

Паттерн «Команда» широко применяется в существующих программных системах, но наиболее явное проявление данного механизма можно наблюдать в классах команд WPF и Java.

4.6.3 Паттерн «Наблюдатель» (Observer)

Данный поведенческий паттерн определяет зависимость типа «один ко многим» между объектами. В результате использования при изменении состояния одного объекта все зависящие от него объекты оповещаются об этом и автоматически обновляются. Существует другое название данного паттерна: «Издатель/Подписчик» (Publisher/Subscriber).

Данный паттерн следует использовать при реализации систем, обладающих свойствами:

1. У абстракции есть два аспекта, один из которых зависит от другого. Инкапсуляции этих аспектов в разные объекты позволяют изменять и повторно использовать их независимо.

2. Когда при модификации одного объекта требуется изменить другие и заранее не известно, сколько именно объектов нужно изменить.

3. Когда один объект должен оповещать других, не делая предположений об уведомляемых объектах. Другими словами, необходимо обеспечить архитектуру при которой объекты не были тесно связаны между собой.

Структура данного паттерна представлена на рис. 4.39

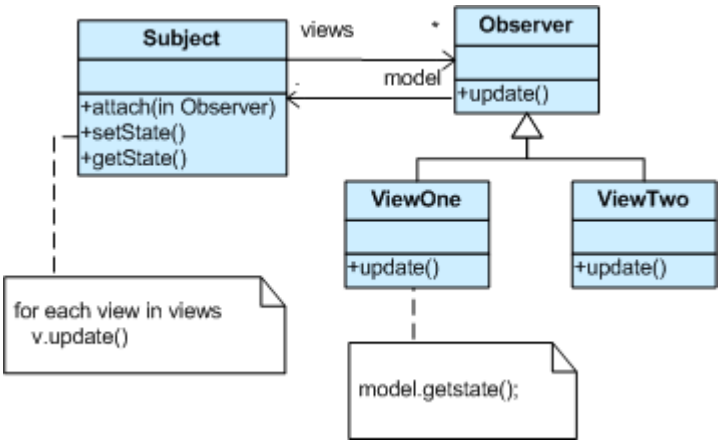


Рисунок 4.39 – Основные классы, участвующие в реализации паттерна «Наблюдатель».

Основные участники паттерна:

1. Subject – субъект. Класс располагает информацией о своих наблюдателях. За субъектом может «следить» любое число наблюдателей. Данный класс предоставляет интерфейс для присоединения и отделения наблюдателей.

2. Observer – наблюдатель. Определяет интерфейс обновления для объектов, которые должны быть уведомлены об изменении субъекта.

3. ConcreteSubject – конкретный субъект. Класс сохраняет состояние, представляющее интерес для конкретного наблюдателя ConcreteObserver и посылает информацию своим наблюдателям, когда происходит изменение.

4. ConcreteObserver – конкретный наблюдатель. Класс хранит ссылку на объект класса ConcreteSubject и сохраняет данные, которые должны быть согласованы с данными субъекта. Данный элемент реализует интерфейс обновления, определенный в классе Observer, чтобы поддерживать согласованность с субъектом.

В результате применения паттерна программная система приобретает следующие достоинства и недостатки:

1. Абстрактная связанность субъекта и наблюдателя. Субъект имеет информацию лишь о том, что у него есть ряд наблюдателей, каждый из которых подчиняется простому интерфейсу абстрактного класса Observer. Субъекту неизвестны конкретные классы наблюдателей. Таким образом, связи между субъектами и наблюдателями носят абстрактный характер и сведены к минимуму.

Поскольку субъект и наблюдатель не являются тесно связанными, то они могут находиться на разных уровнях абстракции системы. Субъект более низкого уровня может уведомлять наблюдателей, находящихся на верхних уровнях, не нарушая иерархии системы. Если бы субъект и наблюдатель представляли собой единое целое, то получающийся объект либо пересекал бы границы уровней (нарушая принцип их формирования), либо должен был находиться на каком-то одном уровне (компрометируя абстракцию уровня).

2. Поддержка широковещательных коммуникаций. В отличие от обычного запроса для уведомления, посылаемого субъектом, не нужно задавать определенного получателя. Уведомление автоматически поступает всем подписавшимся на него объектам. Субъекту не нужна информация о количестве таких объектов, от него требуется всего лишь уведомить своих наблюдателей. Поэтому мы можем в любое время добавлять и удалять наблюдателей. Наблюдатель сам решает, обработать полученное уведомление или игнорировать его.

3 Неожиданные обновления. Поскольку наблюдатели не располагают информацией друг о друге, им неизвестно и о том, во что обходится изменение субъекта. Безобидная, на первый взгляд, операция над субъектом может вызвать целый ряд обновлений наблюдателей и зависящих от них объектов. Более того, нечетко определенные или плохо поддерживаемые критерии зависимости могут стать причиной непредвиденных обновлений, отследить которые очень сложно.

Эта проблема усугубляется еще и тем, что простой протокол обновления не содержит никаких сведений о том, что именно изменилось в субъекте. Без дополнительного протокола, помогающего выяснить характер изменений, наблюдатели будут вынуждены проделать сложную работу для косвенного получения такой информации.

При практической реализации данного паттерна, программист должен обращать внимание на следующие аспекты:

1. Отображение субъектов на наблюдателей. С помощью этого простейшего способа субъект может отследить всех наблюдателей, которым он должен посылать уведомления, то есть хранить на них явные ссылки. Однако при наличии большого числа субъектов и всего нескольких наблюдателей это может оказаться накладно. Один из возможных компромиссов в пользу экономии памяти за счет времени состоит в том, чтобы использовать ассоциативный массив (например, хэш-таблицу) для хранения отображения между субъектами и наблюдателями. Тогда субъект, у которого нет наблюдателей, не будет зря расходовать память. С другой стороны, при таком подходе увеличивается время поиска наблюдателей.

2. Наблюдение более чем за одним субъектом. Иногда наблюдатель может зависеть более чем от одного субъекта. Например, у электронной таблицы бывает более одного источника данных. В таких случаях необходимо расширить интерфейс Update(), чтобы наблюдатель мог «узнать», какой субъект прислал уведомление. Субъект может просто

передать себя в качестве параметра операции Update(), тем самым сообщая наблюдателю, что именно нужно обследовать.

3. Источник обновления. Чтобы сохранить согласованность, субъект и его наблюдатели полагаются на механизм уведомлений. Но какой именно объект вызывает операцию Notify() для инициирования обновления? Существуют различные варианты реализации.

4. Висячие ссылки на удаленные субъекты. Удаление субъекта не должно приводить к появлению висячих ссылок у наблюдателей. Избежать этого можно, например, поручив субъекту уведомлять все свои наблюдатели о своем удалении, чтобы они могли уничтожить хранимые у себя ссылки. В общем случае простое удаление наблюдателей не годится, так как на них могут ссылаться другие объекты и под их наблюдением могут находиться другие субъекты.

5. Гарантии непротиворечивости состояния субъекта перед отправкой уведомления. Важно быть уверенным, что перед вызовом операции Notify() состояние субъекта непротиворечиво, поскольку в процессе обновления собственного состояния наблюдатели будут опрашивать состояние субъекта.

Паттерн Observer является более сложным по сравнению с ранее рассмотренными, но он чаще применяется и предоставляет разработчикам широкие возможности по модификации и доработке.

4.6.4 Применение поведенческих паттернов

Инкапсуляция вариаций – элемент многих паттернов поведения. Если определенная часть программы подвержена периодическим изменениям, эти паттерны позволяют определить объект для инкапсуляции такого аспекта. Другие части программы, зависящие от данного аспекта, могут кооперироваться с ним. Обычно паттерны поведения определяют абстрактный класс, с помощью которого описывается инкапсулирующий объект. Своим названием паттерн как раз и обязан этому объекту. Например:

– объект стратегия инкапсулирует алгоритм;

– объект состояние инкапсулирует поведение, зависящее от состояния;

– объект посредник инкапсулирует протокол общения между объектами;

– объект итератор инкапсулирует способ доступа и обхода компонентов составного объекта.

Когда взаимодействующие объекты напрямую ссылаются друг на друга, они становятся зависимыми, а это может отрицательно сказаться на повторном использовании системы и разбиении ее на уровни. Паттерны команда, наблюдатель, посредник и цепочка обязанностей указывают разные способы разделения получателей и отправителей запросов.

Как правило, паттерны поведения дополняют и усиливают друг друга. Например, класс в цепочке обязанностей, скорее всего, будет содержать хотя бы один шаблонный метод.

Паттерны поведения хорошо сочетаются и с другими паттернами. Например, система, в которой применяется паттерн компоновщик, время от времени использует посетитель для выполнения операций над компонентами.

Контрольные вопросы

1. Назовите основные типы паттернов. Какие критерии положены в основание классификации паттернов?
2. Опишите назначение и ключевые особенности каждого типа паттернов.
3. Для чего предназначен паттерн «Одиночка»? Приведите диаграмму классов, участвующих в реализации данного паттерна. Как реализовать данный паттерн с наследованием?
4. В чем особенность порождающих паттернов?

5. Нарисуйте диаграмму классов, участвующих в паттерне «Абстрактная фабрика» (Abstract Factory). Опишите варианты использования данного паттерна и основные элементы, участвующие в его реализации.

6. Нарисуйте диаграмму классов, участвующих в паттерне «Строитель» (Builder). Опишите варианты использования данного паттерна и основные элементы, участвующие в его реализации.

7. Нарисуйте диаграмму классов, участвующих в паттерне «Прототип» (Prototype). Опишите варианты использования данного паттерна и основные элементы, участвующие в его реализации.

8. Что общего у структурных паттернов? Чем они отличаются от порождающих паттернов?

9. Нарисуйте диаграмму классов, участвующих в паттерне «Адаптер» (Adapter). Опишите варианты использования данного паттерна и основные элементы, участвующие в его реализации.

10. Нарисуйте диаграмму классов, участвующих в паттерне «Мост» (Bridge). Опишите варианты использования данного паттерна и основные элементы, участвующие в его реализации.

11. Нарисуйте диаграмму классов, участвующих в паттерне «Декоратор» (Decorator). Опишите варианты использования данного паттерна. Поясните, каким образом обеспечивается вложенность декораторов?

12. Чем поведенческие паттерны отличаются от порождающих и структурных?

13. Поясните структуру паттерна «Цепочка событий» (Chain of responsibility). В каких случаях применяется данный механизм?

РАЗДЕЛ 5. ПРАКТИЧЕСКИЕ ПРИЕМЫ РАЗРАБОТКИ ПРИЛОЖЕНИЙ

Разработка приложений с графическим интерфейсом, как одно из направлений программирования, отражает практически все достижения и наработки современной объектно-ориентированной технологии.

ООП – это инструментарий для разработки сложных программ. Следует понимать, что графические приложения, традиционно, являются более сложными. Объектно-ориентированное программирование и проектирование в полной мере проявляют свои преимущества при создании приложений с графическим интерфейсом пользователя.

Технологии построения подобных приложений непрерывно изменяются, совершенствуются. Наблюдается тенденция к возникновению единого технологического и методологического подхода при разработке приложений как для настольных решений, так и для сетевых веб-приложений. Технология WPF в полной мере отражает данную тенденцию: язвк XAML обеспечивает независимую от бизнес-логики разработку интерфейса. При этом сама структура языка XAML приближена к HTML.

В библиотеке Swing, представленной в технологии Java, унификация технологий настольных и сетевых приложений явно не прослеживается. Но следует понимать, что Java, как технология, изначально ориентирована на использование в сетевой среде: технология Swing применяется как для создания приложений по принципу «толстый клиент», так и для разработки веб-интерфейсов на базе сервлетов (использование Servlet, Java Server Pages, Java Server Faces).

Унификация технологий построения графического интерфейса на современном этапе развития программных систем приводит к повышению роли HTML, но значимости объектно-ориентированного подхода данный

факт не уменьшает: программирование бизнес-логики приложений остается уделом ООП.

5.1 Построение графического интерфейса средствами Java

Существуют различные технологии разработки приложений с графическим интерфейсом пользователя, включая мультиплатформенные. В рамках данного пособия рассматривается инструментарий, доступный программисту Windows-приложений, но затрагиваются вопросы разработки с использованием библиотеки Swing, которая предполагает однотипное использование элементов при создании приложений для разных операционных систем.

5.1.1 Основные концепции, библиотеки

Объектно-ориентированное программирование как технология претерпевала развитие параллельно с развитием технологий взаимодействия компьютеров с пользователями. Появление графического интерфейса пользователя (ГИП) существенно усложнило разработку прикладного ПО и вывело на первый план именно ООП-парадигму, которая наиболее полно учитывает потребности разработчиков графических приложений.

В большинстве высокоуровневых языков разработки приложений с ГИП реализованы библиотеки для построения интерфейса пользователя. В языке C# используются технологии Windows Forms и Windows Presentation Foundation. В Java существуют библиотеки AWT и Swing.

Стандартная поставка Java SE включает в себя библиотеку классов, обеспечивающих создание графического интерфейса пользователя (ГИП) (Graphical User Interface, GUI). Эта графическая библиотека получила название JFC (Java Foundation Classes). В библиотеке JFC можно выделить шесть основных частей:

1) AWT (Abstract Window Toolkit) – базовая библиотека классов с несколько устарелым набором «тяжелых» (heavyweight) графических компонентов, расположенная в пакете java.awt и его подпакетах.

2) Swing – библиотека «легких» (lightweight) графических компонентов, дополняющая и во многом заменяющая библиотеку AWT. Занимает почти двадцать пакетов с префиксом javax.swing;

3) Java 2D – часть библиотеки AWT, обеспечивающая рисование графики, выбор цвета, вывод изображений и фигурного текста, а также преобразование их перед выводом.

4) DnD (Drag and Drop) – библиотека классов, позволяющих перемещать объекты из одного компонента в другой с помощью буфера обмена (clipboard). Классы этой библиотеки помещены в пакеты java.awt.datatransfer и java.awt.dnd;

5) Input Method Framework – классы для создания новых методов ввода/вывода. Они занимают пакеты java.awt.im и java.awt.im.spi;

6) Accessibility – библиотека классов для взаимодействия с нестандартными устройствами ввода/вывода: клавиатурой Брайля, световым пером и др. Она расположена в пакете javax.accessibility.

Интерфейс программирования графики (Java Graphics API) представлен двумя технологиями в Java: AWT и Swing.

Возможности технологии AWT разнообразны – AWT состоит из 12 пакетов. В большинстве приложений используются пакеты java.awt и java.awt.event. Это наиболее часто используемые пакеты.

Пакета java.awt содержит:

1) Классы, представляющие компоненты ГИП (GUI Component Classes). Эти классы представляют такие элементы как кнопки (Button), текстовые поля (TextField), надписи (Label) и пр.

2) Классы, представляющие контейнеры ГИП (GUI Container Classes). Представляют такие элементы как панель (Panel), фрейм (Frame), диалог (Dialog) и пр.

3) Менеджеры расположения элементов (Layout managers). Это элементы FlowLayout, BorderLayout и GridLayout.

4) Вспомогательные классы и графические классы общего назначения (Custom graphics classes). Например классы для представления цвета (Color), шрифтов (Font).

Пакет java.awt.event, как видно из названия, содержит классы для работы с системой событий и содержит:

1) Классы событий (Event classes): ActionEvent, MouseEvent, KeyEvent и WindowEvent.

2) Интерфейсы слушателей событий (Event Listener Interfaces): ActionListener, MouseListener, KeyListener и WindowListener.

3) Классы адаптеров слушателей событий (Event Listener Adapter classes): MouseAdapter, KeyAdapter и WindowAdapter.

AWT обеспечивает интерфейс для разработки графических приложений, независимый от платформы и устройства. Эти приложения запускаются на всех ОС.

В AWT выделяют два типа GUI элементов: компоненты (Component) и контейнеры (Container). На рис. 5.1 представлена диаграмма классов AWT, на которой разделены эти две группы элементов.

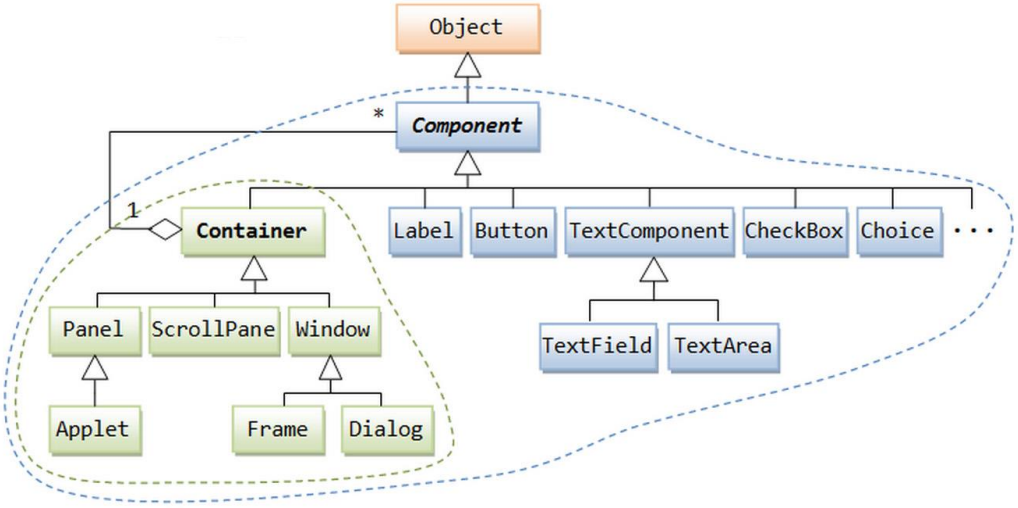


Рисунок 5.1 – Диаграмма классов AWT.

Понимание принципов функционирования библиотеки AWT необходимо для программирования на Java с использованием любой другой библиотеки ГИП, включая Swing. Но детальное описание AWT выходит за рамки данного учебного пособия. В данной главе акцент сделан на разработке с использованием Swing.

Технология разработки ГИП Swing основана на AWT. На рис. 5.2 представлена диаграмма классов, отражающая преемственность двух технологий.

