МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
федеральное государственное бюджетное образовательное учреждение высшего образования
«КУЗБАССКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ ИМЕНИ Т. Ф. ГОРБАЧЕВА»
Филиал КузГТУ в г. Белово

Кафедра инженерно-экономическая

ПМ.02 Осуществление интеграции программных модулей
МДК.02.01 Технология разработки программного обеспечения

Методические рекомендации
по выполнению лабораторных работ
для специальности
09.02.07 «Информационные системы и программирование»

Составитель: Витвицкий М.Н.
Рассмотрены и утверждены на заседании кафедры
Протокол № 6 от 14.02.2026 г.
Рекомендовано учебно-методической комиссией специальностей СПО в качестве электронного издания для использования в учебном процессе
Протокол № 6 от 17.02.2026 г.

Долганова Жанна Александровна
Подписано цифровой подписью: Долганова Жанна Александровна
Дата: 2026.02.18 11:15:28 +07'00'

Белово 2026

СОДЕРЖАНИЕ

ОРГАНИЗАЦИЯ ЛАБОРАТОРНОЙ РАБОТЫ

Лабораторная работа обучающихся может рассматриваться как организационная форма обучения, обеспечивающих управление учебной деятельностью или деятельность обучающихся по освоению общих и профессиональных компетенций, знаний и умений учебной и научной деятельности на примере выполнения тематических практических заданий.

Лабораторной работа обучающихся проводится с целью: систематизации и закрепления полученных теоретических знании и закрепления практических навыков студентов. В рамках дисциплины «Технология разработки программного обеспечения», студент должен закрепить на примере, разработанных заданий следующие навыки и умения:

1. Изучение основных принципов процесса разработки программного обеспечения.

2. Изучение встроенных и основных специализированные инструментов анализа качества программных продуктов.

3. Использование специализированных графических средств построения и анализа архитектуры программных продуктов.

4. Умение выполнять ручное и автоматизированное тестирование программного модуля.

ПЛАНИРОВАНИЕ ЛАБОРАТОРНЫХ РАБОТ

При разработке рабочей программы по учебной дисциплине или профессиональному модулю при планировании содержания практической работы преподавателей устанавливается содержание и объем учебной информации или практических заданий, которые выносятся на лабораторную работу, определяются формы и методы контроля результатов.

Содержание лабораторной работы определяется в соответствии с рекомендуемыми видами заданий согласно программе учебной дисциплины модуля.

В соответствии с ведущей дидактической целью содержанием практических занятий являются решение разного рода задач, в том числе профессиональных (анализ производственных ситуаций, решение ситуационных производственных задач, выполнение профессиональных функций в деловых играх и т.п.), выполнение вычислений, расчетов, чертежей, работа с измерительными приборами, оборудованием, аппаратурой, работа с нормативными документами, инструктивными материалами, справочниками, составление проектной, плановой и другой технической и специальной документации и др.

Виды заданий для лабораторной работы, их содержание и характер имеют вариативный и дифференцированный характер, учитывают специфику данной дисциплины и индивидуальные особенности студента.

Перед выполнением студентами лабораторной работы преподаватель проводит инструктаж по выполнению задания, который включает цель задания, его содержание, сроки выполнения, ориентировочный объем работы, основные требования к результатам работы, критерии оценки. В процессе инструктажа преподаватель предупреждает обучающихся о возможных типичных ошибках, встречающихся при выполнении задания. Инструктаж проводится преподавателем за счет объема времени, отведенного на изучение дисциплины.

Лабораторная работа может осуществляться индивидуально или группами обучающихся в зависимости от цели, объема, конкретной тематики работы, уровня сложности уровня умений обучающихся.

Отчет по лабораторной работе обучающихся предоставляется в электронном виде.

КОНТРОЛЬ РЕЗУЛЬТАТОВ ВЫПОЛНЕНИЯ ЛАБОРАТОРНОЙ РАБОТЫ

Контроль результатов лабораторной работы студентов осуществляться в пределах времени, отведенного на обязательные учебные занятия по дисциплине и практическую работу обучающихся по дисциплине, может проходить в письменной, устной или смешанной форме, с представлением продукта деятельности учащегося.

В качестве форм и методов контроля работы обучающихся, могут быть использованы, зачеты, тестирование, самоотчеты, контрольные работы, защита творческих работ и др., которые могут осуществляться на учебном занятии или вне его (например, оценки за реферат).

Критериями оценки результатов работы обучающегося являются:

- уровень освоения учащимся учебного материала;
- умение обучающегося использовать теоретические знания при выполнении задач;
- сформированность общих и профессиональных компетенций;
- обоснованность и четкость изложения ответа;
- оформление материала в соответствии с требованиями.

МЕТОДИЧЕСКИЕ МАТЕРИАЛЫ

ЛАБОРАТОРНАЯ РАБОТА

Лабораторная работа - это вид учебного занятия или практического задания, в ходе которого обучающиеся под руководством преподавателя проводят опыты, измерения или исследования для закрепления и применения теоретических знаний на практике. Она подразумевает самостоятельную работу с материальными объектами или моделями, развитие практических и интеллектуальных навыков (планирование, наблюдение, анализ) и получение практического опыта.

Цель лабораторной работы

Применение знаний - усвоение теоретического материала через практическое выполнение заданий.

Формирование навыков - развитие экспериментальных, измерительных и аналитических умений.

Проверка знаний - экспериментальное подтверждение теоретических положений.

Получение опыта - развитие навыков работы с приборами и технологиями, а также умения делать выводы на основе полученных данных.

Критерии оценки:

- соответствие теме;
- глубина проработки материала:
- правильность и полнота использования источников;
- оформление отчета.

Лабораторная работа состоит из 2 частей:

1. Теоретическое часть (теоретическая часть работы);
2. Практическое задание (выдается преподавателем индивидуально согласно перечню тем лабораторных работ).

Темы лабораторных работ

№ раздела (темы)	Вопросы, выносимые на самостоятельное изучение	Количество часов
ТЕМА 2.1.2. ОПИСАНИЕ И АНАЛИЗ ТРЕБОВАНИЙ. ДИАГРАММЫ IDEF, МЕТОДОЛОГИЯ UML.	Лабораторная работа №1. «Построение диаграммы Вариантов использования и диаграммы последовательности».	4
	Лабораторная работа №2. «Построение диаграммы Кооперации и диаграммы развертывания».	4
	Лабораторная работа №3. «Построение диаграммы Деятельности, диаграммы состояний и диаграммы классов».	4
	Лабораторная работа №4. «Построение диаграммы компонентов».	4
	Лабораторная работа №5. «Построение диаграмм потоков данных».	2
ТЕМА 2.1.3. ОЦЕНКА КАЧЕСТВА ПРОГРАММНЫХ СРЕДСТВ.	Лабораторная работа №6. «Разработка тестового сценария».	2
	Лабораторная работа №7. «Оценка необходимого количества тестов».	2
	Лабораторная работа №8. «Разработка тестовых пакетов».	2
	Лабораторная работа №9. «Оценка программных средств с помощью метрик».	2
	Лабораторная работа №10. «Инспекция программного кода на предмет соответствия стандартам кодирования».	2

Лабораторная работа №1. Построение диаграммы вариантов использования и диаграммы последовательности

Целью работы является изучение порядка построения диаграмм вариантов

использования и диаграммы последовательности.

Для выполнения работы студент должен изучить приведенный ниже теоретический материал. Отчет сдается в распечатанном и электронном (файл Word) видах.

Построение диаграммы Вариантов использования и диаграммы последовательности

Визуальное моделирование в UML можно представить, как некоторый процесс по уровневого спуска от наиболее обшей и абстрактной концептуальной модели исходной системы к логической, а затем и к физической модели соответствующей программной системы. Для достижения этих целей вначале строится модель в форме так называемой диаграммы вариантов использования (use case diagram), которая описывает функциональное назначение системы или, другими словами, то, что система будет делать в процессе своего функционирования. Диаграмма вариантов использования является исходным концептуальным представлением или концептуальной моделью системы в процессе ее проектирования и разработки.

Разработка диаграммы вариантов использования преследует цели:

- Определить общие границы и контекст моделируемой предметной области на начальных этапах проектирования системы.
- Сформулировать общие требования к функциональному поведению проектируемой системы.
- Разработать исходную концептуальную модель системы для ее последующей детализации в форме логических и физических моделей.
- Подготовить исходную документацию для взаимодействия разработчиков системы с ее заказчиками и пользователями.

Суть данной диаграммы состоит в следующем: проектируемая система представляется в виде множества сущностей или актеров, взаимодействующих с системой с помощью так называемых вариантов использования. При этом актером (actor) или действующим лицом называется любая сущность, взаимодействующая с системой извне. Это может быть человек, техническое устройство, программа или любая другая система, которая может служить источником воздействия на моделируемую систему так, как определит сам разработчик. В свою очередь, вариант использования (use case) служит для описания сервисов, которые система

предоставляет актеру. Другими словами, каждый вариант использования определяет некоторый набор действий, совершаемый системой при диалоге с актером. При этом ничего не говорится о том, каким образом будет реализовано взаимодействие актеров с системой.

Примечание

Рассматривая диаграмму вариантов использования в качестве модели системы, можно ассоциировать ее с моделью черного ящика». Действительно, подробная детализация данной диаграммы на начальном этапе проектирования скорее имеет отрицательный характер, поскольку предопределяет способы реализации поведения системы. А согласно рекомендациям, RUP именно эти аспекты должны быть скрыты от разработчика на диаграмме вариантов использования.

В самом общем случае, диаграмма вариантов использования представляет собой граф специального вида, который является графической нотацией для представления конкретных вариантов использования, актеров, возможно некоторых интерфейсов, и отношений между этими элементами. При этом отдельные компоненты диаграммы могут быть заключены в прямоугольник, который обозначает проектируемую систему в целом. Следует отметить, что отношениями данного графа могут быть только некоторые фиксированные типы взаимосвязей между актерами и вариантами использования, которые в совокупности описывают сервисы или функциональные требования к моделируемой системе.

Рациональный унифицированный процесс разработки модели сложной системы представляет собой разбиение ее на составные части с минимумом взаимных связей на основе выделения пакетов. В самом языке UML-пакет Варианты использования является под пакетом пакета Элементы поведения. Последний специфицирует понятия, при помощи которых определяют функциональность моделируемых систем. Элементы пакета вариантов использования являются первичными по отношению к тем, с помощью которых могут быть описаны сущности, такие как системы и подсистемы. Однако внутренняя структура этих сущностей никак не описывается. Базовые элементы этого пакета – вариант использования и актер. С этих понятий мы и приступим к изучению диаграмм вариантов использования.

Вариант использования

Конструкция или стандартный элемент языка UML вариант использования

применяется для спецификации общих особенностей поведения системы или любой другой сущности предметной области без рассмотрения внутренней структуры этой сущности. Каждый вариант использования определяет последовательность действий, которые должны быть выполнены проектируемой системой при взаимодействии ее с соответствующим актером. Диаграмма вариантов может дополняться пояснительным текстом, который раскрывает смысл или семантику составляющих ее компонентов. Такой пояснительный текст получил название примечания или сценария.

Отдельный вариант использования обозначается на диаграмме эллипсом, внутри которого содержится его краткое название или имя в форме глагола с пояснительными словами (Рисунок 1).

Рисунок 1. Графическое обозначение варианта использования

Цель варианта использования заключается в том, чтобы определить законченный аспект или фрагмент поведения некоторой сущности без раскрытия внутренней структуры этой сущности. В качестве такой сущности может выступать исходная система или любой другой элемент модели, который обладает собственным по ведением, подобно подсистеме или классу в модели системы.

Каждый вариант использования соответствует отдельному сервису, который предоставляет моделируемую сущность или систему по запросу пользователя (актера), т. е. определяет способ применения этой сущности. Сервис, который инициализируется по запросу пользователя, представляет собой законченную последовательность действий. Это означает, что после того как система закончит обработку запроса пользователя, она должна возвратиться в исходное состояние, в котором готова к выполнению следующих запросов.

Варианты использования описывают не только взаимодействия между пользователями и сущностью, но также реакции сущности на получение отдельных сообщений от пользователей и восприятие этих сообщений за пределами сущности. Варианты использования могут включать в себя описание особенностей способов

реализации сервиса и различных исключительных ситуаций, таких как корректная обработка ошибок системы. Множество вариантов использования в целом должно определять все возможные стороны ожидаемого поведения системы. Для удобства множество вариантов использования может рассматриваться как отдельный пакет.

С системно-аналитической точки зрения варианты использования могут применяться как для спецификации внешних требований к проектируемой системе, так и для спецификации функционального поведения уже существующей системы. Кроме этого, варианты использования неявно устанавливают требования, определяющие, как пользователи должны взаимодействовать с системой, чтобы иметь возможность корректно работать с предоставляемыми данной системой сервисами!

Примечание

Каждый выполняемый вариантом использования метод реализуется как неделимая транзакция, т.е. выполнение сервиса не может быть прервано никаким другим экземпляром варианта использования.

Применение вариантов использования на всех уровнях диаграммы позволяет не только достичь требуемого уровня унификации обозначений для представления функциональности подсистем и системы в целом, но и является мощным средством последовательного уточнения требований к проектируемой системе на основе по уровневого спуска от пакетов системы к операциям классов. С другой стороны, модификация отдельных операций класса может оказать обратное влияние на уточнение сервиса соответствующего варианта использования, т. е. реализовать эффект обратной связи с целью уточнения спецификаций или требований на уровне пакетов системы.

В метамодели UML вариант использования является подклассом классификатора, который описывает последовательности действий, выполняемых отдельным экземпляром варианта использования. Эти действия включают изменения состояния и взаимодействия со средой варианта использования. Эти последовательности могут описываться различными способами, включая такие, как графы деятельности и автоматы.

Примерами вариантов использования могут являться следующие действия: проверка состояния текущего счета клиента, оформление заказа на покупку товара, получение дополнительной информации о кредитоспособности клиента, отображение

графической формы на экране монитора и другие действия.

Актеры

Актер представляет собой любую внешнюю по отношению к моделируемой системе сущность, которая взаимодействует с системой и использует ее функциональные возможности для достижения определенных целей или решения частных задач. При этом актеры служат для обозначения согласованного множества ролей, которые могут играть пользователи в процессе взаимодействия с проектируемой системой. Каждый актер может рассматриваться как некая отдельная роль относительно конкретного варианта использования. Стандартным графическим обозначением актера на диаграммах является фигурка «человечка», под которой записывается конкретное имя актера (Рисунок 2).

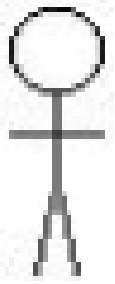


Рисунок2. Графическое обозначение актера

В некоторых случаях актер может обозначаться в виде прямоугольника класса с ключевым словом «актер» и обычными составляющими элементами класса. Имена актеров должны записываться заглавными буквами и следовать рекомендациям использования имен для типов и классов модели. При этом символ отдельного актера связывает соответствующее описание актера с конкретным именем. Имена абстрактных актеров, как и других абстрактных элементов языка UML, рекомендуется обозначать курсивом.

Примечание

Имя актера должно быть достаточно информативным с точки зрения семантики. Вполне подходят для этой цели наименования должностей в компании (например, продавец, кассир, менеджер, президент). Не рекомендуется давать актерам имена собственные (например, «О. Бендер») или моделей конкретных устройств (например, «маршрутизатор Cisco 3640»), даже если это с очевидностью следует из контекста проекта. Дело в том, что одно и то же лицо может выступать в нескольких ролях и, соответственно, обращаться к различным сервисам системы. Например, посетитель

банка может являться как потенциальным клиентом, и тогда он востребует один из его сервисов, а может быть и налоговым инспектором или следователем прокуратуры. Сервис для последнего может быть совершенно исключительным по своему характеру.

Примерами актеров могут быть: клиент банка, банковский служащий, продавец магазина, менеджер отдела продаж, пассажир авиарейса, водитель автомобиля, администратор гостиницы, сотовый телефон и другие сущности, имеющие отношение к концептуальной модели соответствующей предметной области.

Примечание

В метамодели актер является подклассом классификатора. Актеры могут взаимодействовать с множеством вариантов использования и иметь множество интерфейсов, каждый из которых может представлять особенности взаимодействия других элементов с отдельными актерами.

Актеры используются для моделирования внешних по отношению к проектируемой системе сущностей, которые взаимодействуют с системой и используют ее в качестве отдельных пользователей. В качестве актеров могут выступать другие системы, подсистемы проектируемой системы или отдельные классы. Важно понимать, что каждый актер определяет некоторое согласованное множество ролей, в которых могут выступать пользователи данной системы в процессе взаимодействия с ней. В каждый момент времени с системой взаимодействует вполне определенный пользователь, при этом он играет или выступает в одной из таких ролей. Наиболее наглядный пример актера – конкретный пользователь системы со своими собственными параметрами аутентификации.

Любая сущность, которая согласуется с подобным неформальным определением актера, представляет собой экземпляр или пример актера. Для моделируемой системы актерами могут быть как субъекты-пользователи, так и другие системы. Поскольку пользователи системы всегда являются внешними по отношению к этой системе, то они всегда представляются в виде актеров.

Так как в общем случае актер всегда находится вне системы, его внутренняя структура никак не определяется. Для актера имеет значение только его внешнее представление, т. е. то, как он воспринимается со стороны системы. Актеры взаимодействуют с системой посредством передачи и приема сообщений от вариантов

использования. Сообщение представляет собой запрос актером сервиса от системы и получение этого сервиса. Это взаимодействие может быть выражено посредством ассоциаций между отдельными актерами и вариантами использования или классами. Кроме этого, с актерами могут быть связаны интерфейсы, которые определяют, каким образом другие элементы модели взаимодействуют с этими актерами.

Два и более актера могут иметь общие свойства, т. е. взаимодействовать с одним и тем же множеством вариантов использования одинаковым образом. Такая общность свойств и поведения представляется в виде рассматриваемого ниже отношения обобщения с другим, возможно, абстрактным актером, который моделирует соответствующую общность ролей. Совокупность отношений, которые могут присутствовать на диаграмме вариантов использования, будет рассмотрена ниже в данной главе.

Интерфейсы

Интерфейс (interface) служит для спецификации параметров модели, которые видимы извне без указания их внутренней структуры. В языке UML интерфейс является классификатором и характеризует только ограниченную часть поведения моделируемой сущности. Применительно к диаграммам вариантов использования, интерфейсы определяют совокупность операций, которые обеспечивают необходимый набор сервисов или функциональности для актеров. Интерфейсы не могут содержать ни атрибутов, ни состояний, ни направленных ассоциаций. Они содержат только операции без указания особенностей их реализации. Формально интерфейс эквивалентен абстрактному классу без атрибутов и методов с наличием только абстрактных операций.

На диаграмме вариантов использования интерфейс изображается в виде маленького круга, рядом с которым записывается его имя(Рисунок3,а). В качестве имени может быть существительное, которое характеризует соответствующую информацию или сервис (например, «датчик», «сирена», «видеокамера»), но чаще строка текста (например, «запрос к базе данных», «форма ввода», «устройство подачи звукового сигнала»). Если имя записывается на английском, то оно должно начинаться с заглавной буквы I, например, ISecurelnformation, ISensor (Рисунок 3, б).

Рисунок3. Графическое изображение интерфейсов
на диаграммах вариантов использования

Примечание

Имена интерфейсов подчиняются общим правилам наименования компонентов языка UML, т. е. имя может состоять из любого числа букв, цифр и некоторых знаков препинания, таких как двойное двоеточие «::». Последний символ используется для более сложных имен, включающих в себя не только имя самого интерфейса (после знака), но и имя сущности, которая включает в себя данный интерфейс (перед знаком). Примерами таких имен являются: «Сеть предприятия сервер» для указания на сервер сети предприятия или «Система аутентификации клиентов::форма ввода пароля».

Графический символ отдельного интерфейса может соединяться на диаграмме сплошной линией с тем вариантом использования, который его поддерживает. Сплошная линия в этом случае указывает на тот факт, что связанный с интерфейсом

вариант использования должен реализовывать все операции, необходимые для данного интерфейса, а возможно и больше (Рисунок 4, а). Кроме этого, интерфейсы могут соединяться с вариантами использования пунктирной линией со стрелкой (Рисунок 4, б), означающей, что вариант использования предназначен для спецификации только того сервиса, который необходим для реализации данного интерфейса.

Рисунок 4. Графическое изображение взаимосвязей интерфейсов
с вариантами использования

С системно-аналитической точки зрения интерфейс не только отделяет

спецификацию операций системы от их реализации, но и определяет общие границы проектируемой системы. В последующем интерфейс может быть уточнен явным указанием тех операций, которые специфицируют отдельный аспект поведения системы. В этом случае он изображается в форме прямоугольника класса с ключевым словом «interface» в секции имени, с пустой секцией атрибутов и с непустой секцией операций. Однако подобное графическое представление используется на диаграммах классов или диаграммах, характеризующих поведение моделируемой системы.

Важность интерфейсов заключается в том, что они определяют стыковочные узлы в проектируемой системе, что совершенно необходимо для организации коллективной работы над проектом. Более того, спецификация интерфейсов способствует «безболезненной» модификации уже существующей системы при переходе на новые технологические решения. В этом случае изменению подвергается только реализация операций, но никак не функциональность самой системы. А это обеспечивает совместимость последующих версий программ с первоначальными при спиральной технологии разработки программных систем.

Примечания

Примечания (notes) в языке UML предназначены для включения в модель произвольной текстовой информации, имеющей непосредственное отношение к контексту разрабатываемого проекта. В качестве такой информации могут быть комментарии разработчика (например, дата и версия разработки диаграммы или ее отдельных компонентов), ограничения (например, на значения отдельных связей или экземпляры сущностей) и помеченные значения. Применительно к диаграммам вариантов использования примечание может носить самую общую информацию, относящуюся к общему контексту системы.

Графически примечания обозначаются прямоугольником с «загнутым» верхним правым уголком (Рисунок 5). Внутри прямоугольника содержится текст примечания. Примечание может относиться к любому элементу диаграммы, в этом случае их соединяет пунктирная линия. Если примечание относится к нескольким элементам, то от него проводятся, соответственно, несколько линий. Разумеется, примечания могут присутствовать не только на диаграмме вариантов использования, но и на других канонических диаграммах.

Рисунок5. Примеры примечаний в языке UML

Если в примечании указывается ключевое слово «constraint», то данное примечание является ограничением, налагаемым на соответствующий элемент модели, но не на саму диаграмму. При этом запись ограничения заключается в фигурные скобки и должна соответствовать правилам правильного построения выражений языка OCL. Однако для диаграмм вариантов использования ограничения включать в модели не рекомендуется, поскольку они достаточно жестко регламентируют отдельные аспекты системы. Подобная регламентация противоречит неформальному характеру общей модели системы, в качестве которой выступает диаграмма вариантов использования.

Отношения на диаграмме вариантов использования

Между компонентами диаграммы вариантов использования могут существовать различные отношения, которые описывают взаимодействие экземпляров одних актеров и вариантов использования с экземплярами других актеров и вариантов. Один актер может взаимодействовать с несколькими вариантами использования. В этом случае этот актер обращается к нескольким сервисам данной системы. В свою очередь один вариант использования может взаимодействовать с несколькими актерами, предоставляя для всех них свой сервис. Следует заметить, что два варианта использования, определенные для одной и той же сущности, не могут взаимодействовать друг с другом, поскольку каждый из них самостоятельно описывает законченный вариант использования этой сущности. Более того, варианты использования всегда предусматривают некоторые сигналы или сообщения, когда взаимодействуют с актерами за пределами системы. В то же время могут быть определены другие способы для взаимодействия с элементами внутри системы.

В языке UML имеется несколько стандартных видов отношений между актерами

и вариантами использования:

- Отношение ассоциации (association relationship)
- Отношение расширения (extend relationship)
- Отношение обобщения (generalization relationship)
- Отношение включения (include relationship)

При этом общие свойства вариантов использования могут быть представлены тремя различными способами, а именно с помощью отношений расширения, обобщения и включения.

Отношение ассоциации

Отношение ассоциации является одним из фундаментальных понятий в языке UML и в той или иной степени используется при построении всех графических моделей систем в форме канонических диаграмм.

Применительно к диаграммам вариантов использования оно служит для обозначения специфической роли актера в отдельном варианте использования. Другими словами, ассоциация специфицирует семантические особенности взаимодействия актеров и вариантов использования в графической модели системы. Таким образом, это отношение устанавливает, какую конкретную роль играет актер при взаимодействии с экземпляром варианта использования. На диаграмме вариантов использования, так же как и на других диаграммах, отношение ассоциации обозначается сплошной линией между актером и вариантом использования. Эта линия может иметь дополнительные условные обозначения, такие, например, как имя и кратность (Рисунок 6).

Рисунок 6. Пример графического представления отношения ассоциации между актерами

Кратность (multiplicity) ассоциации указывается рядом с обозначением компонента диаграммы, который является участником данной ассоциации. Кратность характеризует общее количество конкретных экземпляров данного компонента,

которые могут выступать в качестве элементов данной ассоциации. Применительно к диаграммам вариантов использования кратность имеет специальное обозначение в форме одной или нескольких цифр и, возможно, специального символа «*» (звездочка).

Примечание

Для диаграмм вариантов использования наиболее распространенными являются четыре основные формы записи кратности отношения ассоциации:

- Целое неотрицательное число (включая цифру 0). Предназначено для указания кратности, которая является строго фиксированной для элемента соответствующей ассоциации. В этом случае количество экземпляров актеров или вариантов использования, которые могут выступать в качестве элементов отношения ассоциации, в точности равно указанному числу.

Примером этой формы записи кратности ассоциации является указание кратности «1» для актера «Клиент банка» (Рисунок 7). Эта запись означает, что каждый экземпляр варианта использования «Оформить кредит для клиента банка» может иметь в качестве своего элемента единственный экземпляр актера «Клиент банка». Другими словами, при оформлении кредита в банке необходимо иметь в виду, что каждый конкретный кредит оформляется на единственного клиента этого банка.

- Два целых неотрицательных числа, разделенные двумя точками и записанные в виде: «первое число .. второе число». Данная запись в языке UML соответствует нотации для множества или интервала целых чисел, которая применяется в некоторых языках программирования для обозначения границ массива элементов. Эту запись следует понимать как множество целых неотрицательных чисел, следующих в последовательно возрастающем порядке:

{первое_число, первое_число+1, первое_число+2, ..., второе_число].

Очевидно, что первое число должно быть строго меньше второго числа в арифметическом смысле, при этом первое число может быть равно 0.

Пример такой формы записи кратности ассоциации – «1..5». Эта запись означает, что количество отдельных экземпляров данного компонента, которые могут выступать в качестве элементов данной ассоциации, равно некоторому заранее неизвестному числу из множества целых чисел {1, 2, 3, 4, 5}. Эта ситуация может иметь место, например, в случае рассмотрения в качестве актера – клиента банка, а в качестве

варианта использования – процедуру открытия счета в банке. При этом количество отдельных счетов каждого клиента в данном банке, исходя из некоторых дополнительных соображений, может быть не больше 5. Эти дополнительные соображения как раз и являются внешними требованиями по отношению к проектируемой системе и определяются ее заказчиком на начальных этапах.

• Два символа, разделенные двумя точками. При этом первый из них является целым неотрицательным числом или 0, а второй – специальным символом «*». Здесь символ «*» обозначает произвольное конечное целое неотрицательное число, значение которого неизвестно на момент задания соответствующего отношения ассоциации.

Пример такой формы записи кратности ассоциации – «2..*». Запись означает, что количество отдельных экземпляров данного компонента, которые могут выступать в качестве элементов данной ассоциации, равно некоторому заранее неизвестному числу из подмножества натуральных чисел: {2, 3, 4}.

• Единственный символ «*», который является сокращением записи интервала «0..*». В этом случае количество отдельных экземпляров данного компонента отношения ассоциации может быть любым целым неотрицательным числом. При этом 0 означает, что для некоторых экземпляров соответствующего компонента данное отношение ассоциации может вовсе не иметь места.

В качестве примера этой записи можно привести кратность отношения ассоциации для варианта использования «Оформить кредит для клиента банка» (Рисунок 6). Здесь кратность «*» означает, что каждый отдельный клиент банка может оформить для себя несколько кредитов, при этом их общее число заранее неизвестно и ничем не ограничивается. При этом некоторые клиенты могут совсем не иметь оформленных на свое имя кредитов (вариант значения 0).

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

Отношение расширения

Отношение расширения определяет взаимосвязь экземпляров отдельного варианта использования с более общим вариантом, свойства которого определяются на основе способа совместного

объединения данных экземпляров. В метамодели отношение расширения является направленным и указывает, что применительно к отдельным примерам

некоторого варианта использования должны быть выполнены конкретные условия, определенные для расширения данного варианта использования. Так, если имеет место отношение расширения от варианта использования A к варианту использования B, то это означает, что свойства экземпляра варианта использования B могут быть дополнены благодаря наличию свойств у расширенного варианта использования A.

Отношение расширения между вариантами использования обозначается пунктирной линией со стрелкой (вариант отношения зависимости), направленной от того варианта использования, который является расширением для исходного варианта использования. Данная линия со стрелкой помечается ключевым словом «extend» («расширяет»), как показано на Рис. 7.

Рисунок 7. Пример графического изображения отношения расширения между вариантами использования

Отношение расширения отмечает тот факт, что один из вариантов использования может присоединять к своему поведению некоторое дополнительное поведение, определенное для другого варианта использования. Данное отношение включает в себя некоторое условие и ссылки на точки расширения в базовом варианте использования. Чтобы расширение имело место, должно быть выполнено определенное условие данного отношения. Ссылки на точки расширения определяют те места в базовом варианте использования, в которые должно быть помещено соответствующее расширение при выполнении условия.

Один из вариантов использования может быть расширением для нескольких базовых вариантов, а также иметь в качестве собственных расширений несколько других вариантов. Базовый вариант использования может дополнительно никак не зависеть от своих расширений.

Семантика отношения расширения определяется следующим образом. Если экземпляр варианта использования выполняет некоторую последовательность действий, которая определяет его поведение, и при этом имеется точка расширения на экземпляр другого варианта использования, которая является первой из всех точек расширения у исходного варианта, то проверяется условие данного отношения. Если

условие выполняется, исходная последовательность действий расширяется посредством включения действий экземпляра другого варианта использования. Следует заметить, что условие отношения расширения проверяется лишь один раз – при первой ссылке на точку расширения, и если оно выполняется, то все расширяющие варианты использования вставляются в базовый вариант.

В представленном выше примере (Рисунок 7) при оформлении заказа на приобретение товара только в некоторых случаях может потребоваться предоставление клиенту каталога всех товаров. При этом условием расширения является запрос от клиента на получение каталога товаров. Очевидно, что после получения каталога клиенту необходимо некоторое время на его изучение, в течение которого оформление заказа приостанавливается. После ознакомления с каталогом клиент решает либо в пользу выбора отдельного товара, либо отказа от покупки вообще. Сервис или вариант использования «Оформить заказ на приобретение товара» может отреагировать на выбор клиента уже после того, как клиент получит для ознакомления каталог товаров.

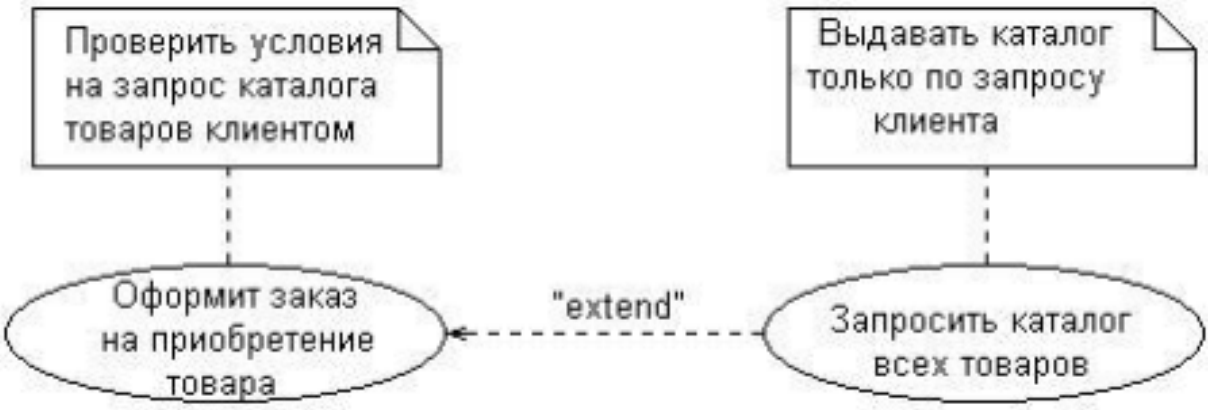


Рисунок9. Графическое изображение отношения расширения с примечаниями условий выполнения вариантов использования

Точка расширения может быть как отдельной точкой в последовательности действий, так и множеством отдельных точек. Важно представлять себе, что если отношение расширения имеет некоторую последовательность точек расширения, только первая из них может определять множество отдельных точек. Все остальные должны определять в точности одну такую точку. Какая из точек должна быть первой точкой расширения, т.е. определяться единственным расширением. Такие ссылки на расположение точек расширения могут быть представлены различными способами, например, с помощью текста примечания на естественном языке (Рисунок 9), пред- и

постусловий, а также с использованием имен состояний в автомате.