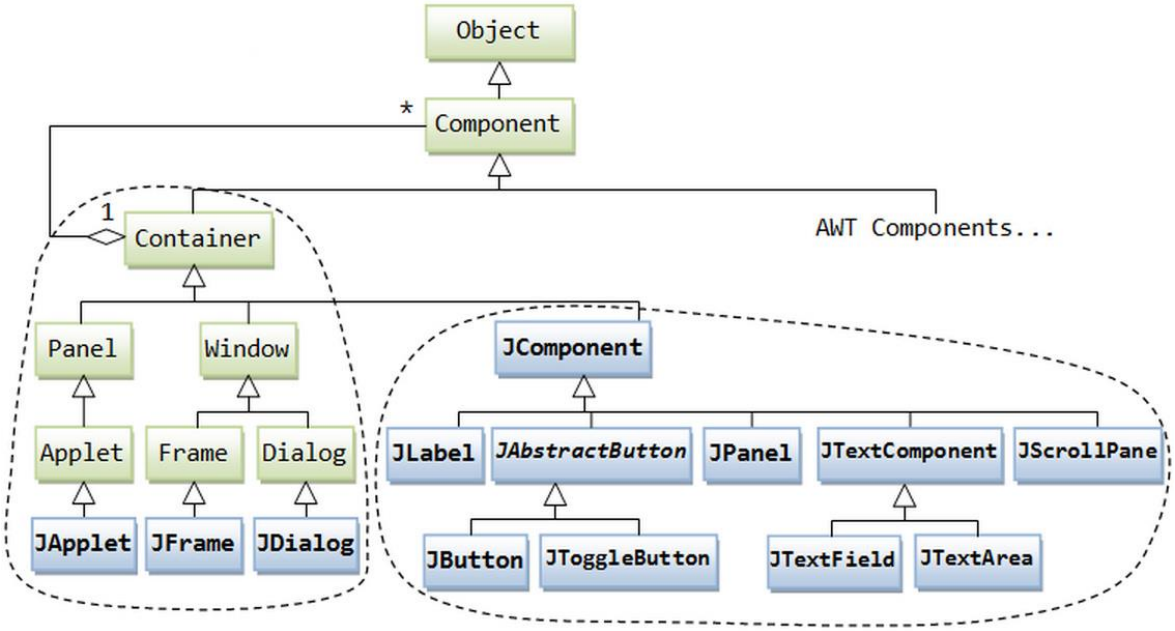


Рисунок 5.2 – Связь AWT и Swing.

Особенности библиотеки Swing:

1) Swing написан на «чистом» Java (pure Java).

2) Компоненты Swing являются «легковесными». Тогда как компоненты AWT – «тяжеловесные». Это означает, что компоненты Swing вообще не нуждаются в поддержке операционной системы и поэтому гораздо более стабильны и быстры.

3) Компоненты Swing поддерживают подключаемые вид и поведение (pluggable look-and-feel). Такое поведение компонентов реализовано на основе архитектурного подхода MVC.

4) В компоненты Swing встроена поддержка всплывающих подсказок, рамок, клавиатурных действий, средств для пользователей с ограниченными возможностями.

5) Компоненты Swing – это компоненты JavaBeans (модель визуального программирования, основанная на компонентах).

6) Приложения Swing используют классы AWT для обработки событий (пакет java.awt.event). Swing добавляет новые классы в пакет javax.swing.event, но они используются редко.

7) Приложения Swing используют классы AWT для управления размещением компонентов (FlowLayout и BorderLayout пакета java.awt). При этом добавлены новые менеджеры размещения, такие как Springs, Struts, и BoxLayout (пакет javax.swing).

8) В Swing реализована двойная буферизация и оптимизирован механизм прорисовки компонентов.

9) В Swing реализована поддержка много документального интерфейса (JLayeredPane и JInternalFrame).

10) В Swing существенно расширен список поддерживаемых компонентов.

На рис. 5.3 представлена иерархия классов Swing. На рисунке представлены наиболее часто используемые компоненты.

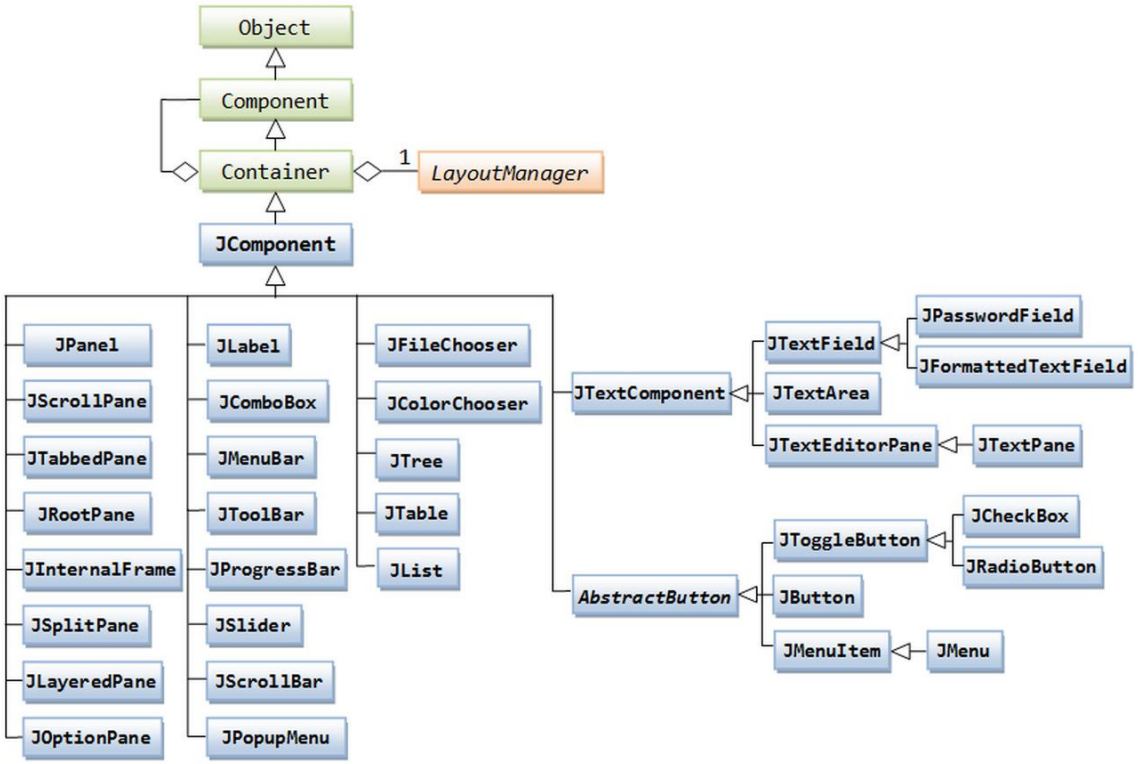


Рисунок 5.3 – Диаграмма классов Swing.

Описание и принципы работы с каждым из представленных компонентов представлены в документации Java. Перед рассмотрением отдельных компонентов пользовательского интерфейса, необходимо освоить работу с главным элементом графического пользовательского интерфейса – окном.

5.1.2 Контейнеры высшего уровня

Контейнеры высшего уровня (top level containers) представляют собой окна операционной системы, в которых программист может размещать элементы пользовательского интерфейса. К контейнерам высшего уровня относятся окна JFrame и JWindow, диалоговое окно JDialog, а также апплет JApplet (не является окном, но предназначен для построения интерфейса в браузере).

Для добавления элементов в окно (например, добавление кнопки на JFrame) логично было бы использовать метод add(), но контейнеры высшего уровня не позволяют добавлять компоненты напрямую. Для выполнения

подобных задач применяется специальная панель JRootPane, которая расположена внутри контейнера высшего уровня. Эта панель называется корневой панелью (root panel). Следует понимать, что корневая панель не является отдельным компонентом – она представляет собой набор слоев (панелей), каждый из которых отвечает за определенный функционал. JRootPane содержит следующие слои (рис. 5.4):

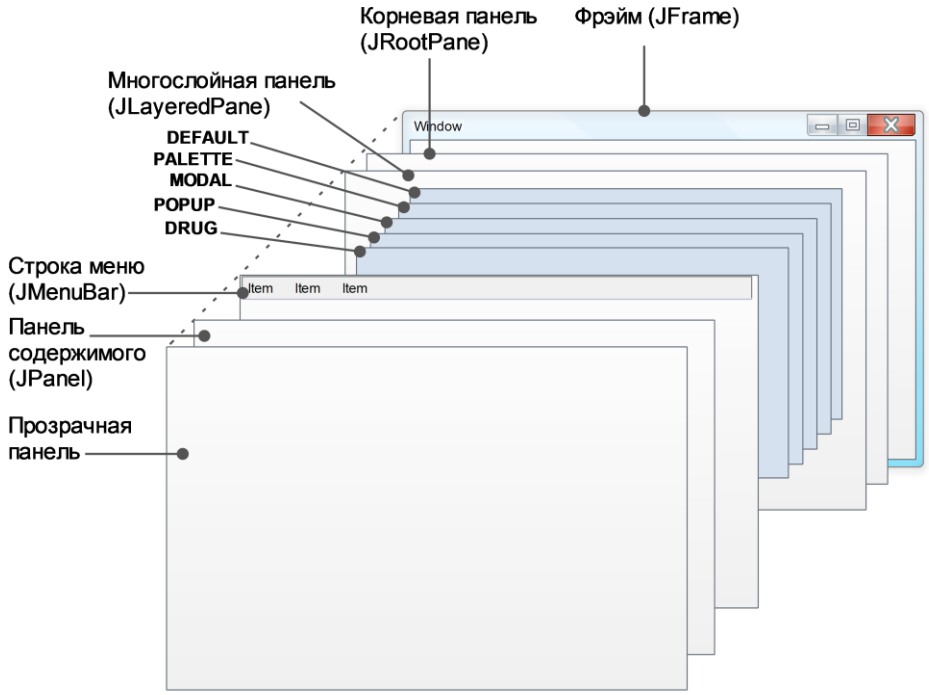


Рисунок 5.4 – Структура JFrame.

1. Многослойная панель JLayeredPane. Представляет собой совокупность слоев, в которых располагаются вызуальные компоненты Swing. Каждый слой имеет свой идентификатор и программист может создавать произвольное количество слоев и располагать в них свои элементы. Существуют стандартные (предопределенные слои), номера которых заданы идентификаторами, представленными в таблице 5.2.

Таблица 5.2 – Слои JLayeredPanel.

Идентификатор слоя	Описание слоя
DEFAULT_LAYER	Этот слой используется для всех обычных компонентов, которые программист добавляет в контейнер. В нем же располагаются внутренние окна многодокументных приложений.
PALETTE_LAYER	Слой предназначен для размещения так называемых палитр, или окон с набором инструментов, которые обычно перекрывают остальные элементы интерфейса. Создавать такие окна позволяет панель JDesktopPane, которая размещает их как раз в этом слое.
MODAL_LAYER	Судя по названию, разработчики планировали использовать этот слой для размещения легковесных модальных диалоговых окон. Однако такие диалоговые окна пока не реализованы. Этот слой в Swing в настоящее время не используется.
POPUP_LAYER	Наиболее часто используемый слой, служащий для размещения всплывающих меню и подсказок.
DRAG_LAYER	Самый верхний в стопке спой. Предназначен для обслуживания операций перетаскивания (drag and drop).

2. Строка меню (menu bar). Строка меню также размещается в многослойной панели, в том же специальном слое FRAME_CONTENT_LAYER, и занимает небольшое пространство в верхней части окна, в длину равное размеру окна, ширина используемого пространства при этом зависит от самой строки меню и содержащихся в ней элементов. Корневая панель следит за тем, чтобы панель содержимого и строка меню не перекрывались. Если строка меню в контейнере высшего уровня не требуется, корневая панель использует все пространство окна для размещения панели содержимого.

3. Панель содержимого (content pane) –часть корневой панели, служащая для размещения компонентов пользовательского интерфейса программы. Она занимает большую часть пространства многослойной панели (за исключением того места, что занимает строка меню). Чтобы

панель содержимого не закрывала добавляемые впоследствии в окно компоненты, многослойная панель размещает ее в специальном очень низком слое с названием. FRAME_CONTENT_LAYER, с номером -30000. Именно в панель содержимого следует добавлять все компоненты вашего пользовательского интерфейса, это основное отличие контейнеров высшего уровня Swing от их аналогов из AWT. Попытка добавить компоненты напрямую в контейнер (JFrame) высшего уровня Swing приведет к ошибке, так что программисту Java следует запомнить важный вызов:

getContentPane().add(Компонент);

Панель содержимого – это экземпляр панели JPanel.

5. Прозрачная панель (glass pane). Прозрачная панель используется в приложениях достаточно редко, поэтому по умолчанию корневая панель делает ее невидимой. Данный слой может использоваться для организации автоматизированного тестирования интерфейса приложения, а также для реализации различных анимационных эффектов.

Подробные описания каждого слоя заслуживают отдельного обсуждения, и выходит за рамки данного пособия.

5.1.3 Классы для представления окон

Окно с рамкой JFrame унаследовано от класса JWindow (в рамках данного курса не рассматривается) и представляет собой наиболее часто используемое в приложениях окно. Окна JFrame обладают рамкой (которая позволяет пользователям легко изменять их размер), заголовком с названием приложения (хотя можно оставлять этот заголовок пустым), иногда системным меню (позволяющим проводить манипуляции с этим окном) и кнопками для управления окном (чаще всего для его закрытия и свертывания). Именно класс JFrame применяется в подавляющем большинстве приложений для размещения компонентов пользовательского интерфейса.

Рассмотрим пример исходного кода, демонстрирующий создание окна:

```
1  package lesson8.Theory;
2
3  import javax.swing.JFrame;
4
5  public class JFrameExample extends JFrame{
6
7      public JFrameExample(){
8          // задание заголовка
9          super("Пример JFrame");
10         // что делать, если окно закрывается
11         setDefaultCloseOperation(EXIT_ON_CLOSE);
12         // иконка для окна
13         setIconImage(getToolkit().getImage("1.png"));
14         // установление размера
15         setSize(300, 100);
16         // показ окна
17         setVisible(true);
18     }
19
20     public static void main(String[] args) {
21         new JFrameExample();
22     }
23
24 }
```

Рисунок 5.5 – Исходный код приложения с ГИП.

На рис. 5.6 показан результат работы программы.

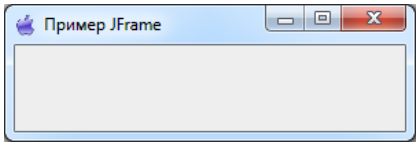


Рисунок 5.6 – Окно приложения, демонстрирующее базовый функционал JFrame.

В таблице 5.3 представлены методы класса JFrame.

Таблица 5.3 – Методы класса JFrame.

Метод	Описание
JFrame()	Конструктор. Создает окно, которое является невидимым.
JFrame(String title)	Конструктор. Создает окно с заданным заголовком. По-умолчанию окно является невидимым.
setTitle	Метод устанавливает заголовок окна.
setSize(int width, int height)	Устанавливает ширину и высоту окна.
setDefaultCloseOperation(int operation)	Устанавливает действие, которое выполнится при закрытии окна. Запоминать численные значения

Метод	Описание
	параметра operation не требуется – для них существуют символьные определения в классе JFrame: DO_NOTHING_ON_CLOSE – не производит никаких действий; HIDE_ON_CLOSE – просто скрывает окно; DISPOSE_ON_CLOSE – скрывает и удаляет окно из памяти, приложение продолжает выполняться; EXIT_ON_CLOSE – после закрытия окна полностью завершает приложение.
setLocationRelativeTo(Component c)	Устанавливает относительную позицию окна (относительно компонента c). Если передано значение null, то окно выравнивается по центру экрана.
void setVisible(boolean b)	Метод показывает (параметр true) или скрывает (false) форму на экране.
getTitle()	Возвращает строку – заголовок формы.

Программисту доступен еще один класс для создания фрейма верхнего уровня – JDialog. Диалоговые окна довольно часто используются в приложениях для получения информации, не имеющей прямого отношения к основному окну, например, для установки параметров, вывода важной вспомогательной информации, получения дополнительной информации от пользователя.

Диалоговые окна могут быть модальными (modal) – они запрещают всем остальным окнам приложения получать информацию от пользователя, пока тот не закончит работу с модальным диалоговым окном. Модальные диалоговые окна полезны в тех ситуациях, когда информация, получаемая от пользователя, коренным образом меняет работу приложения и поэтому до ее получения продолжение работы невозможно. Внешний вид диалоговых окон повторяет окна с рамкой JFrame, но обычно у них меньше элементов управления (чаще всего, имеется только кнопка закрытия окна) и отсутствует системное меню. Как правило, диалоговые окна всегда располагаются поверх

основного окна приложения, напоминая о том, что необходимо завершить ввод информации.

Принципы работы с диалоговыми окнами JDialog схожи с основными операциями класса JFrame. При работе с диалоговыми окнами следует помнить, что они бывают модальными и немодальными. Рассмотрим пример:

```
1  package lesson8.Theory;
2  import java.awt.Dialog;
3  import javax.swing.JDialog;
4
5  public class JDialogExample extends JDialog {
6
7      public JDialogExample(Dialog parent, String caption, boolean isModal){
8
9          super(parent, caption, isModal);
10         setDefaultCloseOperation(DISPOSE_ON_CLOSE);
11         setSize(200, 100);
12         setVisible(true);
13     }
14
15     public static void main(String[] args) {
16         JDialog diag1 = new JDialogExample(null, "Немодальный диалог", false);
17         new JDialogExample(diag1, "Модальный диалог", true);
18     }
19 }
```

Рисунок 5.7 – Исходный код создания диалоговых окон.

Таким образом, очевидно, что работа с классами JDialog и JFrame отличается не значительно.

5.1.4 Использование менеджеров расположения

Поддержка менеджеров расположения встроена в базовый класс всех контейнеров java.awt.Container. Задать и получить расположение элементов можно методами setLayout() и getLayout(). В библиотеки существуют стандартные менеджеры расположения (семь различных классов), но если программист хочет расположить компоненты особенным образом, ему необходимо реализовать интерфейс LayoutManager.

Стандартные менеджеры расположения: FlowLayout, GridLayout, BorderLayout, GridBagLayout, CardLayout, SpringLayout, BoxLayout. Рассмотрим наиболее простые менеджеры.

Полярное расположение BorderLayout. Позволяет создать простую структуру окна. При этом окно разбивается на четыре полярных области и одну область, заполняющую центр окна (рис. 5.8).