Отношение обобщения

Отношение обобщения служит для указания того факта, что некоторый вариант использования А может быть обобщен до варианта использования В. В этом случае вариант А будет являться специализацией варианта В. При этом В называется предком или родителем по отношению А, а вариант А – потомком по отношению к варианту использования В. Следует подчеркнуть, что потомок наследует все свойства и поведение своего родителя, а также может быть дополнен новыми свойствами и особенностями поведения. Графически данное отношение обозначается сплошной линией со стрелкой в форме не закрашенного треугольника, которая указывает на родительский вариант использования (Рисунок 9). Эта линия со стрелкой имеет специальное название – стрелка «обобщение».

Рисунок 9. Пример графического изображения отношения обобщения между вариантами использования

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

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

Между отдельными актерами также может существовать отношение обобщения. Данное отношение является направленным и указывает на факт специализации одних актеров относительно других. Например, отношение обобщения от актера А к актеру

В отмечает тот факт, что каждый экземпляр актера А является одновременно экземпляром актера В и обладает всеми его свойствами. В этом случае актер В является родителем по отношению к актеру А, а актер А, соответственно, потомком актера В. При этом актер А обладает способностью играть такое же множество ролей, что и актер В. Графически данное отношение также обозначается стрелкой обобщения, т.е. сплошной линией со стрелкой в форме не закрашенного треугольника, которая указывает на родительского актера (Рисунок 10).

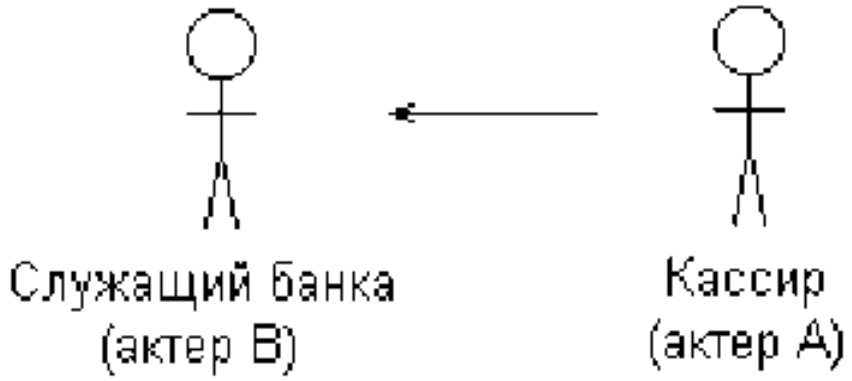


Рисунок10. Пример графического изображения отношения обобщения между актерами

Отношение включения

Отношение включения между двумя вариантами использования указывает, что некоторое заданное поведение для одного варианта использования включается в качестве составного компонента в последовательность поведения другого варианта использования. Данное отношение является направленным бинарным отношением в том смысле, что пара экземпляров вариантов использования всегда упорядочена в отношении включения.

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

Один вариант использования может быть включен в несколько других вариантов, а также включать в себя другие варианты. Включаемый вариант использования может быть независимым от базового варианта в том смысле, что он предоставляет последнему некоторое инкапсулированное поведение, детали реализации которого скрыты от последнего и могут быть легко перераспределены между несколькими включаемыми вариантами использования. Более того, базовый вариант может зависеть только от результатов выполнения включаемого в него поведения, но не от структуры включаемых в него вариантов.

Отношение включения, направленное от варианта использования А к варианту использования В, указывает, что каждый экземпляр варианта А включает в себя функциональные свойства, заданные для варианта В. Эти свойства специализируют поведение соответствующего варианта А на данной диаграмме. Графически данное отношение обозначается пунктирной линией со стрелкой (вариант отношения зависимости), направленной от базового варианта использования к включаемому. При этом данная линия со стрелкой помечается ключевым словом «include» («включает»), как показано на Рисунок 11.

Рисунок11. Пример графического изображения отношения включения между вариантами использования

Примечание

Следует заметить, что рассмотренные три последних отношения могут существовать только между вариантами использования, которые определены для одной и той же сущности. Причина этого заключается в том, что поведение некоторой сущности обусловлено вариантами использования только этой сущности. Другими словами, все экземпляры варианта использования выполняются лишь внутри данной сущности. Если некоторый вариант использования должен иметь отношение обобщения, включения или расширения с вариантом использования другой сущности, получаемые в результате экземпляры вариантов должны быть включены в обе сущности, что противоречит семантическим правилам представления элементов языка

UML. Однако эти отношения, определенные в пределах одной сущности, могут быть использованы в пределах другой сущности, если обе сущности связаны между собой отношением обобщения. В этом случае поведение соответствующих вариантов использования подчиняется общим правилам наследования свойств и поведения сущности-предка всеми дочерними сущностями.

Пример построения диаграммы вариантов использования

В качестве примера рассмотрим процесс моделирования системы продажи товаров по каталогу, которая может быть использована при создании соответствующих информационных систем.

В качестве актеров данной системы могут выступать два субъекта, один из которых является продавцом, а другой – покупателем. Каждый из этих актеров взаимодействует с рассматриваемой системой продажи товаров по каталогу и является ее пользователем, т. е. они оба обращаются к соответствующему сервису «Оформить заказ на покупку товара». Как следует из существа выдвигаемых к системе требований, этот сервис выступает в качестве варианта использования разрабатываемой диаграммы, первоначальная структура которой может включать в себя только двух указанных актеров и единственный вариант использования (Рисунок 12).

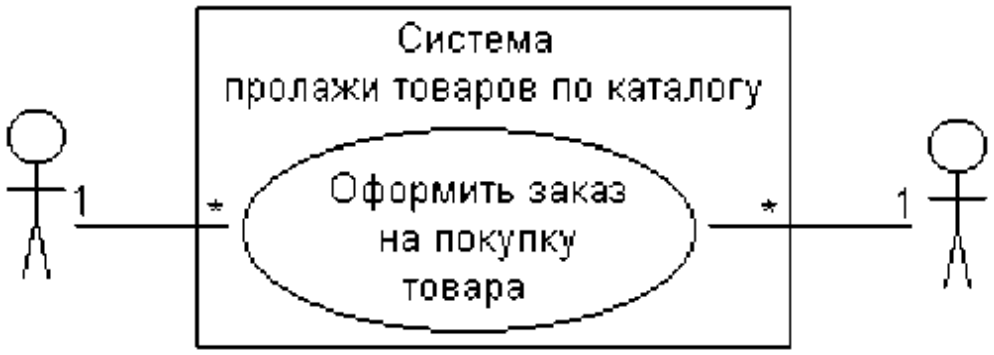


Рисунок12. Исходная диаграмма вариантов использования для примера разработки системы продажи товаров по каталогу

Значения указанных на данной диаграмме кратностей отражают общие правила или логику оформления заказов на покупку товаров. Согласно этим правилам, один продавец может участвовать в оформлении нескольких заказов, в то же время каждый заказ может быть оформлен только одним продавцом, который несет ответственность за корректность его оформления и, в связи с этим, будет иметь агентское вознаграждение за его оформление. С другой стороны, каждый покупатель может

оформлять на себя несколько заказов, но, в то же время, каждый заказ должен быть оформлен на единственного покупателя, к которому переходят права собственности на товар после его оплаты.

Примечание

Рассмотренные выше примеры значений для кратности отношения ассоциации могут вызвать невольное восхищение глубиной своей семантики, которая в единственном специальном символе отражает вполне определенные логические условия реализации соответствующих компонентов диаграммы вариантов использования.

На следующем этапе разработки данной диаграммы вариант использования «Оформить заказ на покупку товара» может быть уточнен на основе введения в рассмотрение четырех дополнительных вариантов использования. Это следует из более детального анализа процесса продажи товаров, что позволяет выделить в качестве отдельных сервисов такие действия, как обеспечить покупателя информацией о товаре, согласовать условия оплаты товара и заказать товар со склада. Вполне очевидно, что указанные действия раскрывают поведение исходного варианта использования в смысле его конкретизации, и поэтому между ними будет иметь место отношение включения.

С другой стороны, продажа товаров по каталогу предполагает наличие самостоятельного информационного объекта – каталога товаров, который в некотором смысле не зависит от реализации сервиса по обслуживанию покупателей. В нашем случае, каталог товаров может запрашиваться покупателем или продавцом при необходимости выбора товара и уточнения деталей его продажи. Вполне резонно представить сервис «Запросить каталог товаров» в качестве самостоятельного варианта использования.

Полученная в результате последующей детализации уточненная диаграмма вариантов использования будет содержать 5 вариантов использования и 2 актеров (Рисунок 13), между которыми установлены отношения включения и расширения.

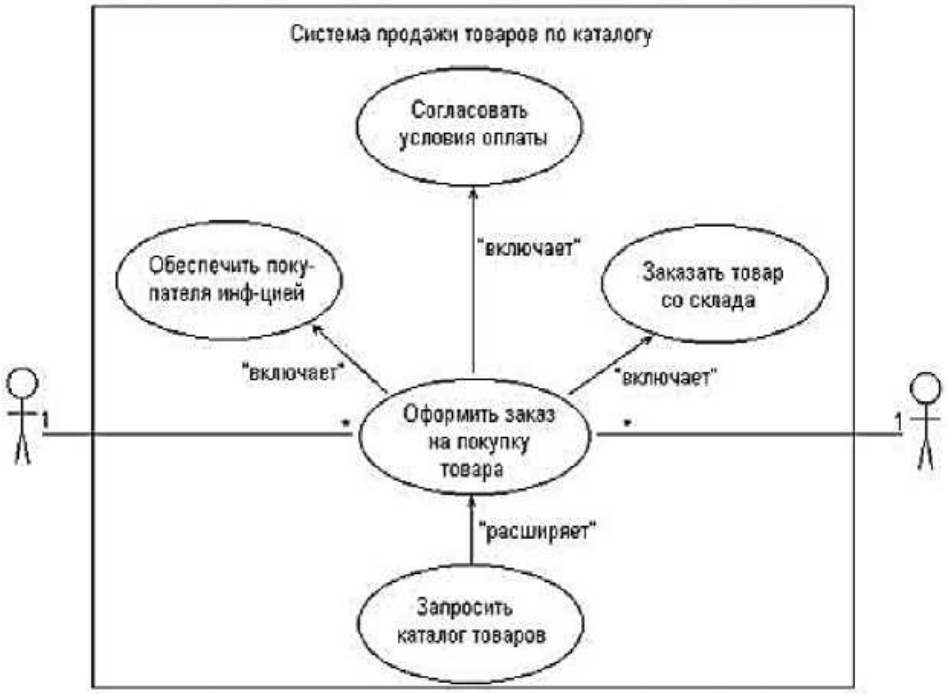


Рисунок13. Уточненный вариант диаграммы вариантов использования для примера системы продажи товаров по каталогу

Приведенная выше диаграмма вариантов использования, в свою очередь, может быть детализирована далее с целью более глубокого уточнения предъявляемых к системе требований и конкретизации деталей ее последующей реализации. В рамках общей парадигмы ООАП подобная детализация может выполняться в двух основных направлениях.

С одной стороны, детализация может быть выполнена на основе установления дополнительных отношений типа отношения «обобщение-специализация» для уже имеющихся компонентов диаграммы вариантов использования. Так, в рамках рассматриваемой системы продажи товаров может иметь самостоятельное значение и специфические особенности отдельная категория товаров – компьютеры. В этом случае диаграмма может быть дополнена вариантом использования «Оформить заказ на покупку компьютера» и актерами «Покупатель компьютера» и «Продавец компьютеров», которые связаны с соответствующими компонентами диаграммы отношением обобщения (Рисунок 14).

Уточненный таким способом вариант диаграммы вариантов использования содержит одну важную особенность, которую необходимо отметить. А именно, хотя на данной диаграмме (Рисунок 14) отсутствуют изображения линий отношения ассоциации между актером «Продавец компьютеров» и вариантом использования «Оформить заказ на покупку компьютера», а также между актером «Покупатель компьютера» и

вариантом использования «Оформить заказ на покупку компьютера», наличие отношения обобщения между соответствующими компонентами позволяет им наследовать отношение ассоциации от своих предков. Поскольку принцип наследования является одним из фундаментальных принципов объектно-ориентированного программирования, в нашем примере можно с уверенностью утверждать, что эти линии отношения ассоциации с соответствующими кратностями присутствуют на данной диаграмме в скрытом виде.

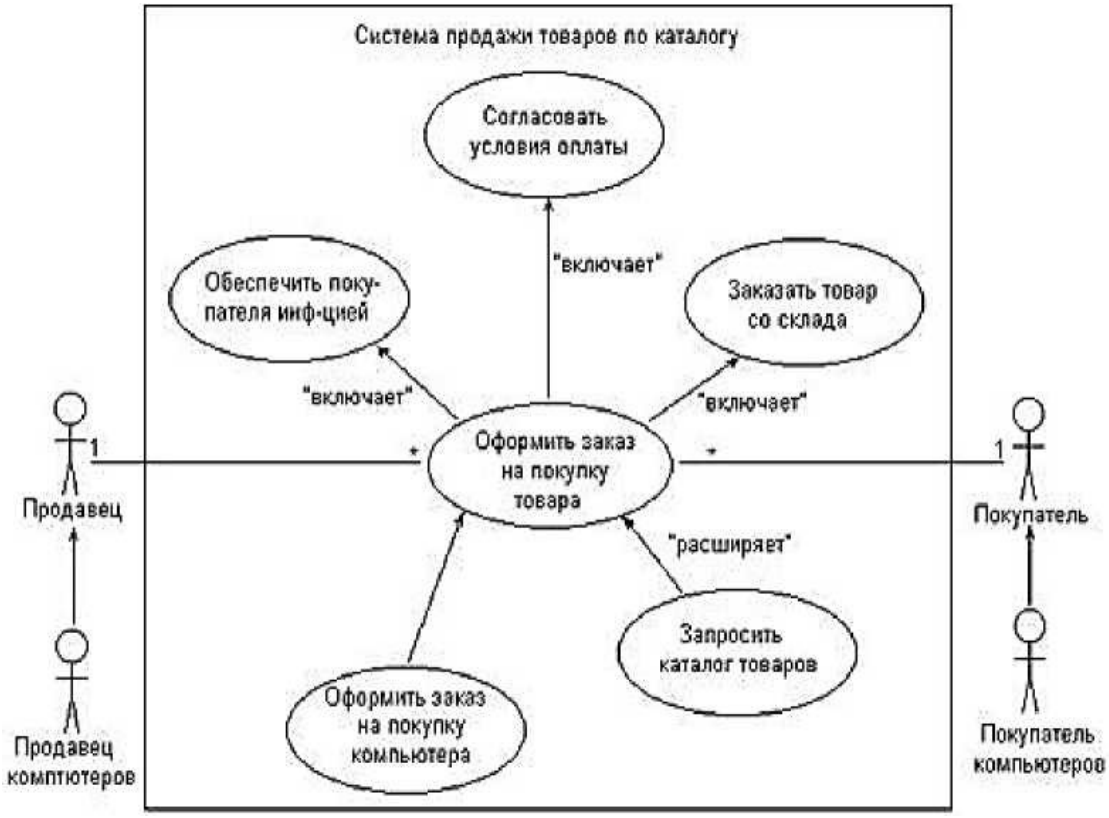


Рисунок 14.Один из вариантов последующего уточнения диаграммы вариантов использования для примера рассматриваемой системы продажи

Для пояснения изложенного можно привести фрагмент диаграммы вариантов использования для рассмотренного примера, на котором явно указаны отношения ассоциации между дочерними компонентами (Рисунок 15). Данное изображение фрагмента диаграммы приводится с методической целью, при этом остальные компоненты диаграммы, которые остались без изменений, условно отмечены многоточием.

Рисунок 15. Фрагмент диаграммы вариантов использования с отношениями ассоциации между дочерними компонентами

Примечание

Строго говоря, приведенное выше изображение фрагмента диаграммы не является допустимым с точки зрения нотации языка UML Причиной этого следует считать многоточие, которое не может быть использовано в подобной интерпретации. Тем не менее, данное изображение иллюстрирует основные идеи наследования свойств и поведения, которые неявно могут присутствовать в графических моделях сложных систем. С другой стороны, следует всегда помнить, что эта информация является избыточной с точки зрения семантики языка UML, а значит может быть опущена, что и было сделано на предыдущей диаграмме (см. Рисунок 14).

Второе из основных направлений детализации диаграмм вариантов использования связано с последующей структуризацией ее отдельных компонентов в форме элементов других диаграмм. Например, конкретные особенности реализации вариантов использования в терминах взаимодействующих объектов, определенных в виде классов данной сущности, могут быть заданы на диаграмме кооперации.

Построение диаграммы вариантов использования является самым первым этапом процесса объектно-ориентированного анализа и проектирования (ООАП), цель которого – представить совокупность требований к поведению проектируемой системы. Спецификация требований к проектируемой системе в форме диаграммы вариантов использования представляет собой самостоятельную модель, которая в языке UML получила название модели вариантов использования и имеет свое специальное стандартное имя или стереотип «use Case Model».

В последующем все заданные в этой модели требования представляются в виде

общей модели системы, которая состоит из пакета Системы. Последний в свою очередь может представлять собой иерархию пакетов, на самом верхнем уровне которых содержится множество классов модели проектируемой системы. Если же пакет системы со стандартным именем «top Level Package» является подсистемой, то ее абстрактное поведение в точности такое же, как и у исходной системы.

Рекомендации по разработке диаграмм вариантов использования

Главное назначение диаграммы вариантов использования заключается в формализации функциональных требований к системе с помощью понятий соответствующего пакета и возможности согласования полученной модели с заказчиком на ранней стадии проектирования. Любой из вариантов использования может быть подвергнут дальнейшей декомпозиции на множество подвариантов использования отдельных элементов, которые образуют исходную сущность. Рекомендуемое общее количество актеров в модели – не более 20, а вариантов использования – не более 50. В противном случае модель теряет свою наглядность и, возможно, заменяет собой одну из некоторых других диаграмм.

Семантика построения диаграммы вариантов использования должна определяться следующими особенностями рассмотренных выше элементов модели. Отдельный экземпляр варианта использования по своему содержанию является выполнением последовательности действий, которая инициализируется посредством экземпляра сообщения от экземпляра актера. В качестве отклика или ответной реакции на сообщение актера экземпляр варианта использования выполняет последовательность действий, установленную для данного варианта использования. Экземпляры актеров могут генерировать новые экземпляры сообщений для экземпляров вариантов использования.

Подобное взаимодействие будет продолжаться до тех пор, пока не закончится выполнение требуемой последовательности действий экземпляром варианта использования, и соответствующий экземпляр актера (и никакой другой) не получит требуемый экземпляр сервиса. Окончание взаимодействия означает отсутствие инициализации экземпляров сообщений от экземпляров актеров для соответствующих экземпляров вариантов использования.

Варианты использования могут быть специфицированы в виде текста, а в последующем – с помощью операций и методов вместе с атрибутами, в виде графа

деятельности, посредством автомата или любого другого механизма описания поведения, включающего предусловия и постусловия. Взаимодействие между вариантами использования и актерами может уточняться на диаграмме кооперации, когда описываются взаимосвязи между сущностью, содержащей эти варианты использования, и окружением или внешней средой этой сущности.

В случае, когда для представления иерархической структуры проектируемой системы используются подсистемы, система может быть определена в виде вариантов использования на всех уровнях. Отдельные подсистемы или классы могут выступать в роли таких вариантов использования. При этом вариант, соответствующий некоторому из этих элементов, в последующем может уточняться множеством более мелких вариантов использования, каждый из которых будет определять сервис элемента модели, содержащийся в сервисе исходной системы. Вариант использования в целом может рассматриваться как суперсервис для уточняющих его подвариантов, которые, в свою очередь, могут рассматриваться как подсервисы исходного варианта использования.

Функциональность, определенная для более общего варианта использования, полностью наследуется всеми вариантами нижних уровней. Однако следует заметить, что структура элемента-контейнера не может быть представлена вариантами использования, поскольку они могут представлять только функциональность отдельных элементов модели. Подчиненные варианты использования кооперируются для совместного выполнения супер сервиса варианта использования верхнего уровня. Эта кооперация также может быть представлена на диаграмме кооперации в виде совместных действий отдельных элементов модели.

Отдельные варианты использования нижнего уровня могут участвовать в нескольких кооперациях, т. е. играть определенную роль при выполнении сервисов нескольких вариантов верхнего уровня. Для отдельных таких кооперацій могут быть определены соответствующие роли актеров, взаимодействующих с конкретными вариантами использования нижнего уровня. Эти роли будут играть актеры нижнего уровня модели системы. Хотя некоторые из таких актеров могут быть актерами верхнего уровня, это не противоречит принятым в языке UML семантическим правилам построения диаграмм вариантов использования. Более того, интерфейсы вариантов использования верхнего уровня могут полностью совпадать по своей

структуре с соответствующими интерфейсами вариантов нижнего уровня.

Окружение вариантов использования нижнего уровня является самостоятельным элементом модели, который в свою очередь содержит другие элементы модели, определенные для этих вариантов использования. Таким образом, с точки зрения общего представления верхнего уровня взаимодействие между вариантами использования нижнего уровня определяет результат выполнения сервиса варианта верхнего уровня. Отсюда следует, что в языке UML вариант использования является элементом- контейнером.

Варианты использования классов соответствуют операциям этого класса, поскольку сервис класса является по существу выполнением операций данного класса. Некоторые варианты использования могут соответствовать применению только одной операции, в то время как другие – конечного множества операций, определенных в виде последовательности операций. В то же время одна операция может быть необходима для выполнения нескольких сервисов класса и поэтому будет появляться в нескольких вариантах использования этого класса.

Реализация варианта использования зависит от типа элемента модели, в котором он определен. Например, поскольку варианты использования класса определяются посредством операций этого класса, они реализуются соответствующими методами. С другой стороны, варианты использования подсистемы реализуются элементами, из которых состоит данная подсистема. Поскольку подсистема не имеет своего собственного поведения, все предлагаемые подсистемой сервисы должны представлять собой композицию сервисов, предлагаемых отдельными элементами этой подсистемы, т. е., в конечном итоге, классами. Эти элементы могут взаимодействовать друг с другом для совместного обеспечения требуемого поведения отдельного варианта использования, посвященной построению диаграмм кооперации. Здесь лишь отметим, что кооперации используются как для уточнения спецификаций в виде вариантов использования нижних уровней диаграммы, так и для описания особенностей их последующей реализации.

Если в качестве моделируемой сущности выступает система или подсистема самого верхнего уровня, то отдельные пользователи вариантов использования этой системы моделируются актерами. Такие актеры, являясь внутренними по отношению к моделируемым подсистемам нижних уровней, часто в явном виде не указываются,

хотя и присутствуют неявно в модели подсистемы. Вместо этого варианты использования непосредственно обращаются к тем модельным элементам, которые содержат в себе подобные неявные актеры, т. е. экземпляры которых играют роли таких актеров при взаимодействии с вариантами использования. Эти модельные элементы могут содержаться в других пакетах или подсистемах. В последнем случае роли определяются в том пакете, к которому относится соответствующая подсистема.

С системно-аналитической точки зрения построение диаграммы вариантов использования специфицирует не только функциональные требования к проектируемой системе, но и выполняет исходную структуризацию предметной области. Последняя задача сочетает в себе не только следование техническим рекомендациям, но и является в некотором роде искусством, умением выделять главное в модели системы. Хотя рациональный унифицированный процесс не исключает итеративный возврат в последующем к диаграмме вариантов использования для ее модификации, не вызывает сомнений тот факт, что любая подобная модификация потребует, как по цепочке, изменений во всех других представлениях системы. Поэтому всегда необходимо стремиться к возможно более точному представлению модели именно в форме диаграммы вариантов использования.

Если же варианты использования применяются для спецификации части системы, то они будут эквивалентны соответствующим вариантам использования в модели подсистемы для части соответствующего пакета. Важно понимать, что все сервисы системы должны быть явно определены на диаграмме вариантов использования, и никаких других сервисов, которые отсутствуют на данной диаграмме, проектируемая система не может выполнять по определению. Более того, если для моделирования реализации системы используются сразу несколько моделей (например, модель анализа и модель проектирования), то множество вариантов использования всех пакетов системы должно быть эквивалентно множеству вариантов использования модели в целом.

Диаграммы последовательности

В ходе проектирования ИС аналитик поэтапно спускается от общей концепции, через понимание ее логической структуры к наиболее детальным моделям, описывающим физическую реализацию.

С помощь диаграммы прецедентов (вариантов использования) выявляются

основные пользователи системы и задачи, которые данная система должна решать. С помощью диаграммы деятельности мы описываем последовательность действий для каждого прецедента, необходимая для достижения поставленной цели.

Далее проектируется логическая структура системы с помощью диаграммы классов. На данном этапе выделяются классы, формирующие структуру базы данных системы, а также классы реализующие некий набор операций, способствующий достижению целей в рамках выбранного прецедента. Для описания сложного поведения некоторых объектов (экземпляров класса) составляется диаграмма состояний.

Таким образом, аналитиками фиксируются такие поведенческие аспекты как алгоритм действий в рамках одного или нескольких прецедентов, необходимый для достижения определенного результата, а также изменение состояния объектов в ходе выполнения приведенных действий.

Зачастую на этапе спецификации требований необходимо показать не только алгоритм действий или изменение состояния объекта, но и обмен сообщениями между отдельными объектами системы. Данную задачу решает диаграмма взаимодействия.

Диаграмма взаимодействия предназначена для моделирования отношений между объектами (ролями, классами, компонентами) системы в рамках одного прецедента.

Данный вид диаграмм отражает следующие аспекты проектируемой системы:

• обмен сообщениями между объектами (в том числе в рамках обмена сообщениями со сторонними системами) ограничения, накладываемые на взаимодействие объектов события, инициирующие взаимодействия объектов.

В отличие от диаграммы деятельности, которая показывает только последовательность (алгоритм) работы системы, диаграммы взаимодействия акцентируют внимание разработчиков на сообщениях, инициирующих вызов определенных операций объекта (класса) или являющихся результатом выполнения операции.

Диаграмма последовательности является одной из разновидности диаграмм взаимодействия и предназначена для моделирования взаимодействия объектов системы во времени, а также обмена сообщениями между ними.

Одним из основных принципов ООП является способ информационного обмена между элементами системы, выражающийся в отправке и получении сообщений друг

от друга. Таким образом, основные понятия диаграммы последовательности связаны с понятием объект и сообщение.

На диаграмме последовательности объекты в основном представляю экземпляры поведением (Рисунок 16).

В качестве объектов могут выступать пользователи, инициирующие взаимодействие, классы, обладающие поведением в системе или программные компоненты, а иногда и системы в целом.

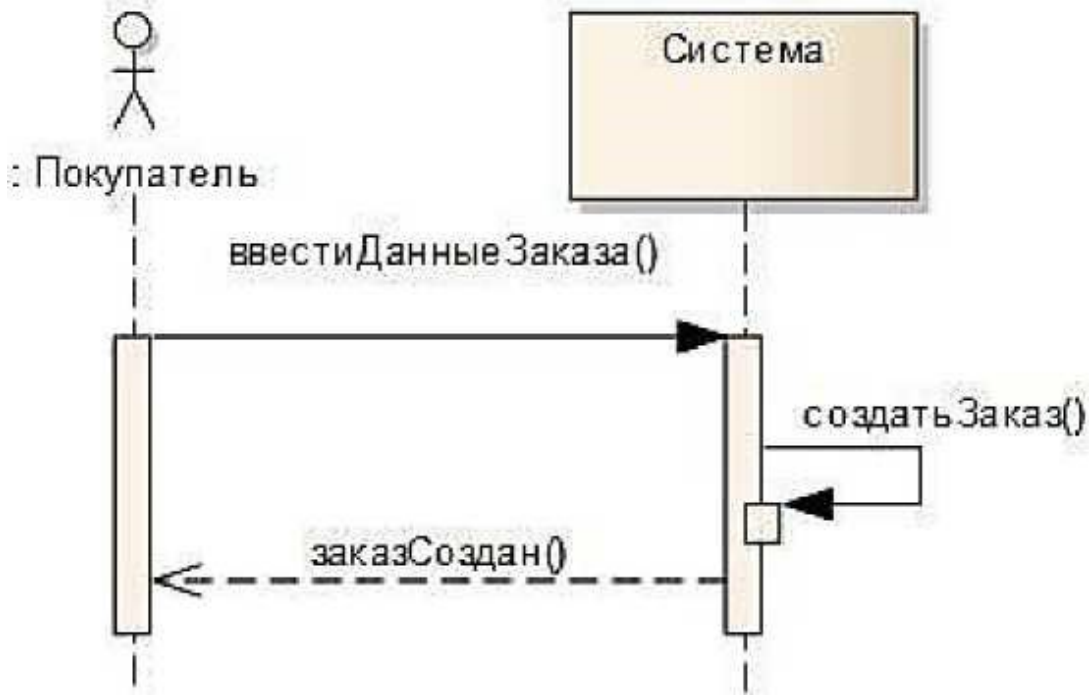


Рисунок 16. Фрагмент диаграммы последовательности класса или сущности, обладающие

Объекты располагаются с лева на права таким образом, чтобы крайним с лева был тот объект, который инициирует взаимодействие. Неотъемлемой частью объекта на диаграмме последовательности является линия жизни объекта. Линия жизни показывает время, в течение которого объект существует в системе. Периоды активности объекта в момент взаимодействия показываются с помощью фокуса управления. Временная шкала на диаграмме направлена сверху вниз.

На диаграммах последовательности допустимо использование стандартных стереотипов класса, представленных в Таблице 1 и Рисунок17.

Таблица 1 .Стандартные стереотипы класса для диаграмм последовательности

	**Actor**–экземпляр участника процесса (роль на диаграмме прецедентов)
	**Boundary**– Класс-Разграничитель – используется для классов, отделяющих внутреннюю структуру системы от внешней среды (экранная форма, пользовательский интерфейс, устройство ввода-вывода). Объект со стереотипом <<bondary>> отличается от, привычного нам, класса <<Интерфейс>>, который по большей части предназначен для вызова методов класса, с которым он связан. Объект boundary показывает именно экранную форму, которая принимает и передает данные обработчику.</bondary>>
	**Control** – Класс-контроллер – активный элемент, который используются для выполнения некоторых операций над объектами (программный компонент, модуль, обработчик)
	**Entity**– Класс-сущность – обычно применяется для обозначения классов, которые хранят некую информацию о бизнес-объектах (соответствует таблице или элементу БД)

Также одним из основных понятий, связанных с диаграммой последовательности, является Сообщение.

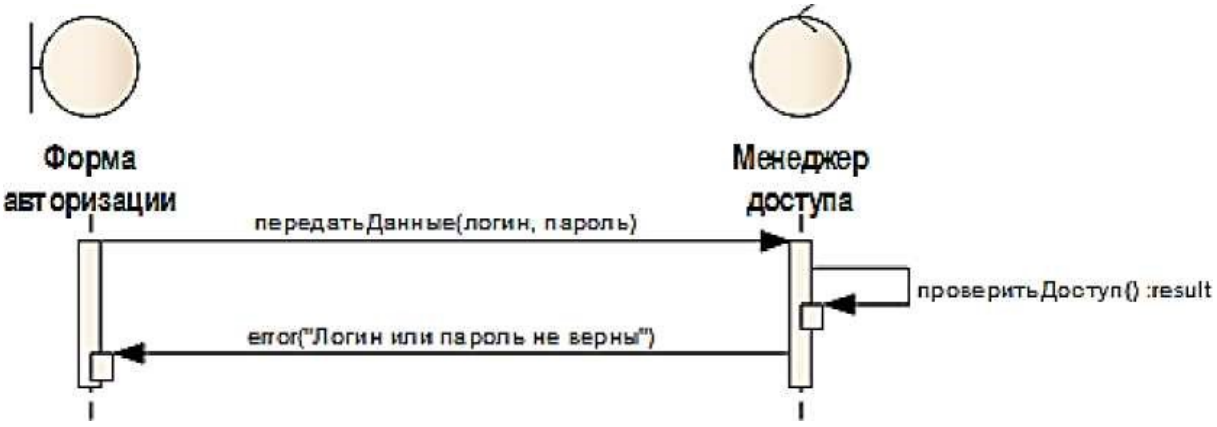


Рисунок17. Фрагмент диаграммы последовательности с использованием стандартных стереотипов класса

На диаграмме деятельности выделяются сообщения, инициирующие ту или иную деятельность или являющиеся ее следствием. На диаграмме состояний частично показан обмен сообщениями в рамках сообщений инициирующих изменение

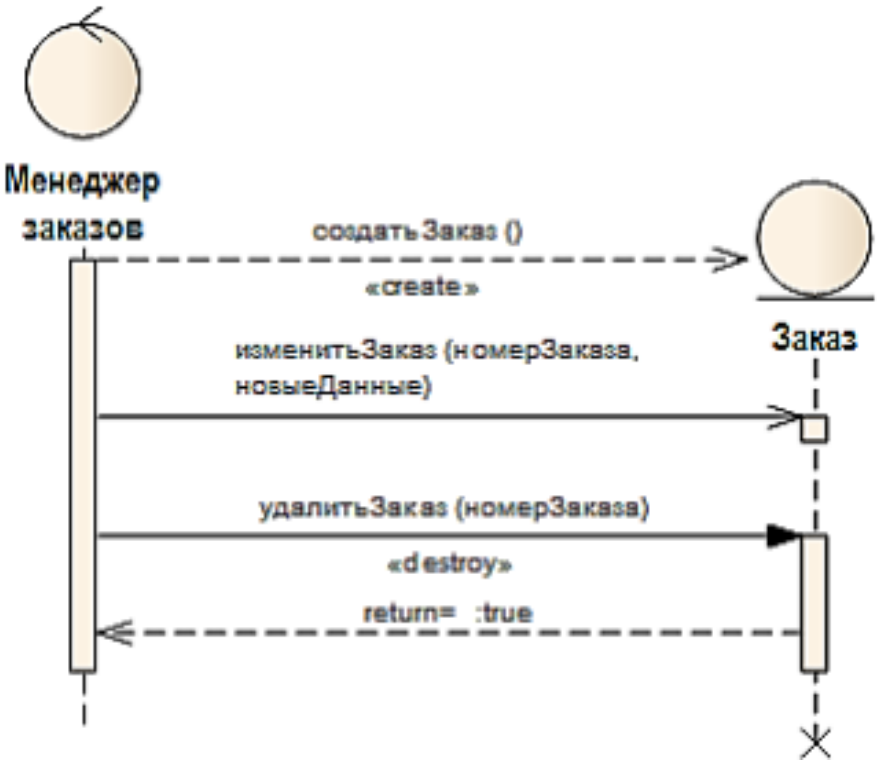


Рис.18. Пример сообщений на диаграмме последовательности состояния объекта.

Диаграмма последовательности объединяет диаграмму деятельности, диаграмму состояний и диаграмму классов.

Таким образом, на диаграмме последовательности мы можем увидеть следующие аспекты:

- Сообщения, побуждающие объект к действию
- Действия, которые вызываются сообщениями (методы) – зачастую это передача сообщения следующему объекты или возвращение определенных данных объекта
- Последовательность обмена сообщениями между объектами.

Итак, прием сообщения инициирует выполнение определенных действий, направленных на решение отдельной задачи тем объектом, которому это сообщение отправлено. Сообщение в большинстве случаев (за исключением диаграмм, описывающих концептуальный уровень Системы) это вызов методов отдельных объектов, поэтому для корректного исполнения метода в сообщении необходимо передать какие-то данные и определить, что мы хотим видеть в ответ. При именовании сообщения на уровне проектирования реализации системы в качестве имени сообщения следует использовать имя метода (Рисунок 19).

- синхронное сообщение (**synchCall**) – соответствует синхронному вызову операции и, подразумевает ожидание ответа от объекта получателя. Пока ответ не поступит, никаких действий в системе не производится.

• асинхронное сообщение (**asynchCall**) – которое соответствует асинхронному вызову операции и подразумевает, что объект может продолжать работу, не ожидая ответа.

• ответное сообщение (**reply**) – ответное сообщение от вызванного метода. Данный вид сообщения показывается на диаграмме по мере необходимости или, когда возвращаемые им данные несут смысловую нагрузку.

• потерянное сообщение (**lost**) – сообщение, не имеющее адресата сообщения, т. е. для него существует событие передачи и отсутствует событие приема

• найденное сообщение (**found**) – сообщение, не имеющее инициатора сообщения, т. е. для него существует событие приема и отсутствует событие передачи

Для сообщений также доступен ряд предопределенных стереотипов. Наиболее часто используемые стереотипы – это **create** и **destroy.**

Сообщение со стереотипом **create**, вызывает в классе метод, которые создает экземпляр класса. На диаграмме последовательности не обязательно показывать с самого начала все объекты, участвующие во взаимодействии. При использовании сообщения со стереотипом **create**, создаваемый объект отображается на уровне конца сообщения.

Для уничтожения экземпляра класса используется сообщение со стереотипом **destroy**, при этом в конце линии жизни объекта отображаются две перекрещенные линии.

При отображении работы с сообщениями иногда возникает необходимость указать некоторые временные ограничения. Например, длительность передачи сообщения или ожидание ответа от объекта не должно превышать определенный временной интервал (Рисунок20). Можно указать следующие временные параметры:

• ограничение продолжительности (Duration Constraint) – минимальное и максимальное значение продолжительности передачи сообщения;

• ограничение продолжительности ожидания между передачей и получением сообщения (Duration Constraint Between Messages);

• перехват продолжительности сообщения (Duration Observation);

• временное ограничение (Timing Constraint) – временной интервал, в течение которого сообщение должно прийти к цели (устанавливается на стороне получателя);

• перехват времени, когда сообщение было отправлено (Timing Observation).

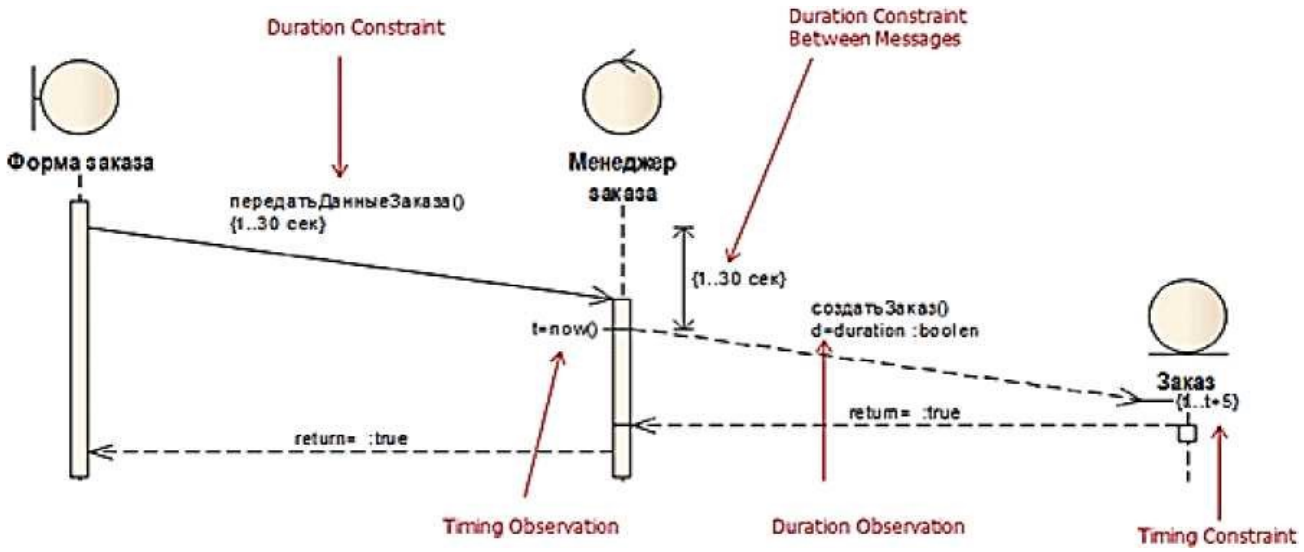


Рисунок 19. Пример временных ограничений сообщений на диаграмме последовательности

Форма заказа передает данные Менеджеру заказа, при этом передача данных не должна длиться больше 30 секунд – данное ограничение может понадобиться при выявлении требований к быстродействию Системы. Далее получение данных с формы инициирует запуск метода для создания экземпляра класса Заказ. Между получением данных от Формы заказа и инициализацией создания объекта должно пройти не более 30 секунд, в противном случае пользователю может быть предоставлено сообщение об ошибке или недоступности сервера. Длительность передачи сообщения о создании объекта может быть зафиксирована в переменной d.

Данное значение может понадобиться при установке временного ограничения на получение ответного сообщения клиентом. В момент передачи сообщения фиксируется временное значение и заносится в переменную t. Таким образом, можно установить ограничение на стороне приемника, указав переменную t в качестве минимального значения и t+< допустимый интервал > в качестве максимального значения.

До появления UML 2.0 диаграмма последовательности рассматривалась только в рамках моделирования последовательности обмена сообщениями. Расширение сценария отображались с помощью ветвления линий сообщений, что не давало полной картины взаимодействия объектов. Таким образом, для целей моделирования расширения сценария, параллельности процессов или цикличности использовались диаграммы деятельности. Для решения данных задач в UML 2.0 было введено понятие фрейма взаимодействия и операторов взаимодействия (Рисунок 20).

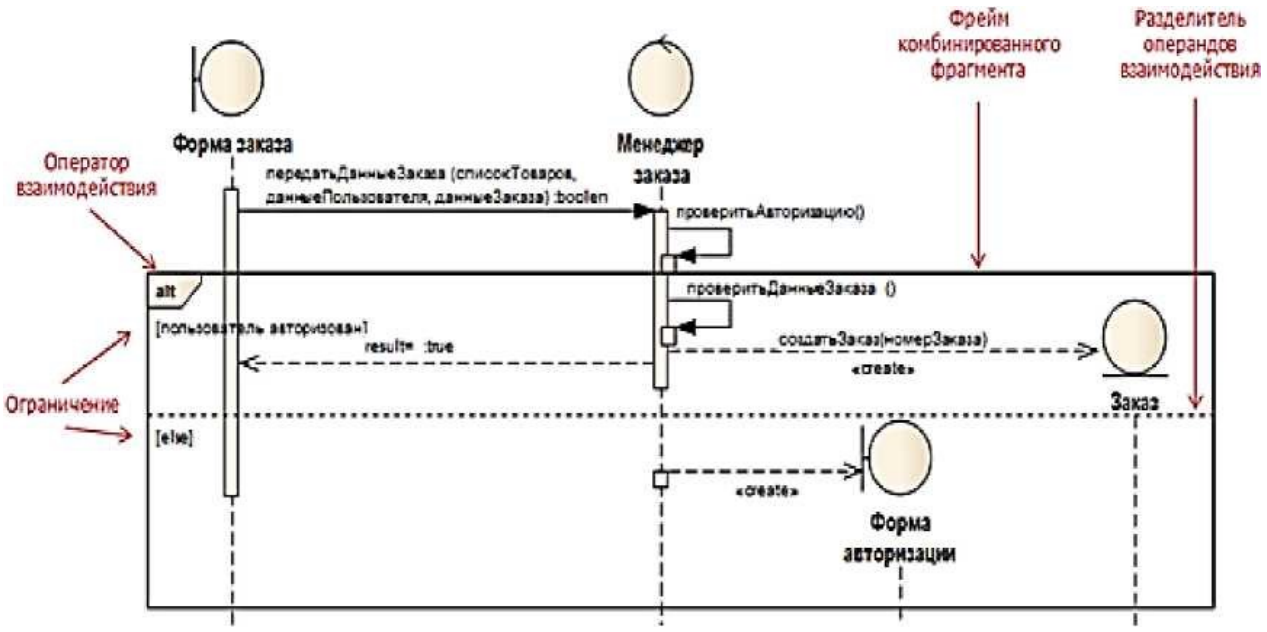


Рисунок 20. Пример фрейма взаимодействия и оператора взаимодействия на диаграмме последовательности

Отдельные фрагменты диаграммы взаимодействия можно выделить с помощью фрейма. Фрейм должен содержать метку оператора взаимодействия. UML содержит следующие операнды:

- **Alt –** Несколько альтернативных фрагментов (alternative); выполняется только тот фрагмент, условие которого истинно.
- **Opt –** Необязательный (optional) фрагмент; выполняется, только если условие истинно. Эквивалентно **alt** с одной веткой.
- **Par –** Параллельный (parallel); все фрагменты выполняются параллельно.
- **loop –** Цикл (loop); фрагмент может выполняться несколько раз, а защита обозначает тело итерации.
- **region –** Критическая область (critical region); фрагмент может иметь только один поток, выполняющийся за один прием.
- **Neg –** Отрицательный (negative) фрагмент обозначает неверное взаимодействие.
- **ref –** Ссылка (reference); ссылается на взаимодействие, опеделенное на другой диаграмме. Фрейм рисуется, чтобы охватить линии жизни, вовлеченные во взаимодействие. Можно определять параметры и возвращать значение.
- **Sd –** Диаграмма последовательности (sequence diagram); используется для очерчивания всей диаграммы последовательности, если это необходимо.

При использовании фрагмента условного операнда фрейм должен содержать

условие для ограничения взаимодействия. При использовании условного или параллельного операнда фрейм делится на регионы взаимодействия с помощью разделителя операторов взаимодействия.

К условным операндам относятся alt и opt. Операнд alt используется при моделировании расширения сценария, т.е. при наличии альтернативного потока взаимодействия. Оператор opt используется, если сообщение должно быть передано, только при истинности какого-то условия. Данный фрейм используется без разделения на регионы.

Параллельность потоков взаимодействия можно изобразить с помощью операнда par. Внутри фрейма моделируются потоки взаимодействия в отдельных регионах.

Цикличность потока взаимодействия может быть представлена на диаграмме последовательности с помощью операнда loop (Рисунок 21). При использовании оператора цикла можно указать минимальное и максимальное число итераций. Также фрейм должен содержать условие, при наступлении которого взаимодействие повторяется.

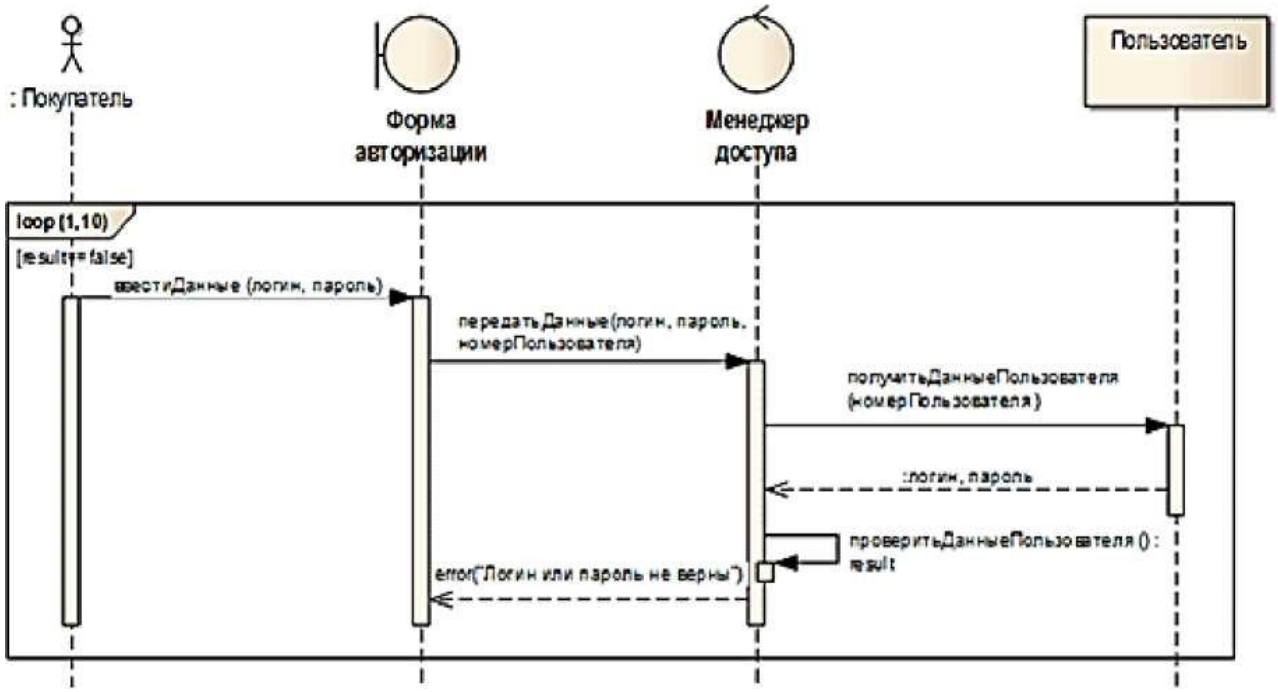


Рисунок 21. Пример использование оператора взаимодействия Loop на диаграмме последовательности

Диаграммы последовательности предназначены для моделирования взаимодействия между несколькими объектами. Зачастую диаграммы последовательности создаются для моделирования взаимодействия в рамках одного прецедента.

На концептуальном уровне можно использовать диаграммы последовательности для моделирования взаимодействия между Бизнес-актерами, но зачастую подобные диаграммы обрастают лишними подробностями и плохо читаются. На данном уровне лучше подойдут диаграммы деятельности, исключение составляют случаи, когда необходимо смоделировать обмен сообщениями между двумя независимыми Системами.

Также диаграммы последовательности подойдут для моделирования взаимодействия пользователя и Системы в целом.

На уровне детальной спецификации требований диаграммы последовательности используются для моделирования взаимодействия компонентов Системы и пользовательских классов в рамках выбранного прецедента.

На уровне реализации с помощью диаграммы последовательности моделируется взаимодействие между отдельными компонентами Системы. На данном уровне детализации лучше подойдет диаграмма коммуникации.

Задание

Заданием работы является построение диаграммы вариантов использования и диаграммы последовательности для предметной области из Приложения 1.

Контрольные вопросы

1. Что такое диаграмма последовательности действий?
2. Какие элементы содержит диаграмма последовательности действий?
3. Что такое диаграмма использования?

Лабораторная работа №2. Построение диаграммы деятельности, диаграммы состояний и диаграммы классов

Целью работы является изучение порядка построения различных диаграмм.

Для выполнения работы студент должен изучить приведенный ниже теоретический материал. Отчет сдается в распечатанном и электронном (файл Word) видах.

Диаграммы деятельности

Создание Информационной Системы – сложный процесс, который можно представить как поэтапный спуск от общей концепции будущей ИС, через понимание ее логической структуры к на и более детальным моделям, описывающим физическую реализацию. Диаграмма деятельности принадлежит к логической модели (Рисунок23).

В качестве графического представления для выделения основных функций Системы мы применяем диаграмму вариантов использования (use case). Диаграмма вариантов использования дает нам представление, что должна делать Система.

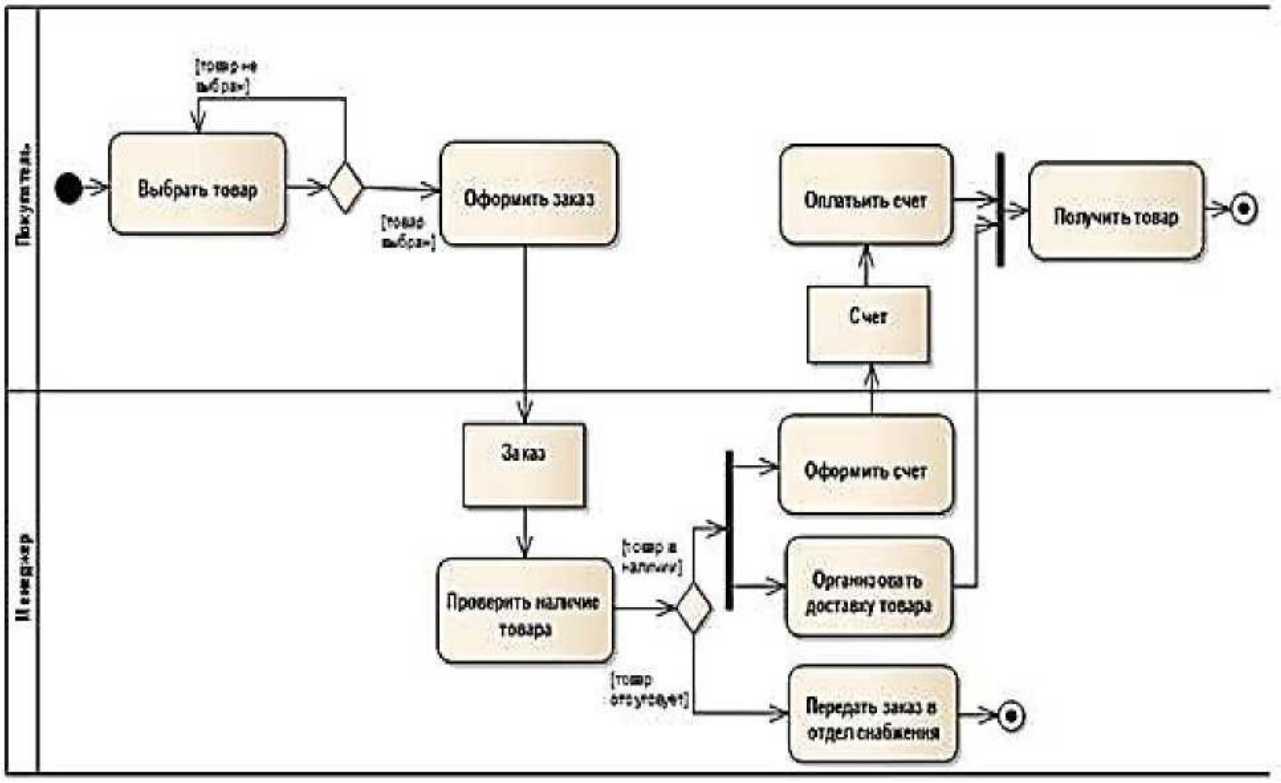


Рисунок 22. Пример диаграммы деятельности

То есть если варианты использования ставят перед Системой цель, то диаграмма деятельности показывает последовательность действий, необходимых для ее достижения. Действия (action) это элементарные шаги, которые не предполагают дальнейшую декомпозицию.

Деятельность может содержать входящие и/или исходящие дуги деятельности, показывающие потоки управления и потоки данных. Если поток соединяет две деятельности, он является потоком управления. Если поток заканчивается объектом, он является потоком данных.

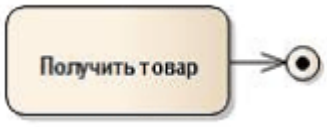
Деятельность выполняется, только тогда, когда готовы все его «входы», после выполнения, деятельность передает управление и(или) данные на свои «выходы». Саму диаграмму деятельности принято располагать таким образом, чтобы действия следовали слева направо или сверху вниз.

Чтобы указать, где именно находится процесс, используется абстрактная точка «**маркер**» (или «токен»). Визуально на диаграмме маркер не показывается, данное понятие вводится только для удобства описания динамического процесса.

Переход маркера осуществляется между узлами. Маркер может не содержать никакой дополнительной информации (пустой маркер) и тогда он называется маркером управления (control flow token) или же может содержать ссылку на объект или структуру данных, и тогда маркер называется маркером данных (data flow token).

Для создания диаграммы деятельности используются следующие узлы (Таблица 2).

Таблица 2. Схематичное изображение узлов, используемых в диаграммах деятельности

	Узел управления(control node) - это абстрактный узел действия, которое координирует потоки действий
	Начальный узел деятельности (или начальное состояние деятельности) (activity initial node) является узлом управления, в котором начинается поток (или потоки) при вызове данной деятельности извне
	Конечный узел деятельности (или конечное состояние деятельности) (activity final node) является узлом управления, который останавливает (stop) все потоки данной диаграммы деятельности. На диаграмме может быть более одного конечного узла
	Конечный узел потока (или конечное состояние потока) (flow final node) является узлом управления, который завершает данный поток. На другие потоки и деятельность данной диаграммы это не влияет
	Объект, над которым выполняются действия. Это не обязательный элемент диаграммы, но в некоторых случаях необходимо показать объект, инициирующий выполнение действий, или являющийся его результатом

Для отображения расширений сценария на диаграмме деятельности используются, так называемые узлы решения (Рисунок 23). **Узел решения** предназначен для определения правила ветвления и различных вариантов дальнейшего развития сценария.

Рисунок 23. Узлы решения на диаграмме деятельности

В точку ветвления входит ровно один переход, а выходит – два или более. Для каждого исходящего перехода задается булевское выражение, которое вычисляется только один раз при входе в точку ветвления. Ни для каких двух исходящих переходов эти сторожевые условия не должны одновременно принимать значение «истина», иначе поток управления окажется неоднозначным. Желательно чтобы условия покрывали все возможные варианты, иначе поток остановится.

Для пометки исходящего перехода, который должен быть выбран в случае, если условия, заданные для всех остальных переходов не выполнены, разрешается использовать ключевое слово else.

Далее следует обратить внимание на такой элемент, как **узел объединение**. Узел объединения имеет два и более входящих узла и один исходящий. Узлы решения объединения аналогичны логическому выражению «строгое или», т. е. для узла объединения – **только** при выполнении того **или** иного действия осуществляется переход к следующему узлу управления. Соответственно для узла решения – **только** при выполнении того **или** иного условия становится доступна возможность перехода к одному из следующих действий.

Для отображения условий соответствующих логическому оператору «и» на диаграмме используются синхронизационная черта (Рисунок 24).

Рисунок 26. Пример реализации логического оператора и на диаграмме деятельности

Точка разделения обеспечивает разделение одного потока на несколько

параллельных потоков:

- входит ровно один поток;
- выходит два и более потока, каждый из которых далее выполняется параллельно с другими.

Точка слияния обеспечивает синхронизацию нескольких параллельных потоков.

- входят два или более потока, причем эти потоки выполняются параллельно;
- выходит ровно один поток, причем в точке слияния входящие параллельные потоки синхронизируются, то есть каждый из них ждет, пока все остальные достигнут этой точки, после чего выполнение продолжается в рамках одного потока.

Также диаграмма действия может описывать поведение, на которое оказывают влияние внешние события, происходящие за пределами данной Системы.

На диаграмме это может быть показано при помощи изображения передачи сигнала (Рисунок 25). Передача сигнала может изображаться путем помещения между двумя действиями соответствующего элемента. Данная семантика была принята в UML 2.0.

Рисунок 25. Изображение передачи сигнала на диаграмме деятельности

Передача сигнала (send signal action) – действие, которое на основе своих входов создает экземпляр сигнала и передает его внешней Системе.

Прием события (receive eventaction) – действие, которое ожидает некоторого события, принимает и обрабатывает полученное сообщение.

На диаграмме представлено взаимодействие двух независимых Систем: «Учетная система» и «Веб-магазин» (Рисунок 25).

• Результатом действия по приему банковской выписки и разнесению оплаты является входящий сигнал для ИС «Веб-магазин» сообщающей об оплате товара.

• Соответственно при получении входящего сигнала в ИС «Веб-магазин» фиксируется факт оплаты, который инициализирует действие «Разрешить отгрузку».

Для изображения передачи сигнала мы можем поместить между двумя узлами деятельности символ деятельности передачи или ожидания сигнала, или непосредственно узел объекта, который будет символизировать сигнал.

Для отображения объекта, осуществляющего управление потоками из нескольких источников, в UML 2 появилось два специальных узла: центральный буфер и хранилище данных.

Центральный буфер – объект, который управляет потоками между множественными источниками и приемниками. На диаграмме центральный буфер представляется в виде объекта со стереотипом <<centralbuffer>>. </centralbuffer>>

Данный объект может применяться на уровне описания реализации функций Системы для визуализации временных таблиц.

На Рисунок 26 представлена диаграмма, которая отражает сценарий формирования списка сравнения товаров:

• В центральный буфер поступает информация о товаре, выбранном для сравнения пользователем из каталога товаров.

• Данные товара хранятся в центральном буфере какой-то промежуток времени.

• Далее пользователь вызывает для просмотра список сравнения товаров, просматривает его и сохраняет наиболее подходящий товар в корзину.

• Товар, выбранный для покупки, сохраняется в таблице БД, хранящей заказы пользователя, в то время как другие товары из временной таблицы сравнения удаляются.

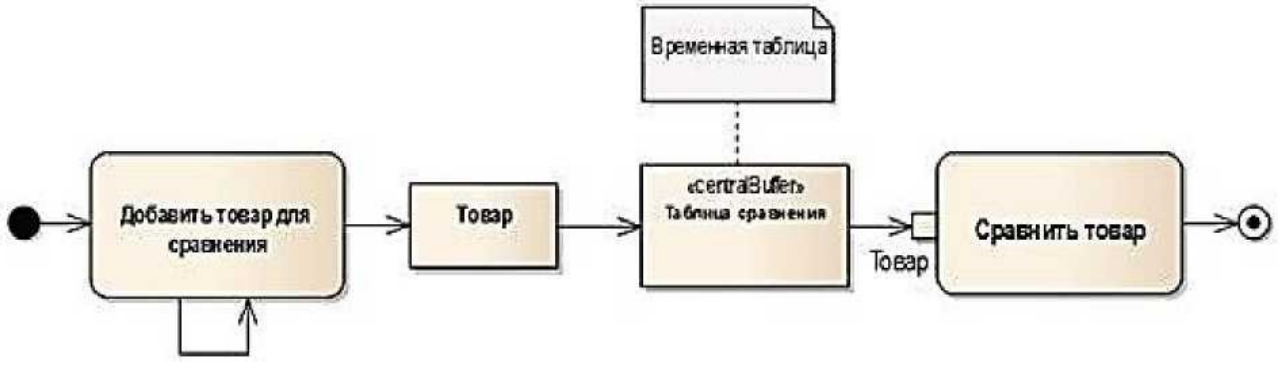


Рисунок 27. Диаграмма сценария формирования списка сравнения товаров

Для оптимизации диаграммы входные и выходные объекты могут заменяться изображением «контакт». Входной контакт, в данном случае, является узлом объекта, который принимает значения от других действий в форме потока объектов. Соответственно выходной контакт поставляет значения другим действиям в форме потока объекта.

Частный случай Центрального буфера – Хранилище данных представлен на Рисунок 27.

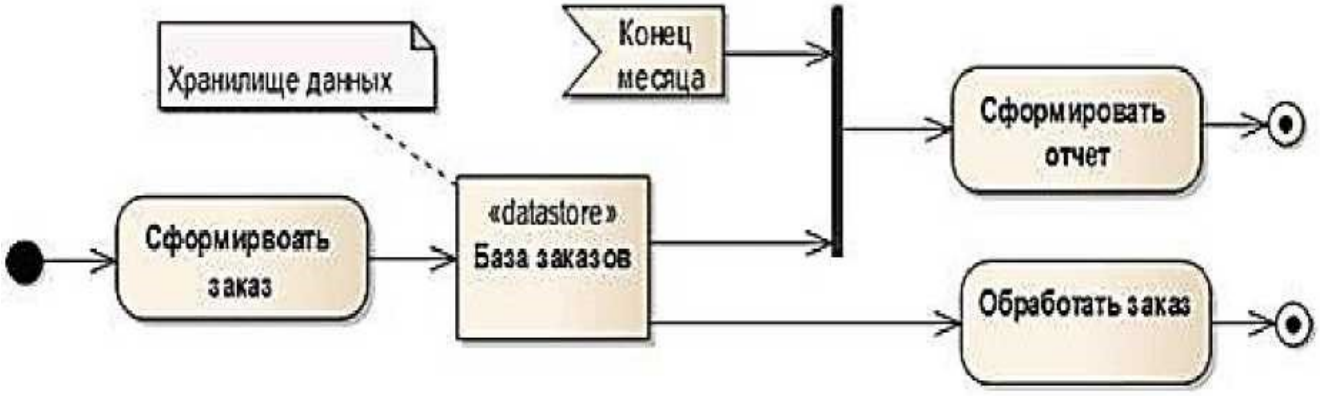


Рисунок 27. Частный случай диаграммы деятельности

Принципиальным отличием Хранилища данных является то, что оно содержит все поступившие данные и на выходе отдает лишь копии. Таким образом, результатом действия «Сформировать заказ» является непосредственно заказ, который помещается в базу заказов. Для дальнейшей обработки заказа или мониторинга выполнения заказов, из базы осуществляется запрос данных заказа. Данные предоставляются в виде копий, в то время как оригинал продолжает оставаться в Базе заказов. Копирование данных осуществляется каждый раз, когда заказ выбирается для осуществления каких-либо действий.