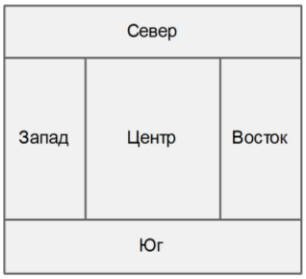


Рисунок 5.8 – Схема расположения областей менеджера BorderLayout.

Добавление компонента в контейнер производится методом add() с дополнительным параметром, указывающим на место расположения:

1. Значение BorderLayout.NORTH или строка «North» – компонент располагается вдоль верхней (северной) границы окна и растягивается на всю его ширину (панель инструментов).

2. Значение BorderLayout.SOUTH или строка «South» – компонент располагается вдоль нижней (южной) границы и растягивается на всю ширину окна (строка состояния).

3. Значение BorderLayout.WEST или строка «West» – компонент располагается вдоль левой (западной) границы окна и растягивается на всю его высоту, однако при этом учитываются размеры северных и южных компонентов (они имеют приоритет).

4. Значение BorderLayout.EAST или строка «East» – компонент располагается вдоль правой (восточной) границы окна.

5. Значение BorderLayout.CENTER или строка «Center» – компонент помещается в центр окна, занимая максимально возможное пространство.

Пример использования расположения BorderLayout представлен на рис. 5.9.

```
7  public class BorderLayoutExample extends JFrame {
8
9      public static void main(String[] args) {
10         new BorderLayoutExample();
11     }
12
13     public BorderLayoutExample(){
14         super("Пример BorderLayout");
15         setSize(400, 300);
16         setDefaultCloseOperation(EXIT_ON_CLOSE);
17         // Получаем панель содержимого
18         Container con = getContentPane();
19         // con.setLayout(new BorderLayout());
20         // Добавляем компоненты
21         con.add(new JButton("Кнопка Север"), "North");
22         con.add(new JButton("Кнопка Юг"), "South");
23         con.add(new JLabel("Надпись Запад"), BorderLayout.WEST);
24         con.add(new JTextField("Текст Восток"), BorderLayout.EAST);
25         con.add(new JEditorPane(JEditorPane.DEFAULT_KEYMAP,
26                 "Текстовый редактор"),
27                 BorderLayout.CENTER);
28         setIconImage(getToolkit().getImage("1.png"));
29         setVisible(true);
30     }
31
32 }
```

Рисунок 5.9 – Листинг с использованием менеджера BorderLayout.

На представленном фрагменте необходимо обратить внимание на строку 18, в которой получается объект панели содержимого. Именно на этом объекте вызываются методы add() в строках 21-25.

В строке 19 вызов метода setLayout() закомментирован. Этот метод устанавливает конкретный менеджер расположения, и все элементы, добавляемые на панель содержимого, будут расставлены в соответствии с данным менеджером расположения. В данном случае этот метод можно не использовать, так как BorderLayout – это менеджер расположения по умолчанию.

В строках 21-25 используется метод add(), в качестве второго параметра которого передается характер расположения элемента. Расположение можно определить как строку: «NORTH», «West» и т.д. Но можно использовать предопределенные значения: BorderLayout.NORTH, BorderLayout.WEST. Очевидны преимущества второго подхода.

Результат выполнения программы представлен на рис. 5.10.

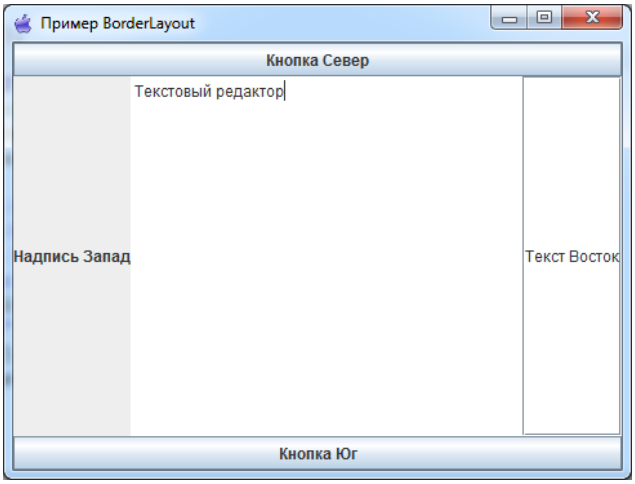


Рисунок 5.10 – Приложение с использованием BorderLayout.

Последовательное расположение FlowLayout. Менеджер задает последовательное расположение компонентов, «стопкой» (вертикально или горизонтально). Класс FlowLayout имеет три перегруженные версии:

– без параметров (расположение по умолчанию);

– с заданием выравнивания FlowLayout(int align), где переменная align может принимать значения FlowLayout.LEFT, FlowLayout.RIGHT, FlowLayout.CENTER, FlowLayout.LEADING, или FlowLayout.TRAILING.

– с выравниваем и заданием расстояния между элементами FlowLayout(int align, int hdist, int vdist), где hdist и vdist – расстояние между элементами, помещаемыми в контейнер, по горизонтали и вертикали соответственно.

Пример использования FlowLayout показан на рис. 5.11.

```java
package lesson8.Theory;
import java.awt.Container;
import java.awt.FlowLayout;
import javax.swing.*;

public class FlowLayoutExample extends JFrame {
    public static void main(String[] args) {
        new FlowLayoutExample();
    }

    public FlowLayoutExample(){
        super("Пример FlowLayout");
        setSize(400, 300);
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        // Получаем панель содержимого
        Container con = getContentPane();
        // Устанавливаем менеджер расположения
        con.setLayout(new FlowLayout());
        // Добавляем компоненты
        con.add(new JButton("Кнопка Север"));
        con.add(new JButton("Кнопка Юг"), "South");
        con.add(new JLabel("Надпись Запад"));
        con.add(new JTextField("Текст Восток"));
        con.add(new JEditorPane(JEditorPane.DEFAULT_KEYMAP,
                "Текстовый редактор"));
        setIconImage(getToolkit().getImage("1.png"));
        setVisible(true);
    }
}
```

Рисунок 5.11 – Расположение элементов с использованием менеджера FlowLayout.

На рис. 5.12 показан результат работы программы. Следует обратить внимание, что элементы располагаются по горизонтали, а элементы, которым не хватило места в горизонтальном ряду, перемещается на следующую строку.

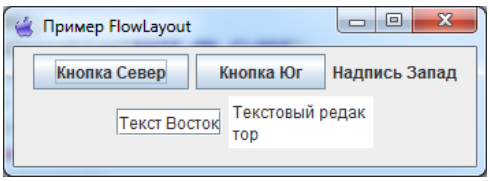


Рисунок 5.12 – Приложение с использованием менеджера FlowLayout.

Табличное расположение GridLayout. Менеджер расположения разделяет контейнер на ячейки, в которые затем помещаются элементы управления. Все ячейки таблицы имеют одинаковый размер, равный максимальному размеру добавляемого элемента.

Конструкторы менеджера расположения:

– конструктор без параметров GridLayout() – устанавливает один столбец и одну строку;

– конструктор GridLayout(int rows, int cols) с указанием количества строк (rows) и количества столбцов (cols);

– конструктор GridLayout(int rows, int cols, int hdist, int vdist) – задает количество строк (rows) и столбцов (cols), а также горизонтальное (hdist) и вертикальное (vdist) расстояния между элементами.

Если вместо rows или cols передано значение 0, то предполагается, что возможно неограниченное количество строк или столбцов. Сетка будет увеличиваться динамически. Все параметры, задаваемые конструкторами можно изменить с помощью методов объекта GridLayout.

Пример использования менеджера показан GridLayout на рис. 5.13.

```
7  public class GridLayoutExample extends JFrame{
8      public GridLayoutExample(){
9          super("Пример GridLayout");
10         setSize(500, 500);
11         setDefaultCloseOperation(EXIT_ON_CLOSE);
12         // Получаем панель содержимого
13         Container con = getContentPane();
14         // Устанавливаем менеджер расположения
15         con.setLayout(new GridLayout(10, 10, 1, 1));
16         // Добавляем компоненты
17         for (int i = 0; i < 10; i++)
18             for (int j = 0; j < 10; j++)
19                 con.add(new JButton(String.valueOf(i)+String.valueOf(j)));
20         setVisible(true);
21     }
22     public static void main(String[] args) {
23         new GridLayoutExample();
24     }
25 }
```

Рисунок 5.13 – Использование менеджера GridLayout.

Результат работы программы представлен на рис. 5.14.

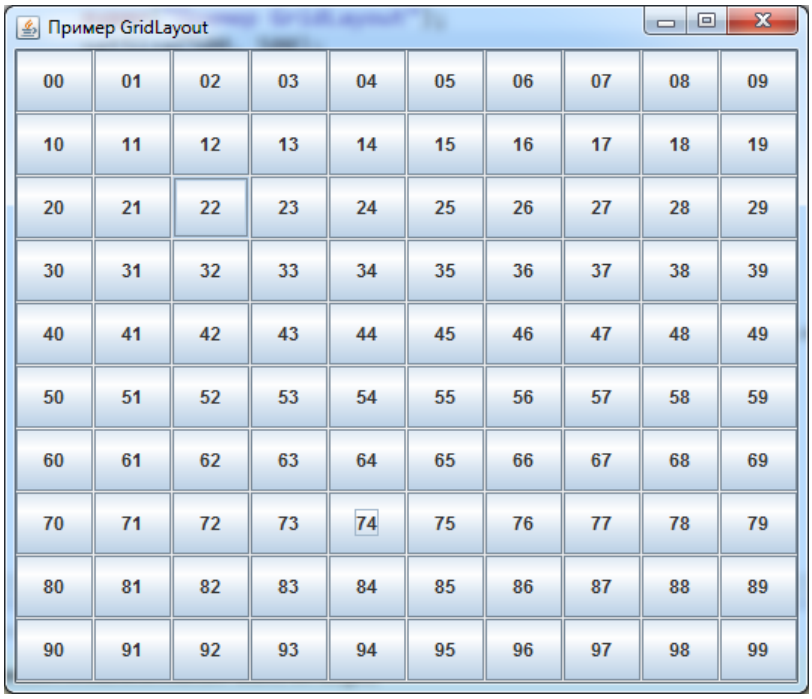


Рисунок 5.14 – Приложение с использованием менеджера GridLayout.

Рассмотрение прочих менеджеров расположения выходит за рамки данного пособия. Необходимо отметить, что если программист намерен применить абсолютное позиционирование элементов, то следует использовать вызов setLayout(null).

Блочное расположение BoxLayout. Менеджер расположения, обладающий широкими одновременно возможностями GridBagLayout (сложного менеджера с большим количеством настроек) и BorderLayout (простого менеджера, возможности которого были рассмотрены ранее). Это относительно новый компонент (пакет javax.swing), который рекомендуется использовать для размещения компонентов в современных приложениях. Конструктору класса BoxLayout необходимо указать контейнер, в котором он будет применен, и характер расположения элементов (Y_AXIS – по вертикали; X_AXIS – по горизонтали). Пример использования BoxLayout представлен на рис. 5.15.

```
 9 public class BoxLayoutExample extends JFrame {
10     public BoxLayoutExample(int axis){
11         super("Пример BoxLayout");
12         setSize(500, 500);
13         setDefaultCloseOperation(EXIT_ON_CLOSE);
14         // Получаем панель содержимого
15         Container container = getContentPane();
16         // Создаем объект BoxLayout
17         BoxLayout manager = new BoxLayout(container, axis);
18         // Устанавливаем менеджер расположения
19         container.setLayout(manager);
20         // Добавляем компоненты
21         URL imgURL = BoxLayoutExample.class.getResource("B-Favorites.png");
22         JButton btn = new JButton("Favorites", new ImageIcon(imgURL));
23         container.add(btn);
24         imgURL = BoxLayoutExample.class.getResource("B-Logoff.png");
25         btn = new JButton("Logoff", new ImageIcon(imgURL));
26         container.add(btn);
27         imgURL = BoxLayoutExample.class.getResource("B-Previous.png");
28         btn = new JButton("Previous", new ImageIcon(imgURL));
29         container.add(btn);
30         imgURL = BoxLayoutExample.class.getResource("B-Play.png");
31         btn = new JButton("Play", new ImageIcon(imgURL));
32         container.add(btn);
33         imgURL = BoxLayoutExample.class.getResource("B-Next.png");
34         btn = new JButton("Next", new ImageIcon(imgURL));
35         container.add(btn);
36         setVisible(true);
37     }
38     public static void main(String[] args) {
39         new BoxLayoutExample(BoxLayout.X_AXIS);
40         new BoxLayoutExample(BoxLayout.Y_AXIS);
41     }
42 }
```

Рисунок 5.15 – Использование менеджера BoxLayout.

В данном листинге видно, что конструктор BoxLayout отличается от конструкторов менеджеров расположения, рассмотренных ранее. Конструктору в качестве параметра передается контейнер, в котором будет применен данный менеджер.

В результате работы программы создаются две формы с различным расположением элементов (рис. 5.16).

Рисунок 5.16 – Приложение с использованием менеджера BoxLayout.

При создании промышленных приложений менеджеры обычно комбинируют и вкладывают друг в друга.

Существую более сложные контейнеры компоновки с широкими возможностями настройки расположения элементов, рассмотрение подобных контейнеров выходит за рамки данного пособия.

5.2 Построение графического интерфейса средствами .NET Framework

Технология .NET Framework предлагает программисту две модели разработки графического интерфейса пользователя для приложений: Windows Forms и Windows Presentation Foundateion (WPF).

5.2.1 Основные концепции и библиотеки

Наиболее перспективной технологией является WPF.

WPF – это графическая система отображения для Windows. Платформа WPF спроектирована для .NET под влиянием таких современных технологий отображения, как HTML и Flash, и использует аппаратное ускорение.

Наиболее существенные изменения, связанные с внедрением технологии WPF (прежде всего по сравнению с технологиями Windows Forms и Swing):

1. Веб-подобная модель компоновки. Вместо того чтобы фиксировать элементы управления на месте с определенными координатами, WPF поддерживает гибкий поток, размещающий элементы управления на основе их содержимого. В результате получается пользовательский интерфейс, который может быть адаптирован для отображения высоко динамичного содержимого или к разным языкам.

2. Новая модель рисования. Вместо рисования пикселей в WPF программист имеет дело с примитивами – базовыми фигурами, блоками текста и прочими графическими ингредиентами. Кроме того, доступны такие новые средства, как действительно прозрачные элементы управления, возможность укладывания друг на друга множества уровней с разной степенью прозрачности, а также встроенная поддержка трехмерной графики.

3. Развитая текстовая модель. После многих лет нестандартной обработки текстов WPF наконец-то предоставляет Windows-приложениям возможность отображения расширенного стилизованного текста в любом месте пользовательского интерфейса. И если нужно отображать значительные объемы текста, для повышения читабельности можно воспользоваться развитыми средствами отображения документов, такими как переносы, разбиение на колонки и выравнивание.

4. Анимация как программная концепция. В WPF нет необходимости использовать таймер для того, чтобы заставить форму перерисовать себя. Вместо этого доступна анимация – неотъемлемая часть платформы. Анимация определяется декларативными дескрипторами, и WPF запускает ее в действие автоматически.

5. Поддержка аудио и видео. Прежние инструментальные наборы для построения пользовательских интерфейсов, такие как Windows Forms, были

весьма ограничены в работе с мультимедиа. Однако WPF включает поддержку воспроизведения любого аудио- или видеофайла, поддерживаемого проигрывателем Windows Media, позволяя воспроизводить более одного медиафайла одновременно.

6. Стили и шаблоны. Стили позволяют стандартизировать форматирование и многократно использовать его по всему приложению. Шаблоны дают возможность изменить способ отображения элементов, даже таких основополагающих, как кнопки. Построение интерфейса с обложками еще никогда не было таким простым.

7. Команды. Большинству пользователей известно, что не имеет значения, откуда они инициируют команду – через меню или панель инструментов; конечный результат один и тот же. Теперь эта абстракция доступна коду – можно определять команды приложения в одном месте и привязывать их к множеству элементов управления.

8. Декларативный пользовательский интерфейс. Содержимое каждого окна сериализуется в виде XML-дескрипторов в документе XAML. Преимущество состоит в том, что пользовательский интерфейс полностью отделяется от кода, и дизайнеры графики могут использовать профессиональные инструменты для редактирования файлы XAML, улучшая внешний вид всего приложения. (XAML – Extensible Application Markup Language).

5.2.2 Окно верхнего уровня. Класс Window

Объект Window – основной элемент традиционных приложений, в нем отображается их основное содержимое. В Windows Presentation Foundation (WPF) за объектом Window скрывается обычное окно Win32. Операционная система не различает окна с WPF- и Win32-содержимым: обрамление рисуется одинаково, на панели задач они неотличимы друг от друга и т. д. Обрамление (Chrome) – это просто другое название области окна, которая

обрамляет клиентскую область и в числе прочего содержит кнопки Minimize (Свернуть), Maximize (Развернуть) и Close (Закрыть).

Объект Window – это абстракция окна Win32 (так же, как класс Form в Windows Forms), содержащая ряд простых методов и свойств. Создав объект Window, программист может показать его на экране с помощью метода Show, скрыть – методом Hide() (это то же самое, что присвоить свойству Visibility значение Hidden или Collapsed) и закрыть навсегда – методом Close().

Внешним видом Window можно управлять с помощью таких свойств, как Icon (иконка приложения), Title (интерпретируется как заголовок окна) и WindowStyle. Для управления положением на экране служат свойства Left и Top. Более осмысленного поведения можно добиться, присваивая свойству WindowStartupLocation значение CenterScreen или CenterOwner. Короче говоря, с помощью установки свойств можно делать почти все, что принято ожидать от окна. Скажем, если установить для Topmost значение true, то окно будет всегда отображаться поверх других, а если присвоить ShowInTaskbar значение false, то значок окна не будет показываться на панели задач.

Всякий раз, как окно становится активным или неактивным (например, из-за того, что пользователь переключается между разными окнами), возникает событие Activated или Deactivated объекта Window.

На рис. 5.17 представлено определение окна с использованием языка разметки XAML. Данное определение генерируется Visual Studio по умолчанию.

```
1  <Window x:Class="WpfExample.Window1"
2          xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
3          xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
4          Title="Window1" Height="300" Width="300">
5      <Grid>
6
7      </Grid>
8  </Window>
```

Рисунок 5.17 – Шаблон окна приложения, генерируемого по умолчанию.

Окно может содержать только один дочерний элемент. Для размещения в окне множества дочерних элементов управления в WPF используется подход: сначала в фрейме окна размещается специализированная панель (контейнер компоновки), а затем, в данном контейнере располагаются элементы управления.

5.2.3 Контейнеры компоновки WPF

Для размещения элементов управления в WPF используются специализированные компоненты: контейнеры компоновки. В технологии Windows Forms применяется механизм абсолютного позиционирования элементов управления: задаются координаты верхнего левого угла элемента (свойства Top, Left) и его размер (свойства Width и Height).

В WPF компоновка определяется используемым контейнером. Окно в WPF должно реализовывать следующие принципы:

1. Элементы (такие как элементы управления) не должны иметь явно установленных размеров. Вместо этого они растут, чтобы уместить свое содержимое. Например, кнопка увеличивается при добавлении в нее текста. Можно ограничить элементы управления приемлемыми размерами, устанавливая максимальное и минимальное их значение.

2. Элементы не указывают свою позицию в экранных координатах. Вместо этого они упорядочиваются своим контейнером на основе размера, порядка и (необязательно) другой информации, специфичной для контейнера компоновки. Для добавления пробелов между элементами служит свойство Margin.

3. Контейнеры компоновки «разделяют» доступное пространство между своими дочерними элементами. Они пытаются обеспечить для каждого элемента его предпочтительный размер (на основе его содержимого), если только позволяет свободное пространство. Они могут

также выделять дополнительное пространство одному или более дочерним элементам.

4. Контейнеры компоновки могут быть вложенными. Типичный пользовательский интерфейс начинается с Grid – наиболее развитого контейнера, и содержит другие контейнеры компоновки, которые организуют меньшие группы элементов, такие как текстовые поля с метками, элементы списка, значки в панели инструментов, колонка кнопок и т.д.

Все контейнеры компоновки WPF являются панелями, которые унаследованы от абстрактного класса System.Windows.Controls.Panel (рис. 5.18). Класс Panel добавляет небольшой набор возможностей всем производным классам.

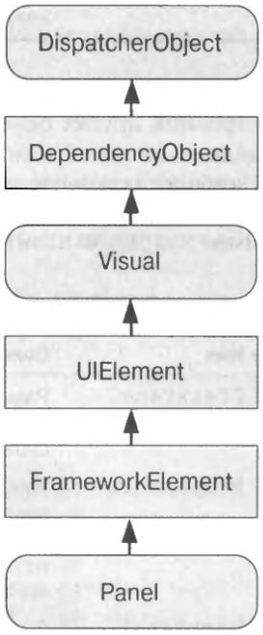


Рисунок 5.18 – Иерархия классов панелей WPF.

Сам по себе базовый класс Panel – это абстрактный класс, не допускающий создания объектов. WPF предлагает набор производных от Panel классов, которые можно использовать для организации компоновки. Основные контейнеры компоновки перечислены в табл. 5.4. Как и все элементы управления WPF, а также большинство визуальных элементов, эти классы находятся в пространстве имен System.Windows.Controls.

Таблица 5.4 – Контейнеры компоновки WPF.

Контейнер	Описание
StackPanel	Размещает элементы в горизонтальном или вертикальном стеке. Этот контейнер компоновки обычно используется в небольших разделах крупного и более сложного окна.
WrapPanel	Размещает элементы в последовательностях строк с переносом. В горизонтальной ориентации WrapPanel располагает элементы в строке слева направо, затем переходит к следующей строке. В вертикальной ориентации WrapPanel располагает элементы сверху вниз, используя дополнительные колонки для дополнения оставшихся элементов.
DockPanel	Выравнивает элементы по краю контейнера.
Grid	Выстраивает элементы в строки и колонки невидимой таблицы. Это один из наиболее гибких и широко используемых контейнеров компоновки.
UniformGrid	Помещает элементы в невидимую таблицу, устанавливая одинаковый размер для всех ячеек. Данный контейнер компоновки используется нечасто.
Canvas	Позволяет элементам позиционироваться абсолютно — по фиксированным координатам. Этот контейнер компоновки более всего похож на традиционный компоновщик Windows Forms, но не предусматривает средств привязки и стыковки. В результате это неподходящий выбор для окон переменного размера.

Контейнер StackPanel.

На рис. 5.19 представлен простой пример компоновки элементов с использованием контейнера StackPanel.

```
1  <Window x:Class="WpfExample.Window31"
2          xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
3          xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
4          Title="Window31" Height="127" Width="300">
5      <StackPanel>
6          <Button>Кнопка 1</Button>
7          <Button>Кнопка 2</Button>
8          <Button>Кнопка 3</Button>
9          <Button>Кнопка 4</Button>
10     </StackPanel>
11 </Window>
```

Рисунок 5.19 – Использование StackPanel.

Результат подобной компоновки представлен на рис. 5.20. Без дополнительной информации о порядке, координатах и размерах кнопок с помощью контейнера StackPanel удалось расположить элементы вертикально.

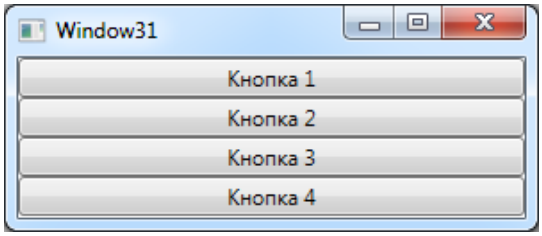


Рисунок 5.20 – Окно с контейнером StackPanel.

По умолчанию панель StackPanel располагает элементы сверху вниз, устанавливая высоту каждого из них такой, которая необходима для отображения его содержимого.

В данном примере это значит, что размер кнопок устанавливается достаточно большим для спокойного размещения текста внутри них. Все элементы растягиваются на полную ширину StackPanel, которая равна ширине окна. Если вы расширите окно, StackPanel также расширится, и кнопки растянутся, чтобы заполнить ее.

StackPanel может также использоваться для упорядочивания элементов в горизонтальном направлении за счет установки свойства Orientation (рис. 5.21):

<StackPanel Orientation=llHorizontalll>

```
1  <Window x:Class="WpfExample.Window31"
2          xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
3          xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
4          Title="Window31" Height="127" Width="300">
5      <StackPanel Orientation="Horizontal">
6          <Button>Кнопка 1</Button>
7          <Button>Кнопка 2</Button>
8          <Button>Кнопка 3</Button>
9          <Button>Кнопка 4</Button>
10     </StackPanel>
11 </Window>
```

Рисунок 5.21 – Использование StackPanel.

Результат представлен на рис. 5.22.

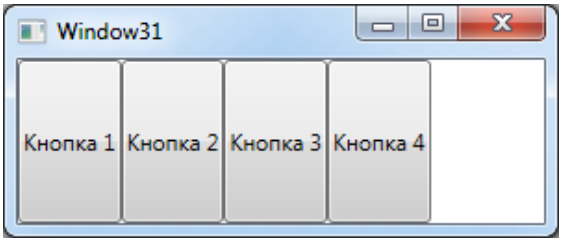


Рисунок 5.22 – Использование горизонтального размещения StackPanel.

Размещение элементов управления определяется контейнером, но элементы управления могут повлиять на этот процесс, явно задавая свойства. Свойства элементов, которые влияют на размещение в контейнере представлены в таблице 5.5.

Таблица 5.5 – Свойства компоновки.

Наименование свойства	Описание
HonzontalAlignment	Определяет позиционирование дочернего элемента внутри контейнера компоновки, когда имеется дополнительное пространство по горизонтали. Доступные значения: Center, Left, Right или Stretch.
VerticalAlignment	Определяет позиционирование дочернего элемента внутри контейнера компоновки, когда имеется дополнительное пространство по вертикали. Доступные значения: Center, Top, Bottom или Stretch.
Margin	Добавляет некоторое пространство вокруг элемента. Свойство Margin – это экземпляр структуры System.Windows.Thickness, с отдельными компонентами для верхней, нижней, левой и правой граней.
MinWidth и MinHeight	Устанавливает минимальные размеры элемента. Если элемент слишком велик, чтобы поместиться в его контейнер компоновки, он будет усечен.
MaxWidth и MaxHeight	Устанавливает максимальные размеры элемента. Если контейнер имеет свободное пространство, элемент не будет увеличен сверх указанных пределов, даже если свойства HonzontalAlignment и VerticalAlignment установлены в Stretch.

Width и Height	Явно устанавливают размеры элемента. Эта установка переопределяет значение Stretch для свойств HorizontalAlignment и VerticalAlignment. Однако данный размер не будет установлен, если выходит за пределы, заданные в MinWidth, MinHeight, MaxWidth и MaxHeight.

Элемент Border.

Border не является одной из панелей компоновки, но это удобный элемент, который часто используется.

Класс Border принимает единственную порцию вложенного содержимого (которым часто является панель компоновки) и добавляет фон или рамку вокруг него. Для работы с Border понадобятся свойства, перечисленные в табл. 5.6.

Таблица 5.6 – Свойства класса Border.

Наименование свойства	Описание
Background	С помощью объекта Brush устанавливает фон, который появляется под содержимым. Можно использовать сплошной цвет либо что-нибудь более сложное.
BorderBrush и BorderThickness	Устанавливают цвет рамки, который появляется на границе объекта Border, и толщину рамки. Для отображения рамки потребуется установить оба свойства.
CornerRadius	Позволяет скруглить углы рамки. Чем больше значение CornerRadius, тем заметнее эффект.
Padding	Добавляет пространство между рамкой и содержимым внутри нее.

На рис. 5.23 представлен фрагмент кода XAML, использующий данные свойства.

```
<Window x:Class="WpfExample.Window31"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="Window31" Height="127" Width="300">
    <Border Background="GreenYellow" BorderBrush="Red"
            BorderThickness="5" CornerRadius="10" Padding="10">
        <StackPanel Orientation="Vertical">
            <Button>Кнопка 1</Button>
            <Button>Кнопка 2</Button>
            <Button>Кнопка 3</Button>
            <Button>Кнопка 4</Button>
        </StackPanel>
    </Border>

</Window>
```

Рисунок 5.23 – Использование свойств Border.

На рис. 5.24 показано результирующее окно.

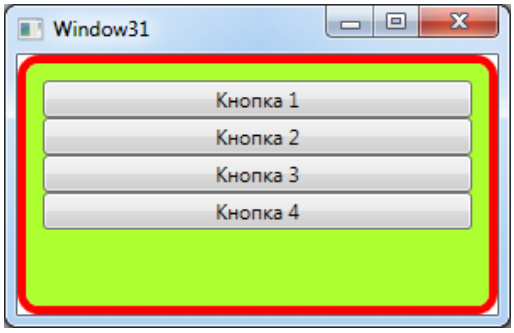


Рисунок 5.24 – Результат применения Border.

Элементы WrapPanel и DockPanel.

Панель WrapPanel располагает элементы управления в доступном пространстве – по одной строке или колонке за раз. По умолчанию свойство WrapPanel.Orientation установлено в Horizontal; элементы управления располагаются слева направо, затем – в следующих строках. Установка значения Vertical для свойства WrapPanel. Orientation приводит к размещению элементов в нескольких колонках.

Подобно StackPanel, панель WrapPanel действительно предназначена для управления мелкими деталями пользовательского интерфейса, а не компоновкой всего окна. Например, WrapPanel можно использовать для удержания вместе кнопок в элементе управления, подобном панели инструментов.

Панель DockPanel обеспечивает более интересный вариант компоновки. Эта панель растягивает элементы управления вдоль одной из внешних границ. Простейший способ представить это – вообразить панель инструментов, которая присутствует в верхней части многих Windows-приложений. Такие панели инструментов пристыковываются к верхней части окна. Как и в случае StackPanel, пристыкованные элементы должны выбрать один аспект компоновки. Например, если вы пристыковать кнопку к верхней части DockPanel, она растянется на всю ширину DockPanel, но получит высоту, которая ей понадобится (на основе своего содержимого и свойства MinHeight). С другой стороны, если пристыковать кнопку к левой стороне контейнера, ее высота будет растянута для заполнения контейнера, но ширина будет установлена по необходимости.

Дочерние элементы DockPanel прикрепляются к одной из сторон панели посредством задания присоединенного свойства Dock. Элементы на самом деле не имеют этого свойства – они приобретают его после помещения в контейнер.

На рис. 5.25 показан пример использования DockPanel.

```
<Window x:Class="WpfExample.DockPanelExample"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="DockPanelExample" Height="300" Width="300">
    <DockPanel>
        <Button Content="кнопка 1" DockPanel.Dock="Top"/>
        <Button Content="кнопка 2" DockPanel.Dock="Bottom"/>
        <Button Content="кнопка 3" DockPanel.Dock="Left"/>
        <Button Content="кнопка 4" DockPanel.Dock="Right"/>
        <Button Content="кнопка 5"/>
    </DockPanel>
</Window>
```

Рисунок 5.25 – Использование DockPanel.

На рис. 5.26 представлен результат.

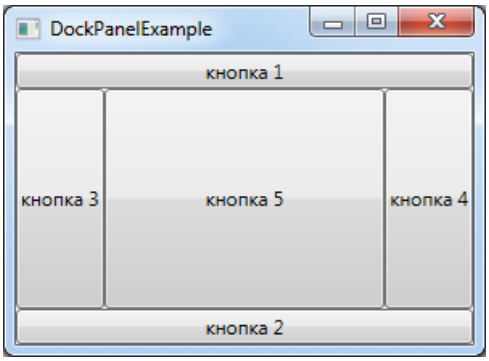


Рисунок 5.26 – Контейнер DockPanel в окне приложения.

Контейнер Grid.

Элемент управления Grid – это наиболее мощный контейнер компоновки в WPF. Большая часть того, что можно достичь с помощью других элементов управления компоновкой, также возможно и в Grid. Контейнер Grid является идеальным инструментом для разбиения окна на меньшие области, которыми можно управлять с помощью других панелей.

При добавлении в Visual Studio нового документа XAML для окна автоматически добавляются дескрипторы Grid в качестве контейнера первого уровня, вложенного внутрь корневого элемента Window.

Хотя Grid задуман как невидимый элемент, можно установить свойство Grid.ShowGridLines в true и получить наглядное представление о нем. Это средство на самом деле не предназначено для украшения окна. В действительности это средство для облегчения отладки, которое предназначено для того, чтобы наглядно показать, как Grid разделяет пространство на отдельные области. Благодаря ему, появляется возможность точно контролировать то, как Grid выбирает ширину колонок и высоту строк.

Создание компоновки на основе Grid – поэтапный процесс. Сначала выбирается необходимое количество колонок и строк. Затем каждому содержащемуся элементу назначается соответствующая строка и колонка, тем самым помещая его в правильное место.

Колонки и строки создаются путем заполнения объектами коллекции Grid.ColumnDefinitions и Grid.RowDefinitions. Например, если необходимо создать две строки и три колонки, то используются следующие дескрипторы:

```
<Window x:Class="WpfExample.GridExample"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="GridExample" Height="300" Width="300">
    <Grid>
        <Grid Background="GreenYellow">
            <Grid.ColumnDefinitions>
                <ColumnDefinition/>
                <ColumnDefinition/>
                <ColumnDefinition/>
            </Grid.ColumnDefinitions>
            <Grid.RowDefinitions>
                <RowDefinition/>
                <RowDefinition/>
                <RowDefinition/>
            </Grid.RowDefinitions>
            <Button Grid.Column="0" Grid.Row="0" Content="Кнопка (0, 0)"/>
            <Button Grid.Column="1" Grid.Row="1" Content="Кнопка (1, 1)"/>
            <Button Grid.Column="2" Grid.Row="2" Content="Кнопка (2, 2)"/>
            <Button Grid.Column="2" Grid.Row="0" Content="Кнопка (2, 0)"/>
            <Button Grid.Column="0" Grid.Row="2" Content="Кнопка (0, 2)"/>
        </Grid>
    </Grid>
</Window>
```

Рисунок 5.27 – Контейнер Grid.

Результирующее окно представлено на рис. 5.28.

Рисунок 5.28 – Расположение кнопок в контейнере Grid.

5.3 Обработка событий

5.3.1 Событийная модель Java

В большинстве современных технологий (.NET Framework, Java) разработки приложений с ГИП используется специализированный механизм

– механизм событий. Говорят, что графические приложения – это приложения управляемые событиями.

При рассмотрении событийной модели Java необходимо выделить три группы компонентов:

1. Источники событий. Источниками событий чаше всего выступают различные элементы управления (кнопки, списки и их элементы, текстовые поля, контейнеры, окна и т.д.). Объекты-источники событий способны информировать объекты-слушатели событий (посылать им сообщение или генерировать событие). Объекты-источники посылают информацию об изменении своего состояния (кнопка нажата, элемент списка выделен, текст в текстовом поле изменен и т.д.).

2. События. Объект-источник должен переслать некоторую информацию объектам-слушателям. Упрощенно можно утверждать, что именно эта пересылаемая информация оформляется в виде события – объекта специального класса, содержащего данные для детального описания возникшего события. Говорят, что сведения о событии инкапсулируются в объекте события.

3. Слушатели событий (подписчики). Это объекты, которые принимают сообщение от источников событий. Для того, чтобы слушатель получил сообщение о возникновении события, он должен быть подписан на «рассылку» событий. В объекте-источнике производится регистрация слушателей (подписчиков). Слушателем может являться любой объект, но на практике указывается такой объект, который может отреагировать на событий надлежащим образом. Такая модель делегирования событий является очень гибкой.

На рис. 5.29 представлен механизм обработки событий в Java (отношение между задействованными классами).

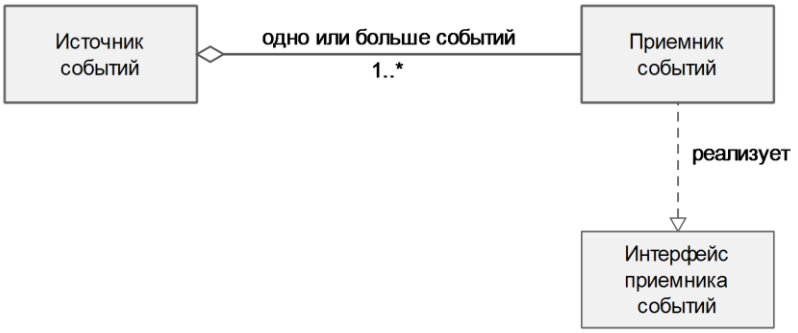


Рисунок 5.29 – Результат работы приложения.