Если пришедший заказ уже содержится в хранилище, то предыдущий объект будет заменен.

Далее следует подробно рассмотреть разбиение деятельности на разделы.

Разделы группируют действия относительно какой-либо общей характеристики, при этом на течение потоков эта группировка никак не влияет. В более ранних версиях UML использовалось такое понятие как дорожки (swimlanes) по аналогии с дорожками в плавательном бассейне (Рисунок 28).

Дорожки, используемые при данной структуре диаграммы деятельности, зачастую символизируют роль пользователя или организационное подразделение,

осуществляющее определенные действия в рамках данной деятельности.

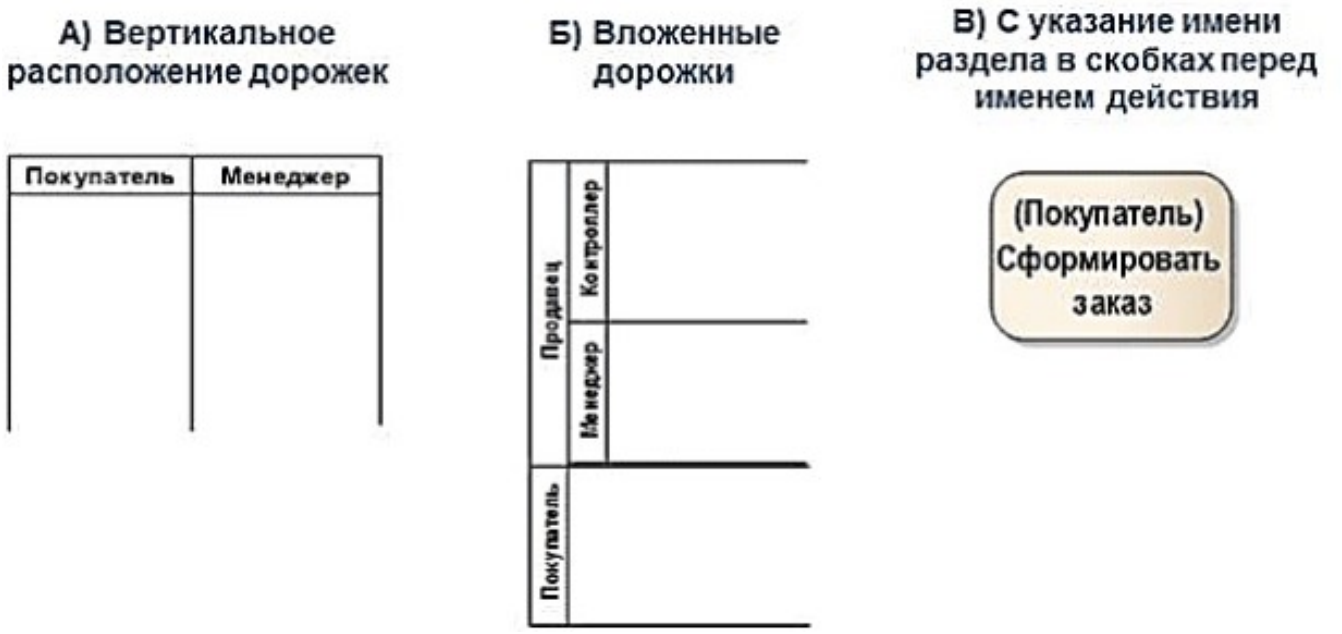


Рисунок 28. Расположение дорожек на диаграммах деятельности

Расположение дорожек может быть как вертикальным, так и горизонтальным. Несколько дорожек могут быть объединены по организационному принципу.

В UML 2 принято правило применять горизонтальное расположение дорожек для отображения модели бизнес- процесса.

Для оптимизации диаграммы деятельности, использование дорожек можно заменить указанием наименования раздела перед наименованием действия.

Как уже говорилось, для описания процессов верхнего уровня на диаграмме мы показываем переход между деятельностями, которые в свою очередь содержат свою последовательность деятельностей или действий.

Спецификация UML дает несколько способов представления декомпозиции деятельности на диаграмме. Мы можем использовать обозначение поддеятельности (subactivity state), где указываем:

- наименование деятельности;
- предусловие и постусловие
- пиктограмму, информирующую о наличие развернутой диаграммы для данной деятельности .

Данная форма не дает нам представления о последовательности действий для данной деятельности, а лишь предоставляет ссылку на более детальную диаграмму.

При необходимости описание последовательности действий для поддеятельности может быть размещено непосредственно на основной диаграмме. Для этого описание

поддеятельности размещается в отдельный фрейм.

При таком способе декомпозиции мы можем указать:

- предусловия и постусловия;
- входные и выходные параметры(объекты);
- внутреннее устройство деятельности.

Диаграмма деятельности – мощный инструмент, который интенсивно используется при создании ИС.

В зависимости от, поставленной перед нами задачи мы создаем диаграмму деятельности, используя тот набор элементов, который необходим для отражения определенного уровня детализации.

Таким образом, диаграмма деятельности может применяться как для описания бизнес-процесса, так и функциональных требований к Системе.

Цель концептуального описания – показать целостную картину бизнес-процессов предметной области (Рисунок 29).

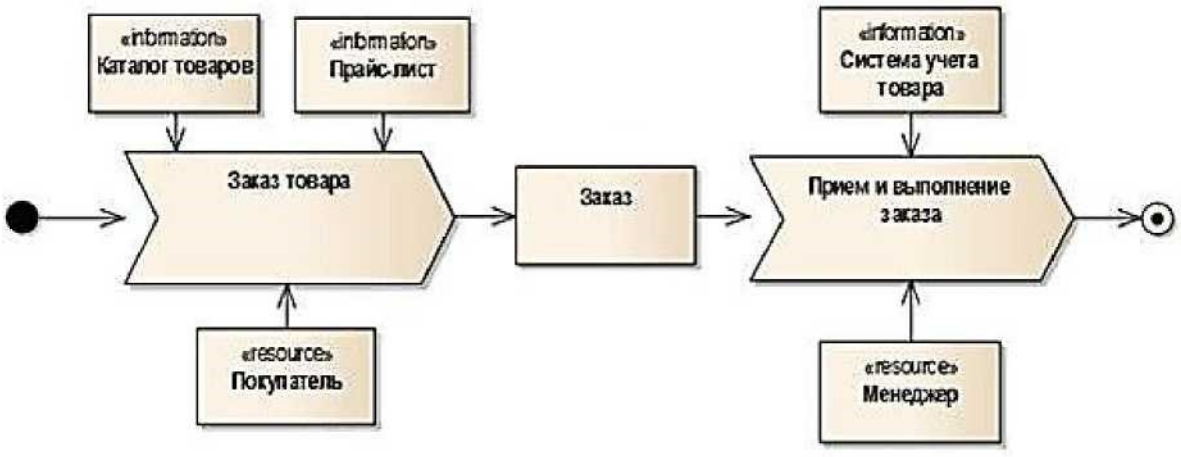


Рисунок 29. Концептуальное описание бизнес процессов предметно области с помощью диаграммы деятельности

Для описания концепции процесса совершения покупки через интернет-магазин можно использовать диаграмму действия в стиле IDEF0, где указываются следующие параметры:

- входные и выходные данные;
- объекты, управляющие процессом (в нашем случае это каталог товаров и прайс-лист);
- используемые ресурсы (в нашем примере это Покупатель и Менеджер).

На Рисунок 30 показаны стандартные UML-объекты «действие» и «объект», но со

специальными стереотипами:

- Для действия – стереотип<<process>></process>>
- Для объекта – стереотип <<information>></information>>

Для создания концептуальной модели необходимо выделять только основные процессы, так как вспомогательные процессы и сущности могут перегружать диаграмму. Также концептуальная модель должна включать только процессы верхнего уровня. Далее можно детализировать каждый процесс на отдельной диаграмме, как это показано на Рисунок 30.

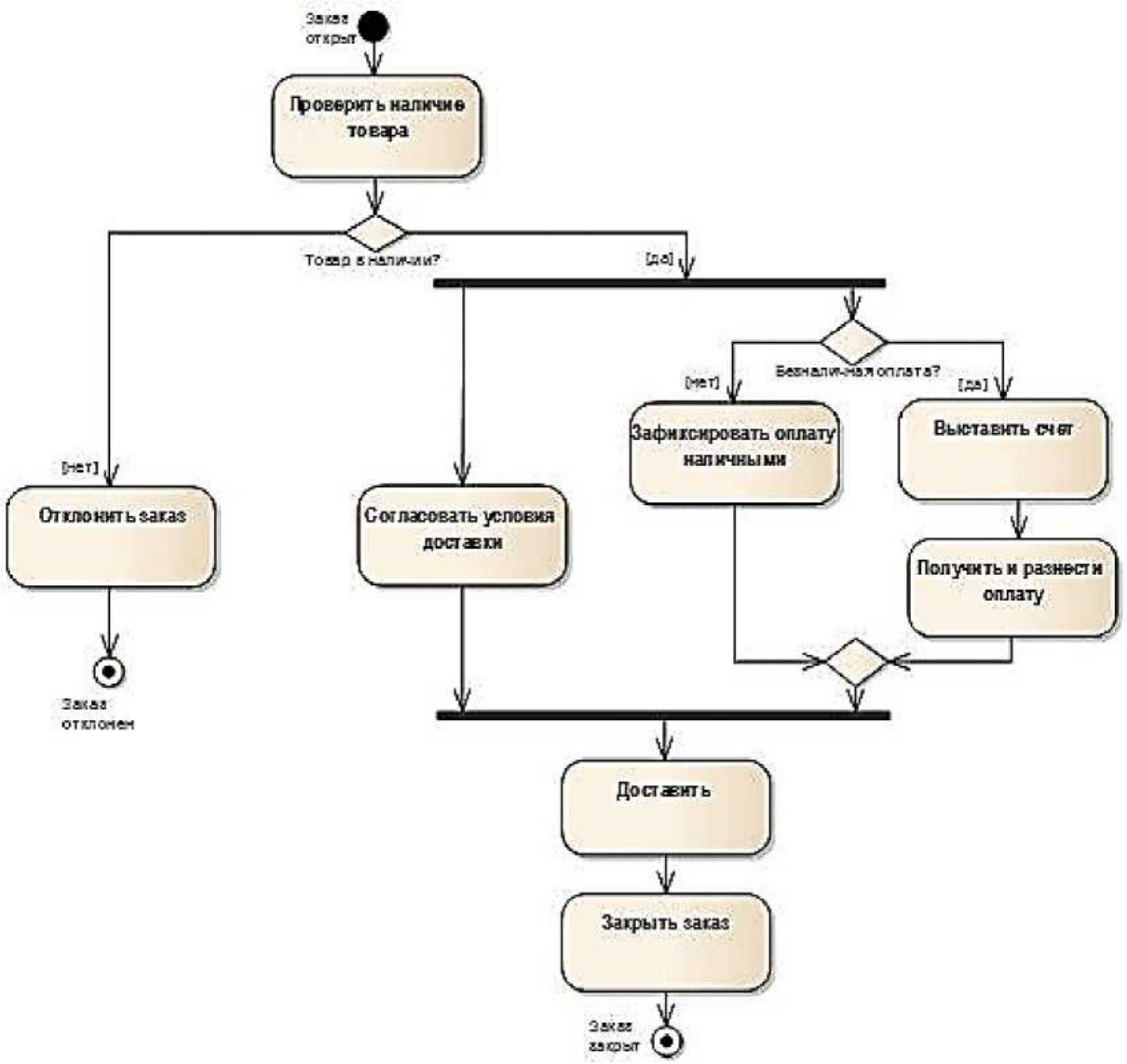


Рисунок 30. Детализация процессов диаграммы деятельности

Диаграмма деятельности данного вида хорошо отражает:

- последовательность действий;
- события, инициирующие действия или являющиеся конечным результатом;
- условия расширения сценария;

Для того чтобы отобразить соответствие деятельности определенному

пользователю или Системе к данной диаграмме можно применить «дорожки». Так как в нашем случае все действия выполняются менеджером, применение разделителей не целесообразно.

Данный способ иллюстрации бизнес-процесса может охватывать не только действия, происходящие внутри, разрабатываемой системы, но производящиеся за ее пределами, что необходимо для формирования четкого представления о процессе в целом.

Наш пример содержит только действия, совершаемые в рамках ИС, поэтому данная диаграмма может быть помещена в качестве иллюстрации к сценарию использования в раздел «Общее описание функций» документа «Техническое задание на разработку ИС». Если, на диаграмме последовательность действий будет включать деятельность, выходящую за рамки ИС (например, «оплатить товар»), она может быть размещена в разделе «Сведения об объекте автоматизации».

Также диаграмма деятельности целесообразна для описания требований на уровне взаимодействия компонентов Системы. Целевой аудиторией в данном случае будет являться команда разработчиков.

Если на диаграмме необходимо показать последовательность действий, вызываемых сторонними Системами, то целесообразно добавить элементы получения и приема сигналов.

Диаграмма состояний

Диаграмма деятельности полезна для описания алгоритма действий, но она не дает представления о поведении определенного объекта в рамках отдельного варианта использования или системы в целом, что необходимо при объектно-ориентированном программирования.

На сегодняшний день при проектировании сложной Системы

принято делить ее на части, каждую из которых затем рассматривать отдельно. Таким образом, при объектной декомпозиции Система разбивается на объекты или компоненты, которые взаимодействуют друг с другом, обмениваясь сообщениями. Сообщения описывают или представляют собой некоторые события. Получение объектом сообщения активизирует его и побуждает выполнять предписанные его программным кодом действия. При данном подходе Система становится событийно управляемой, поэтому разработчикам зачастую важно знать, как должен реагировать

тот или иной объект на определенные события. Инициаторами событий могут быть как объекты самой Системы, так и ее внешнее окружение.

Описать поведение отдельно взятого объекта помогает диаграмма состояний.

Также зачастую диаграмма состояний используется аналитиками для описания последовательности переходов объекта из одного состояния в другое.

Диаграмма состояний покажет нам все возможные состояния, в которых может находиться объект, а также процесс смены состояний в результате внешнего влияния.

Основными элементами диаграммы состояний являются «Состояние» и «Переход». Диаграмма состояний имеет схожую семантику с диаграммой деятельности, только деятельность здесь заменена состоянием, переходы символизируют действия. Таким образом, если для диаграммы деятельности отличие между понятиями «Деятельность» и «Действие» заключается в возможности дальнейшей декомпозиции, то на диаграмме состояний деятельность символизирует состояние, в котором объект находится продолжительное количество времени, в то время как действие моментально (Рисунок 31).

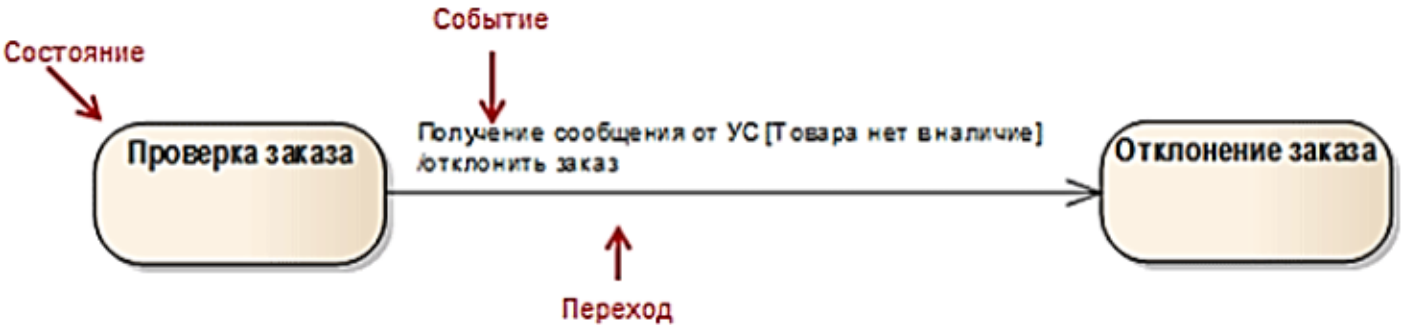


Рисунок 31. Переход от состояния к событию на диаграмме состояний

Переход может быть инициирован событием, которое также отражается на диаграмме состояний (Рисунок 32).

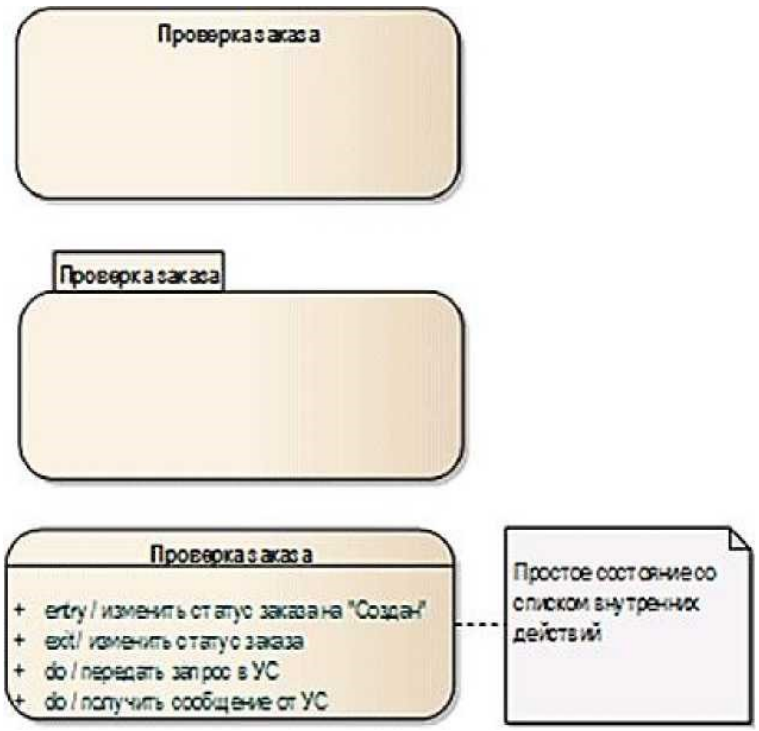


Рисунок 32. Инициирование перехода событием на диаграмме состояний

Состояние может содержать только имя или имя и дополнительно список внутренних действий. Список внутренних действий содержит перечень действий или деятельностей, которые выполняются во время нахождения объекта в данном состоянии (Рисунок 33). Данный список фиксированный. Список основных действий включает следующие значения:

- entry – действие, которое выполняется в момент входа в данное состояние (входное действие);
- exit – действие, которое выполняется в момент выхода из данного состояния (выходное действие);
- do – выполняющаяся деятельность («do activity») в течение всего времени, пока объект находится в данном состоянии;
- defer – событие, обработка которого предписывается в другом состоянии, но после того, как все операции в текущем будут завершены.

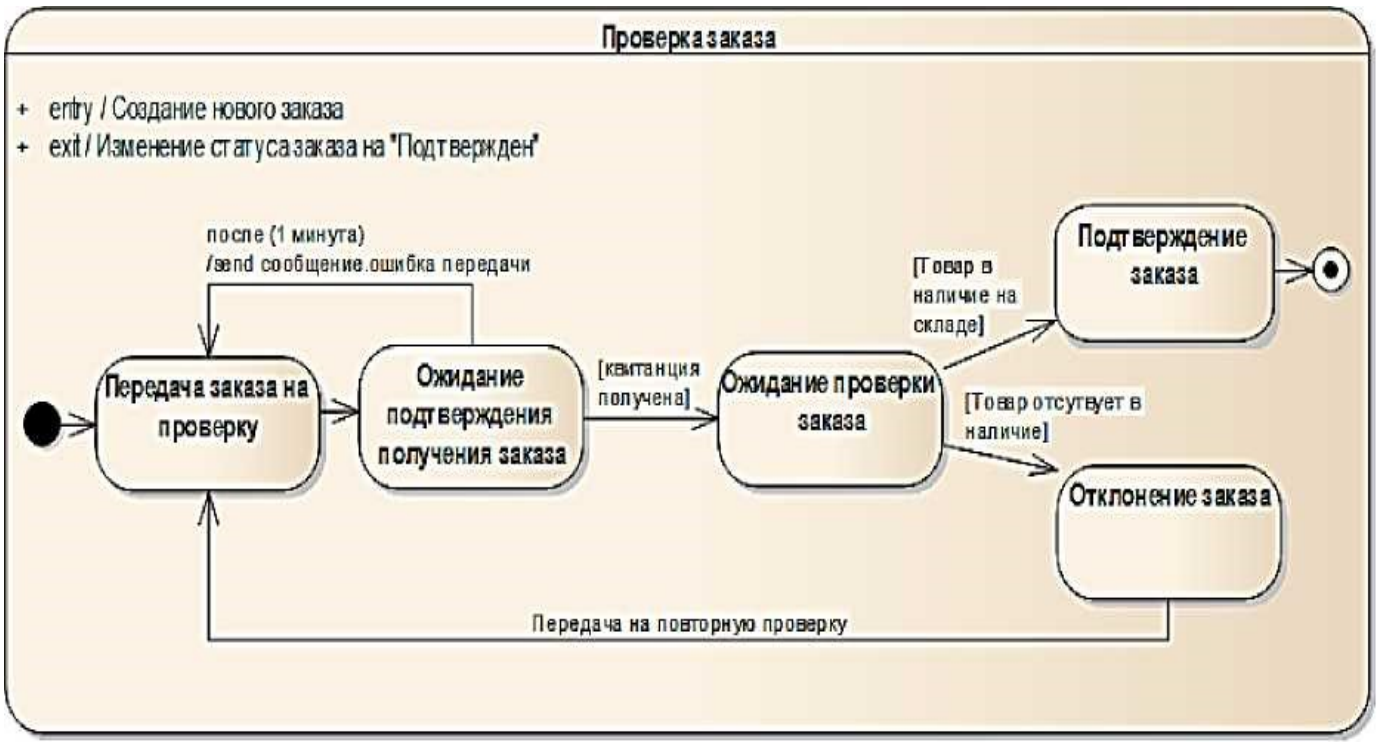


Рисунок 33. Список внутренних действий на диаграмме состояний

Факт смены одного состояния другим изображается с помощью **перехода**. Переход осуществляется при наступлении некоторого события: окончания выполнения деятельности (do activity), получении объектом сообщения или приемом сигнала. Переход может быть **тригерным** и **не тригерным**. Если переход срабатывает, когда все операции исходного состояния завершены, он называется не тригерным или переходом по завершении. Если переход инициируется каким-либо событием, он считается тригерным. Для тригерного перехода характерно наличие имени, которое может быть записано в следующем формате:

<имя события>'('<список параметров, разделенных запятыми>')'['<сторожевое условие>']'<выражение действия>.

Обязательным параметром является только имя события.

В качестве **события** могут выступать сигналы, вызовы, окончание фиксированных промежутков времени или моменты окончания выполнения определенных действий. После имени события могут следовать круглые скобки для явного задания параметров соответствующего события-триггера (например, пользователь инициирующий действие).

Различаются следующие виды событий:

- Событие вызова (callevent) – событие, возникающее при вызове метода класса. При срабатывании данного вида события объектом ассинхронно создается

Сигнал, который принимается другим объектом. Для указания на то, что некоторая операция посылает сигнал, можно воспользоваться зависимостью со стереотипом send.

- Событие сигнала (signalevent) – событие, возникающее при посылке сигнала. Если событие сигнала представляет возбуждение сигнала, то событие вызова предназначено для описания выполнения операции. То есть переход осуществляется при получении сигнала от другого объекта. В то время как сигнал является событием асинхронным, событие вызова обычно синхронно. Сигнал также может быть представлен на диаграмме в виде объекта со стереотипом «signal».
- Событие таймера (time event) – возникает, когда истек заданный интервал времени с момента попадания автомата в данное состояние. В UML событие времени моделируется с помощью ключевого слова after(после), за которым следует выражение, вычисляющее некоторый промежуток времени.
- Событие изменения (change event) – событие, которое возникает, когда некоторое логическое условие становится истинным, будучи до этого ложным. Данное событие моделируется с помощью ключевого слова when

Если при срабатывании перехода возможно ветвление, в имени перехода используется сторожевое условие. **Сторожевое условие (guard condition)** всегда записывается в прямых скобках после события-триггера и представляет собой некоторое булевское выражение. В общем, случае из одного состояния может быть несколько переходов с одним и тем же событием- триггером, при этом целевое состояние будет зависеть от того какое из сторожевых условий примет значение «истина».

Также имя перехода может содержать **выражение действия (action expression)**. В данном случае указанное действие выполняется сразу при срабатывании перехода и до начала каких бы то ни было действий в целевом состоянии. В общем случае выражение действия может содержать целый список отдельных действий, разделенных символом «;».

При создании диаграммы состояний для отдельных компонентов Системы выражение действия записывается на одном из языков программирования, который предполагается использовать для реализации модели.

Помимо основных узлов, на диаграмме состояний могут использоваться, так называемые, псевдосостояния–вершины которые не обладают поведением, и объект не

находится в ней, а «мгновенно» ее проходит.

Под псевдосостояниями на диаграмме состояний понимаются, знакомые уже нам начальное и конечное состояние. **Начальное состояние** обычно не содержит никаких внутренних действий и определяет точку, в которой находится объект по умолчанию в начальный момент времени. **Конечное состояние** также не содержит никаких внутренних действий и служит для указания на диаграмме области, в которой завершается процесс изменения состояний в контексте конечного автомата.

Если необходимо отразить уничтожение объекта используется узел завершения (terminate node), псевдосостояние, вход в который означает завершение выполнения поведения конечного автомата в контексте его объекта (Рисунок 34).

Рисунок 34. Пример уничтожение объекта (terminate) на диаграмме состояний

Узлы ветвления и объединения аналогичны узлам на диаграмме деятельности (Рисунок 35). Основная цель данных подсостояний показать параллельную работу подавтоматов. На диаграмме состояний обычно данные подсостояния используются распараллеливанием переходов в композитных состояниях, о которых речь пойдет позже. После срабатывания перехода моделируемый объект одновременно будет находиться во всех целевых состояниях этого перехода.

Варианты принятия решений на диаграмме состояний могут быть показаны также как и на диаграмме деятельности с помощью узла выбора. При этом переход в состояние выбора должен быть триггерным и содержать имя события. Переходы из псевдосостояния выбора в целевые состояния должны содержать сторожевые условия. Переход, который должен срабатывать, если ни одно из условий не примет значение «истина» должен содержать метку «else».

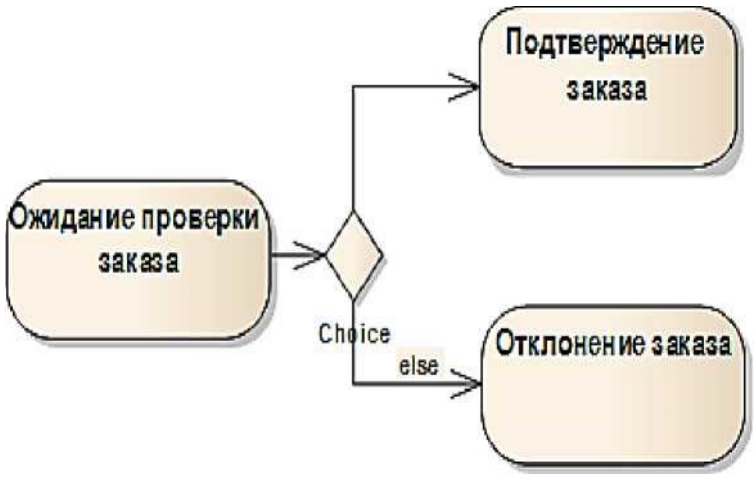


Рисунок 35. Узлы ветвления и объединения на диаграмме состояний

В отличие от диаграммы деятельности, при отображении возможных вариантов перехода на диаграмме состояний узел выбора использовать не обязательно. Диаграмма состояний должна показывать возможное изменение состояния объекта, и не имеет своей целью выстраивать четкую последовательность переходов. Таким образом, из одного состояния могут выходить несколько переходов, конечной целью которых будут различные целевые состояния. Для отображения возможности выбора в данном случае достаточно в имени всех переходов добавить триггер и сторожевое условие.

Также узел выбора на диаграмме состояний может быть заменен узлом соединения. Данное псевдосостояние имеет достаточно свободную семантику и предназначено для соединения нескольких переходов. В отличие от псевдосостояния выбора узел соединения может иметь несколько входящих переходов и несколько исходящих переходов, т.е. он может принимать значение логического выражения «или».

На диаграмме могут быть представлены как простые состояния, так и сложные состояния. Сложные или составные состояния (composite state) включают в себя вложенные подсостояния (Рисунок 36). Декомпозиция сложного состояния может осуществляться как на основной диаграмме, так и отдельно, при этом на основной диаграмме следует использовать элемент спиктограммой декомпозиции.

Составное состояние может содержать два или более параллельных подавтомата или несколько последовательных подсостояний.

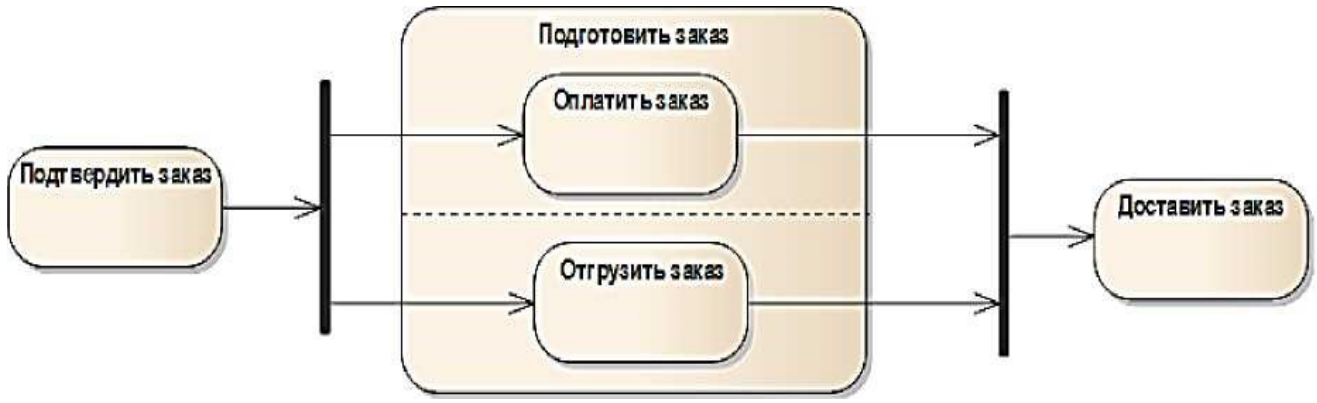


Рисунок 36. Сложные или составные состояния

Последовательные подсостояния (sequential substates) используются для моделирования такого поведения объекта, во время которого в каждый момент времени объект может находиться в одном и только одном подсостояний. Поведение объекта в этом случае представляет собой последовательную смену подсостояний, начиная от начального и заканчивая конечным подсостояниями.

Параллельные подсостояния (concurrent substates) позволяют специфицировать два и более подавтомата, которые могут выполняться параллельно внутри составного события. Каждый из подавтоматов занимает некоторую область (регион) внутри составного состояния.

Переходы могут осуществляться как в само композитное состояние, так и в одно из его подсостояний. Таким образом, переход, стрелка которого соединена с границей некоторого составного состояния, обозначает переход в составное состояние. Он эквивалентен переходу в начальное состояние каждого из подавтоматов. Переход, выходящий из составного состояния относится к каждому из вложенных подсостояний. Это означает, что объект может покинуть составное суперсостояние, находясь в любом из его подсостояний. Если необходимо указать конкретное подсостояние из которого может осуществиться выход из композитного состояния, достаточно добавить переход от подсостояния в целевое состояние.

В случае перехода в сложное состояние для каждого из начальных подсостояний выполняются необходимые входные («entry») действия. При выходе из сложного состояния для каждого из конечных подсостояний выполняются необходимые выходные («exit») действия.

Иногда возникает ситуация, когда необходимо показать переход из одного состояния в подсостояние композитного состояния, декомпозиция которого производится на отдельной диаграмме.

В UML 1.0 для подобных случаем использовались элементы «заглушка» и «ссылочное состояние». В UML 2.0 данные элементы были замены «точкой входа» и «точкой выхода» (Рисунок 37).