На рис. 5.29 видно, что в механизме присутствуют интерфейсы слушателей событий. Следует учитывать, существует множество различных классов событий, при этом все классы событий описываются подклассами, производными от класса java.util.ObjectEvent.

На рис. 5.30 представлена диаграмма классов событий.

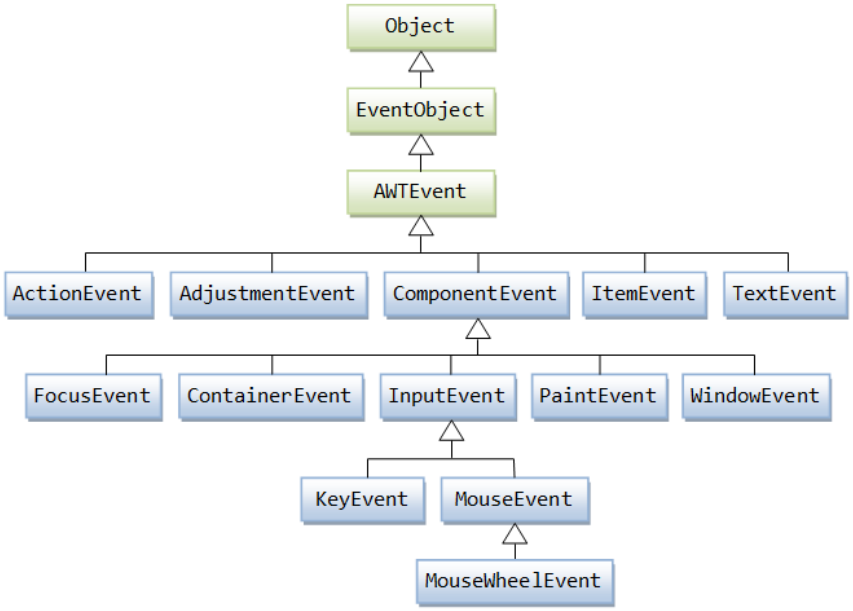


Рисунок 5.30 – Диаграмма классов событий.

Для каждого типа события предусмотрен свой класс – это значит, что объект-слушатель должен реализовать специальный метод (или набор методов) для корректной обработки этого события.

События в Java иногда подразделяют на низкоуровневые (low-level events) и высокоуровневые (high-level events).

К низкоуровневым событиям относят те, что происходят непосредственно в результате действий пользователя: это движения мыши, передача фокуса, выделение элемента списка, нажатия клавиш и т. Все низкоуровневые события унаследованы от класса java.util.AWTEvent. Они поступают в Java-программу от операционной системы или от внутренних механизмов виртуальной машины.

Высокоуровневые события происходят в результате изменения состояния компонента. Такие события поступают не от операционной системы, а создаются самим компонентом. Процесс создания события еще называют запуском (fire). Во многих компонентах Swing можно встретить методы вида fireXXX(). Именно в таких методах создаются объекты с информацией о событиях, которые затем рассылаются объектам-слушателям.

В таблице 5.7 представлены события и методы, которые должен реализовать объект-слушатель события для обработки соответствующего события.

Таблица 5.7 – События Java и методы их обработки.

Класс события	Интерфейс слушателя	Методы слушателя	Источник события
ContainerEvent	ContainerListener	componentAdded(), componentRemoved()	Все контейнеры (наследники класса java.awt.Container)
FocusEvent	FocusListener	focusGained(), focusLost()	Все компоненты
KeyEvent	KeyListener	keyPressed(), keyReleased(), keyTyped()	Все компоненты
MouseEvent	MouseListener	mouseClicked(), mouseEntered(), mouseExited(), mousePressed(), mouseReleased()	Все компоненты
	MouseMotionListener	mouseDragged(), mouseMoved()	Все компоненты
MouseWheelEvent	MouseWheelListener	mouseWheelMoved()	Все компоненты

Класс события	Интерфейс слушателя	Методы слушателя	Источник события
WindowEvent	WindowListener	windowActivated(), windowClosed(), windowClosing(), windowDeactivated(), windowDeiconified(), windowIconified(), windowOpened()	Окна (наследники java.awt.Window), например, JFrame или JDialog)
ActionEvent	ActionListener	actionPerformed()	Компоненты, у которых есть какое-то стандартное действие.
ChangeEvent	ChangeListener	stateChanged	Некоторые компоненты Swing. Многие модели используют это событие для связи с UI-представителями.
PropertyChangeEvent		propertyChange()	Компоненты JavaBeans (в том числе компоненты Swing).
ItemEvent	ItemListener	itemStateChanged()	Событие выделения элемента списка.
ComponentEvent	ComponentListener	componentHidden() componentMoved() componentResized() componentShown()	Событие связано с состоянием (размер, положение и т.п.) компонента.
TextEvent	TextListener	textValueChanged()	Событие связано с изменением текста компонента.

В таблице 5.7 представлены не все классы событий, а наиболее распространенные. Каждый метод интерфейса принимает в качестве параметра соответствующий объект класса события: keyPressed(KeyEvent), mouseClicked(MouseEvent), windowOpened(WindowEvent) и т.д. Представленные классы содержатся в пакетах java.awt.event и javax.swing.event.

При изучении интерфейсов некоторых слушателей становится очевидным, что при их реализации на практике программисту придется реализовать каждый метод интерфейса. При этом может понадобиться только один метод. Для таких случаев в библиотеке Java предусмотрены адаптеры

(adapters) – специальные классы, реализующие интерфейс определенного слушателя, а значит все его методы. Методы слушателя адаптер оставляет пустыми, без всякой реализации, что позволяет программисту вместо реализации интерфейса слушателя наследовать адаптер от класса адаптера. При наследовании в дочерний класс переходят все методы базового класса, и остается переопределить только те из них, которые требуется.

8.1.2 Методы организации механизма обработки событий.

Существует несколько стратегий структурной организации программы для использования событий Java.

1. Применение вложенных классов. Чаще всего обработка событий происходит во внутренних классах, относящихся к классу, в котором создается основная часть пользовательского интерфейса. Причина такой организации кода: при наступлении события может потребоваться информация не только о целевом элементе, но и то, в каком состоянии находятся соседние компоненты пользовательского интерфейса (каков текст в текстовых полях, сколько выбрано элементов в списке, какие флажки отмечены и т. д.). Пример такой реализации представлен на рис. 5.31.

```
1  package lr8_Event;
2  import javax.swing.*;
3  import java.awt.*;
4  import java.awt.event.*;
5
6  public class FrameWithInnerListener extends JFrame {
7      // кнопка формы
8      private JButton btnOK;
9
10     public FrameWithInnerListener() {
11         super("Слушатель как внутренний класс");
12         setDefaultCloseOperation(EXIT_ON_CLOSE); // Действие при закрытии
13         setSize(400, 100); // Установка размера окна
14         getContentPane().setLayout(new FlowLayout()); // Пока не учили!!!
15         btnOK = new JButton("Нажмите"); // Создание кнопки
16         btnOK.setSize(29, 100); // Установка размера кнопки
17         getContentPane().add(btnOK); // Пока не учили!!!
18
19         // Добавление объекта-слушателя внутреннего класса:
20         btnOK.addActionListener(new ButtonOKListener());
21         setVisible(true); // Вывод окна на экран
22     }
23     // Определение класса-слушателя как внутреннего
24     class ButtonOKListener implements ActionListener{
25         public void actionPerformed(ActionEvent e) {
26             System.out.println("Нажата кнопка \"OK\"");
27         }
28     }
29     // Главная функция
30     public static void main(String[] args){
31         new FrameWithInnerListener();
32     }
33 }
```

Рисунок 5.31 – Реализация внутреннего класса-слушателя.

2. Применение анонимных классов. Данный метод обработки событий не рекомендуется использовать. Это самый быстрый путь создания обработчика событий. Класс, который используется в качестве слушателя, является анонимным (anonymous class). Такой класс объявляется и наполняется содержимым в момент создания его объекта (рис. 5.32).

```java
package lr8_Event;
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class FrameWithInnerListener extends JFrame {
    // кнопка формы
    private JButton btnOK;

    public FrameWithInnerListener() {
        super("Слушатель как внутренний класс");
        setDefaultCloseOperation(EXIT_ON_CLOSE); // Действие при закрытии
        setSize(400, 100); // Установка размера окна
        getContentPane().setLayout(new FlowLayout()); // Пока не учили!!!
        btnOK = new JButton("Нажмите"); // Создание кнопки
        btnOK.setSize(29, 100); // Установка размера кнопки
        getContentPane().add(btnOK); // Пока не учили!!!

        // Добавление слушателя с использованием пользовательского метода:
        btnOK.addActionListener(getListenerForButton());
        setVisible(true); // Вывод окна на экран
    }
    // Метод, возвращающий объект-слушаетель
    public ActionListener getListenerForButton(){
        // Возврат объекта анонимного класса
        // Класс создается и наполняется в момент необходимости (после new)
        return new ActionListener(){
            public void actionPerformed(ActionEvent e) {
                System.out.println("Нажата кнопка \"OK\"");
            }
        };
    }
    // Главная функция
    public static void main(String[] args){
        new FrameWithInnerListener();
    }
}
```

Рисунок 5.32 – Обработка события с использованием анонимного класса.

3. Применение фабрики слушателей. Как было рассмотрено ранее, анонимные классы – некорректное решение для создания больших приложений. Использование внутренних классов – может запутать код при наследовании. Для повышения структурной гибкости программы используют подход на основе шаблона «фабрика» (factory). Суть метода заключается в том, что объекты-слушатели возвращаются только методами одного класса – фабрикой слушателей. Описание фабрики слушателей показано на рис. 5.33.

```java
package lr8_Event;
import java.awt.event.*;

public class FactoryOfListeners {
    // Метод фабрики слушателей, возвращающий объект-слушаетель
    public ActionListener getListenerForButton(){
        return new ActionListener(){
            public void actionPerformed(ActionEvent e) {
                System.out.println("Нажата кнопка \"OK\"");
            }
        };
    }
}
```

Рисунок 5.33 – Класс фабрики (файл FactoryOfListeners.java).

На рис. 5.34 показан класс окна приложения, использующий данный подход.

```java
package lr8_Event;
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class FrameWithInnerListener extends JFrame {
    // кнопка формы
    private JButton btnOK;
    private FactoryOfListeners factory = new FactoryOfListeners();

    public FrameWithInnerListener() {
        super("Фабрика слушателей");
        setDefaultCloseOperation(EXIT_ON_CLOSE); // Действие при закрытии
        setSize(400, 100); // Установка размера окна
        getContentPane().setLayout(new FlowLayout()); // Пока не учили!!!
        btnOK = new JButton("Нажмите"); // Создание кнопки
        btnOK.setSize(29, 100); // Установка размера кнопки
        getContentPane().add(btnOK); // Пока не учили!!!

        // Добавление слушателя с использованием фабрики слушателей:
        btnOK.addActionListener(factory.getListenerForButton());
        setVisible(true); // Вывод окна на экран
    }

    // Главная функция
    public static void main(String[] args){
        new FrameWithInnerListener();
    }
}
```

Рисунок 5.34 – Обработка события с использованием фабрики слушателей.

Таким образом, все слушатели для окна приложения можно разместить в одном классе.

4. Применение диспетчера (перенаправления) событий. Данный подход предполагает создание класса, внутри которого происходит определение: какой слушатель требуется конкретному источнику (рис. 5.35).

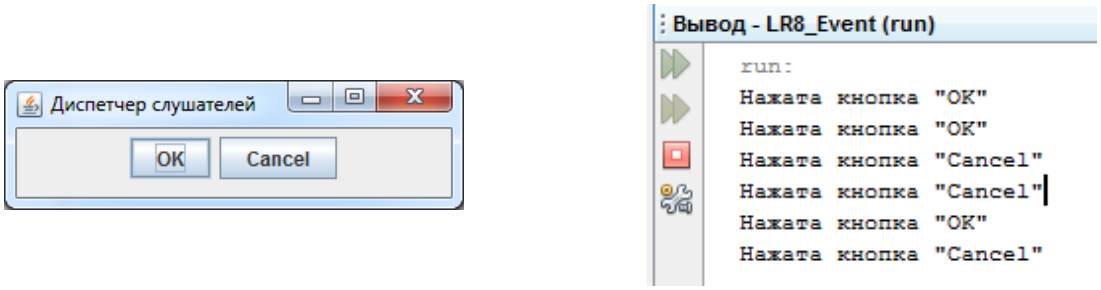
```
1   package lr8_Event;
2   import javax.swing.*;
3   import java.awt.*;
4   import java.awt.event.*;
5
6   public class FrameWithInnerListener extends JFrame {
7       // кнопка формы
8       private JButton btnOK;
9       private JButton btnCancel;
10
11      public FrameWithInnerListener() {
12          super("Диспетчер слушателей");
13          setDefaultCloseOperation(EXIT_ON_CLOSE); // Действие при закрытии
14          setSize(400, 100); // Установка размера окна
15          getContentPane().setLayout(new FlowLayout()); // Пока не учили!!!
16          btnOK = new JButton("OK"); // Создание кнопки OK
17          btnCancel = new JButton("Cancel"); // Создание кнопки Cancel
18          getContentPane().add(btnOK); // Пока не учили!!!
19          getContentPane().add(btnCancel); // Пока не учили!!!
20          // Добавление слушателей с использованием диспетчера слушателей:
21          DispetcherOfListeners dispetcher = new DispetcherOfListeners();
22          btnOK.addActionListener(dispetcher);
23          btnCancel.addActionListener(dispetcher);
24                  setVisible(true); // Вывод окна на экран
25      }
26
27      class DispetcherOfListeners implements ActionListener{
28          public void actionPerformed(ActionEvent e) {
29              if (e.getSource() == btnOK) getListenerForOK();
30              if (e.getSource() == btnCancel) getListenerForCancel();
31          }
32      }
33
34      private void getListenerForOK() {
35          System.out.println("Нажата кнопка \"OK\"");
36      }
37
38      private void getListenerForCancel() {
39          System.out.println("Нажата кнопка \"Cancel\"");
40      }
41
42  // Главная функция
43      public static void main(String[] args){
44          new FrameWithInnerListener();
45      }
46  }
```

Рисунок 5.35 – Обработка события с использованием диспетчера.

Использование диспетчера позволяет не создавать отдельный класс слушателя для каждого события, а использовать один класс диспетчера

(DispetcherOfListeners) и для каждого события реализовывать только метод – обработчик события (методы getListenerForOk и getListenerForCancel).

На рис. 5.36 показан результат работы программы.



а) б)

Рисунок 5.36 – Программа обработки событий с использованием диспетчера: а) окно программы; б) вывод в консоль через обработчик события.

5.3.2 Событийная модель WPF

Программист, работающий в .NET, знаком с понятием события: это сообщение, которое посылается объектом (например, элементом WPF) для уведомления кода о том, что произошло что-то важное. WPF дополняет модель событий .NET новой концепцией маршрутизации событий.

Маршрутизация позволяет событию возникать в одном элементе, а генерироваться в другом: например, щелчок на кнопке панели инструментов генерируется в панели инструментов, а затем в содержащем эту панель окне, и только тогда передается на обработку коду.

Маршрутизация событий дает возможность писать лаконичный и простой код, который может обрабатывать события в наиболее удобном для этого месте. Она необходима также для работы с моделью содержимого WPF, позволяющей создавать простые элементы (например, кнопки) из десятков отдельных ингредиентов, каждый из которых имеет свой собственный набор событий.

Создать обработчик события (на примере кнопки) можно несколькими способами:

1. Добавлять соответствующий атрибут события в разметку XAML (рис. 5.37).

```
<Window x:Class="WpfExample.EventExample"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="EventExample" Height="300" Width="300">
    <Grid>
        <Button Width="150" Height="150">
            <Border BorderThickness="2" BorderBrush="Black">
                <Image Source="Images/1.png" Stretch="Fill" Width="100" Height="100"
                   MouseLeave="Image_OnMouseLeave"
                   MouseEnter="Image_OnMouseEnter"/>
            </Border>
        </Button>
    </Grid>
</Window>
```

Рисунок 5.37 – Добавление обработчика в разметке XAML.

В данном случае добавлены два обработчика: для события попадания мыши в область рисунка, и для события покидания указателем мыши пределов рисунка.

2. Событие можно соединить и с кодом. Вот эквивалент приведенного на рис. 5.38 кода разметки XAML:

```
public partial class EventExample : Window
{
    0 references
    public EventExample()
    {
        InitializeComponent();

        Image.MouseEnter += Image_OnMouseEnter;
        Image.MouseLeave += Image_OnMouseLeave;
    }
```

Рисунок 5.38 – Добавление обработчика в коде C#.

Если необходимо отсоединить обработчик события, то сделать это можно только в коде:

```
private void Button_OnClick(object sender, RoutedEventArgs e)
{
    Image.MouseEnter -= Image_OnMouseEnter;
    Image.MouseLeave -= Image_OnMouseLeave;
}
```

Рисунок 5.39 – Отключение обработчика в коде C#.

Маршрутизируемые события. Многие элементы управления в WPF являются элементами управления содержимым, которые могут иметь разный тип и разный объем вложенного содержимого.

Например, на рис. 5.40 показан подобный элемент:

```
<Window x:Class="WpfExample.RoutedEventEx"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="RoutedEventEx" Height="300" Width="300">
    <Label HorizontalAlignment="Center">
        <StackPanel VerticalAlignment="Center">
            <TextBlock Margin="10">
                Пример маршрутизируемого события
            </TextBlock>
            <Button Width="150">
                <Button.Content>
                    <StackPanel>
                        <Image Source="Images/1.png" Width="100"/>
                        <Border>
                            <TextBlock FontWeight="ExtraBold" FontSize="14">
                                Кнопка с яблоком
                            </TextBlock>
                        </Border>
                    </StackPanel>
                </Button.Content>
            </Button>
        </StackPanel>
    </Label>
</Window>
```

Рисунок 5.40 – Вложенность элементов управления WPF.

На рис. 5.41 показано дерево элементов WPF.