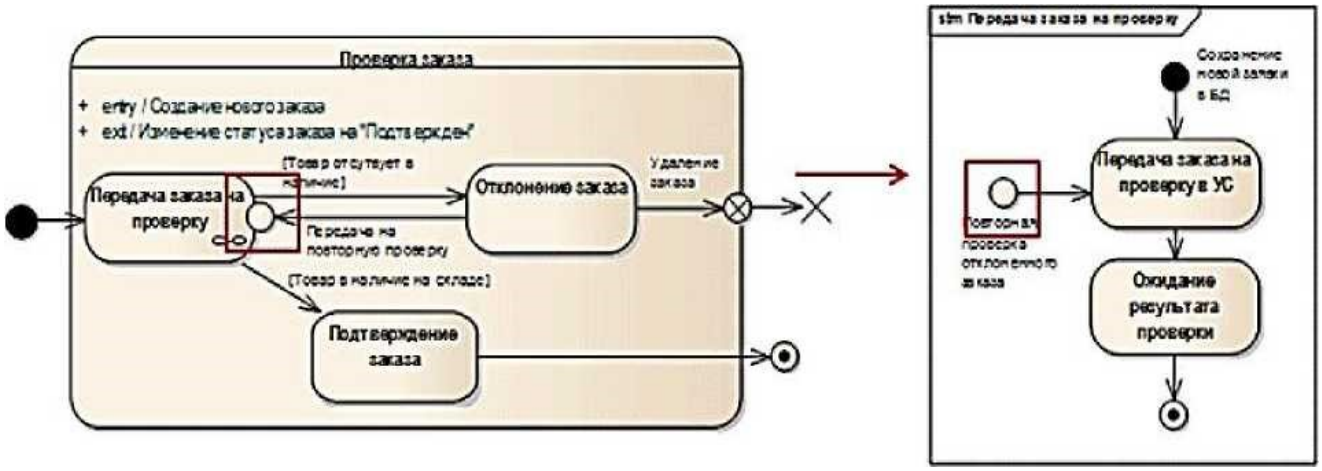


Рисунок 37. Пример декомпозиции состояния на диаграмме состояний

Точка входа – псевдосостояние, моделирующее вход в композитное состояние. При этом данная точка вход должна представлять альтернативный вход в композитное состояние, т. е. целевое подсостояние должно отличаться от начального подсостояния данного суперсостояния.

Точка выхода также символизирует альтернативный выход из композитного состояния. Данная семантика также может применяться при отображении повторяющихся действий (Рисунок 38).

Также с понятием композитного состояния тесно связано понятие исторического состояния.

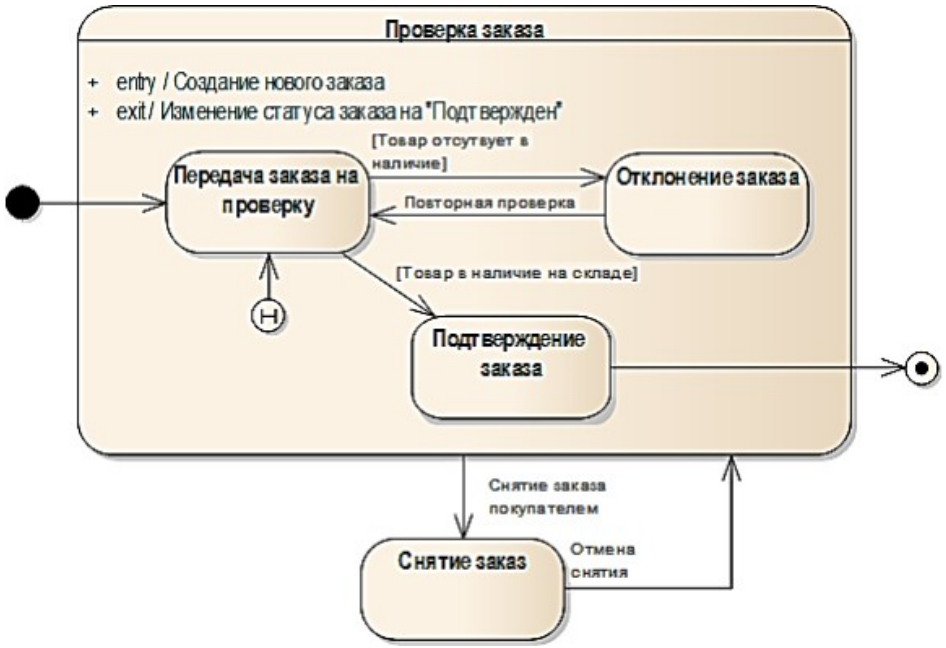


Рисунок 38. Отображение повторяющихся действий на диаграмме состояний

Историческое состояние используется для запоминания того из последовательных подсостояний, которое было текущим в момент выхода из составного состояния. Существует две разновидности исторического состояния: недавнее и давнее.

Недавнее историческое состояние (shallow history state) является первым подсостоянием в составном состоянии, и переход извне в это составное состояние должен вести непосредственно в это историческое состояние. При первом попадании в недавнее историческое состояние оно не хранит никакой истории (история пуста), то есть заменяет собой начальное состояние подавтомата. Далее следует последовательное изменение вложенных подсостояний.

Если в некоторый момент происходит выход из вложенного состояния (например, в случае некоторого внешнего события), то это историческое состояние запоминает то из подсостояний, которое являлось текущим на момент выхода. При следующем входе в это же составное состояние историческое подсостояние уже имеет непустую историю и сразу отправляет подавтомат в запомненное подсостояние, минуя все предшествующие ему подсостояния. В момент перехода в конечное состояние подавтомата, историческое состояние теряет свою историю.

Недавнее историческое состояние запоминает историю только того подавтомата, к которому он относится. Если запомненное состояние, в свою очередь, также являться композитным, для запоминания его подсостояния необходимо использовать давнее историческое состояние (deep history state). Давнее историческое состояние служит для запоминания всех подсостояний любого уровня вложенности для текущего подавтомата.

Автоматы состояний можно использовать при моделировании поведения графического интерфейса, как реакции на действия пользователя, различные приложения с множеством разных режимов работы в которых система ведет себя по-разному, моделирование объектов. Диаграмма автоматов зачастую используется в системах реального времени, где требуется высокая вычислительная скорость, поскольку за счет статического анализа, изменения состояний и переходы осуществляются очень быстро, как правило, это сводится к присваиванию полю класса, нескольких вызовов и запуска событий. При использовании диаграммы состояний для классов можно на ее основе сразу сгенерировать код (прямое

проектирование).

Если система ожидает наступления каких-либо событий и выполняет определенные действия в ответ, конечный автомат может быть легко использован для спецификации требуемого поведения в определенном варианте использования.

При использовании диаграммы состояний важно следовать следующим правилам:

- Диаграмма состояний должна создаваться только для объектов, обладающих реактивным поведением. Не следует делать диаграмму автоматов для всех классов или объектов, достаточно выбрать только основные классы или объекты, обладающие сложным поведением.
- Диаграмма состояний должна быть сосредоточена на описании только одного аспекта поведения объекта. Следует создавать диаграмму автомата, моделирующую поведение только одного объекта. Если необходимо показать поведение

нескольких, взаимосвязанных объектов, допустимо создавать для них диаграмму состояний в рамках определенного варианта использования (диаграмма состояний для варианта использования).

- На диаграмме состояний целесообразно использовать только те элементы, которые существенны для понимания описываемого аспекта.

Диаграммы классов UML. Логическое моделирование

Диаграммы классов используются при моделировании ПС наиболее часто. Они являются одной из форм статического описания системы с точки зрения ее проектирования, показывая ее структуру. Диаграмма классов не отображает динамическое поведение объектов изображенных на ней классов. На диаграммах классов показываются классы, интерфейсы и отношения между ними.

Представление классов

Класс – это основной строительный блок ПС. Это понятие присутствует и в ОО языках программирования, то есть между классами UML и программными классами есть соответствие, являющееся основой для автоматической генерации программных кодов или для выполнения реинжиниринга. Каждый класс имеет название, атрибуты и операции. Класс на диаграмме показывается в виде прямоугольника, разделенного на 3 области. В верхней содержится название класса, в средней – описание атрибутов (свойств), в нижней – названия операций – услуг, предоставляемых объектами этого

класса (Рисунок 39).

Рисунок 39. Изображение класса в нотации UML

Атрибуты класса определяют состав и структуру данных, хранимых в объектах этого класса. Каждый атрибут имеет имя и тип, определяющий, какие данные он представляет. При реализации объекта в программном коде для атрибутов будет выделена память, необходимая для хранения всех атрибутов, и каждый атрибут будет иметь конкретное значение в любой момент времени работы программы. Объектов одного класса в программе может быть сколь угодно много, все они имеют одинаковый набор атрибутов, описанный в классе, но значения атрибутов у каждого объекта свои и могут изменяться в ходе выполнения программы.

Для каждого атрибута класса можно задать видимость (visibility). Эта характеристика показывает, доступен ли атрибут для других классов. В UML определены следующие уровни видимости атрибутов:

- Открытый (public) – атрибут виден для любого другого класса (объекта);
- Защищенный (protected) – атрибут виден для потомков данного класса;
- Закрытый (private) – атрибут не виден внешними классами (объектами) и может использоваться только объектом, его содержащим.

Последнее значение позволяет реализовать свойство инкапсуляции данных. Например, объявив все атрибуты класса закрытыми, можно полностью скрыть от внешнего мира его данные, гарантируя отсутствие несанкционированного доступа к ним. Это позволяет сократить число ошибок в программе. При этом любые изменения в составе атрибутов класса никак не скажутся на остальной части ПС. Класс содержит объявления операций, представляющих собой определения запросов, которые должны выполнять объекты данного класса. Каждая операция имеет сигнатуру, содержащую имя операции, тип возвращаемого значения и список параметров, который может быть пустым. Реализация операции в виде процедуры – это метод, принадлежащий классу.

Для операций, как и для атрибутов класса, определено понятие «видимость». Закрытые операции являются внутренними для объектов класса и недоступны из других объектов. Остальные образуют интерфейсную часть класса и являются средством интеграции класса в ПС.

Отношения

На диаграммах классов обычно показываются ассоциациии обобщения. Каждая ассоциация несет информацию о связях между объектами внутри ПС. Наиболее часто используются бинарные ассоциации, связывающие два класса. Ассоциация может иметь название, которое должно выражать суть отображаемой связи (Рисунок 42). Помимо названия, ассоциация может иметь такую характеристику, как множественность. Она показывает, сколько объектов каждого класса может участвовать в ассоциации. Множественность указывается у каждого конца ассоциации (полюса) и задается конкретным числом или диапазоном чисел. Множественность, указанная в виде звездочки, предполагает любое количество (в том числе, и ноль). Например, на Рисунок42 ассоциация связывает один объект класса «Набор товаров» с одним или более объектами класса «товар». Связаны между собой могут быть и объекты одного класса, поэтому ассоциация может связывать класс с самим собой. Например, для класса «Житель города» можно ввести ассоциацию «Соседство», которая позволит находить всех соседей конкретного жителя.

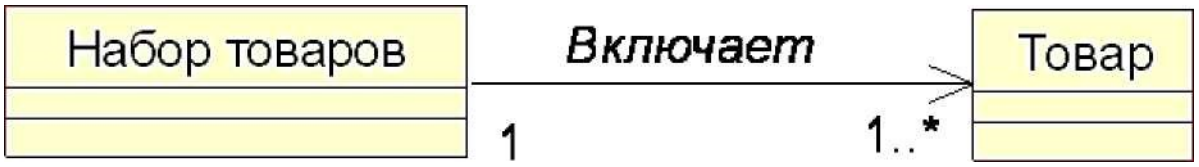


Рисунок 40. Применение ассоциаций

Ассоциация «включает» показывает, что набор может включать несколько различных товаров. В данном случае направленная ассоциация позволяет найти все виды товаров, входящие в набор, но не дает ответа на вопрос, входит ли товар данного вида в какой-либо набор.

Ассоциация сама может обладать свойствами класса, то есть, иметь атрибуты и операции. В этом случае она называется класс- ассоциацией и может рассматриваться как класс, у которого помимо явно указанных атрибутов и операций есть ссылки на оба связываемых ею класса. В примере на Рисунок 44 ассоциация «включает» по существу есть класс-ассоциация, у которой есть атрибут «Количество»,

показывающий, сколько единиц каждого товара входит в набор (Рисунок 42).

Обобщение на диаграммах классов используется, чтобы показать связь между классом-родителем и классом-потомком. Оно вводится на диаграмму, когда возникает разновидность какого-либо класса (например, при развитии ПС – см. Рисунок 42), а также в тех случаях, когда в системе обнаруживаются несколько классов, обладающих сходным поведением (в этом случае общие элементы поведения выносятся на более высокий уровень, образуя класс-родитель – Рисунок 41).

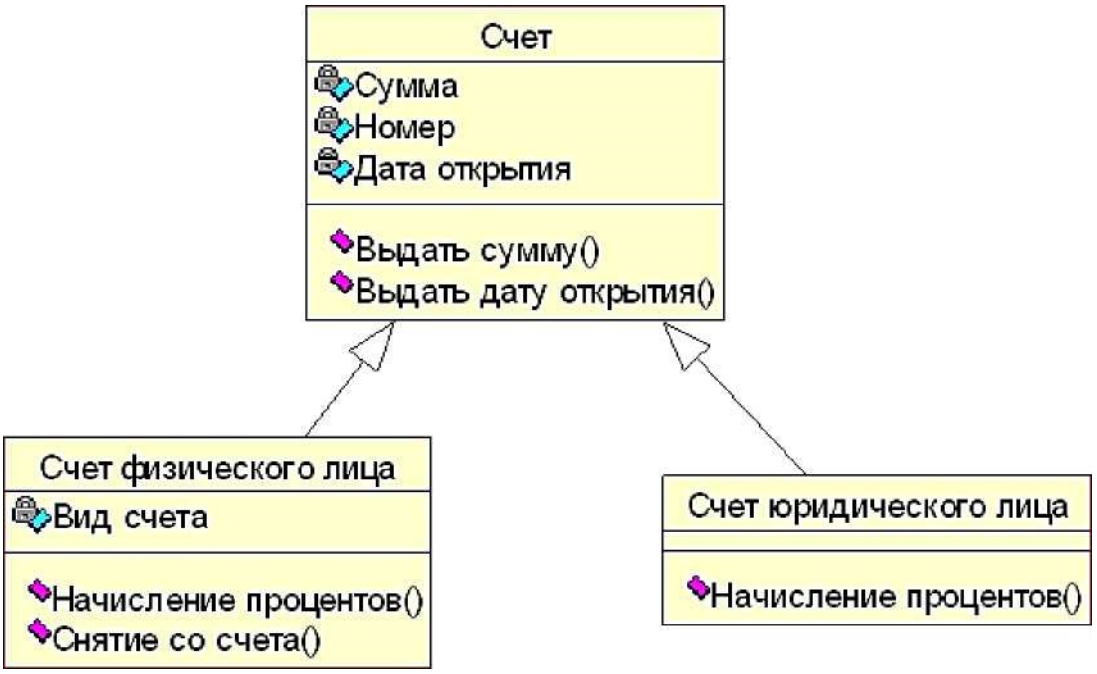


Рисунок 41. Наследуются атрибуты и операции

Как уже говорилось ранее, UML позволяет строить модели с различным уровнем детализации. На Рисунок 40 показана детализация модели, представленной на Рисунок 42.

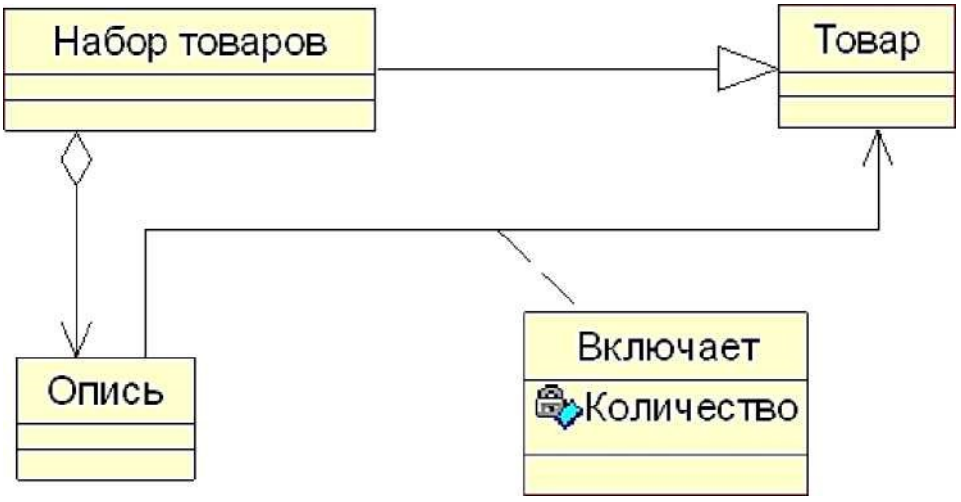


Рисунок 42. Детализация модели набора товаров

Обобщение показывает, что набор товаров – это тоже товар, который может быть

предметом заказа, продажи, поставки и т.д. Набор включает опись, в которой указывается, какие товары входят в набор, а класс-ассоциация «включает» определяет количество каждого вида товаров в наборе.

Стереотипы классов

При создании диаграмм классов часто пользуются понятием «стереотип». В дальнейшем речь пойдет о стереотипах классов. Стереотип класса – это элемент расширения словаря UML, который обозначает отличительные особенности в использовании класса. Стереотип имеет название, которое задается в виде текстовой строки. При изображении класса на диаграмме стереотип показывается в верхней части класса в двойных угловых скобках. Есть четыре стандартных стереотипа классов, для которых предусмотрены специальные графические изображения (Рисунок 43).

Стереотип используется для обозначения классов-сущностей (классов данных), стереотип описывает пограничные классы, которые являются посредниками между ПС и внешними по отношению к ней сущностями – актерами, обозначаемыми стереотипом <>. Наконец, стереотип описывает классы и объекты, которые управляют взаимодействиями. Применение стереотипов позволяет, в частности, изменить вид диаграмм классов.

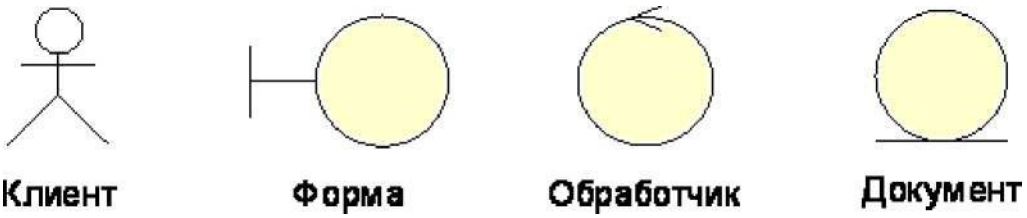


Рисунок 43.Стереотипы для классов

Применение диаграмм классов

Диаграммы классов создаются при логическом моделировании ПС и служат для следующих целей:

- Для моделирования данных. Анализ предметной области позволяет выявить основные характерные для нее сущности и связи между ними. Это удобно моделируется с помощью диаграмм классов. Эти диаграммы являются основой для построения концептуальной схемы базы данных.
- Для представления архитектуры ПС. Можно выделить архитектурно значимые классы и показать их на диаграммах, описывающих архитектуру ПС.
- Для моделирования навигации экранов. На таких диаграммах показываются

пограничные классы и их логическая взаимосвязь. Информационные поля моделируются как атрибуты классов, а управляющие кнопки – как операции и отношения.

- Для моделирования логики программных компонент.
- Для моделирования логики обработки данных.

Задание

Заданием работы является построение диаграммы деятельности, диаграммы состояний и диаграммы классов для предметной области из Приложения 1.

Контрольные вопросы

1. Что такое диаграмма деятельности?
2. Какие данные отражаются на диаграмме состояний?
3. По каким принципам строится диаграмма классов?

Лабораторная работа №3. Построение диаграммы компонентов

Целью работы является изучение порядка построения диаграммы компонентов.

Для выполнения практической работы № 7 студент должен изучить приведенный ниже теоретический материал. Отчет сдается в распечатанном и электронном (файл Word) видах.

Диаграмма компонентов и особенности ее построения

Все рассмотренные ранее диаграммы отражали концептуальные и логические аспекты построения модели системы. Особенность логического представления заключается в том, что оно оперирует понятиями, которые не имеют материального воплощения. Другими словами, различные элементы логического представления, такие как классы, ассоциации, состояния, сообщения, не существуют материально или физически. Они лишь отражают понимание статической структуры той или иной системы или динамические аспекты ее поведения.

Для создания конкретной физической системы необходимо реализовать все элементы логического представления в конкретные материальные сущности. Для описания таких реальных сущностей предназначен другой аспект модельного представления, а именно – физическое представление модели. В контексте языка UML это означает совокупность связанных физических сущностей, включая программное и аппаратное обеспечение, а также персонал, которые организованы для выполнения специальных задач.

Физическая система (physical system) – реально существующий прототип модели системы.

С тем чтобы пояснить отличие логического и физического представлений, необходимо в общих чертах рассмотреть процесс разработки программной системы. Ее исходным логическим представлением могут служить структурные схемы алгоритмов и процедур, описания интерфейсов и концептуальные схемы баз данных. Однако для реализации этой системы необходимо разработать исходный текст программы на языке

программирования. При этом уже в тексте программы предполагается организация программного кода, определяемая синтаксисом языка программирования и предполагающая разбиение исходного кода на отдельные модули.

Однако исходные тексты программы еще не являются окончательной реализацией проекта, хотя и служат фрагментом его физического представления. Программная система может считаться реализованной в том случае, когда она будет способна выполнять функции своего целевого предназначения. А это возможно, только если программный код системы будет реализован в форме исполняемых модулей, библиотек классов и процедур, стандартных графических интерфейсов, файлов баз данных. Именно эти компоненты являются базовыми элементами физического представления системы в нотации языка UML.

Полный проект программной системы представляет собой совокупность моделей логического и физического представлений, которые должны быть согласованы между собой. В языке UML для физического представления моделей систем используются так называемые диаграммы реализации, которые включают в себя две отдельные канонические диаграммы: диаграмму компонентов и диаграмму развертывания.

Диаграмма компонентов, в отличие от ранее рассмотренных диаграмм, описывает особенности физического представления системы. Диаграмма компонентов позволяет определить архитектуру разрабатываемой системы, установив зависимости между программными компонентами, в роли которых может выступать исходный, бинарный и исполняемый код. Во многих средах разработки модуль или компонент соответствует файлу. Пунктирные стрелки, соединяющие модули, показывают отношения взаимозависимости, аналогичные тем, которые имеют место при компиляции исходных текстов программ. Основными графическими элементами

диаграммы компонентов являются компоненты, интерфейсы и зависимости между ними.

В разработке диаграмм компонентов участвуют как системные аналитики и архитекторы, так и программисты. Диаграмма компонентов обеспечивает согласованный переход от логического представления к конкретной реализации проекта в форме программного кода. Одни компоненты могут

существовать только на этапе компиляции программного кода, другие – на этапе его исполнения. Диаграмма компонентов отражает общие зависимости между компонентами, рассматривая последние в качестве отношений между ними.

Компоненты

Для представления физических сущностей в языке UML применяется специальный термин – компонент.

Компонент (component) – физически существующая часть системы, которая обеспечивает реализацию классов и отношений, а также функционального поведения моделируемой программной системы.

Компонент предназначен для представления физической организации ассоциированных с ним элементов модели. Дополнительно компонент может иметь текстовый стереотип и помеченные значения, а некоторые компоненты – собственное графическое представление. Компонентом может быть исполняемый код отдельного модуля, командные файлы или файлы, содержащие интерпретируемые скрипты.

Компонент служит для общего обозначения элементов физического представления модели и может реализовывать некоторый набор интерфейсов. Для графического представления компонента используется специальный символ – прямоугольник со вставленными слева двумя более мелкими прямоугольниками (Рисунок46). Внутри объемлющего прямоугольника записывается имя компонента и, возможно, дополнительная информация. Этот символ является базовым обозначением компонента в языке UML.

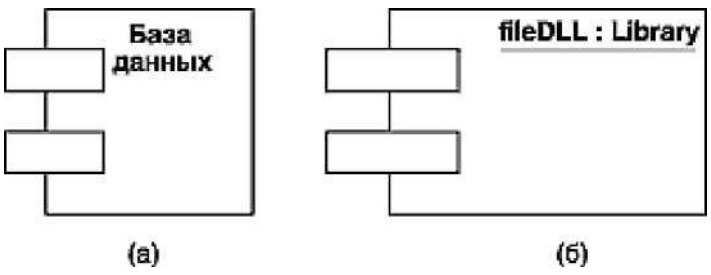


Рисунок 44. Графическое изображение компонента

Графическое изображение компонента ведет свое происхождение от обозначения модуля программы, применявшегося некоторое время для отображения особенностей

инкапсуляции данных и процедур.

Модуль (module) – часть программной системы, требующая памяти для своего хранения и процессора для исполнения.

В этом случае верхний маленький прямоугольник концептуально ассоциировался с данными, которые реализует этот компонент (иногда он изображается в форме овала). Нижний маленький прямоугольник ассоциировался с операциями или методами, реализуемыми компонентом. В простых случаях имена данных и методов записывались явно в маленьких прямоугольниках, однако в языке UML они не указываются.

Имя компонента подчиняется общим правилам именования элементов модели в языке UML и может состоять из любого числа букв, цифр и знаков препинания. Отдельный компонент может быть представлен на уровне типа или экземпляра. И хотя его графическое изображение в обоих случаях одинаково, правила записи имени компонента несколько отличаются.

Если компонент представляется на уровне типа, то записывается только имя типа с заглавной буквы в форме: <Имя типа>. Если же компонент представляется на уровне экземпляра, то его имя записывается в форме: <имя компонента : Имя типа>. При этом вся строка имени подчеркивается. Так, в первом случае (Рисунок 44,а) для компонента уровня типов указывается имя типа, а во втором (Рисунок 44, б) для компонента уровня экземпляра – собственное имя компонента и имя типа.

Правила именования объектов в языке UML требуют подчеркивания имени отдельных экземпляров, но применительно к компонентам подчеркивание их имени часто опускают. В этом случае запись имени компонента со строчной буквы характеризует компонент уровня примеров.

В качестве собственных имен компонентов принято использовать имена исполняемых файлов, динамических библиотек, Web-страниц, текстовых файлов или файлов справки, файлов баз данных или файлов с исходными текстами программ, файлов скриптов и другие.

В отдельных случаях к простому имени компонента может быть добавлена информация об имени объемлющего пакета и о конкретной версии реализации

данного компонента. Необходимо заметить, что в этом случае номер версии записывается как

помеченное значение в фигурных скобках. В других случаях символ компонента может быть разделен на секции, чтобы явно указать имена реализованных в нем классов или интерфейсов. Такое обозначение компонента называется **расширенным**.

Поскольку компонент как элемент модели может иметь различную физическую реализацию, иногда его изображают в форме специального графического символа, иллюстрирующего конкретные особенности реализации. Строго говоря, эти дополнительные обозначения не специфицированы в нотации языка UML. Однако, удовлетворяя общим механизмам расширения языка UML, они упрощают понимание диаграммы компонентов, существенно повышая наглядность графического представления.

Для более наглядного изображения компонентов были предложены и стали общепринятыми следующие графические стереотипы:

- Во-первых, стереотипы для компонентов развертывания, которые обеспечивают непосредственное выполнение системой своих функций. Такими компонентами могут быть динамически подключаемые библиотеки (Рисунок 45, а), Web-страницы на языке разметки гипертекста (Рисунок 45, б) и файлы справки (Рисунок 45, в).
- Во-вторых, стереотипы для компонентов в форме рабочих продуктов. Как правило – это файлы с исходными текстами программ (Рисунок 45, г).

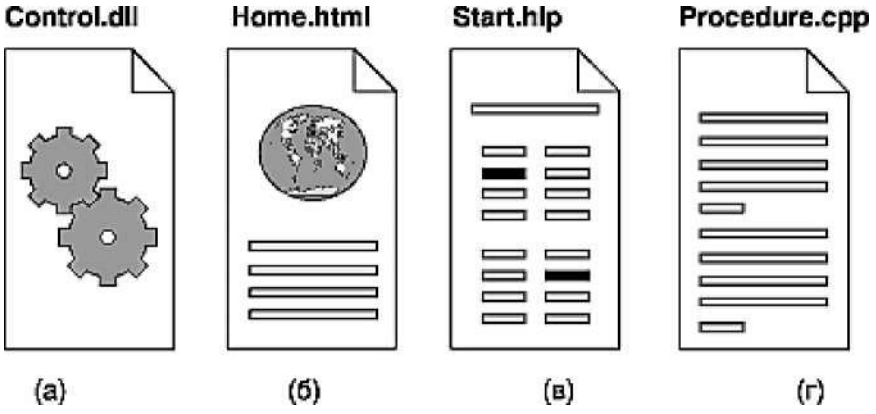


Рисунок 45. Варианты графического изображения компонентов на диаграмме компонентов

Эти элементы иногда называют **артефактами**, подчеркивая при этом их законченное информационное содержание,

зависящее от конкретной технологии реализации соответствующих компонентов.

Более того, разработчики могут для этой цели использовать самостоятельные обозначения, поскольку в языке UML нет строгой нотации для графического представления артефактов.

Другой способ спецификации различных видов компонентов – указание текстового стереотипа компонента перед его именем. В языке UML для компонентов определены следующие стереотипы:

- <<file>> (файл) – определяет наиболее общую разновидность компонента, который представляется в виде произвольного физического файла.
- <<executable>> (исполнимый) – определяет разновидность компонента-файла, который является исполнимым файлом и может выполняться на компьютерной платформе.
- <<document>> (документ) – определяет разновидность компонента-файла, который представляется в форме документа произвольного содержания, не являющегося исполнимым файлом или файлом с исходным текстом программы.
- <<library>> (библиотека) – определяет разновидность компонента-файла, который представляется в форме динамической или статической библиотеки.
- <<source>> (источник) – определяет разновидность компонента-файла, представляющего собой файл с исходным текстом программы, который после компиляции может быть преобразован в исполнимый файл.
- <<table>> (таблица) – определяет разновидность компонента, который представляется в форме таблицы базы данных.

Отдельными разработчиками предлагались собственные графические стереотипы для изображения тех или иных типов компонентов, однако, за не большим исключением они не нашли широкого применения. В свою очередь ряд инструментальных CASE-средств также содержат дополнительный набор графических стереотипов для обозначения компонентов.

Интерфейсы

Следующим графическим элементом диаграммы компонентов являются интерфейсы. В общем случае интерфейс

графически изображается окружностью, которая соединяется с компонентом отрезком линии без стрелок (Рисунок 46, а). При этом имя интерфейса, которое рекомендуется начинать с заглавной буквы «I», записывается рядом с окружностью.

Семантически линия означает реализацию интерфейса, а наличие интерфейсового компонента означает, что данный компонент реализует соответствующий набор интерфейсов.

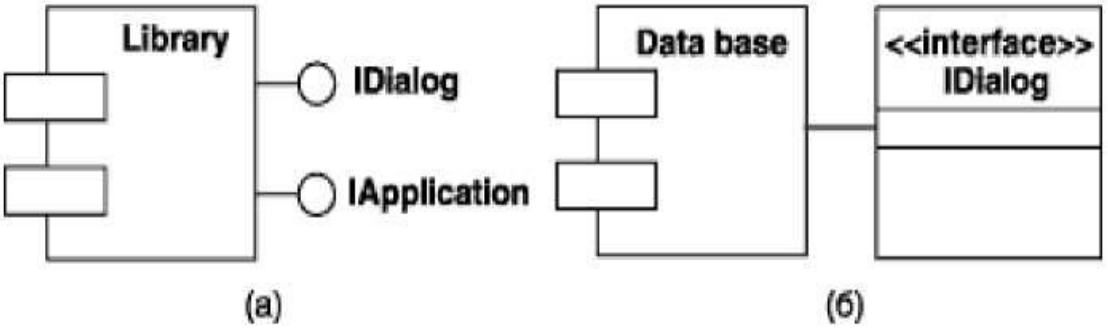


Рисунок 46. Графическое изображение интерфейсов на диаграмме компонентов

Кроме того, интерфейс на диаграмме компонентов может быть изображен в виде прямоугольника, класса со стереотипом <<interface>> и секцией поддерживаемых операций (Рисунок 46, б). Как правило, этот вариант обозначения используется для представления внутренней структуры интерфейса.

При разработке программных систем интерфейсы обеспечивают не только совместимость различных версий, но и возможность вносить существенные изменения в одни части программы, не изменяя другие. Характер применения интерфейсов отдельными компонентами может отличаться.

Различают два способа связи интерфейса и компонента. Если компонент реализует некоторый интерфейс, то такой интерфейс называют экспортируемым или **поддерживаемым**, поскольку этот компонент предоставляет его в качестве сервиса другим компонентам. Если же компонент использует некоторый интерфейс, который реализуется другим компонентом, то такой интерфейс для первого компонента называется **импортируемым**. Особенность импортируемого интерфейса состоит в том, что на диаграмме компонентов это отношение изображается с помощью зависимости.

Зависимости между компонентами

В общем случае отношение зависимости также было рассмотрено ранее. Отношение зависимости служит для

представления факта наличия специальной формы связи между двумя элементами модели, когда изменение одного элемента модели оказывает влияние или приводит к изменению другого элемента модели. Отношение зависимости на диаграмме компонентов изображается пунктирной линией со стрелкой, направленной

от клиента или зависимого элемента к источнику или независимому элементу модели.

Зависимости могут отражать связи отдельных файлов программной системы на этапе компиляции и генерации объектного кода. В других случаях зависимость может указывать на наличие в независимом компоненте описаний классов, которые используются в зависимом компоненте для создания соответствующих объектов. Применительно к диаграмме компонентов зависимости могут связывать компоненты и импортируемые этим компонентом интерфейсы, а также различные виды компонентов между собой.

В этом случае рисуют стрелку от компонента-клиента к импортируемому интерфейсу (Рисунок 49). Наличие такой стрелки означает, что компонент не реализует соответствующий интерфейс, а использует его в процессе своего выполнения. При этом на этой же диаграмме может присутствовать и другой компонент, который реализует этот интерфейс. Отношение реализации интерфейса обозначается на диаграмме компонентов обычной линией без стрелки.

Так, например, изображенный ниже фрагмент диаграммы компонентов представляет информацию о том, что компонент с именем Control зависит от импортируемого интерфейса IDialog, который, в свою очередь, реализуется компонентом с именем DataBase. При этом для второго компонента этот интерфейс является экспортируемым. Изобразить связь второго компонента DataBase с этим интерфейсом в форме зависимости нельзя, поскольку этот компонент реализует указанный интерфейс.

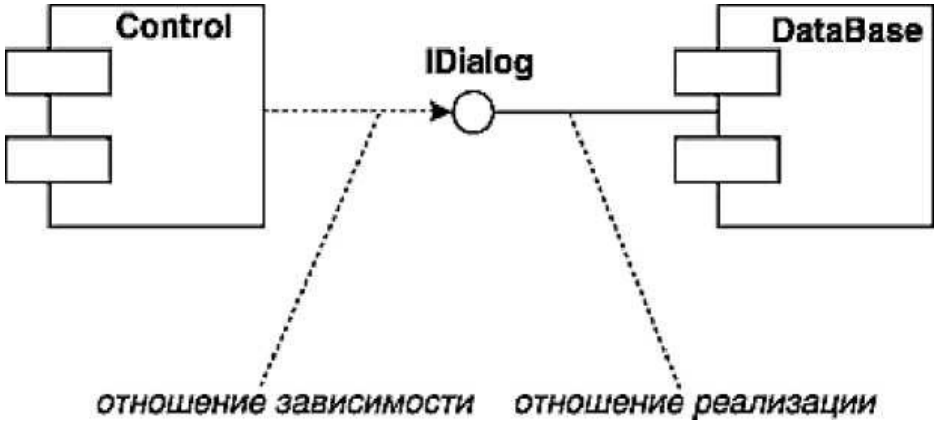


Рисунок 47. Фрагмент диаграммы компонентов с отношениями зависимости реализации

Другим случаем отношения зависимости на диаграмме компонентов является отношение программного вызова и компиляции между различными видами

компонентов. Для рассмотренного фрагмента диаграммы компонентов (Рисунок 50) наличие подобной зависимости означает, что исполнимый компонент Control.exe использует или импортирует некоторую функциональность компонента Library.dll, вызывает страницу гипертекста Home.html и файл помощи Search.hlp, а исходный текст этого исполнимого компонента хранится в файле Control.cpp. При этом характер отдельных видов зависимостей может быть отмечен дополнительно с помощью текстовых стереотипов.

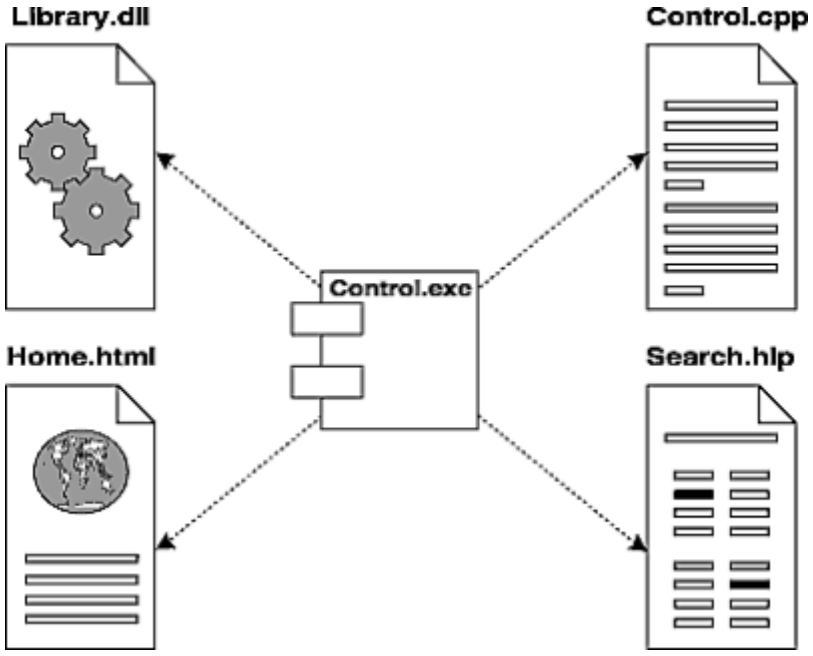


Рисунок 48. Графическое изображение отношения зависимости между компонентами

На диаграмме компонентов могут быть также представлены отношения зависимости между компонентами и реализованными в них классами. Эта информация имеет значение для обеспечения согласования логического и физического представлений модели системы. Разумеется, изменения в структуре описаний классов могут привести к изменению этой зависимости. Ниже приводится фрагмент зависимости подобного рода, когда исполнимый компонент Control.exe зависит от соответствующих классов (Рисунок 49).

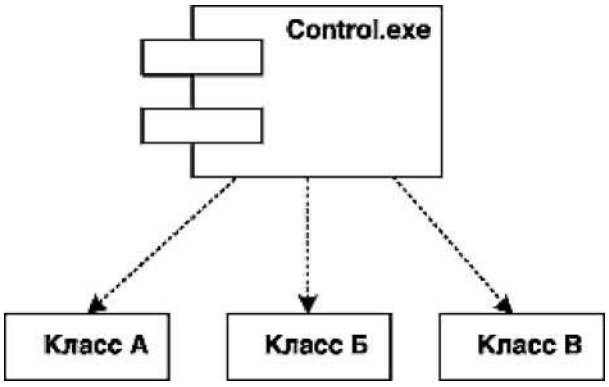


Рисунок 49. Графическое изображение зависимости между компонентом и классами

В этом случае из диаграммы компонентов не следует, что классы реализованы данным компонентом. Если требуется подчеркнуть, что некоторый компонент реализует отдельные классы, то для обозначения компонента используется расширенный символ прямоугольника. При этом прямоугольник компонента делится на две секции горизонтальной линией. Верхняя секция служит для записи имени компонента и, возможно, дополнительной информации, а нижняя секция – для указания реализуемых данным компонентом классов (Рисунок 50).

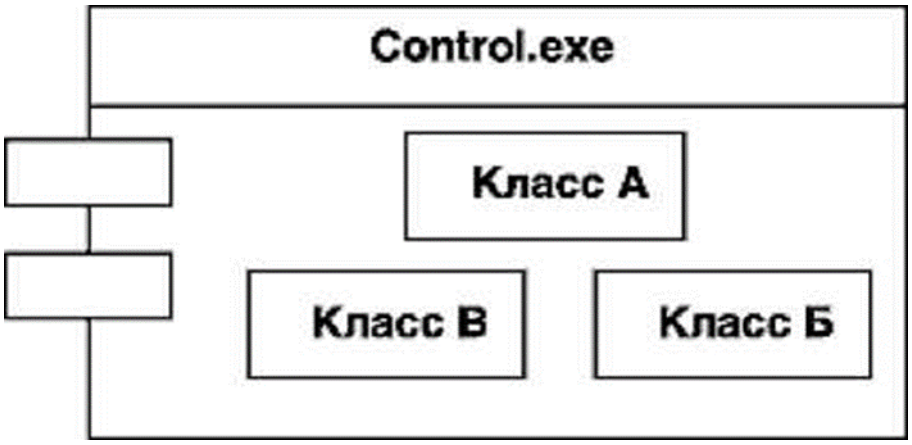


Рисунок 50. Графическое изображение компонента с информацией о реализуемых им классах

В случае если компонент является экземпляром и реализует три отдельных объекта, он изображается в форме компонента уровня экземпляров (Рисунок 51). Объекты, которые находятся в отдельном компоненте-экземпляре, изображаются вложенными в символ данного компонента. Подобная вложенность означает, что выполнение компонента влечет за собой выполнение операций соответствующих объектов. При этом существование компонента в течение времени исполнения программы обеспечивает функциональность всех вложенных в него объектов. Что касается доступа к этим объектам, то он может быть дополнительно специфицирован с помощью видимости, подобно видимости пакетов.

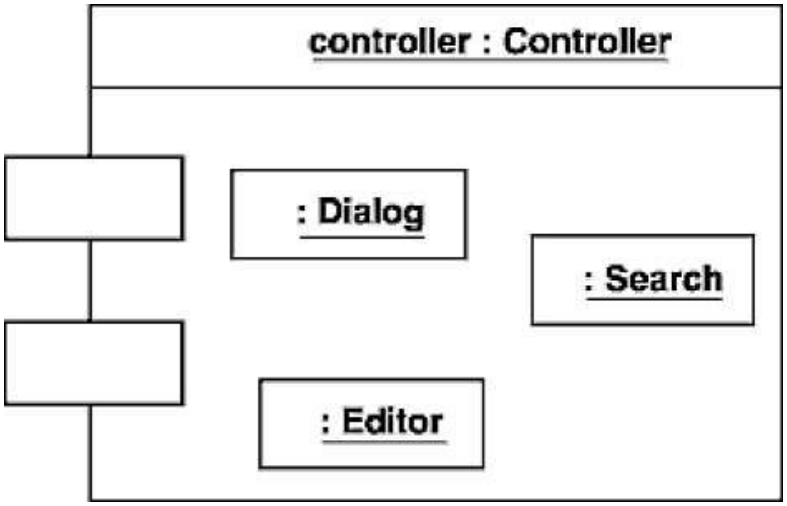


Рисунок 51. Графическое изображение компонента-экземпляра, реализующего отдельные объекты

Для компонентов с исходным текстом программы видимость может означать возможность внесения изменений в соответствующие тексты программ с их последующей перекомпиляцией. Для компонентов с исполняемым кодом программы видимость может характеризовать возможность запуска на исполнение соответствующего компонента или вызова реализованных в нем операций или методов.

Рекомендации по построению диаграммы компонентов

Разработка диаграммы компонентов предполагает использование информации не только о логическом представлении модели системы, но и об особенностях ее физической реализации. В первую очередь, необходимо решить, из каких физических частей или файлов будет состоять программная система. На этом этапе следует обратить внимание

на такую реализацию системы, которая обеспечивала бы возможность повторного использования кода за счет рациональной декомпозиции компонентов, а также создание объектов только при их необходимости.

Общая производительность программной системы существенно зависит от рационального использования вычислительных ресурсов. Для этой цели необходимо большую часть описаний классов, их операций и методов вынести в динамические библиотеки, оставив в исполняемых компонентах только самые необходимые для инициализации программы фрагменты программного кода.

После общей структуризации физического представления системы необходимо дополнить модель интерфейсами и схемами базы данных. При разработке

интерфейсов следует обращать внимание на согласование различных частей программной системы. Включение в модель схемы базы данных предполагает спецификацию отдельных таблиц и установление информационных связей между ними.

Завершающий этап построения диаграммы компонентов связан с установлением и нанесением на диаграмму взаимосвязей между компонентами, а также отношений реализации. Эти отношения должны иллюстрировать все важнейшие аспекты физической реализации системы, начиная с особенностей компиляции исходных текстов программ и заканчивая исполнением отдельных частей программы на этапе ее выполнения. Для этой цели можно использовать различные графические стереотипы компонентов.

При разработке диаграммы компонентов следует придерживаться общих принципов создания моделей на языке UML. В частности, в первую очередь необходимо использовать уже имеющиеся в языке UML и общепринятые графические и текстовые стереотипы. В большинстве типовых проектов этого набора достаточно для представления компонентов и зависимостей между ними.

Если же проект содержит физические элементы, описание которых отсутствует в языке UML, то следует воспользоваться механизмом расширения. В частности, можно применить дополнительные стереотипы для отдельных нетиповых

компонентов или помеченные значения для уточнения отдельных характеристик компонентов.

Задание

Заданием работы является построение диаграммы компонентов для предметной области из Приложения 1.

Контрольные вопросы

1. Какие компоненты изображаются на диаграмме компонентов?
2. Каким символом изображается библиотека?
3. Как изображаются зависимости между компонентами?

Лабораторная работа №4. Построение диаграммы потоков данных

Целью работы является изучение порядка построения диаграммы потоков данных

Для выполнения работы студент должен изучить приведенный ниже теоретический материал. Отчет сдается в распечатанном и электронном (файл Word)

видах.

Диаграмма потоков данных

Диаграммы потоков данных (Data Flow Diagrams – DFD) представляют собой иерархию функциональных процессов, связанных потоками данных. Цель такого представления – продемонстрировать, как каждый процесс преобразует свои входные данные в выходные, а также выявить отношения между этими процессами.

Для построения DFD традиционно используются две различные нотации, соответствующие методам Йордона Де Марко и Гейна Сэрсона. Эти нотации незначительно отличаются друг от друга графическим изображением символов (далее в примерах используется нотация Гейна Сэрсона).

В соответствии с данным методом модель системы определяется как иерархия диаграмм потоков данных, описывающих асинхронный процесс преобразования информации от ее ввода в систему до выдачи потребителю. Источники информации (внешние сущности) порождают информационные потоки (потоки данных), переносящие информацию к подсистемам или процессам. Те, в свою очередь, преобразуют информацию и порождают новые потоки, которые переносят информацию к другим процессам или подсистемам, накопителям данных или внешним сущностям – потребителям информации.

Диаграммы верхних уровней иерархии (контекстные диаграммы) определяют основные процессы или подсистемы с внешними входами и выходами. Они детализируются при помощи диаграмм нижнего уровня. Такая декомпозиция продолжается, создавая многоуровневую иерархию диаграмм, до тех пор, пока не будет достигнут уровень декомпозиции, на котором детализировать процессы далее не имеет смысла.

Состав диаграмм потоков данных

Основными компонентами диаграмм потоков данных являются:

- внешние сущности;
- системы и подсистемы;
- процессы;
- накопители данных;
- потоки данных.

Внешняя сущность представляет собой материальный объект или физическое

лицо, являющиеся источником или приемником информации, например, заказчики, персонал, поставщики, клиенты, склад. Определение некоторого объекта или системы в качестве внешней сущности указывает на то, что она находится за пределами границ анализируемой системы. В процессе анализа некоторые внешние сущности могут быть перенесены внутрь диаграммы анализируемой системы, если это необходимо, или, наоборот, часть процессов может быть вынесена за пределы диаграммы и представлена как внешняя сущность.

Внешняя сущность обозначается квадратом (Рисунок 52), расположенным над диаграммой и бросающим на нее тень для того, чтобы можно было выделить этот символ среди других обозначений.

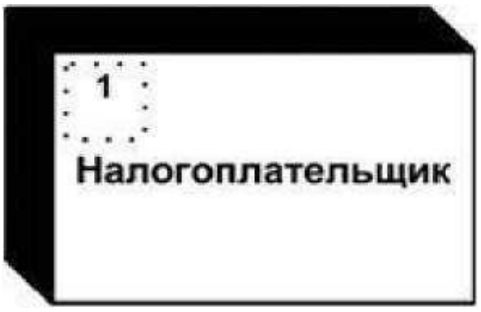


Рисунок 52. Графическое изображение внешней сущности

При построении модели сложной системы она может быть представлена в самом общем виде на так называемой контекстной диаграмме в виде одной системы как единого целого, либо может быть декомпозирована на ряд подсистем.

Подсистема (или система) на контекстной диаграмме изображается так, как она представлена на Рисунок 53.

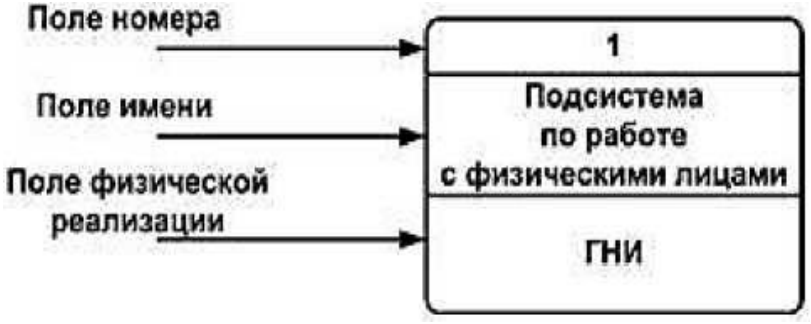


Рисунок53. Подсистема по работе с физическими лицами (ГНИ – Государственная налоговая инспекция)

Номер подсистемы служит для ее идентификации. В поле имени вводится наименование подсистемы в виде предложения с подлежащим и соответствующими определениями и дополнениями.

Процесс представляет собой преобразование входных потоков данных в

выходные в соответствии с определенным алгоритмом. Физически процесс может быть реализован различными способами: это может быть подразделение организации (отдел), выполняющее обработку входных документов и выпуск отчетов, программа, аппаратно реализованное логическое устройство и т. д.

Процесс на диаграмме потоков данных изображается, как показано на Рисунок 54.

Рисунок 54. Графическое изображение процесса

Номер процесса служит для его идентификации. В поле имени вводится наименование процесса в виде предложения с активным недвусмысленным глаголом в неопределенной форме (вычислить, рассчитать, проверить, определить, создать, получить), за которым следуют существительные в винительном падеже, например: «Ввести сведения о налогоплательщиках», «Выдать информацию о текущих расходах», «Проверить поступление денег».

Информация в поле физической реализации показывает, какое подразделение организации, программа или аппаратное устройство выполняет данный процесс.

Накопитель данных – это абстрактное устройство для хранения информации, которую можно в любой момент поместить в накопитель и через некоторое время извлечь, причем способы помещения и извлечения могут быть любыми.

Накопитель данных может быть реализован физически в виде микрофиши, ящика в картотеке, таблицы в оперативной памяти, файла на магнитном носителе и т. д. Накопитель данных на диаграмме потоков данных изображается, как показано на Рисунок 55.

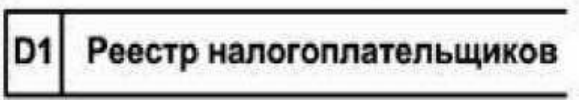


Рисунок 55. Графическое изображение накопителя данных

Накопитель данных идентифицируется буквой «D» и произвольным числом. Имя

накопителя выбирается из соображения наибольшей информативности для проектировщика.

Накопитель данных в общем случае является прообразом будущей базы данных, и описание хранящихся в нем данных должно соответствовать модели данных.

Поток данных определяет информацию, передаваемую через некоторое соединение от источника к приемнику. Реальный поток данных может быть информацией, передаваемой по кабелю между двумя устройствами, пересылаемыми по почте письмами, магнитными лентами или дискетами, переносимыми с одного компьютера на другой и т. д.

Поток данных на диаграмме изображается линией, оканчивающейся стрелкой, которая показывает направление потока (Рисунок 56). Каждый поток данных имеет имя, отражающее его содержание.

Рисунок 56. Поток данных

Построение иерархии диаграмм потоков данных

Главная цель построения иерархии DFD заключается в том, чтобы сделать описание системы ясным и понятным на каждом уровне детализации, а также разбить его на части с точно определенными отношениями между ними. Для достижения этого целесообразно пользоваться следующими рекомендациями:

- Размещать на каждой диаграмме от 3 до 6–7 процессов (аналогично SADT). Верхняя граница соответствует человеческим возможностям одновременного восприятия и понимания структуры сложной системы с множеством внутренних связей, нижняя граница выбрана по соображениям здравого смысла: нет необходимости детализировать процесс диаграммой, содержащей всего один или два процесса.
- Не загромождать диаграммы несущественными на данном уровне деталями.

• Декомпозицию потоков данных осуществлять параллельно с декомпозицией процессов. Эти две работы должны выполняться одновременно, а не одна после завершения другой.

• Выбирать ясные, отражающие суть дела имена процессов и потоков, при этом стараться не использовать аббревиатуры.

Первым шагом при построении иерархии DFD является построение контекстных диаграмм. Обычно при проектировании относительно простых систем строится единственная контекстная диаграмма со звездообразной топологией, в центре которой находится так называемый главный процесс, соединенный с приемниками и источниками информации, посредством которых с системой взаимодействуют пользователи и другие внешние системы. Перед построением контекстной DFD необходимо проанализировать внешние события (внешние сущности), оказывающие влияние на функционирование системы. Количество потоков на контекстной диаграмме должно быть по возможности небольшим, поскольку каждый из них может быть в дальнейшем разбит на несколько потоков на следующих уровнях диаграммы.

Для проверки контекстной диаграммы можно составить список событий. Список событий должен состоять из описаний действий внешних сущностей (событий) и соответствующих реакций системы на события. Каждое событие должно

соответствовать одному или более потокам данных: входные потоки интерпретируются как воздействия, а выходные потоки – как реакции системы на входные потоки.

Для сложных систем (признаками сложности могут быть наличие большого количества внешних сущностей (десять и более), распределенная природа системы или ее многофункциональность) строится иерархия контекстных диаграмм. При этом контекстная диаграмма верхнего уровня содержит не единственный главный процесс, а набор подсистем, соединенных потоками данных. Контекстные диаграммы следующего уровня детализируют контекст и структуру подсистем.

Для каждой подсистемы, присутствующей на контекстных диаграммах, выполняется ее детализация при помощи DFD. Это можно сделать путем построения диаграммы для каждого события. Каждое событие представляется в виде процесса с соответствующими входными и выходными потоками, накопителями данных, внешними сущностями и ссылки на другие процессы для описания связей между этим

процессом и его окружением. Затем все построенные диаграммы сводятся в одну диаграмму нулевого уровня.

Каждый процесс на DFD, в свою очередь, может быть детализирован при помощи DFD или (если процесс элементарный) спецификации. Спецификация процесса должна формулировать его основные функции таким образом, чтобы в дальнейшем специалист, выполняющий реализацию проекта, смог выполнить их или разработать соответствующую программу.

Спецификация является конечной вершиной иерархии DFD. Решение о завершении детализации процесса и использовании спецификации принимается аналитиком исходя из следующих критериев:

- Наличия у процесса относительно небольшого количества входных и выходных потоков данных (2–3 потока);
- возможности описания преобразования данных процессов в виде последовательного алгоритма;
- выполнения процессом единственной логической функции преобразования входной информации в выходную;
- возможности описания логики процесса при помощи спецификации небольшого объема (не более 20–30 строк).

Спецификации представляют собой описания алгоритмов задач, выполняемых процессами. Они содержат номер и/или имя процесса, списки входных и выходных данных и тело (описание) процесса, являющееся спецификацией алгоритма или операции, трансформирующей входные потоки данных в выходные. Языки спецификаций могут варьироваться от структурированного естественного языка или псевдокода до визуальных языков моделирования.

Структурированный естественный язык применяется для понятного, достаточно строгого описания спецификаций процессов. При его использовании приняты следующие соглашения:

- логика процесса выражается в виде комбинации последовательных конструкций, конструкций выбора и итераций;
- глаголы должны быть активными, недвусмысленными и ориентированными на целевое действие (заполнить, вычислить, извлечь, а не модернизировать, обработать);
- логика процесса должна быть выражена четко и недвусмысленно.

При построении иерархии DFD переходить к детализации процессов следует только после определения содержания всех потоков и накопителей данных, которое описывается при помощи структур данных. Для каждого потока данных формируется список всех его элементов данных, затем элементы данных объединяются в структуры данных, соответствующие более крупным объектам данных (например, строкам документов или объектам предметной области). Каждый объект должен состоять из элементов, являющихся его атрибутами. Структуры данных могут содержать альтернативы, условные вхождения и итерации. Условное вхождение означает, что данный компонент может отсутствовать в структуре (например, структура «данные о страховании» для объекта «служащий»). Альтернатива означает, что в структуру может входить один из перечисленных элементов. Итерация означает вхождение любого числа элементов в указанном диапазоне (например, элемент «имя ребенка» для объекта «служащий»). Для каждого элемента данных может указываться его тип (непрерывные или дискретные данные). Для непрерывных данных могут указываться единица измерения, диапазон значений, точность представления и форма физического кодирования. Для дискретных данных может указываться таблица допустимых значений.

После построения законченной модели системы ее необходимо верифицировать (проверить на полноту и согласованность). В полной модели все ее объекты (подсистемы, процессы, потоки данных) должны быть подробно описаны и детализированы. Выявленные не детализированные объекты следует детализировать, вернувшись на предыдущие шаги разработки. В согласованной модели для всех потоков данных и накопителей данных должно выполняться правило сохранения информации: все поступающие куда-либо данные должны быть считаны, а все считываемые данные должны быть записаны.

При моделировании бизнес-процессов диаграммы потоков данных (DFD) используются для построения моделей «AS-IS» и «AS-TO-BE», отражая, таким образом, существующую и предлагаемую структуру бизнес-процессов организации и взаимодействие между ними. При этом описание используемых в

организации данных на концептуальном уровне, независимом от средств реализации базы данных, выполняется с помощью модели «сущность-связь» (Рисунок 57).

Ниже перечислены основные виды и использованием последовательность работ при построении бизнес-моделей с методики Йордона:

1. Описание контекста процессов и построение начальной контекстной диаграммы

Начальная контекстная диаграмма потоков данных должна содержать нулевой процесс с именем, отражающим деятельность организации, внешние сущности, соединенные с нулевым процессом посредством потоков данных. Потоки данных соответствуют документам, запросам или сообщениям, которыми внешние сущности обмениваются с организацией.

2. Спецификация структур данных

Определяется состав потоков данных и готовится исходная информация для построения концептуальной модели данных в виде структур данных. Выделяются все структуры и элементы данных типа «итерация», «условное вхождение» и «альтернатива». Простые структуры и элементы данных объединяются в более крупные структуры. В результате для каждого потока данных должна быть сформирована иерархическая (древовидная) структура, конечные элементы (листья) которой являются элементами данных, узлы дерева являются структурами данных, а верхний узел дерева соответствует потоку данных в целом.

3. Построение начального варианта концептуальной модели данных

Для каждого класса объектов предметной области выделяется сущность. Устанавливаются связи между сущностями и определяются их характеристики. Строится диаграмма «сущность-связь» (без атрибутов сущностей).

4. Построение диаграмм потоков данных нулевого и последующих уровней

Для завершения анализа функционального аспекта деятельности организации детализируется (декомпозируется) начальная контекстная диаграмма. При этом можно построить диаграмму для каждого события, поставив ему в соответствие процесс и описав входные и выходные потоки, накопители данных, внешние сущности и ссылки на другие процессы для описания связей между этим процессом и его окружением. После этого все построенные диаграммы сводятся в одну диаграмму нулевого уровня.

Процессы разделяются на группы, которые имеют много общего (работают с одинаковыми данными и/или имеют сходные функции). Они изображаются вместе на диаграмме более низкого (первого) уровня, а на диаграмме нулевого уровня

объединяются в один процесс. Выделяются накопители данных, используемые процессами из одной группы.

Декомпозируются сложные процессы и проверяется соответствие различных уровней модели процессов.

Накопители данных описываются посредством структур данных, а процессы нижнего уровня – посредством спецификаций.

5. Уточнение концептуальной модели данных

Определяются атрибуты сущностей. Выделяются атрибуты- идентификаторы. Проверяются связи, выделяются (при необходимости) связи «супертип-подтип».

Проверяется соответствие между описанием структур данных и концептуальной моделью (все элементы данных должны присутствовать на диаграмме в качестве атрибутов).

Рисунок 57. Модель «сущность-связь»

Задание

Заданием работы является построение диаграммы потоков данных для предметной области из Приложения 1.

Контрольные вопросы

1. Что такое поток данных?

2. Как изображается объект на DFD?
3. Как изображается накопитель на DFD?

Лабораторная работа №5. Разработка тестового сценария

Целью работы является изучение порядка построения тестовых сценариев.

Для выполнения работы студент должен изучить приведенный ниже теоретический материал. Отчет сдается в распечатанном и электронном (файл Word) видах.

Разработка тестового сценария

Написать тест кейсы для «полного» тестирования продукта – просто невозможно. Мы можем разработать миллионы тестов, но будет ли время у нас их выполнить? Вероятнее всего – нет. Поэтому приходится тщательно выбирать тест-кейсы, которые мы будем проводить.

В последние несколько лет наметилась тенденция, когда усилия фирм-разработчиков программного обеспечения направлены на повышение качества своих программных продуктов. Постепенно производители отказываются от «интуитивного» тестирования программ и переходят к формальному тестированию, с написанием тест-кейсов.

Но, как известно, полностью протестировать программу невозможно по следующим причинам:

- Количество всех возможных комбинаций входных данных слишком велико, чтоб его можно было проверить полностью.
- Пользовательский интерфейс программы (включающий все возможные комбинации действий пользователя и его перемещений по программе) обычно слишком сложен для полного тестирования.

Характеристики хорошего теста:

- Набор тестов не должен быть избыточным.
- Тест должен быть наилучшим в своей категории.
- Тест не должен быть слишком простым или слишком сложным

В настоящее время наблюдается несколько методологий разработки тест-кейсов. Они отличаются и теоретическим подходом, и практической реализацией. Наиболее часто употребляемая методология разработки тестовых случаев – методология, при которой источниками тестовых случаев выступают случаи использования.

Методология разработки тестовых случаев на основе сценариев использования

Случай использования состоит из некоторого множества сценариев: нормальный случай, расширения и исключительные ситуации. Для разработки тест-кейсов на основе одного случая использования разрабатываются несколько сценариев, соответственно: Сценарий использования представляет собой оптимистический сценарий, который выбирается чаще других. В раздел Альтернативные пути могут быть включены несколько сценариев, которые отличаются от сценария использования в различных аспектах, однако остаются полноценными путями исполнения. В раздел Исключительные пути попадают те сценарии, которые приводят к возникновению ошибок. Каждый сценарий предусматривает действия, предпринимаемые действующим субъектом, и требует от системы отклика, который соответствует основной части тестового случая. Тестовый случай состоит из некоторого набора предусловий, стимулирующего воздействия (входные данные) и ожидаемого отклика.

При разработке нужно определить, сколько необходимо использовать тестовых случаев из каждого случая использования, после чего построить эти случаи. Первым шагом на пути определения количества тестовых случаев, приходящихся на один случай использования, является построение профилей использования.

Профиль использования системы – это упорядочивание индивидуальных случаев использования, в основу которого положено некоторое сочетание значений частоты использования и критичности для отдельных случаев использования.

Комбинация рейтингов частоты использования и критичности, применяемая для того, чтобы упорядочить случаи использования, обеспечивает получение определенного критерия качества. Например, можно нарисовать эмблему в правом нижнем углу каждого окна. Это повторяется довольно часто, но, если это не получится, система все еще способна выполнять наиболее важные функции для пользователя. Аналогично, соединение с сервером локальной базы данных происходит крайне редко, однако неудача этой операции сделает невозможным успешное выполнение множества других функций. Количество тестовых случаев, приходящихся на один случай использования, выбирается в зависимости от положения этого случая использования в рейтинговой таблице (чем чаще встречается данный случай использования и чем критичней его неверное выполнение для системы – там больше

тест-кейсов должно быть разработано). На этом этапе тестирования поддерживается проведение тестирования приложения в таком режиме, в каком оно будет использовано на практике.

Построение профилей использования начинается с определения действующих субъектов на диаграмме случаев использования. Там, где имеется один действующий субъект, это значение профиля должно соответствовать значению поля частоты использования в случаях использования. Но также интерес представляют и пользу приносят случаи, в которых участие принимают несколько действующих субъектов.

Очень редко все эти действующие субъекты используют систему одним и тем же способом. Поле частоты случая использования представляет собой композицию значений частоты использования отдельных профилей действующих субъектов.

Этот подход весьма полезен для систем, которые ни разу не устанавливались. Он дает более точную оценку того, как действующий субъект будет использовать систему, по сравнению с простым угадыванием того, каким окажется совокупный результат отдельных случаев использования.