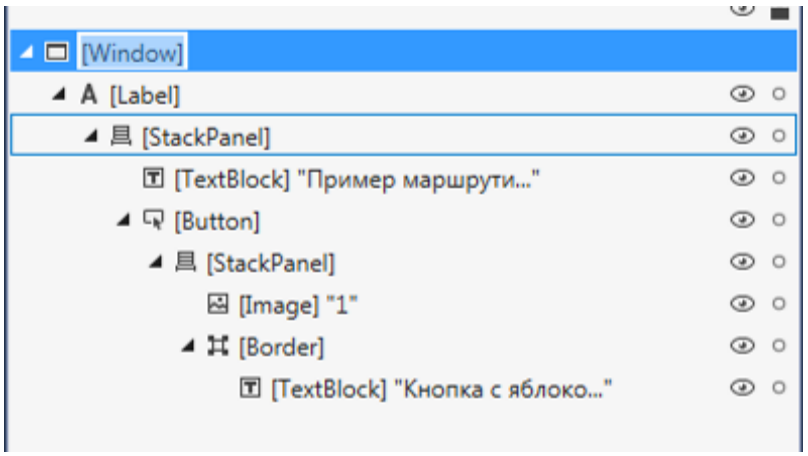


Рисунок 5.41 – Вложенность элементов управления WPF.

На рис. 5.42 представлен результат такой компоновки.

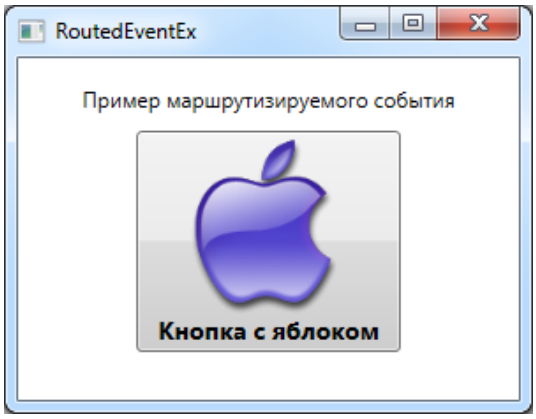


Рисунок 5.42 – Результирующее окно WPF.

Каждый компонент, помещаемый в окно WPF, так или иначе является наследником класса UIElement, включая Label, StackPanel, TextBlock и Image. Класс UIElement определяет несколько ключевых событий. Например, каждый класс, являющийся потомком UIElement, обеспечивает события MouseUp и MouseDown.

А теперь подумайте, что произойдет при щелчке на изображении в такой метке. Понятно, что при этом возникнут события Image.MouseDown и Image.MouseUp. А если вам нужно обрабатывать все щелчки на метке одинаковым образом? То есть неважно, где щелкнул пользователь: на изображении, на тексте или на пустом месте в области метки. В любом из этих случаев нужно реагировать на щелчок с помощью одного и того же кода.

Понятно, что к событиям MouseDown и MouseUp каждого элемента можно привязать один и тот же обработчик, однако это может загромоздить код и усложнить сопровождение разметки. WPF предлагает более удобное решение с помощью модели маршрутизируемых событий.

Маршрутизируемые события бывают трех видов:

1. Прямые (direct) события подобны обычным событиям .NET. Они возникают в одном элементе и не передаются в другой. Например, прямым является событие MouseEnter, которое возникает, когда указатель мыши наводится на элемент.

2. Пузырьковые (bubbling) события поднимаются по иерархии содержания. Например, пузырьковым событием является MouseDown. Оно возникает в элементе, на котором был произведен щелчок, потом передается от этого элемента к родителю, затем к родителю этого родителя, и т.д., пока WPF не достигнет вершины дерева элементов.

3. Туннелируемые (tunneling) события опускаются по иерархии содержания. Они позволяют предварительно просматривать (и, возможно, останавливать) событие, прежде чем оно дойдет до подходящего элемента управления. Например, PreviewKeyDown позволяет перехватить нажатие клавиши, сначала на уровне окна, а затем в более специфических контейнерах, вплоть до элемента, содержавшего фокус в момент нажатия клавиши.

Поскольку события MouseUp и MouseDown являются пузырьковыми событиями, можно определить, что произойдет в примере с составной меткой. При щелчке на изображении с яблоком (рис. 4.7) событие MouseDown возникнет в следующем порядке:

1. Image.MouseDown.
2. StackPanel.MouseDown
3. Button.MouseDown.
4. StackPanel.MouseDown.
5. Label.MouseDown.

После того как событие MouseDown возникнет в метке, оно передается следующему элементу управления (в данном случае это окно Window). Окно находится на самом верху иерархии содержания и в самом конце в последовательности пузырькового распространения события. Здесь последний шанс обработать пузырьковое событие наподобие MouseDown. Если пользователь отпускает кнопку мыши, в такой же последовательности возникает событие MouseUp.

Пузырьковые события не обязательно обрабатывать в одном месте: например, ничто не мешает обрабатывать события MouseDown и MouseUp на каждом уровне. Однако, как правило, для каждой задачи выбирается наиболее подходящая маршрутизация событий.

Класс RoutedEventArgs. Объект данного класса передается в качестве параметра обработчика события. При обработке пузырькового события параметр отправителя содержит ссылку на последнее звено в цепочке. Например, если событие перед обработкой всплывает от изображения до метки, то параметр отправителя будет ссылаться на объект метки.

В некоторых случаях требуется знать, где первоначально произошло событие. Эту информацию, а также другие подробности, можно получить из свойств класса RoutedEventArgs (таблица 5.8).

Таблица 5.8 – Свойства класса RoutedEventArgs.

Имя свойства	Описание
Source	Указывает, какой объект сгенерировал событие. Если речь идет о событии клавиатуры, то это элемент управления, имевший фокус ввода в момент возникновения события (например, когда была нажата клавиша). В случае события мыши это самый верхний элемент под указателем мыши в момент возникновения события (например, когда был произведен щелчок кнопкой мыши).
OriginalSource	Указывает, какой объект первоначально сгенерировал событие. Как правило, совпадает с Source. Однако в некоторых случаях OriginalSource спускается глубже по дереву объектов, чтобы дойти до внутреннего элемента, являющегося частью элемента более высокого уровня. Например, если вы щелкнете кнопкой мыши близко к границе окна, то получите объект Window в качестве источника события и Border в качестве первоначального источника. Это объясняется тем, что Window состоит из отдельных меньших элементов.

RoutedEvent	Предоставляет объект RoutedEvent для события, сгенерированного вашим обработчиком события (например, статический объект UIElement.MouseUpEvent). Эта информация бывает полезна при обработке разных событий одним и тем же обработчиком.
Handled	Позволяет остановить процесс пузырькового распространения или туннельного распространения события. Если элемента управления заносит в свойство Handled значение true, событие прекращает продвижение и не будет возникать в любых других элементах.

Туннелируемые события работают так же, как и пузырьковые, но в обратном направлении. Например, если бы событие MouseUp было туннельным (а это не так), то при щелчке на изображении в примере с меткой событие MouseUp возникло бы сначала в окне, затем в элементе Label, затем в StackPanel и так далее до достижения источника Image, т.е. изображения кнопки.

Туннелируемые события легко распознать: они начинаются со слова Preview. Более того, WPF обычно определяет события попарно. Это означает, что если имеется пузырьковое событие MouseUp, то, скорее всего, существует и туннелируемое событие PreviewMouseUp. Туннелируемые событие всегда возникает перед пузырьковым событием, как показано на рис. 5.43.

Рисунок 5.43 – Механизм пузырьковых и туннелируемых событий WPF.

5.3.3 Асинхронный шаблон на основе событий

Механизм событий, как шаблон взаимодействия объектов в ООП, является распространенных явлением в мире современного программирования. При программировании графических приложений с событием тесно связывают механизм обработки приложением действий пользователя. Но область применения событий обширней. Существуют шаблоны приложений, основанные на событийной модели. Широкое распространение использование событий получило в стеках технологий: .NET Framework, Java SE, Java EE.

При программировании многопоточных приложений .NET Framework говорят о так называемом асинхронном шаблоне на основе событий.

Пример асинхронного шаблона основан на использовании интерфейса IAsyncResult. В случае использования асинхронного обратного вызова поток обратного вызова отличается от вызывающего потока. В Windows Forms и WPF это становится проблемой, потому что элементы управления Windows Forms и WPF связываются с единственным потоком. В случае каждого из этих элементов управления методы могут вызываться только из того потока, с помощью которого был создан данный элемент управления. Это также означает и то, что при наличии фонового потока получать напрямую доступ к элементам управления пользовательского интерфейса из этого потока тоже нельзя.

Единственными методами в случае элементов управления Windows Forms, которые могут вызываться из потока, отличного от потока-создателя, являются Invoke(), Beginlnvoke() и Endlnvoke(), а также свойство InvokeRequired.

Методы Beginlnvoke() и Endlnvoke() являются асинхронными вариантами Invoke(). Эти методы переключаются на поток-создатель для вызова метода, назначенного параметру делегата, который может передаваться этим методам. Использовать эти методы не так-то просто, из-за

чего, начиная с версии .NET 2.0, и был добавлен новый компонент с новым асинхронным шаблоном на основе событий (event-based asynchronous pattern).

В случае применения асинхронного шаблона на основе событий асинхронный компонент предлагает метод с суффиксом Async; например, синхронный метод DoATask() в асинхронной версии будет иметь имя DoATaskAsync(). Для извлечения результирующей информации компоненту также необходимо определить событие с суффиксом Completed, например, DoATaskCompleted. Пока указанное в методе DoATask() действие выполняется в фоновом потоке, событие DoATaskCompleted срабатывает в том же потоке как вызывающий метод (caller).

В случае применения асинхронного шаблона на основе событий асинхронный компонент может при необходимости поддерживать функцию отмены и функцию отчета о ходе выполнения. Для обеспечения функции отмены необходимо использовать метод с именем CancelAsync(), а для обеспечения функции отчета о ходе выполнения – событие с суффиксом ProgressChanged, например, DoATaskProgressChanged.

Класс BackgroundWorker. Класс BackgroundWorker представляет собой одну из реализаций асинхронного шаблона на основе событий. Метод RunWorkerAsync() в этом классе асинхронно запускает метод, ассоциированный с событием DoWork. Сразу же после выполнения работы происходит србытие RunWorkerCompleted.

Класс BackgroundWorker может также поддерживать возможность отмены, если его свойство WorkerSupportsCancellation установлено в true. В таком случае выполнение работы может отменяться вызовом метода CancelAsync. При этом выполняющееся задание должно с помощью свойства CancellationPending удостовериться в том, действительно ли ему следует остановить выполнение своей работы. Подобно новому механизму отмены,

класс BackgroundWorker поддерживает осуществление отмены в кооперативной манере.

Класс BackgroundWorker умеет отображать информацию о ходе выполнения работы. Если выполняемая задача возвращает сведения о том, как продвигается ее выполнение, то нужную информацию можно получить через свойство WorkerReportsProgress. В таком случае можно назначить событие ProgressChanged для получения информации об изменениях в ходе выполнения. Задание должно сообщать о прохождении работы с помощью метода ReportProgress(), которому передается выполненный объем в процентах.

Рассмотрим реализацию механизма асинхронного выполнения потока приложения, который взаимодействует с интерфейсом пользователя.

Создадим приложение WPF (рис. 5.44).

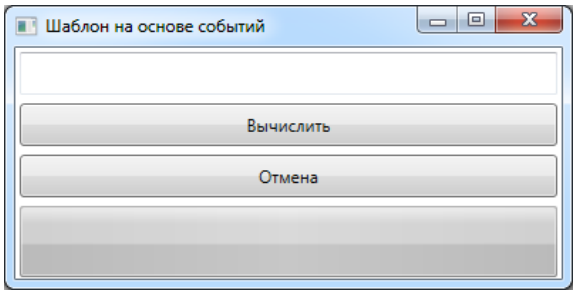


Рисунок 5.44 – Окно приложения WPF.

На рис. 5.45 представлен код XAML данного окна.

```
sign   ↑↓   XAML
<Window x:Class="WpfApplication1.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="Шаблон на основе событий" Height="200" Width="400">
    <Grid>
        <StackPanel Orientation="Vertical">
            <TextBox Margin="3" Height="30" Name="Input"></TextBox>
            <Button Margin="3" Height="30" Name="Calculate">Вычислить</Button>
            <Button Margin="3" Height="30" Name="Cancel">Отмена</Button>
            <ProgressBar Margin="3" Height="50" Name="Progress"/>
        </StackPanel>
    </Grid>
</Window>
```

Рисунок 5.45 – Код XAML окна WPF.

Пользователь вводит число в поле Input, нажимает кнопку Calculate и запускает фоновый процесс вычислений. Состояние процесса вычислений показывает прогресс-бар Progress.

Модифицируем исходный код класса MainWindow следующим образом:

```
public partial class MainWindow : Window
{
    private BackgroundWorker bw;

    0 references
    public MainWindow()
    {
        InitializeComponent();

        bw = new BackgroundWorker();
        bw.DoWork += bw_DoWork;
        bw.RunWorkerCompleted += (sender, args) =>
        {
            this.Title = args.Result.ToString();
            this.Calculate.IsEnabled = true;
            this.Cancel.IsEnabled = false;
        };
    }

    1 reference
    void bw_DoWork(object sender, DoWorkEventArgs e)
    {
        var input = (int)e.Argument;
        Thread.Sleep(5000);
        e.Result = input*input;
    }

    1 reference
    private void Calculate_Click(object sender, RoutedEventArgs e)
    {
        this.Calculate.IsEnabled = false;
        this.Title = String.Empty;
        this.Cancel.IsEnabled = true;
        this.Progress.Value = 0;
        bw.RunWorkerAsync(int.Parse(this.Input.Text));
    }
}
```

Рисунок 5.46 – Исходный код класса MainWindow.

В классе определено закрытое поле bw; в конструкторе класса выполнено создание объекта BackgroundWorker и осуществлена привязка методов к событиям DoWork и RunWorkerCompleted объекта bw. Обратите

внимание, что в качестве одного метода обработчика указывается метод самого класса окна, в качестве второго – лямбда-выражение.

Затем определяется обработчик нажатия кнопки Calculate.На данном этапе приложение готово к работе. Запустите и потестируйте приложение на разных значениях задержки.

Добавим в обработчик нажатия кнопки Cancel код отмены выполнения фонового потока:

```
private void Cancel_Click(object sender, RoutedEventArgs e)
{
    bw.CancelAsync();
}
```

Рисунок 5.47 – Отмена выполнения фонового потока.

Для того, чтобы полученный обработчик работал, необходимо установить свойство WorkerSupportsCancellation объекта bw в значение true. Это указывает, что фоновый поток поддерживает отмену выполнения.

Выполнение асинхронной задачи автоматически не отменяется. Поэтому внутри обработчика bw_DoWork(), отвечающего за ее выполнение, необходимо так изменить реализацию, чтобы она предусматривала проверку свойства CancellationPending элемента управления BackGroundWorker bw. Значение для этого свойства устанавливается сразу же, как только вызывается метод CancelAsync(). В случае, если значение этого свойства показывает, что запрос отмены ожидает обработки, нужно установить свойство Cancel параметра DoWorkEventArgs в true и выйти из обработчика.

```
void bw_DoWork(object sender, DoWorkEventArgs e)
{
    var input = (int)e.Argument;
    for (int i = 0; i < 10; i++)
    {
        Thread.Sleep(500);
        if (bw.CancellationPending)
        {
            e.Cancel = true;
            return;
        }
    }

    e.Result = input * input;
}
```

Рисунок 5.48 – Модификация метода bw_DoWork().

Необходимо модифицировать обработчик события RunWorkerCompleted объекта bw для корректного вывода результата в случае прерывания фонового потока.

```
bw.RunWorkerCompleted += (sender, args) =>
{
    if (args.Cancelled)
    {
        this.Title = "Выполнение операции было отменено";
    }
    else
    {
        this.Title = args.Result.ToString();
    }

    this.Calculate.IsEnabled = true;
    this.Cancel.IsEnabled = false;
};
```

Рисунок 5.49 – Обработчик события RunWorkerCompleted объекта bw.

Запустите приложение и протестируйте разработанный механизм отмены и запуска фонового потока. На данном этапе остался не реализован механизм визуализации хода выполнения фоновой задачи с использованием прогрессбара.

Для реализации подобного механизма необходимо выполнить следующие действия:

1. Разрешить передачу информации о ходе выполнения фонового процесса. Для этого необходимо установить свойство WorkerReportsProgress в значение true.

```
bw = new BackgroundWorker();
bw.WorkerSupportsCancellation = true;
bw.WorkerReportsProgress = true;
bw.DoWork += bw_DoWork;
bw.RunWorkerCompleted += (sender, args) =>
{
    if (args.Cancelled)
    {
        this.Title = "Выполнение операции был
```

Рисунок 5.50 – Разрешение передачи информации о ходе выполнения.

2. В методе bw_DoWork() добавить вызов метода отправки информации на каждой итерации цикла:

```
void bw_DoWork(object sender, DoWorkEventArgs e)
{
    var input = (int)e.Argument;
    for (int i = 0; i < 10; i++)
    {
        bw.ReportProgress(i*10);
        Thread.Sleep(500);
        if (bw.CancellationPending)
        {
            e.Cancel = true;
            return;
        }
    }

    e.Result = input * input;
}
```

Рисунок 5.51 – Отправка информации о ходе выполнения.

3. Метод ReportProgress класса BackgroundWorker генерирует событие ProgressChanged. Необходимо обработать это событие – написать код, которые будет менять текущее значение прогрессбара Progress. Сделаем это с использованием лямбда-выражения:

```
bw = new BackgroundWorker();
bw.WorkerSupportsCancellation = true;
bw.WorkerReportsProgress = true;
bw.ProgressChanged += (sender, args) =>
{
    Progress.Value = args.ProgressPercentage;
};
bw.DoWork += bw_DoWork;
bw.RunWorkerCompleted += (sender, args) =>
{
    if (args.Cancelled)
```

Рисунок 5.52 – Отправка информации о ходе выполнения.

На этом этапе выполнение задачи и постоение приложения с использованием асинхронного шаблона на основе событий завершено. Запустите приложение (рис. 5.53), протестируйте команды запуска и отмены процессов выполнения.

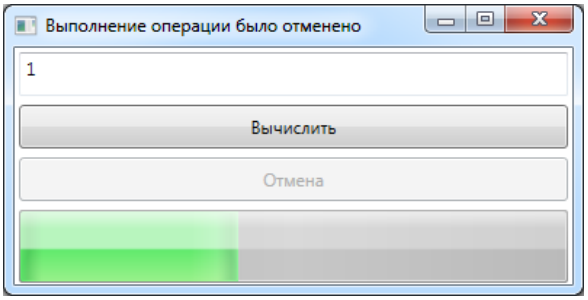


Рисунок 5.53 – Результирующее приложение.