Случай использования обычно содержит многочисленные сценарии, которые могут быть преобразованы в тестовые случаи.

Разработка сценария для случая использования предусматривает выполнение четырех действий:

- Идентификация всех значений, которые вводятся действующими субъектами, содержащимися в модели случая использования.
- Выделение классов эквивалентности значений каждого типа входных данных.
- Построение таблиц, в которые помещен список комбинаций значений из различных классов эквивалентности.
- Построение тестовых случаев, в которых сочетаются одна перестановка значений с необходимыми внешними ограничениями.

Как пример рассматривается некая система управления персоналом. В случаях использования этой системы употребляются три переменных. Каждый служащий представлен в системе именем и переменными, показывающими, является ли он новым сотрудником фирмы или уже работает в ней в течении определенного времени, и уровнем полномочий, санкционированных системой безопасности.

В таблице 3 показаны классы эквивалентности этих трех переменных.

Таблица 3. Классы эквивалентности переменных

Имя переменной	Тип объекта	Классы эквивалентности
Имя	Строка	• Имя, которое выходит за пределы максимальной длины строки • Имя, которое в точности соответствует максимальной длине строки • Полное имя с оставшимся пустым пространством • Пустое имя
Служащий	Штатная единица	• Специально созданная штатная единица • Ранее существовавшая единица
Авторизация	Код безопасности	• Санкционирован только локальный доступ • Санкционирован доступ в масштабах всей системы

В таблице 4 каждой из этих переменных отводится отдельный столбец. В эти столбцы помещены значения из различных классов эквивалентности рассматриваемых переменных. Каждая строка таблицы представляет собой описание конкретного теста.

Количество тестовых случаев, которые необходимо построить, зависит от значения атрибута частоты использования каждого случая использования. Один из способов оценки соответствующего числа тестовых случаев заключается в том, что вычисляется произведение количества различных входов и количества классов эквивалентности каждого типа ввода с целью получения максимального количества перестановок.

Таблица 4

Имя	Штатная единица	Авторизация
Полное имя с остающимся пустым	Ранее существовавшая штатная единица	Санкционирован только локальный доступ
Полное имя с остающимся пустым	Новая штатная единица	Санкционирован только локальный доступ
...	...	...

На практике количество тестовых случаев может быть ограничено, если принимать во внимание важность того или иного случая использования или объем доступных системных ресурсов. Можно предпринять пробную попытку либо

воспользоваться подходящими статистическими данными для определения, какой объем ресурсов необходим для выполнения типичного случая использования. Если известно количество случаев использования, то можно получить оценку трудозатрат, необходимых для реализации проекта в полном объеме.

При тестировании сложных систем одна из наиболее трудных задач заключается в том, чтобы определить результаты, ожидаемые от прогона конкретного теста. Телекоммуникационные системы, программное обеспечение управления космическим кораблем, информационные системы многонациональных корпораций – это случаи систем, для которых построение тестовых данных и тестовых результатов обходится особенно дорого. Некоторые методы разработки тест- кейсов могут оказаться полезными для снижения затрат усилий на разработку и описание ожидаемых результатов. Первый из них предусматривает построение результатов в так называемом инкрементальном режиме. По условиям этого подхода тестовые случаи создаются с целью покрытия некоторого поднабора случаев использования системы, возможно, только некоторых процедур ввода данных. В последующих случаях покрытия расширяются с целью проверки использования системы в полном объеме. С расширением тестовых случаев, тестовые результаты тоже расширяются.

Тестовые случаи расширяются в итеративном режиме. То есть, мы начинаем написание тест-кейсов с описания небольших тестовых случаев, после чего постепенно увеличиваются размеры и сложность тестовых случаев и продолжается этот процесс до тех пор, пока тесты не станут реалистичными с позиций промышленной среды. В системе управления базами данных можно начать с базы данных, содержащей 50 записей, и постепенно увеличивать это число до нескольких тысяч. Результаты, ожидаемые на каждом новом уровне, должны включать любые взаимодействия, которые возникают в силу появления новых случаев использования. Например, присутствие одной записи может препятствовать выбору другой, которая была выбрана в процессе выполнения предыдущего теста.

Второй подход заключается в разработке **тестовых случаев большого цикла** (grand tour test cases), в котором каждый тестовый случай генерирует данные, которые служат входом для следующего тестового случая. По условиям такого подхода каждый тест переносит тестовые данные через весь жизненный цикл. Полученное при этом состояние базы данных используется в качестве входного для следующего теста.

Этот метод особенно эффективен при тестировании жизненного цикла после того, как тестирование нижнего уровня позволило выявить большую часть дефектов, вызывающих отказы. Если выстроить тестовые случаи в соответствующую последовательность, то после успешного выполнения тестового случая 1 устанавливается такое состояние программы, которое ожидается как входное для тестового случая 2. Очевидная проблема в условиях очерченного подхода заключается в том, что неудачное выполнение тестового случая 1 оставляет программу в состоянии, которого мы не ожидали, в результате чего мы не можем выполнять прогон тестового случая 2 или даже вернуть программу в рабочее состояние.

Рекомендуется проводить следующие виды тестирования:

- Тестирование на соответствие функциональным требованиям.
- Проверка качественных системных атрибутов.

Добротная организация разработок программного обеспечения предусматривает методы подтверждения всех

системных «требований», включая и претензии на придание программному продукту особых качеств. Существуют два вида претензий, с которыми может столкнуться программа при разработке продукта. Первый вид претензий представляет интерес только для организаций, занимающейся разработкой программных продуктов. Например, утверждение, что «программный код допускает многократное использование». Второй тип претензий представляет интерес для пользователей системы. Например, утверждение о том, что система является более полной, нежели другие системы подобного класса, предлагаемые на текущий момент на рынке программных продуктов. Вполне понятно, что не все эти претензии могут подвергаться проверке через тестирование. Однако на это следует обратить внимание.

- Тестирование механизма развертывания системы.
- Тестирование после развертывания системы.

Естественное расширение тестирования механизма развертывания системы заключается в добавлении в тестируемый программный продукт функциональных средств самопроверки. Считается, что система «изнашивается» во времени по причине изменений, имеющих место в ее взаимодействии с окружением, примерно так же, как со временем изнашивается механическая система из-за трений между ее компонентами. По мере того, как устанавливаются все более новые версии

стандартных драйверов и библиотек, несоответствия возрастают вместе с ростом вероятности возникновения отказов. Каждая новая версия dll- библиотеки привносит возможность появления новых областей нестыковки стандартных интерфейсов или появления состояния гонок между этой библиотекой и приложением. Функциональные средства самотестирования должны обеспечивать выполнение тестов, которые исследуют работу интерфейсов между этими программными продуктами.

- Тестирование взаимодействий окружения.
- Тестирование системы безопасности.

При разработке тест кейсов на основе случаев использования необходимо обратить внимание на все эти аспекты функционирования программного обеспечения.

Задание

Заданием работы является разработка тестовых сценариев для информационной системы в соответствующей предметной области из Приложения 1.

Контрольные вопросы

1. Что такое тестовый сценарий?
2. Опишите порядок построения тестового сценария.
3. Что используется в качестве основы для разработки тестового сценария?

Лабораторная работа №6. Оценка необходимого количества тестов

Целью работы является изучение порядка оценки необходимого количества тестов.

Для выполнения работы студент должен изучить приведенный ниже теоретический материал. Отчет сдается в распечатанном и электронном (файл Word) видах.

Оценка необходимого количества тестов

Тестовое Покрытие – это одна из метрик оценки качества тестирования, представляющая из себя плотность покрытия тестами требований либо исполняемого кода.

Если рассматривать тестирование как «проверку соответствия между реальным и ожидаемым поведением программы, осуществляемая на конечном наборе тестов, то именно этот конечный набор тестов и будет определять тестовое покрытие:

Чем выше требуемый уровень тестового покрытия, тем больше тестов будет выбрано, для проверки тестируемых требований или исполняемого кода.

Сложность современного программного обеспечения и инфраструктуры сделало невыполнимой задачу проведения тестирования со 100% тестовым покрытием. Поэтому для разработки набора тестов, обеспечивающего более менее высокий уровень покрытия можно использовать специальные инструменты либо техники тест-дизайна.

Существуют следующие подходы к оценке и измерению тестового покрытия:

1. **Покрытие требований (Requirements Coverage)** – оценка покрытия тестами функциональных и нефункциональных требований к продукту путем построения матриц трассировки (traceability matrix).

2. **Покрытие кода (Code Coverage)** – оценка покрытия исполняемого кода тестами, путем отслеживания непроверенных в процессе тестирования частей программного обеспечения.

3. **Тестовое покрытие на базе анализа потока управления**

– оценка покрытия, основанная на определении путей выполнения кода программного модуля и создания выполняемых тест кейсов для покрытия этих путей.

Различия:

Метод покрытия требований сосредоточен на проверке соответствия набора проводимых тестов требованиям к продукту, в то время как анализ покрытия кода – на полноте проверки тестами, разработанной части продукта (исходного кода), а анализ потока управления – на прохождении путей в графе или модели выполнения тестируемых функций (Control Flow Graph).

Ограничения:

Метод оценки покрытия кода не выявит нереализованные требования, так как работает не с конечным продуктом, а с существующим исходным кодом. Метод покрытия требований может оставить непроверенными некоторые участки кода, потому что не учитывает конечную реализацию.

Покрытие требований (Requirements Coverage)

Расчет тестового покрытия относительно требований проводится по формуле:

$$T_{cov} = \left(\frac{L_{cov}}{L_{total}}\right) * 100\% \quad (1)$$

где Tcov – тестовое покрытие; Lcov – количество требований, проверяемых тест кейсами; Ltotal – общее количество требований.

Для измерения покрытия требований, необходимо проанализировать требования к продукту и разбить их на пункты. Опционально каждый пункт связывается с тест кейсами, проверяющими его. Совокупность этих связей и является матрицей трассировки. Проследив связи, можно понять какие именно требования проверяет тестовый случай.

Тесты, не связанные с требованиями, не имеют смысла. Требования, не связанные с тестами – это «белые пятна», т.е. выполнив все созданные тест-кейсы, нельзя дать ответ реализовано данное требование в продукте или нет.

Для оптимизации тестового покрытия при тестировании на основании требований, наилучшим способом будет использование стандартных техник тест-дизайна.

Покрытие кода (CodeCoverage)

Расчет тестового покрытия относительно исполняемого кода программного обеспечения проводится по формуле:

$$T_{cov} = \left(\frac{L_{tc}}{L_{code}}\right) * 100\% \quad (2)$$

где Tcov – тестовое покрытие; Ltc – кол-ва строк кода, покрытых тестами; Lcode – общее кол-во строк кода.

В настоящее время существует инструментарий (например: **Clover**), позволяющий проанализировать в какие строки были вхож- дения во время проведения тестирования, благодаря чему можно значительно увеличить покрытие, добавив новые тесты для конкретных случаев, а также избавиться от дублирующих тестов. Проведение такого анализа кода и последующая оптимизация покрытия достаточно легко реализуется в рамках тестирования белого ящика (white-box testing) примодульном, интеграционном и системном тестировании; при тестировании же черногоящика (black-box testing) задача становится довольно дорогостоящей, так как требует много времени и ре- сурсов на установку, конфигурацию и анализ результатов работы, как со стороны тестировщиков, так и разработчиков.

Тестовое покрытие на базе анализа потока управления

Тестирование потоков управления (Control Flow Testing) – это одна из техник тестирования белого ящика, основанная на определении путей выполнения кода программного модуля и создания выполняемых тест-кейсов для покрытия этих путей.

Фундаментом для тестирования потоков управления является построение графов потоков управления (Control Flow Graph), основными блоками которых являются:

- блок процесса – одна точка входа и одна точка выхода;
- точка альтернативы – одна точка входа, две и более точки выхода;
- точка соединения – две и более точек входа, одна точка выхода.

Для тестирования потоков управления определены разные уровни тестового покрытия (Таблица 5)

Таблица 5

Уровень	Название	Краткое описание
Уровень 0	--	«Тестируй все что протестируешь, пользователи протестируют остальное»
Уровень 1	Покрытие операторов	Каждый оператор должен быть выполнен как минимум один раз.
Уровень 2	Покрытие альтернатив. Покрытие ветвей	Каждый узел с ветвлением (альтернатива) выполнен как минимум один раз.
Уровень 3	Покрытие условий	Каждое условие, имеющее TRUE и FALSE на выходе, выполнено как минимум один раз.
Уровень 4	Покрытие условий альтернатив	Тестовые случаи создаются для каждого условия и альтернативы
Уровень 5	Покрытие множественных условий	Достигается покрытие альтернатив, условий и условий альтернатив (Уровни 2, 3 и 4)
Уровень 6	Покрытие бесконечного числа путей	Если, в случае зацикливания, количество путей становится бесконечным, допускается существенное их сокращение, ограничивая количество циклов выполнения, для уменьшения числа тестовых случаев.
Уровень 7	Покрытие путей	Все пути должны быть проверены

Основываясь на данных этой таблицы, вы сможете спланировать необходимый уровень тестового покрытия, а также оценить уже имеющийся.

Задание

Заданием работы является оценка необходимого количества тестов для информационной системы в соответствующей предметной области из Приложения 1.

Контрольные вопросы

1. Что такое покрытие кода тестами?
2. Как правильно оценивать покрытие кода тестами?
3. Что используется для определения покрытия тестами?

Лабораторная работа №6. Разработка тестовых пакетов

Целью работы является изучение порядка разработки тестовых пакетов.

Для выполнения работы студент должен изучить приведенный ниже теоретический материал. Отчет сдается в распечатанном и электронном (файл Word) видах.

Разработка тестовых пакетов

1. Создайте проект для тестирования.

Для этого выполните следующие шаги:

1. Запустите Visual Studio.
2. В меню **Файл** выберите пункт **Создать-Проект**. Откроется диалоговое окно **Новый проект**.
3. В области Установленные шаблоны выберите шаблон Visual C#.
4. В списке типов приложения выберите пункт **Библиотека классов**.
5. В поле **Имя** введите **Bank** и нажмите **OK**.

Будет создан новый проект Bank. Этот проект отобразится в **обозревателе решений**, а его файл Class1.cs откроется в редакторе кода.

Примечание

Если файл Class1.cs не откроется в редакторе кода, дважды щелкните Class1.cs в **обозревателе решений**, чтобы открыть его.

6. Скопируйте исходный текст из раздела Пример проекта для создания модульных тестов и замените скопированным текстом исходное содержимое файла Class1.cs.
7. Сохранение файла как BankAccount.cs.
8. В меню **Сборка** выберите **Собрать решение**.

Будет создан проект с именем «Bank». Он содержит исходный код, подлежащий тестированию, и средства для его тестирования. Пространство имен BankAccountNS проекта «Bank», содержит открытый класс «BankAccount», методы которого будут

тестироваться в приведенных ниже процедурах.

Здесь проводится тестирование на примере метода Debit. Метод Debit вызывается, когда денежные средства снимаются со

счета. Так выглядит определение метода:

```
//Methodtobe tested.
publicvoidDebit(doubleamount)
{
if(amount>m_balance)
{
thrownewArgumentOutOfRangeException(«amount»);
}
if(amount<0)
{
thrownewArgumentOutOfRangeException(«amount»);
}
m_balance+=amount;
}
```

2. Создание проекта модульного теста

Для этого сделайте следующие действия:

1. В меню **Файл** выберите **Добавить – Создать проект**.
2. В диалоговом окне **Новый проект** разверните узлы **Установленные** и **Visual C#** и выберите **Тест**.
3. В списке шаблонов выберите **Проект модульного теста**.
4. Вполе **Имя** введите BankTests, а затем нажмите кнопку **ОК**. Проект **BankTests** добавляется в решение **Банк**.
5. В проекте **BankTests** добавьте ссылку на проект **Банк**.

В **обозревателе решений** щелкните **Ссылки** в проекте **BankTests**, а затем выберите в контекстном меню **Добавить ссылку**.

6. В диалоговом окне **Диспетчер ссылок** разверните **Решение** и проверьте элемент **Банк**.

3. Создание тестового класса

Создание тестового класса необходимо, чтобы проверить класс BankAccount. Можно использовать UnitTest1.cs, созданный в шаблоне проекта, но лучше дать файлу и классу более описательные имена. Можно сделать это за один шаг, переименовав файл в **обозревателе решений**.

4. Переименование файла класса

В **обозревателе решений** выберите файл UnitTest1.cs в проекте BankTests. В контекстном меню выберите команду **Переименовать**, а затем переименуйте файл в

BankAccountTests.cs. Выберите **Да** в диалоговом окне, предлагающем переименовать все ссылки на элемент кода UnitTest1 в проекте.

Этот шаг изменяет имя класса на BankAccountTests. Файл BankAccountTests.cs теперь содержит следующий код:

```
usingSystem;
usingMicrosoft.VisualStudio.TestTools.UnitTesting;
namespaceBankTests
{
[TestClass]
publicclassBankAccountTests
{
[TestMethod]
publicvoidTestMethod1()
{
}
}
}
```

5. Добавление оператора using в тестируемый проект

Можно также добавить оператор using в класс, чтобы тестируемый проект можно было вызывать без использования полных имен. Вверху файла класса добавьте:

```
usingBankAccountNS;
```

6. Требования к тестовому классу

Минимальные требования к тестовому классу следующие:

- Атрибут [TestClass] является обязательным для платформы модульных

тестов Microsoft для управляемого кода в любом классе, содержащем методы модульных тестов, которые необходимо вы полнить в обозревателе тестов.

- Каждый метод теста, предназначенный для запуска в обозревателе тестов, должен иметь атрибут [TestMethod].

Можно иметь другие классы в проекте модульного теста, которые не содержат атрибута [TestClass] , а также иметь другие методы в тестовых классах, у которых атрибут – [TestMethod]. Можно использовать эти другие классы и методы в методах теста.

7. Создание первого тестового метода

В этой процедуре мы напишем методы модульного теста для проверки поведения метода Debit класса BankAccount. Метод Debit приведен выше.

Существует по крайней мере три поведения, которые требуется проверить:

- Метод создает исключение ArgumentOutOfRangeException, если сумма по дебету превышает баланс.
- Метод создает исключение ArgumentOutOfRangeException, если сумма по дебету меньше нуля.
- Если значение дебета допустимо, то метод вычитает сумму дебета из баланса счета.

8. Создание метода теста

Первый тест проверяет, снимается ли со счета нужная сумма при допустимом размере кредита (со значением меньшим, чем баланс счета, и большим, чем ноль). Добавьте следующий метод в этот класс BankAccountTests:

```
[TestMethod]
publicvoidDebit_WithValidAmount_UpdatesBalance()
{
//Arrange
doublebeginningBalance=11.99; double debitAmount = 4.55; double expected = 7.44;
BankAccount account = new BankAccount(«Mr. Bryan Walton», beginning- Balance);
// Act account.Debit(debitAmount);
//Assert
doubleactual=account.Balance;
Assert.AreEqual(expected,actual,0.001,«Accountnotdebitedcorrectly» );
```

}

Метод очень прост: он создает новый объект BankAccount с начальным балансом, а затем снимает допустимое значение. Он использует метод AreEqual, чтобы проверить, что конечный баланс соответствует ожидаемому.

9. Требования к методу теста

Метод теста должен удовлетворять следующим требованиям:

- Он декорируется атрибутом[TestMethod].
- Он возвращает void.
- Он не должен иметь параметров.

10. Сборка и запуск теста

Для этого сделайте следующие действия:

1. В меню **Построение** выберите **Построить решение**.

Если ошибок нет, появится **обозреватель тестов** с элементом **Debit_WithValidAmount_UpdatesBalance** в группе **Незапускавшиеся тесты**.

2. Выберите **Запустить все**, чтобы выполнить тест. Во время выполнения теста в верхней части окна отображается анимированная строка состояния. По завершении тестового запуска строка состояния становится зеленой, если все методы теста успешно пройдены, или красной, если какие-либо из тестов не пройдены.

3. В данном случае тест пройден не будет. Метод теста будет перемещен в группу **Неудачные тесты**. Выберите этот метод в **обозревателе тестов** для просмотра сведений в нижней части окна.

11. Исправление кода и повторный запуск тестов

12. Анализ результатов теста

Результат теста содержит сообщение, описывающее возникшую ошибку. Для метода AreEqual сообщение отражает ожидаемый результат (параметр **Ожидается<значение>**) и фактически полученный (параметр **Фактическое<значение>**). Ожидалось, что баланс уменьшится, а вместо этого он увеличился на сумму списания.

Модульный тест обнаружил ошибку: сумма списания добавляется на баланс счета, вместо того чтобы вычитаться.

13. Исправление ошибки

Для исправления ошибки замените строку:

m_balance+=amount;

на:

C#Копировать

m_balance-=amount;

14. Повторный запуск теста

В **обозревателе тестов** выберите **Запустить все**, чтобы запустить тест повторно. Красно-зеленая строка состояния станет зеленой, сигнализируя о том, что тест пройден, а сам тест будет перемещен в группу **Пройденные тесты**.

15. Использование модульных тестов для улучшения кода

В этом разделе рассматривается, как последовательный процесс анализа, разработки модульных тестов и рефакторинга может помочь сделать рабочий код более надежным и эффективным.

16. Анализ проблем

Мы создали тестовый метод для подтверждения того, что допустимая сумма правильно вычитается в методе Debit. Теперь проверим, что методсоздает исключение ArgumentOutOfRangeException, если сумма по дебету:

- больше баланса или
- меньше нуля.

17. Создание методов теста

Создадим метод теста для проверки правильного поведения в случае, когда сумма по дебету меньше нуля:

[TestMethod]

[ExpectedException(typeof(ArgumentOutOfRangeException))] public void De-

bit_WhenAmountIsLessThanZero_ShouldThrowArgumentOutOfRange()

{

//Arrange

doublebeginningBalance=11.99; double debitAmount = -100.00;

BankAccountaccount=newBankAccount(«Mr.BryanWalton»,beginning - Balance);

// Act

account.Debit(debitAmount);

}

Мы используем атрибут ExpectedExceptionAttribute для подтверждения правильности созданного исключения. Данный атрибут приводит к тому, что тест не будет пройден, если не возникнет исключения ArgumentOutOfRangeException . Если временно изменить тестируемый метод для вызова более общего исключения ApplicationException при значении суммы по дебету меньше нуля, то тест работает правильно – то есть завершается неудачно.

Чтобы проверить случай, когда размер списания превышает баланс, выполните следующие действия:

1. Создать новый метод теста с именем

Debit_WhenAmountIsMoreThanBalance_ShouldThrowArgumentOut OfRange.

2. Скопировать тело метода из

Debit_WhenAmountIsLessThanZero_ShouldThrowArgumentOutOfRa nge в новый метод.

3. Присвоить debitAmount значение, превышающее баланс.

18. Запуск тестов

Запуск двух методов теста показывает, что тесты работают правильно.

19. Продолжение анализа

Однако последние два тестовых метода вызывают беспокойство. Нельзя быть уверенным, какое именно условие тестируемого метода создает исключение при запуске любого из тестов. Если каким-либо способом разделить эти два условия, а именно отрицательную сумму по дебету и сумму, большую, чем баланс, то это увеличит достоверность проведения тестов.

Еще раз посмотрев на тестируемый метод и заметив, что оба условных оператора используют конструктор ArgumentOutOfRangeException, который просто получает имя аргумента в качестве параметра:

C#Копировать

thrownewArgumentOutOfRangeException(«amount»);

Так выглядит конструктор, который можно использовать для сообщения более детальной информации:

ArgumentOutOfRangeException(String, Object, String) включает имя аргумента, значения аргумента и определяемое пользователем сообщение. Мы можем выполнить

рефакторинг тестируемого метода для использования данного конструктора. Более того, можно использовать открытые для общего доступа члены типа для указания ошибок.

20. Рефакторинг тестируемого кода

Сначала определим две константы для сообщений об ошибках в области видимости класса. Добавьте это в тестируемый класс BankAccount:

publicconststringDebitAmountExceedsBalanceMessage=«Debitamount exceeds balance»;

publicconststringDebitAmountLessThanZeroMessage=«Debitamountis less than zero»;

ЗатемизменимдвавусловныхоператоравметодеDebit:

C#Копировать

```
if(amount>m_balance)
{
thrownewArgumentOutOfRangeException(«amount»,amount,Debit AmountExceedsBalanceMessage);
}
if(amount<0)
{
thrownewArgumentOutOfRangeException(«amount»,amount,Debit AmountLessThanZeroMessage);
}
```

21. Рефакторинг тестовых методов

Удалим атрибут ExpectedException метода теста, и вместо этого будем перехватывать исключение и проверять соответствующее ему сообщение. Метод StringAssert.Contains обеспечивает возможность сравнения двух строк.

В этом случае метод

Debit_WhenAmountIsMoreThanBalance_ShouldThrowArgument OutOfRange может выглядеть следующим образом:

```
[TestMethod] publicvoidDebit_WhenAmountIsMoreThanBalance_ShouldThrowArgumentOutOfRange()
{
//Arrange
```

```
doublebeginningBalance=11.99; double debitAmount = 20.0;
BankAccountaccount=newBankAccount(«Mr.BryanWalton»,beginning - Balance);
//Act try
{
account.Debit(debitAmount);
}
catch(ArgumentOutOfRangeExceptione)
{
// Assert StringAssert.Contains(e.Message,BankAc-
count.DebitAmountExceedsBalanceMessage);
}
}
```

22. Повторное тестирование, переписывание и анализ

Предположим, что в тестируемом методе есть ошибка, и метод Debit даже не создает исключение ArgumentOutOfRangeException, не говоря уже о выводе правильного сообщения с исключением. В этом случае метод теста не сможет обработать этот случай. Если значение debitAmount допустимо (то есть меньше баланса, но больше нуля), то исключение не перехватывается, а утверждение никогда не сработает. Однако метод теста проходит успешно. Это нехорошо, поскольку метод теста должен был завершиться с ошибкой в том случае, если исключение не создается.

Это является ошибкой в методе теста. Для решения этой проблемы добавим утверждение Fail в конце тестового метода для обработки случая, когда исключение не создается.

Однако повторный запуск теста показывает, что тест теперь оказывается не пройденным при перехватывании верного исключения. Блок catch перехватывает исключение, но метод продолжает выполняться, и в нем происходит сбой на новом утверждении Fail. Чтобы разрешить эту проблему, добавим оператор return после StringAssert в блоке catch. Повторный запуск теста подтверждает, что проблема устранена. Окончательная версия метода Debit_WhenAmountIsMoreThanBalance_ShouldThrowArgumentOut Of Range выглядит следующим образом:

```
[TestMethod]
```

```
public void
Debit_WhenAmountIsMoreThanBalance_ShouldThrowArgumentOutOf Range() {
//Arrange
doublebeginningBalance=11.99; double debitAmount = 20.0;
BankAccountaccount=newBankAccount(«Mr.BryanWalton»,beginning - Balance);
//Act try {
account.Debit(debitAmount);
}
catch(ArgumentOutOfRangeExceptione)
{
// Assert StringAssert.Contains(e.Message,BankAc-
count.DebitAmountExceedsBalanceMessage); return;
}
Assert.Fail(«Theexpectedexceptionwasnotthrown.»);
}
```

Усовершенствования тестового кода привели к созданию более надежных и информативных методов теста. Но что более важно, в результате был также улучшен тестируемый код.

Задание

Заданием работы является разработка тестовых пакетов для информационной системы в соответствующей предметной области из Приложения 1.

Контрольные вопросы

1. Расскажите порядок создания модульного теста в VS.
2. Что такое рефакторинг кода?
3. Как провести рефакторинг, используя модульные тесты?

Лабораторная работа №7. Оценка программных средств с помощью метрик

Целью работы является изучение порядка оценки программных средств с помощью метрик

Для выполнения работы студент должен изучить приведенный ниже теоретический материал. Отчет сдается в распечатанном и электронном (файл Word) видах.

Понятие качества

Сейчас существует несколько определений качества, которые в целом совместимы друг с другом. Приведем наиболее распространенные:

Определение ISO: Качество – это полнота свойств и характеристик продукта, процесса или услуги, которые обеспечивают способность удовлетворять заявленным или подразумеваемым потребностям.

Определение IEEE: Качество программного обеспечения – это степень, в которой оно обладает требуемой комбинацией свойств.

Многомерность качества

Общее качество программной системы включает в себя на верхнем уровне ряд составляющих, которые должны быть приняты во внимание при управлении качеством (Рисунок 58).

Рисунок 58. Многомерность информационной системы качества (ISQ – InformationSystemQuality)

Составляющие качества информационной системы:

- Качество инфраструктуры (infrastructure quality): качество аппаратного и поддерживающего программного обеспечения (например, качество операционных систем, компьютерных сетей и т. п.).
- Качество программного обеспечения (software quality): качество программного обеспечения информационной системы.
- Качество данных (data quality): качество данных, использующихся информационной системой на входе.
- Качество информации (information quality): качество информации, продуцируемое информационной системой.

- Качество организации (administrative quality) - качество менеджмента, включая качество бюджетирования, планирования и календарного контроля.
- Качество сервиса (service quality) - качество обучения, системной поддержки и т. п.

Кроме перечисленных составляющих качества должно быть принято во внимание качество обслуживаемого бизнес процесса.

Анализ всех составляющих качества должен проводится с учетом сфер ответственности заинтересованных сторон, как внутренних участников исполняемого процесса (in-process stakeholder), так и пользователей процесса (end-of-process stakeholders).

Управление качеством будет успешным, если под контролем находятся все измерения качества.

Дерево характеристик качества

Исследования метрического анализа качества показывают, что не существует единственной метрики, которая бы дала универсальный рейтинг качества программного обеспечения. Измерения качества дают спектр проектно-зависимых метрик, которые являются руководящей основой для принятия решений в процессе разработки, заказа и сопровождения программного обеспечения.

Следует отметить, что метрики качества являются изначально неочевидной категорией. Исторически сначала были

выделены ряд универсальных и неполных метрик на основе следующих шагов:

- Определение множества характеристик, которые, являясь важными для программного обеспечения, допускают несложное измерение и не перекрываются.
- Выделение кандидатов в метрики, которые измеряют степень удовлетворения указанным характеристикам.
- Исследование характеристик и связанных метрик, для определения корреляции, значимости, степени автоматизируемости.
- Исследование корреляции между метриками, степени перекрытия, зависимости и недостатков.
- Рафинирование множества метрик в целом во множество метрик, которые в совокупности адекватно отражают качество программного обеспечения.
- Корректировка каждой метрики в итоговом множестве в контексте

зафиксированных множеств характеристик и метрик.

На основе систематического применения данного подхода были выведены примеры универсальных характеристик программного обеспечения, структурно связанные в иерархию «Дерево характеристик качества» (Рисунок 59).

Рисунок 59. Дерево характеристик качества

Нижний слой характеристик в иерархии должен быть строго дифференцирован для того, чтобы исключить (или минимизировать) перекрытия. Данный слой должен состоять из примитивных характеристик, допускающих измерение.

Измерение характеристик нижнего слоя может происходить путем ручного сбора информации, специальными автоматизированными средствами, возможен экспертный способ. Каждая из собранных метрик будет иметь собственные характеристики. Область применения метрик может локализоваться внутри проекта, внутри платформы разработки или быть универсальной. Степень влияния метрик на итоговое качество также является различным. Указанные свойства метрик должны быть документированы и доступны при их практическом использовании.

Шкала измерения характеристик (ISO 12207) – введение в метрики

Для каждой характеристики качества рекомендуется формировать меры и шкалу измерений с выделением требуемых, допустимых и неудовлетворительных значений. Реализация процессов оценки должна коррелировать с этапами жизненного цикла конкретного проекта программного средства в соответствии с применяемой, адаптированной версией стандарта ISO 12207.

Функциональная пригодность – наиболее неопределенная и объективно трудно оцениваемая субхарактеристика программного средства. Области применения, номенклатура и функции комплексов программ охватывают столь разнообразные сферы деятельности человека, что невозможно выделить и унифицировать небольшое число атрибутов для оценки и сравнения этой субхарактеристики в различных комплексах программ.

Оценка корректности программных средств состоит в формальном определении степени соответствия комплекса реализованных программ исходным требованиям контракта, технического задания и спецификаций на программное средство и его компоненты. Путем верификации должно быть определено соответствие исходным требованиям всей совокупности к компонентов комплекса программ, вплоть до модулей и текстов

программ и описаний данных.

Оценка способности к взаимодействию состоит в определении качества совместной работы компонентов программных средств и баз данных с другими прикладными системами и компонентами на различных вычислительных платформах, а также взаимодействия с пользователями в стиле, удобном для перехода от одной вычислительной системы к другой с подобными функциями.