Таким образом, можно сделать вывод, что современные библиотеки программного кода в полной мере инкапсулируют большинство шаблонов проектирования и поведения. Следует обратить внимание, что рассмотренный шаблон с использованием асинхронных событий превносит в объектно-ориентированное программирование параллелизм.

5.4 Шаблон Model-View-Controller (MVC)

MVC отделяет вид от модели, устанавливая между ними протокол взаимодействия «подписка/оповещение» (Publisher/Subscriber). Вид должен гарантировать, что внешнее представление отражает состояние модели. При

каждом изменении внутренних данных модель оповещает все зависящие от нее виды, в результате чего вид обновляет себя. Такой подход позволяет присоединить к одной модели несколько видов, обеспечив тем самым различные представления. Можно создать новый вид, не переписывая модель.

Еще одно свойство MVC заключается в том, что виды могут быть вложенными. Например, панель управления, состоящую из кнопок, допустимо представить как составной вид (содержащий вложенные представления кнопок).

MVC позволяет также изменять реакцию вида на действия пользователя. При этом визуальное представление остается прежним. Например, можно изменить реакцию на нажатие клавиши или использовать всплывающие меню вместо командных клавиш. MVC инкапсулирует механизм определения реакции в объекте контроллера. Обычно разрабатывается иерархия классов контроллеров, и это позволяет без труда создать новый контроллер как вариант уже существующего.

Вид пользуется экземпляром класса, производного от Controller, для реализации конкретной стратегии реагирования. Чтобы реализовать иную стратегию, нужно просто подставить другой контроллер. Можно даже заменить контроллер вида во время выполнения программы, изменив тем самым реакцию на действия пользователя. Например, вид можно деактивировать, так что он вообще не будет ни на что реагировать, если передать ему контроллер, игнорирующий события ввода.

Отношение вид-контроллер – это пример паттерна проектирования стратегия. Стратегия – это объект для представления алгоритма. Он полезен, когда необходимо статически или динамически подменить один алгоритм другим, если существует много вариантов одного алгоритма или когда с алгоритмом связаны сложные структуры данных, которые хотелось бы инкапсулировать.

В МУС используются различные паттерны проектирования: «Наблюдатель», «Компоновщик» и «Стратегия».

Каждый компонент Swing, производный от JComponent, построен по архитектуре подключаемого внешнего вида и поведения (pluggable look and feel architecture). В рамках данной архитектуры выполнено разделение данных компонента (component data) от его модели поведения (component behavior) и внешнего вида (component look). Таким образом, программист может определить особый внешний вид компонента, особенности его поведения, а также подключить компонент к модели данных.

На рис. 5.54 представлена схема компонента Swing. Компонент JComponent – это класс, с которым непосредственно взаимодействует приложение. Модель (Model) и представление (UI) скрыты (инкапсулированы) внутри компонента и используются им через внутренние механизмы.

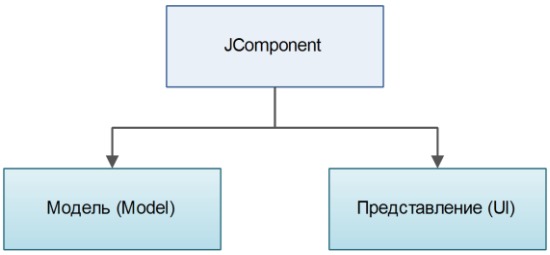


Рисунок 5.54 – Внутреннее представление компонента Swing.

Модель (Model) – класс, который содержит все внутренние атрибуты компонента (нажата кнопка или нет, активна или нет, какой пункт списка выбран и т.д.). Представление (UI) – обрабатывает события от пользователя, осуществляет рисование требуемого вида компонента. В данном случае в представлении (UI) заключены функции представления (view) и контроллера (controller) стандартного шаблона MVC. Два рассмотренных класса взаимодействуют друг с другом через механизм событий. Рассмотрим пример использования класса модели кнопки. На рис. 5.55 представлен

листинг программы, демонстрирующий создание четырех кнопок JButton и одной кнопки JToggleButton.

```
package lr10buttonmodel;
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class LR10ButtonModel extends JFrame {
    public LR10ButtonModel(){
        // Настраиваем форму
        super("Button Model");
        setSize(200, 200);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setLocationRelativeTo(null);
        setVisible(true);

        // Определяем выравнивание компонентов
        Container c = this.getContentPane();
        c.setLayout(new FlowLayout(FlowLayout.CENTER));

        // Создаем первую кнопку и получаем ее модель
        JButton btn1 = new JButton("Кнопка 1");
        ButtonModel model = btn1.getModel();

        // Создаем остальные кнопки на основе модели первой
        JButton btn2 = new JButton("Кнопка 2");
        btn2.setModel(model);
        JButton btn3 = new JButton("Кнопка 3");
        btn3.setModel(model);
        JButton btn4 = new JButton("Кнопка 4");
        btn4.setModel(model);

        // Добавляем кнопки на панель
        c.add(btn1);
        c.add(btn2);
        c.add(btn3);
        c.add(btn4);

        // Создаем кнопку изменения состояния модели
        JToggleButton btnOnOff = new JToggleButton("Отключить модель", true);
        btnOnOff.addItemListener(new ItemListener() {
            @Override
            public void itemStateChanged(ItemEvent e) {
                if(e.getStateChange() == 1){
                    btnOnOff.setText("Отключить модель");
                    model.setEnabled(true);
                } else {
                    btnOnOff.setText("Включить модель");
                    model.setEnabled(false);
                }
            }
        });
        c.add(btnOnOff);
    }

    public static void main(String[] args) {
        new LR10ButtonModel();
    }
}
```

Рисунок 5.55 – Применение модели кнопки.

В строке 21 получаем модель (объект класса ButtonModel) от кнопки btn1. В строках кода 23 – 35 создаются оставшиеся три кнопки и методом add все кнопки добавляются на форму. В строке 38 создается кнопка JToggleButton с двумя состояниями (нажата или отпущена). Кнопка btnOnOff отображает состояние модели: если модель кнопок JButton активна, то кнопка btnOnOff нажата, иначе – отпущена. Для btnOnOff в строках 39 – 50 создается анонимный класс-слушатель ItemListener. Метод itemStateChanged данного класса переопределяется с целью установки нового текста на управляющую кнопку, а также для отключения модели.

Следует также обратить вникание на класс слушателя для btnOnOff: в данном случае не применяется простейший класс ActionListener, так как данный класс обрабатывает просто событие нажатия кнопки, а не смены ее состояния. Запустите приложение. Поясните, почему все кнопки отключаются и включаются одновременно. Поясните также, почему при нажатии любой кнопки JButton остальные тоже нажимаются.

Создание приложения в рамках архитектуры MVC. Рассмотрим пример приложения, в котором произведено разделение модели, представление и контроллера (то есть спроектированное с использованием архитектурного шаблона MVC).

Пример. Требуется разработать приложение, отражающее процессы поступления товара в магазине. Пользователь приложения формирует редактируемый список товарных групп («Продукты», «Бытовая химия», «Одежда», «Фрукты», «Полиграфия»). По факту поступления товара пользователь формирует запись со следующими полями: «Наименование товара», «Партия», «Товарная группа», «Цена» «Количество». Описание полей представлены в таблице 5.9.

Таблица 5.9 – Поля записи регистра поступления товаров.

Поле записи	Описание	Тип данных
Наименование товара	Пользователь заполняет самостоятельно.	String
Партия	Документ (приходная накладная) в соответствии с которым принят товар. Пользователь может ввести самостоятельно или выбрать из уже имеющихся партий.	String
Товарная группа	Пользователь может выбрать только из имеющегося справочника товарных групп.	String
Цена	Пользователь вводит самостоятельно.	float
Количество	Пользователь вводит самостоятельно.	float

В программе должен быть реализован автоматический расчет сумм по каждой партии и товарной группе.

Решение. Для решения этой задачи целесообразно применение технологий баз данных (или файлового ввода-вывода), но в данном пособии необходимо сконцентрировать внимание на классах, реализующих шаблон MVC. Информация не будет сохраняться. Для решения данной задачи необходимо выполнить следующие шаги:

1. Создайте проект приложения Java с главным классом.

2. Начнем реализацию задачи с проектирования интерфейса. Создайте новый пакет «view» и добавьте новую форму JFrame с именем «MainFrame» в данный пакет. На рис. 5.56 показан внешний вид формы в режиме проектирования, а также имена переменных программы, ассоциированных с визуальными компонентами.

В соответствии с условием задания у элемента comboDoc (выпадающий список «Партия») необходимо установить свойство editable в значение true. На рис. 5.57 представлено дерево элементов формы.

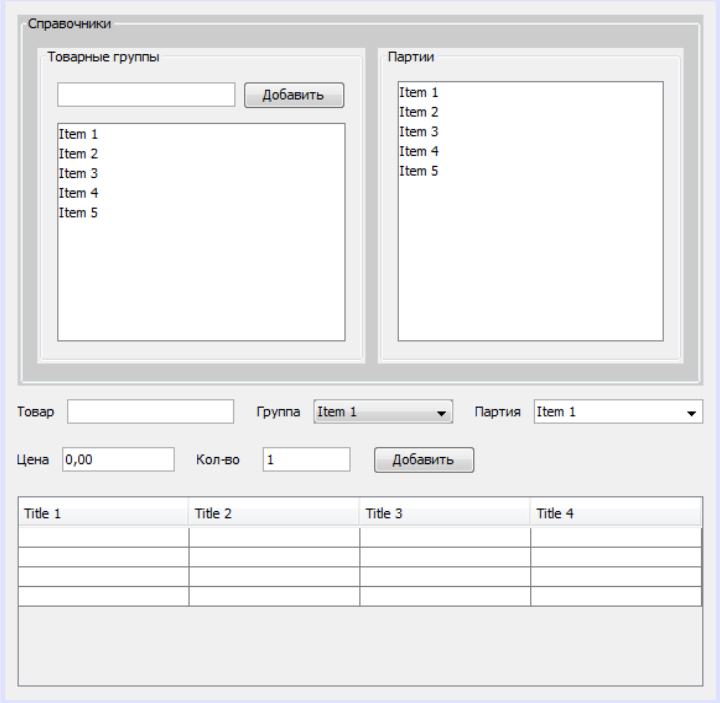


Рисунок 5.56 – Главное окно приложения.

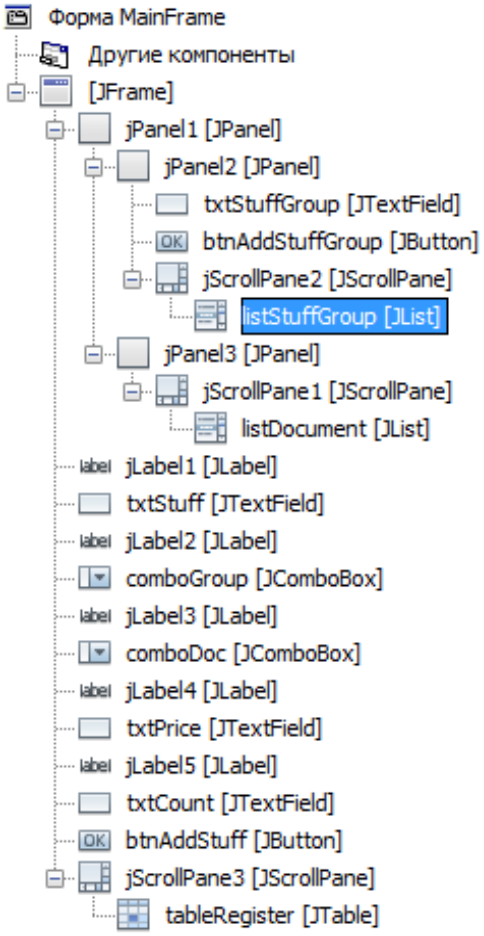


Рисунок 5.57 – Окно «Навигатор».

3. Реализуем классы, которые будут представлять модель данных (data model). Для представления справочников товаров, списка партий, и записей приходного регистра целесообразно использование отдельных классов.

4. Создайте пакет с именем «model»

5. Для представления справочника «Товарный группы» создадим класс StuffGroups в пакете model и реализуем в нем методы инициализации, добавления, удаления и возврата всех записей. На рис. 5.58 представлен листинг данного класса.

```
1   package model;
2   import javax.swing.DefaultListModel;
3
4   public class StuffGroups extends DefaultListModel<String> {
5       // Метод возвращает новый объект StuffGroups
6       public static StuffGroups createNewStuffGroups(){
7           StuffGroups gr = new StuffGroups();
8           return gr;
9       }
10  }
```

Рисунок 5.58 – Класс StuffGroups.

6. Для представления справочника «Партии» добавьте класс Documents в пакет model (рис. 5.59).

```
1   package model;
2   import javax.swing.DefaultListModel;
3
4   public class Documents extends DefaultListModel<String>{
5       // Метод возвращает новый объект Documents
6       public static Documents createNewDocuments(){
7           Documents d = new Documents();
8           return d;
9       }
10  }
```

Рисунок 5.59 – Класс Documents.

7. Для представления приходного регистра предназначен класс Register (рис. 5.60).

```
4   public class Register extends DefaultTableModel {
5       public static Register createNewRegister(){
6           Register r = new Register();
7           r.setColumnIdentifiers(
8                   new String[] {"Товар","Группа",
9                       "Партия","Цена", "Кол-во", "Сумма"});
10          return r;
11      }
12  }
```

Рисунок 5.60 – Класс Register.

8. Классы Register, Documents и StuffGroups объединим в один класс модели MyModel (в пакете model). Листинг класса MyModel представлен на рис. 5.61.

```
1   package model;
2   import javax.swing.DefaultComboBoxModel;
3
4   public class MyModel {
5       // Закрытые данные модели
6       private Register rData;
7       private StuffGroups sData;
8       private Documents dData;
9       private DefaultComboBoxModel sDataCombo, dDataCombo;
10      public MyModel(){
11          // Инициализация данных модели
12          rData = Register.createNewRegister();
13          sData = StuffGroups.createNewStuffGroups();
14          dData = Documents.createNewDocuments();
15          sDataCombo = new DefaultComboBoxModel();
16          dDataCombo = new DefaultComboBoxModel();
17          // Начальное заполнение
18          addNewStuffGroup("Продукты"); addNewStuffGroup("Полиграфия");
19          addNewStuffGroup("Одежда"); addNewStuffGroup("Бытовая химия");
20      }
21      // Открытые данные модели
22      public Register getRegister() { return rData; };
23      public StuffGroups getStuffGroups() { return sData; };
24      public Documents getDocuments() { return dData; };
25      // Модели для полей со списком
26      public DefaultComboBoxModel getDocumentsCombo(){ return dDataCombo; }
27      public DefaultComboBoxModel getStuffGroupsCombo(){ return sDataCombo; }
```

```
28          // Добавление товарной категории
29          public void addNewStuffGroup(String s){
30              if (!sData.contains(s)){
31                  // Добавляем в модель ListBox
32                  sData.add(0, s);
33                  // Добавляем в модель ComboBox
34                  sDataCombo.addElement(s);
35              }
36          }
37          // Добавление партии
38          public void addNewDocument(String s){
39              if(!dData.contains(s)){
40                  // Добавляем в модель ListBox
41                  dData.add(0, s);
42                  // Добавляем в модель ComboBox
43                  dDataCombo.addElement(s);
44              }
45          }
46          // Добавление записи
47          public void addNewStuff(String name, String group,
48                  String doc, float price, float count){
49              rData.addRow(new String[]{ name, group, doc,
50                  String.valueOf(price), String.valueOf(count)});
51          }
52      }
```

Рисунок 5.61 – Класс MyModel.

В данном классе модели данных реализованы только методы добавления данных, а также отсутствует механизм сохранения данных в файлах или в базе данных.

После реализации элементов модели и представления, свяжем эти компоненты контроллером. Контроллер реализуем через механизм событий.

9. Для привязки модели к интерфейсу модифицируем класс MainFrame (рис. 5.62).

```
1  package view;
2  import model.*;
3
4  public class MainFrame extends javax.swing.JFrame {
5
6      private MyModel model;
7
8      public MainFrame(MyModel pModel) {
9          // Установка модели
10         this.model = pModel;
11
12         // Вызов сгенерирован средой разработки
13         initComponents();
14
15         // Задание моделей для компонентов формы
16         if (this.model != null){
17             listStuffGroup.setModel(model.getStuffGroups());
18             comboDoc.setModel(model.getDocumentsCombo());
19             comboGroup.setModel(model.getStuffGroupsCombo());
20             listDocument.setModel(model.getDocuments());
21             tableRegister.setModel(model.getRegister());
22         }
23     }
```

Рисунок 5.62 – Изменения в коде класса MainFrame.

На рис. 5.62 видно, что в класс MainFrame добавлено поле model типа MyModel. В конструкторе класса MainFrame изменения следующие: добавлен параметр конструктора, добавлен код (строки 16 – 21) инициализации модели элементов управления формы.

10. После изменения конструктора необходимо изменить функцию main() приложения как показано на рис. 5.63.

```
1  package lr10mvc;
2  import model.*;
3  import view.*;
4
5  public class LR10MVC {
6
7      public static void main(String[] args) {
8          MainFrame view = new MainFrame(new MyModel());
9          view.setVisible(true);
10     }
11 }
```

Рисунок 5.63 – Функция main().

В данном варианте в конструктор главной формы передается новый объект модели.

11. Запустите приложение обратите внимание на привязку списка товарных групп. Исправьте ошибки.

12. Реализуйте обработчики событий нажатия кнопок добавления новой товарной группы и добавления строки в таблицу поступления товаров (рис. 5.64).

```
248     private void btnAddStuffGroupActionPerformed(java.awt.event.ActionEvent evt) {
249         String text = txtStuffGroup.getText();
250         model.addNewStuffGroup(text);
251     }
252 
253     private void btnAddStuffActionPerformed(java.awt.event.ActionEvent evt) {
254         // Получаем значения, введенные пользователем
255         // Название товара
256         String stuffName = txtStuff.getText();
257         // Наименование товарной группы
258         String stuffGroup = comboGroup.getSelectedItem().toString();
259         // Партия
260         String doc = comboDoc.getSelectedItem().toString();
261         // Цена
262         float price = Float.parseFloat(txtPrice.getText());
263         // Количество
264         float count = Float.parseFloat(txtCount.getText());
265         // Добавляем данные в модель
266         // в модель таблицы
267         model.addNewStuff(stuffName, stuffGroup, doc, price, count);
268         // в модель списка документов
269         model.addNewDocument(doc);
270     }
```

Рисунок 5.64 – Обработчики событий.

13. Запустите приложение, обратите внимание на синхронное отображение данных в списках, списках выбора и таблице (рис. 5.65).

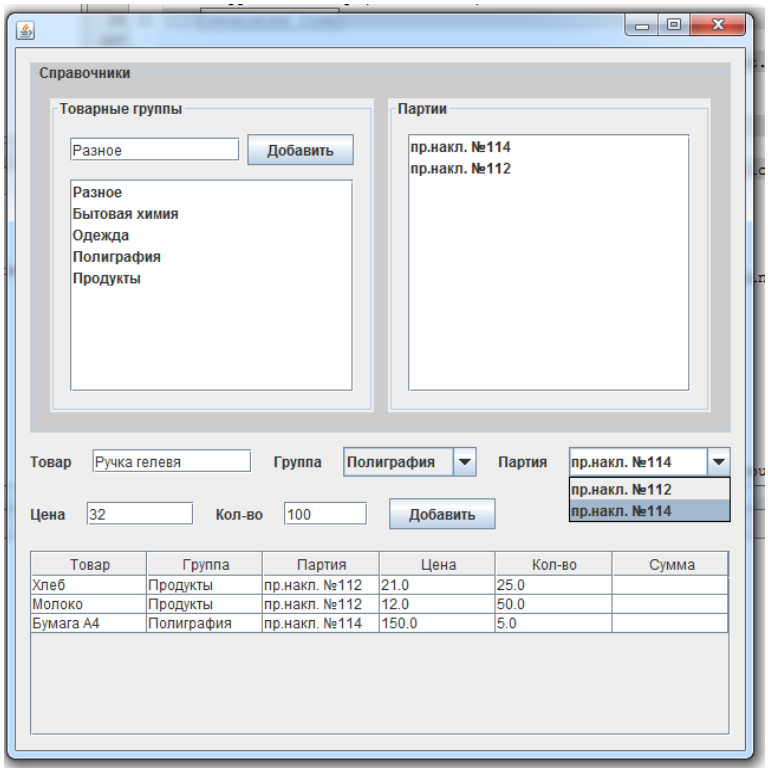


Рисунок 5.65 – Внешний вид работающего приложения.

Таким образом, в результате решения задачи получим приложение, разделенное на отдельные слои, каждый из которых может быть изменен независимо от остальных.

В рассмотренной реализации каждый слой использует интерфейс нижележащего слоя напрямую – не обеспечивается слабая связность слоев.

5.5 Многопоточность

Существуют различные аспекты объектно-ориентированного программирования, которые используются на практике, но их рассмотрение в рамках изучения языков программирования ограничено. К таковым относятся: использование потоков, лямбда-выражения, технологии объектно-реляционного отображения, технологии распределенного взаимодействия объектов. Это связано с достаточно сложной теоретической основой функционирования подобных систем.

В рамках данного учебного пособия рассмотрим отдельные механизмы современных объектно-ориентированных концепций.

Современные приложения немыслимы без многопоточных механизмов, позволяющих создавать эффективные приложения. Свои механизмы реализации многопоточности предлагают все объектно-ориентированные языки. Рассмотрим стратегии реализации многопоточных приложений в рамках технологии .NET Framework.

Наиболее простым способом для создания потока является определение делегата и его вызов асинхронным образом. Класс Delegate поддерживает и возможность асинхронного вызова методов. Для решения поставленной задачи класс Delegate создает отдельный поток.

Чтобы посмотреть в действии на асинхронные возможности делегатов, сначала нужно создать метод, выполнение которого занимает определенное время. Например, ниже показан метод TakesAWhile (), который благодаря вызову Thread.Sleep( ) выполняется минимум столько времени, сколько в миллисекундах задано во втором аргументе (рис. 5.66).

```
static int TakesAWhile(int data, int ms)
{
    Console.WriteLine("TakesAWhile запущен");
    Thread.Sleep(ms);
    Console.WriteLine("TakesAWhile завершен");
    return ++data;
}
```

Рисунок 5.66 – Внешний вид работающего приложения.

Чтобы обеспечить вызов этого метода из делегата, понадобится определить тип делегата с тем де самым параметром и возвращаемым типом:

```
public delegate int TakesAWhileDelegate(int data, int ms);
```

Теперь можно применять различные приемы для асинхронного вызова данного делегата и возврата результатов.

Одним из таких приемов является опрос и проверка, завершил ли делегат свою работу. Созданный класс delegate предоставляет метод

BeginInvoke ( ), в котором могут передаваться входные параметры, определенные вместе с типом делегата. Метод Beginlnvoke ( ) всегда имеет два дополнительных параметра типа AsyncCallback и object, которые будут рассматриваться позже. Сейчас главный интерес представляет возвращаемый тип метода – IAsyncResult. С помощью IAsyncResult можно извлекать информацию о делегате и проверять, завершил ли он свою работу, что и демонстрируется в примере с применением свойства IsCompleted. Цикл while продолжает выполняться в главном потоке программы до тех пор, пока делегат не завершит работу.

```
static void Main(string[] args)
{
    // синхронный вызов метода
    // TakesAWhileA, 3000);

    // асинхронный вызов метода с применением делегата
    TakesAWhileDelegate dl = TakesAWhile;

    IAsyncResult ar = dl.BeginInvoke(1, 3000, null, null);
    while (!ar.IsCompleted)
    {
        // выполнение еще каких-нибудь Сераций в главном потоке
        Console.Write(".");
        Thread.Sleep(50);
    }
    int result = dl.EndInvoke(ar);

    // вывод результата
    Console.WriteLine("result: {0}", result);
}
```

Рисунок 5.67 – Внешний вид работающего приложения.

После запуска этого приложения можно увидеть, что главный поток и поток делегата выполняются параллельно, а после завершения работы потока делегата главный поток прекращает проход по циклу. Если главный поток завершает выполнение, не дожидаясь завершения работы делегата, поток делегата останавливается.

В данном разделе методических указаний рассмотрен самый простой метод запуска метода в отдельном потоке – с использованием асинхронного делегата. Рассмотрен также механизм передачи параметров в поток,

выполняемый параллельно (асинхронно); возврат результата из асинхронно выполняемого метода. Необходимо обратить внимание на реализацию механизма ожидания завершения работы асинхронного метода.

Другой способ ожидания результата от асинхронного делегата предусматривает применение дескриптора ожидания (wait handle), который ассоциируется с IAsyncResult.

Получить доступ к этому дескриптору ожидания можно через свойство AsyncWaitHandle. Это свойство возвращает объект типа WaitHandle, с помощью которого можно организовать ожидание завершения работы потоком делегата. Метод WaitOne ( ) принимает в качестве первого необязательного параметра значение тайм-аута, в котором можно указать максимальный период времени ожидания. В рассматриваемом здесь примере этот период составляет 50 миллисекунд.

```
static void Main(string[] args)
{
    TakesAWhileDelegate dl = TakesAWhile;
    IAsyncResult ar = dl.BeginInvoke(1, 3000, null, null);
    while (true)
    {
        Console.Write (" . ") ;
        if (ar.AsyncWaitHandle.WaitOne(50, false))
        {
            Console.WriteLine("Можно извлечь результат сейчас");
            break;
        }
    }

    int result = dl.EndInvoke(ar);

    // вывод результата
    Console.WriteLine("result: {0}", result);
}
```

Рисунок 5.68 – Внешний вид работающего приложения.

В случае возникновения тайм-аута метод WaitOne ( ) возвращает значение false, и проход по циклу while продолжается. Если во время ожидания до тайм-аута дело не доходит, осуществляется выход из цикла while с помощью break и получение результата методом Endlnvoke ( ). В

пользовательском интерфейсе результат выглядит примерно так же, как в предыдущем примере, отличается лишь способ реализации ожидания.

Еще один способ для ожидания результатов от делегата заключается в применении так называемого асинхронного обратного вызова (asynchronous callback). С помощью третьего параметра в методе BeginInvoke можно передать метод, удовлетворяющий требованиям делегата AsyncCallback. Делегат AsyncCallback требует определять параметр IAsnycResult и возвращаемый тип void. В рассматриваемом здесь примере третьему параметру назначается адрес метода TakesAWhileCompleted, который удовлетворяет требованиям делегата AsyncCallback. В последнем параметре методу Beginlnvoke можно передать объект, к которому будет производиться доступ из метода обратного вызова. Здесь удобно передать экземпляр самого делегата, так что метод обратного вызова сможет использовать его для получения результата асинхронного метода.

```
static void Main(string[] args)
{
    TakesAWhileDelegate dl = TakesAWhile;
    dl.BeginInvoke(1, 3000, TakesAWhileCompleted, dl) ;
    for (int i = 0; i < 100; i++)
    {
        Console.Write(".");
        Thread.Sleep(50);
    }
}

static void TakesAWhileCompleted(IAsyncResult ar)
{
    if (ar == null) throw new ArgumentNullException("ar");
    TakesAWhileDelegate dl = ar.AsyncState as TakesAWhileDelegate;

    // Неверный тип объекта
    Trace.Assert (dl != null, "Invalid object type");

    int result = dl.EndInvoke(ar);

    // вывод результата
    Console.WriteLine("result: {0}", result);

}
```

Рисунок 5.69 – Внешний вид работающего приложения.

По завершении работы делегата TakesAWhileDelegate сразу же вызывается метод TakesAWhileCompleted ( ). Ожидать результатов внутри главного потока нет никакой необходимости. Завершать главный поток перед завершением работы потоков делегатов может быть необязательно, если только остановка работы потоков делегатов при завершении выполнения главного потока не представляет проблемы.

Метод TakesAWhileCompleted ( ) определяется с типами параметров и возврата, которые указаны в делегате AsyncCallback. Последний параметр, передаваемый в Beginlnvoke ( ), может быть прочитан с помощью ar.AsyncState, а результат получен вызовом в TakesAWhileDelegate метода Endlnvoke.

Вместо того чтобы определять отдельный метод и передавать его методу Beginlnvoke ( ), можно воспользоваться лямбда-выражением. Параметр ar имеет тип IAsyncResult. В такой реализации нет необходимости присваивать значение последнему параметру метода Beginlnvoke ( ), потому что лямбда-выражение может напрямую получать доступ к переменной d1, находящейся за пределами контекста данного метода. Но блок реализации лямбда-выражения все равно должен вызываться из потока делегата, что может и не быть очевидным при определении метода подобным образом:

```
static void Main(string[] args)
{
    TakesAWhileDelegate dl = TakesAWhile;
    dl.BeginInvoke (1, 3000, ar =>
    {
        int result = dl.EndInvoke(ar);
        // вывод результата
        Console.WriteLine("result: {0}", result);
    },
    null);

    for (int i = 0; i < 100; i++)
    {
        Console.Write(".");
        Thread.Sleep(50);
    }
}
```

Рисунок 5.70 – Внешний вид работающего приложения.

Модель программирования и приемы, описанные для асинхронных делегатов – опрос, дескрипторы ожидания и асинхронные вызовы доступны не только для делегатов. Эта же модель программирования, которая называется шаблоном асинхронного вызова (asynchronous pattern), встречается в разнообразных местах .NET Framework. Например, с помощью метода BeginGetResponse ( ) класса HttpWebRequest можно асинхронно отправлять HTTP-запросы, а с помощью метода BeginExecuteReader ( ) класса SqlCommand – запросы к базе данных. Параметры похожи на те, что можно передавать в методе Beginlnvoke ( ) делегата, а механизмы, применяемые для получения результатов, выглядят точно так же.

Контрольные вопросы

1. Что такое событие?
2. Что такое контейнер компоновки? Для чего они используются? Опишите свойства класса Panel.
3. Какие типы событий WPF вы знаете? Дайте определение каждого типа WPF
4. Какими способами можно подключить событие? Как можно отсоединить обработчик события от элемента управления?
5. Какие элементы управления вы знаете?
6. Что означают понятия «элемент управления содержимым», «контент», «панель», «контейнер компоновки»?
7. Какие элементы управления содержимым вы знаете? Поясните назначение и отличительные особенности каждого из них.
8. Какие основные компоненты событийной модели можно выделить в технологии Java? Опишите каждый элемент.
9. Назовите основные стратегии построения приложения, основанного на событийной модели.

10. Опишите механизм обработки событий с использованием внутреннего класса-слушателя и в случае применения анонимных классов.

11. Назовите основные особенности библиотеки Swing.

12. Назовите основные компоненты JFC.

13. Как соотносятся библиотеки Swing и AWT? В каких пакетах содержатся классы данных библиотек? Назовите часто используемые классы данных библиотек.

14. Назовите основные методы, доступные в классе JFrame.

15. Поясните назначение метода setDefaultCloseOperation. Какие варианты поведения формы можно реализовать с его использованием?

16. Какие возможные подходы к созданию графического приложения на основе Swing можно выделить? Какой подход наиболее удобен?

17. Внимательно изучите асинхронный шаблон на основе событий. Какие изученные структурные, поведенческие или порождающие паттерны можно выявить в его структуре?

ЗАКЛЮЧЕНИЕ

Развитие информационных технологий затрагивает не только аппаратные системы; происходит интенсивное развитие программных систем. Объектно-ориентированные технологии также быстро развиваются. Новые возможности, доступные в языках программирования Java и C#, увеличивают разницу между «новыми» языками программирования и «традиционными» языками ООП (C++). Появление таких инструментов как делегаты, лямбда-выражения, объектно-реляционное отображение, распределенные объектные системы, развитые модели событий, стимулировало бурное развитие новых паттернов проектирования и реинжиниринг существующих.

В сложном мире ООП необходимо хорошо ориентироваться. Для этого необходимо понимать принципы разработки приложений, такие как S.O.L.I.D. Современному программисту необходимо обладать обширными теоретическими знаниями в области паттернов проектирования, без которых невозможна разработка сложных коммерческих приложений.

Разработка графических приложений и веб-приложений представляют собой пример наиболее сложных областей. При проектировании, реализации, тестировании, отладке и сопровождении подобных объектных систем программист должен задействовать максимальный объем доступного инструментария: модульное тестирование, рефакторинг, методы разработки посредством тестирования и, конечно, большинство паттернов и принципов S.O.L.I.D.

Каждая технология, предназначенная для построения развитого графического интерфейса пользователя, представляет собой синергетическое единство различных паттернов: поведенческих, структурных и порождающих.

В результате изучения материала. Представленного в данном учебном пособии, студент может приступить к внедрению паттернов проектирования при решении учебных и практических задач. Поэтапные методики проектирования и разработки приложений, изложенные в пособии, позволяют максимально эффективно проводить анализ предметной области и избегать различных ошибок при реализации программных систем.

СПИСОК ЛИТЕРАТУРЫ

Список основной литературы

1. Шилдт, Г. Java. Полное руководство. 8-е изд. / Г. Шилдт; пер. с. англ. В.А. Коваленко. – М.: ООО «И.Д. Вильямс», 2012. – 1104 с.

2. Корнелл, Г. Java. Библиотека профессионала, том 1. Основы. 9-е изд. / Г. Корнелл, К.С. Хорстманн; пер. с. англ. И.В. Берштейн. – М.: ООО «И.Д. Вильямс», 2014. – 864 с.

3. Корнелл, Г. Java. Библиотека профессионала, том 2. Расширенные средства. 9-е изд. / Г. Корнелл, К.С. Хорстманн; пер. с. англ. И.В. Берштейн. – М.: ООО «И.Д. Вильямс», 2014. – 864 с.

4 Язык программирования C# 5.0 и платформа .NET 4.5. 6-е изд. Пер. с англ. / Э. Троелсен. – М.:Изд. «Издательский дом Вильямс». 2013. – 1312 с.

5. Албахари Дж. C# 5.0. Справочник. Полное описание языка / Дж. Албахари, Б. Албахари. – М.:Изд. «Издательский дом Вильямс». 2013. – 1008 с.

Список дополнительной литературы

6. Стиллмен Э. Изучаем C#. 3-е изд. / Э. Стиллмен, Дж. Грин. – СПб.: Питер. 2014. – 816 с.

7. Флентов, М. Библия C#. 2-е изд. / М. Флентов. – СПб.: БХВ-Петербург, 2011. – 560 с.

8. Ватсон, Б. C# 4.0 на примерах: пер. с англ. / Б. Ватсон. – СПб.: БХВ-Петербург, 2011. – 608 с.

9. Буч, Г. Объектно-ориентированный анализ и проектирование с примерами приложений, 3-е изд.: Пер. с англ. – М.: 000 «И.Д. Вильяме», 2008. – 720 с.: ил. – Парал. тит. англ.

10. Петцольд Ч. Программирование для Microsoft Windows 8. 6-е изд. / Ч. Петцольд. – СПб.: Питер. 2013 – 1008 с.

11. Гранд, М. Шаблоны проектирования в Java / М. Гранд; пер. с. англ. С. Беликовой. – М.: Новое знание, 2004. – 559 с.

12. Монахов, В. Язык программирования Java и среда NetBeans. 3-е изд., перераб. и доп. / В.В. Монахова. – СПб.: БХВ-Петербург, 2011. – 704 с.

13. Блох, Дж. Java. Эффективное программирование. 2-е изд. / Дж. Блох; пер. с. англ. Е. Коротылева. – М.: Лори, 2014. – 461 с.

14. Блинов, И. Java: методы программирования: уч.-мет. Пособие / И.И. Блинов, В.С. Романчик. – Минск: «Четыре четверти», 2013. – 896 с.

учебное издание

ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ

УЧЕБНОЕ ПОСОБИЕ

Автор

Николаев Евгений Иванович

Редактор, техническая правка,

компьютерная верстка:

Подписано в печать _____г.

Формат 60х84 1/16 Усл.п.л. – 4,25. Уч.-изд.л. – 3,4.

Бумага газетная. Печать офсетная. Заказ № ___ Тираж ___ экземпляров

ФГАОУ ВПО «Северо-Кавказский федеральный университет»

355000, г. Ставрополь, ул. Пушкина, 1

Издательство СКФУ