Оценка защищенности программных средств включает определение полноты использования доступных методов и средств защиты программного средства от потенциальных угроз и достигнутой при этом безопасности функционирования информационной системы. Наиболее широко и детально методологические и системные задачи оценки комплексной защиты информационных систем изложены в трех частях стандарта ISO 15408:1999-1-3 «Методы и средства обеспечения безопасности. Критерии оценки безопасности информационных технологий».

Оценка надежности – измерение количественных метрик атрибутов субхарактеристик в использовании: завершенности, устойчивости к дефектам, восстанавливаемости и доступности/готовности.

Потребность в ресурсах памяти и производительности компьютера в процессе решения задач значительно изменяется в зависимости от состава и объема исходных данных. Для корректного определения предельной пропускной способности информационной системы с данным программным средством нужно измерить

экстремальные и средние значения длительностей исполнения функциональных групп программ и маршруты, на которых они достигаются. Если предварительно в процессе проектирования производительность компьютера не оценивалась, то, скорее всего, понадобится большая доработка или даже замена компьютера на более быстродействующий.

Оценка практичности программных средств проводится экспертами и включает определение понятности, простоты использования, изучаемости и привлекательности программного средства. В основном это качественная (и субъективная) оценка в баллах, однако некоторые атрибуты можно оценить

количественно по трудоемкости и длительности выполнения операций при использовании программного средства, а также по объему документации, необходимой для их изучения.

Сопровождаемость можно оценивать полнотой и достоверностью документации о состояниях программного средства и его компонентов, всех предполагаемых и выполненных изменениях, позволяющей установить текущее состояние версий программ в любой момент времени и историю их развития. Она должна определять стратегию, стандарты, процедуры, распределение ресурсов и планы создания, изменения и применения документов на программы и данные.

Оценка мобильности – качественное определение экспертами адаптируемости, простоты установки, совместимости и замещаемости программ, выражаемое в баллах. Количественно эту характеристик у программного средства и совокупность ее атрибутов можно (и целесообразно) оценить в экономических показателях: стоимости, трудоемкости и длительности реализации процедур переноса на иные платформы определенной совокупности программ и данных.

Пример графического изображения качества

Для мониторинга метрик качества и подготовки информации для принятия решений собранные метрики должны представляются в наглядном виде, обеспечивающим полноту информации, что особенно важно при отсутствии консолидированных метрик качества (Рисунок 60).

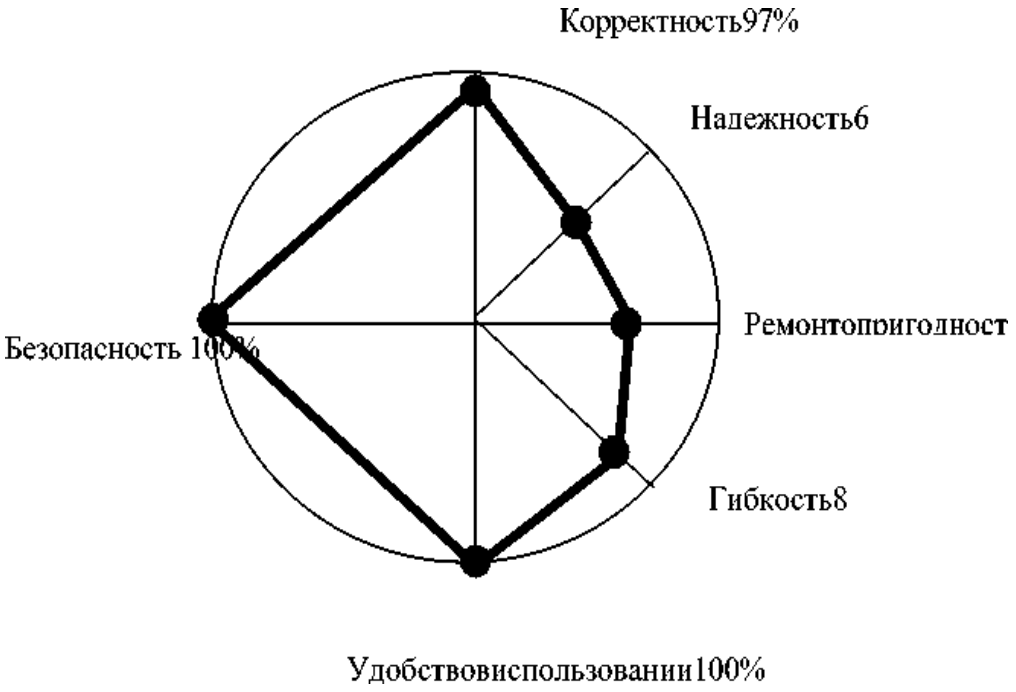


Рисунок 60. Пример графического изображения качества

Для конкретного проекта должно быть разработано или дополнено свое множество метрик, которое отражает назначение и особенности окружения разрабатываемого программного продукта

Задание

Заданием работы является оценка с помощью метрик разработанной информационной системы для соответствующей предметной области из Приложения 1.

Контрольные вопросы

1. Что такое метрика качества кода?
2. Как определить качество кода, используя метрики?
3. Приведите порядок действия для оценки качества кода.

Лабораторная работа №8. Инспекция программного кода на предмет соответствия стандартам кодирования

Целью работы является изучение порядка инспекции программного кода на предмет соответствия стандартам кодирования.

Для выполнения работы студент должен изучить приведенный ниже теоретический материал. Отчет сдается в распечатанном и электронном (файл Word) видах.

Что такое codereview

Code review – инженерная практика в терминах гибкой методологии разработки.

Это анализ (инспекция) кода с целью выявить ошибки, недочеты, расхождения в стиле написания кода, в соответствии написанного кода и поставленной задачи. К очевидным плюсам этой практики можно отнести:

- Улучшается качество кода.
- Находятся «глупые» ошибки(опечатки) в реализации.
- Повышается степень совместного владения кодом.
- Код приводится к единому стилю написания.
- Хорошо подходит для обучения «новичков», быстро набирается навык, происходит выравнивание опыта, обмен знаниями.

Что можно инспектировать

Для ревью подходит любой код. Однако, review обязательно должно проводиться для критических мест в приложении (например: механизмы аутентификации, авторизации, передачи и обработки важной информации – обработка денежных транзакций и пр.).

Также для review подходят и юнит-тесты, так как юнит тесты – это тот же самый код, который подвержен ошибкам, его нужно инспектировать также тщательно как и весь остальной код, потому что, неправильный тест может стоить очень дорого.

Как проводить review

Вообще, ревью кода должен проводиться в совокупности с другими гибкими инженерными практиками: парное программирование, TDD, CI. В этом случае достигается максимальная эффективность ревью. Если используется гибкая методология разработки, то этап code review можно внести в Definition of Done фичи.

Из чего состоит review

Сначала **design review** – анализ будущего дизайна (архитектуры). Данный этап очень важен, так как без него ревью кода будет менее полезным или вообще бесполезным (если программист написал код, но этот код полностью неверен – не решает поставленную задачу, не удовлетворяет требованиям по памяти, времени). Пример: программисту поставили задачу написать алгоритм сортировки массива. Программист реализовал алгоритм bogo-sort, причем с точки зрения качества кода – не придраться (стиль написания, проверка на ошибки), но этот алгоритм совершенно не подходит по времени работы. Поэтому ревью в данном случае бесполезно (конечно –

это утрированный пример, но я думаю, суть ясна), здесь необходимо полностью переписывать алгоритм.

Code review

code review – анализ написанного кода. На данном этапе автору кода отправляются замечания, пожелания по написанному коду.

Также очень важно определиться, за кем будет последнее слово в принятии финального решения в случае возникновения спора. Обычно, приоритет отдается тому кто будет реализовывать код (как в scrum при проведении planning poker), либо специальному человеку, который отвечает за этот код (как в google – code owner).

Как проводить design review

Design review можно проводить за столом, в кругу коллег, у маркерной доски, в корпоративной wiki. На **design review** тот, кто будет писать код, расскажет о выбранной стратегии (примерный алгоритм, требуемые инструменты, библиотеки) решения поставленной задачи. Вся прелесть этого этапазаключается в том, что ошибка проектирования будет стоить 1–2 часа времени (и будет устранена сразу на review).

Как проводить codereview

Code review можно проводить разными способами – дистанционно, когда каждый разработчик сидит за своим

рабочим местом, и совместно – сидя перед монитором одного из коллег, либо в специально выделенным для этого месте, например meeting room. В принципе существует много способов (можно даже распечатать исходный код и вносить изменения на бумаге).

Pre-commit review

Данный вид review проводится перед внесением изменений в VCS. Этот подход позволяет содержать в репозитории только проверенный код. В Microsoft используется этот подход: всем участникам review рассылаются патчи с изменениями. После того как собран и обработан фидбэк, процесс повторяется до тех пор пока все ревьюверы не согласятся с изменениями.

Post-commit review

Данный вид review проводится после внесения изменений в VCS. При этом можно коммитить как в основную ветвь, так и во временную ветку (а в основную ветку вливать уже проверенные изменения).

Тематические review.

Можно также проводить тематические code review – их можно использовать как переходный этап на пути к полноценному code review. Их можно проводить для критического участка кода, либо при поиске ошибок. Самое главное – это определить **цель** данного **review**, при этом цель должна быть обозримой и четкой.

Основное отличие тематических review от полноценного code review– это их узкая специализация. Если в code review мы смотрим на стиль кода, соответствие реализации и постановки задачи, поиск опасного кода, то в тематическом review мы смотрим обычно только один аспект (чаще всего – анализ алгоритма на соответствие ТЗ, обработка ошибок).

Преимущество такого подхода заключается в том, что команда постепенно привыкает к практике review (его можно использовать нерегулярно, по требованию). Получается некий аналог мозгового штурма. Мы использовали такой подход при поиске логических ошибок в нашем ПО: смотрели «старый» код, который был написан за несколько месяцев до review (это можно отнести тоже к отличиям от обычного review – где обычно смотрят свежий код).

Результаты review

Самое главное при проведении review – это использование полученного результата. В результате review могут появиться следующие артефакты:

- Описание способа решения задачи(design review).
- UML диаграммы (design review).
- Комментарии к стилю кода(code review).
- Более правильный вариант (быстрый, легкочитаемый) реализации (design review, code review).
- Указание на ошибки в коде (забытое условие в switch и т.д.) (code review).
- Юниттесты (design review,code review).

При этом очень важно, чтобы все результаты не пропали, и были внесены в VCS, wiki. Этому могут препятствовать:

- Сроки проекта.
- Лень, забывчивость разработчиков.
- Отсутствие удобного механизма внесения изменений review, а также контроль внесения этих изменений.

Для преодоления этих проблем частично может помочь:

- pre-commit hook вVCS.
- Создание ветви в VCS, из которой изменения вливаются в основную ветвь только после review.
- Запрет сборки дистрибутива на CI сервере без проведения review. Например, при сборке дистрибутива проверять специальные свойства (svn:properties), либо специальный файл с результатами review. И отказывать в сборке дистрибутива, если не все ревьюверы одобрили (approve) код.
- Использование методологии в разработке (в которой code review является неотъемлемой частью).

Задание

Заданием работы является инспекция программного кода на предмет соответствия стандартам кодирования разработанной информационной системы для соответствующей предметной области из Приложения 1.

Контрольные вопросы

1. Что такое Code Review?
2. Как проводить Code Review?

Лабораторная работа №9. «Оценка программных средств с помощью метрик».

Цель работы: Изучить базовые метрики качества программного обеспечения, получить практические навыки их расчёта и интерпретации на примере PHP-кода в среде Visual Studio Code с использованием современных инструментов.

1. Теоретическая часть

Метрика программного обеспечения — это численная мера, позволяющая оценить свойство программного кода, такое как размер, сложность, надёжность, сопровождаемость или качество дизайна.

1.1. Классификация и виды метрик:

1. **Метрики размера:**

- **LOC (Lines of Code)** — количество строк кода. Простая, но грубая оценка объёма программы. Различают:
  - **SLOC (Source Lines of Code)** — строки исходного кода без комментариев и пустых строк.

- **NLOC (Logical Lines of Code)** — логические инструкции (например, разделённые ; в PHP).
  - **Количество классов, методов, функций.**

2. **Метрики сложности:**
   - **Цикломатическая сложность (Cyclomatic Complexity, V(G))** — метрика Томаса МакКейба. Отражает количество линейно независимых путей в графе потока управления программы (например, в функции). Рассчитывается по формуле:

$$V(G) = E - N + 2P$$

где E — количество рёбер в графе, N — количество узлов, P — количество компонент связности (обычно 1 для одной функции).

Практическое правило: Число условных операторов (if, else, case, for, while, &&, ||) + 1.

Интерпретация:

- 1-10: Простой код, низкий риск.
- 11-20: Умеренная сложность.
- 21-50: Высокая сложность, требует рефакторинга.
- 50: Очень высокий риск, код трудно тестировать и поддерживать.

3. **Метрики связности и связанности (Cohesion & Coupling):**
   - **Связность (Cohesion)** — степень, с которой элементы одного модуля (класса) связаны между собой. Высокая связность — цель хорошего дизайна (класс делает одну чёткую вещь).
   - **Связанность (Coupling)** — степень зависимости одного модуля от других. Низкая связанность — цель хорошего дизайна (изменения в одном классе мало влияют на другие).

4. **Наследование (Inheritance):**
   - **Глубина дерева наследования (DIT)** — длина максимального пути от класса к корневому классу в иерархии. Большая глубина может усложнить понимание кода.

5. **Объектно-ориентированные метрики Чидамбре (CK):**
   - **WMC (Weighted Methods per Class)** — сумма цикломатической сложности всех методов класса (или просто количество методов). Высокое WMC — признак слишком «умного» класса.

- **RFC (Response For a Class)** — количество методов, которые могут быть вызваны в ответ на сообщение, отправленное объекту класса (размер множества методов класса + вызываемых ими методов).

2. Практическая часть: Расчёт метрик для PHP-кода в VS Code

Задание: Проанализировать предложенный PHP-код, рассчитать ключевые метрики вручную и с помощью инструментов, сделать выводы о качестве кода.

2.1. Исходный код для анализа (`UserProfile.php`)

php

```php
<?php
/**
 * Класс для управления профилем пользователя.
 * Демонстрационный код с намеренными недостатками для анализа метрик.
 */
class UserProfile {
    private $username;
    private $email;
    private $age;
    private $dbConnection;

    public function __construct($username, $email, $age) {
        $this->username = $username;
        $this->email = $email;
        $this->age = $age;
        // Нарушение: класс сам управляет подключением к БД (высокая связанность)
        $this->dbConnection = new mysqli('localhost', 'user', 'pass', 'db');
    }

    // Геттеры
    public function getUsername() { return $this->username; }
    public function getEmail() { return $this->email; }
    public function getAge() { return $this->age; }
```

```php
/**
 * Сложный метод с высокой цикломатической сложностью.
 */
public function validateProfile() {
    $errors = [];

    // Валидация имени
    if (empty($this->username)) {
        $errors[] = "Username is required.";
    } elseif (strlen($this->username) < 3) {
        $errors[] = "Username must be at least 3 characters.";
    } elseif (!preg_match('/^[a-zA-Z0-9_]+$/', $this->username)) {
        $errors[] = "Username can only contain letters, numbers, and underscores.";
    }

    // Валидация email
    if (empty($this->email)) {
        $errors[] = "Email is required.";
    } elseif (!filter_var($this->email, FILTER_VALIDATE_EMAIL)) {
        $errors[] = "Invalid email format.";
    }

    // Валидация возраста
    if ($this->age < 0 || $this->age > 150) {
        $errors[] = "Age must be between 0 and 150.";
    } elseif ($this->age < 18) {
        $errors[] = "You must be at least 18 years old.";
    }

    // Логирование ошибок (дополнительная ответственность)
    if (!empty($errors)) {
```

```
            $this->logErrorsToDatabase($errors);
        }

        return empty($errors);
    }

    private function logErrorsToDatabase(array $errors) {
        $errorString = implode(", ", $errors);
        $query = "INSERT INTO error_logs (message, user) VALUES ('$errorString', '$this->username')";
        $this->dbConnection->query($query);
    }

    // Метод, который делает слишком много (низкая связность)
    public function processUserData() {
        // 1. Валидация
        $isValid = $this->validateProfile();
        if (!$isValid) {
            return false;
        }

        // 2. Сохранение в БД
        $this->saveToDatabase();

        // 3. Отправка email
        $this->sendWelcomeEmail();

        // 4. Генерация отчёта
        $report = $this->generateReport();
        return $report;
    }
```

```
    private function saveToDatabase() {
        // ... код сохранения ...
    }

    private function sendWelcomeEmail() {
        // ... код отправки email ...
    }

    private function generateReport() {
        // Сложный код с циклами и условиями
        $data = [];
        for ($i = 0; $i < 10; $i++) {
            if ($i % 2 == 0) {
                $data[] = "Even: $i";
            } else {
                $data[] = "Odd: $i";
            }
        }
        return $data;
    }
}

// Демонстрация использования
$user = new UserProfile("john_doe", "john@example.com", 25);
if ($user->validateProfile()) {
    echo "Profile is valid.\n";
    $user->processUserData();
} else {
    echo "Profile validation failed.\n";
}
?>
```

2.2. Ручной расчёт метрик (на примере метода validateProfile())

1. **LOC (SLOC) для метода** validateProfile()**:** Подсчитываем строки с кодом (без комментариев и пустых строк). ~25 строк.

2. **Цикломатическая сложность (V(G)) для** validateProfile()**:**

- Подсчитываем условные точки:
  - if (empty(...)) (3 шт.)
  - elseif (strlen(...)) (2 шт.)
  - elseif (!preg_match(...))
  - if (empty(...)) (для email)
  - elseif (!filter_var(...))
  - if ($this->age < 0 || $this->age > 150) (**Внимание:** здесь два условия, но они в одном операторе if — это одна точка принятия решения? Нет, оператор || увеличивает сложность! Правильнее считать так: || и && добавляют +1 к сложности).
  - elseif ($this->age < 18)
  - if (!empty($errors))
- **Более точный подсчёт по формуле (Условные операторы + Логические операторы + 1):**
  - Основные if/elseif: 8 штук.
  - Логические ИЛИ в условии ($this->age < 0 || $this->age > 150): +1.
  - **Итого точек принятия решений:** 9.
  - **Цикломатическая сложность V(G) = 9 + 1 = 10.**
- **Вывод:** Сложность равна 10 (граница приемлемого). Метод требует упрощения (например, вынесения валидации каждого поля в отдельный метод).

2.3. Автоматизированный расчёт с помощью инструментов в VS Code

VS Code — это редактор, поэтому для расчёта метрик нужны расширения (plugins).

Рекомендуемые расширения:

1. **PHP Metrics** (calebporzio.php-metrics) — прямо в редакторе показывает сложность.
2. **PHPMD (PHP Mess Detector)** — выявляет проблемы в коде.
3. **PHPStan / Psalm** — статический анализ, также может давать информацию о сложности.

Установка и использование PHPMD (через терминал):

1. Установите PHPMD глобально: composer global require phpmd/phpmd

2. Проанализируйте файл, запросив отчёт в формате text, включая правило codesize (которое содержит информацию о цикломатической сложности):

bash

phpmd UserProfile.php text codesize

3. **Пример вывода (ожидаемый):**

text

UserProfile.php:37 The method validateProfile has a Cyclomatic Complexity of 10. The configured cyclomatic complexity threshold is 10.

UserProfile.php:80 The method processUserData has a Cyclomatic Complexity of 4.

UserProfile.php:102 The method generateReport has a Cyclomatic Complexity of 3.

Это подтверждает наш ручной расчёт.

2.4. Анализ других метрик по коду

- **Связность (Cohesion) класса** UserProfile**: Низкая.** Класс выполняет множество несвязанных задач: хранит данные пользователя, валидирует их, логирует ошибки, работает с базой данных, отправляет email, генерирует отчёты. Нарушается *Принцип единственной ответственности (SRP)*.
- **Связанность (Coupling): Высокая.** Класс явно создаёт экземпляр mysqli в конструкторе, жёстко привязан к реализации базы данных. Это затруднит тестирование и замену СУБД.
- **WMC (Вес класса):** В классе около 10 методов (считая геттеры). Это умеренное значение, но в сочетании с низкой связностью является проблемой.
- **DIT (Глубина наследования):** 1 (только от базового класса stdClass в PHP). Низкая, проблем нет.

3. Выводы и рекомендации по результатам анализа

1. **Метод** validateProfile() имеет повышенную цикломатическую сложность (V(G)=10). **Рекомендация:** Разбить его на три отдельных метода: validateUsername(), validateEmail(), validateAge().

2. **Класс** UserProfile **в целом** страдает от низкой связности и высокой связанности. **Рекомендации:**

   - Вынести работу с базой данных (логирование, сохранение) в отдельный класс-репозиторий (UserRepository).

- Вынести отправку email в сервис `EmailService`.
- Вынести генерацию отчёта в отдельный класс `ReportGenerator`.
- Внедрять зависимости через конструктор (Dependency Injection), а не создавать их внутри класса.

3. **Итог:** Использование метрик (особенно цикломатической сложности и анализа связности) позволило объективно выявить «запахи кода» (code smells) и наметить конкретные пути рефакторинга для повышения сопровождаемости, тестируемости и надёжности программного продукта.

Заключение: Метрики являются мощным инструментом для количественной оценки качества кода и принятия обоснованных решений о необходимости рефакторинга. Их следует использовать регулярно в процессе разработки, интегрируя в CI/CD пайплайны.

Задание:

Проанализируйте предоставленный PHP-код, рассчитайте метрики (вручную и с помощью инструментов), выявите проблемы и предложите пути рефакторинга.

Часть 1: Ручной расчёт метрик

1. Рассчитайте цикломатическую сложность метода `processCompleteOrder()`
2. Рассчитайте цикломатическую сложность метода `validateCustomer()`
3. Рассчитайте цикломатическую сложность метода `processPayment()`
4. Оцените связность (cohesion) класса `OrderProcessor` по шкале от 1 до 10
5. Оцените связанность (coupling) класса `OrderProcessor`
6. Определите DIT (глубину наследования) для класса `VIPCustomer`

Часть 2: Анализ с помощью инструментов

1. Установите PHPMD или PHP Metrics в VS Code
2. Проанализируйте код и получите отчёт
3. Сравните автоматические расчёты с ручными

Часть 3: Выявление проблем и рефакторинг

1. Выделите 3 основные проблемы в коде на основе метрик
2. Предложите конкретные способы рефакторинга для каждой проблемы
3. Для метода с самой высокой цикломатической сложностью предложите вариант разбиения на более простые методы

Часть 4: Практическое задание

1. Напишите улучшенную версию класса OrderProcessor с лучшими метриками
2. Проведите сравнительный анализ метрик старой и новой версии
3. Сделайте выводы об эффективности рефакторинга

Исходный код: Система обработки заказов (OrderSystem.php)

php

```php
<?php
/**
 * Лабораторная работа №9: Анализ метрик программного кода
 * Пример системы обработки заказов с намеренными архитектурными проблемами
 */

/**
 * Класс для работы с заказами.
 * Содержит намеренные проблемы для анализа метриками.
 */
class OrderProcessor {
    private $dbConnection;
    private $logger;
    private $order;
    private $customer;
    private $payment;

    public function __construct($orderId) {
        // Нарушение: класс создаёт все зависимости сам (высокая связанность)
        $this->dbConnection = new mysqli('localhost', 'user', 'pass', 'ecommerce');
        $this->logger = new FileLogger('/var/log/orders.log');
        $this->order = $this->loadOrder($orderId);
        $this->customer = $this->loadCustomer($this->order['customer_id']);
        $this->payment = new PaymentProcessor();
    }
```

```php
/**
 * Главный метод обработки заказа со СЛОЖНОЙ логикой
 * Цель: высокая цикломатическая сложность
 */
public function processCompleteOrder() {
    $this->log("Начало обработки заказа #" . $this->order['id']);

    // Проверка существования заказа
    if (!isset($this->order['id'])) {
        $this->log("Ошибка: заказ не найден");
        return false;
    }

    // Проверка статуса заказа
    if ($this->order['status'] === 'cancelled') {
        $this->log("Заказ отменён, обработка невозможна");
        return false;
    } elseif ($this->order['status'] === 'completed') {
        $this->log("Заказ уже обработан");
        return true;
    } elseif ($this->order['status'] !== 'pending') {
        $this->log("Неизвестный статус заказа: " . $this->order['status']);
        return false;
    }

    // Проверка клиента
    if (!$this->validateCustomer()) {
        $this->log("Ошибка валидации клиента");
        return false;
    }

    // Проверка наличия товаров на складе
```

```
        $items = $this->getOrderItems();
        $allAvailable = true;
        foreach ($items as $item) {
            if (!$this->checkStock($item['product_id'], $item['quantity'])) {
                $this->log("Недостаточно товара: " . $item['product_id']);
                $allAvailable = false;
            }
        }

        if (!$allAvailable) {
            $this->updateOrderStatus('waiting_for_stock');
            $this->notifyCustomerAboutStock($this->customer['email']);
            return false;
        }

        // Обработка платежа
        $paymentResult = $this->processPayment();
        if (!$paymentResult['success']) {
            if ($paymentResult['reason'] === 'insufficient_funds') {
                $this->updateOrderStatus('payment_failed');
                $this->notifyCustomerAboutPayment($this->customer['email'],
'insufficient_funds');
            } elseif ($paymentResult['reason'] === 'card_declined') {
                $this->updateOrderStatus('payment_failed');
                $this->notifyCustomerAboutPayment($this->customer['email'],
'card_declined');
            } elseif ($paymentResult['reason'] === 'system_error') {
                $this->updateOrderStatus('payment_error');
                $this->notifyAdminAboutError();
            }
            return false;
        }
```

```
    // Обновление остатков
    foreach ($items as $item) {
        $this->updateStock($item['product_id'], $item['quantity']);
    }

    // Генерация и отправка документации
    $this->generateInvoice();
    $this->generateShippingLabel();

    // Выбор службы доставки
    $deliveryService = $this->selectDeliveryService();
    if ($deliveryService === 'express') {
        $this->scheduleExpressDelivery();
    } elseif ($deliveryService === 'standard') {
        $this->scheduleStandardDelivery();
    } elseif ($deliveryService === 'pickup') {
        $this->prepareForPickup();
    }

    // Обновление статуса и уведомления
    $this->updateOrderStatus('completed');
    $this->sendConfirmationEmail();
    $this->updateLoyaltyPoints();

    // Логирование завершения
    $this->log("Заказ #" . $this->order['id'] . " успешно обработан");

    return true;
}

/**
```

```
     * ОЧЕНЬ сложный метод валидации
     */
    private function validateCustomer() {
        $errors = [];

        if (empty($this->customer['name'])) {
            $errors[] = "Имя клиента не указано";
        } elseif (strlen($this->customer['name']) < 2) {
            $errors[] = "Имя слишком короткое";
        } elseif (strlen($this->customer['name']) > 100) {
            $errors[] = "Имя слишком длинное";
        }

        if (empty($this->customer['email'])) {
            $errors[] = "Email не указан";
        } elseif (!filter_var($this->customer['email'], FILTER_VALIDATE_EMAIL)) {
            $errors[] = "Неверный формат email";
        } elseif (!$this->checkEmailDomain($this->customer['email'])) {
            $errors[] = "Домен email не существует";
        }

        if (empty($this->customer['phone'])) {
            $errors[] = "Телефон не указан";
        } elseif (!preg_match('/^\+?[0-9\s\-\(\)]{7,20}$/', $this->customer['phone'])) {
            $errors[] = "Неверный формат телефона";
        }

        if (empty($this->customer['address'])) {
            $errors[] = "Адрес не указан";
        } else {
            $addressParts = explode(',', $this->customer['address']);
            if (count($addressParts) < 3) {
```

```
            $errors[] = "Адрес должен содержать улицу, город и страну";
        } else {
            if (strlen(trim($addressParts[0])) < 5) {
                $errors[] = "Название улицы слишком короткое";
            }
            if (strlen(trim($addressParts[1])) < 2) {
                $errors[] = "Название города слишком короткое";
            }
        }
    }

    if ($this->customer['age'] < 18) {
        $errors[] = "Клиент должен быть старше 18 лет";
    } elseif ($this->customer['age'] > 120) {
        $errors[] = "Некорректный возраст";
    }

    if (!empty($errors)) {
        $this->log("Ошибки валидации клиента: " . implode(', ', $errors));
        return false;
    }

    return true;
}

/**
 * Метод обработки платежа с множеством ветвлений
 */
private function processPayment() {
    $paymentMethod = $this->order['payment_method'];

    if ($paymentMethod === 'credit_card') {
```

```php
            if ($this->customer['credit_score'] > 700) {
                if ($this->order['amount'] < 1000) {
                    return ['success' => true, 'transaction_id' => uniqid()];
                } else {
                    if ($this->customer['account_age'] > 365) {
                        return ['success' => true, 'transaction_id' => uniqid()];
                    } else {
                        return ['success' => false, 'reason' => 'insufficient_funds'];
                    }
                }
            } else {
                return ['success' => false, 'reason' => 'card_declined'];
            }
        } elseif ($paymentMethod === 'paypal') {
            if ($this->customer['paypal_verified']) {
                return ['success' => true, 'transaction_id' => uniqid()];
            } else {
                return ['success' => false, 'reason' => 'system_error'];
            }
        } elseif ($paymentMethod === 'bank_transfer') {
            return ['success' => true, 'transaction_id' => uniqid()];
        } else {
            return ['success' => false, 'reason' => 'system_error'];
        }
    }

    // Методы-заглушки для демонстрации
    private function loadOrder($id) { return ['id' => $id, 'status' => 'pending', /* ... */]; }
    private function loadCustomer($id) { return ['id' => $id, 'name' => 'John Doe', /* ...
*/]; }
    private function log($message) { echo $message . "\n"; }
    private function getOrderItems() { return [['product_id' => 1, 'quantity' => 2]]; }
```

```php
    private function checkStock($id, $qty) { return true; }
    private function updateStock($id, $qty) { /* ... */ }
    private function updateOrderStatus($status) { /* ... */ }
    private function notifyCustomerAboutStock($email) { /* ... */ }
    private function notifyCustomerAboutPayment($email, $reason) { /* ... */ }
    private function notifyAdminAboutError() { /* ... */ }
    private function generateInvoice() { /* ... */ }
    private function generateShippingLabel() { /* ... */ }
    private function selectDeliveryService() { return 'standard'; }
    private function scheduleExpressDelivery() { /* ... */ }
    private function scheduleStandardDelivery() { /* ... */ }
    private function prepareForPickup() { /* ... */ }
    private function sendConfirmationEmail() { /* ... */ }
    private function updateLoyaltyPoints() { /* ... */ }
    private function checkEmailDomain($email) { return true; }
}

/**
 * Класс для обработки платежей (демонстрация связанности)
 */
class PaymentProcessor {
    public function process($amount, $method) {
        // Сложный метод с вложенными условиями
        if ($method === 'credit_card') {
            // Логика обработки карты
            for ($i = 0; $i < 5; $i++) {
                if ($i % 2 == 0) {
                    echo "Processing step $i\n";
                }
            }
        } elseif ($method === 'paypal') {
            // Логика PayPal
```

```
            $attempts = 0;
            while ($attempts < 3) {
                if ($attempts > 1) {
                    echo "Retry attempt $attempts\n";
                }
                $attempts++;
            }
        }
        return true;
    }
}

/**
 * Простой логгер (демонстрация наследования)
 */
class FileLogger {
    private $filePath;

    public function __construct($filePath) {
        $this->filePath = $filePath;
    }

    public function log($message) {
        file_put_contents($this->filePath, $message, FILE_APPEND);
    }
}

/**
 * Демонстрация глубокого наследования (проблема DIT)
 */
class BaseEntity {
    protected $id;
```

```php
    public function getId() { return $this->id; }
}

class User extends BaseEntity {
    protected $name;
    public function getName() { return $this->name; }
}

class Customer extends User {
    private $email;
    private $orders = [];

    public function addOrder($order) {
        $this->orders[] = $order;
        if (count($this->orders) > 10) {
            $this->applyDiscount();
        }
    }

    private function applyDiscount() {
        // Сложная логика применения скидки
        $total = 0;
        foreach ($this->orders as $order) {
            $total += $order['amount'];
            if ($order['amount'] > 100) {
                $total *= 0.9; // 10% скидка
            }
        }
        return $total;
    }
}
```

```php
class VIPCustomer extends Customer {
    private $discountLevel;

    public function getDiscount($amount) {
        if ($this->discountLevel === 'gold') {
            if ($amount > 1000) {
                return $amount * 0.8;
            } else {
                return $amount * 0.9;
            }
        } elseif ($this->discountLevel === 'platinum') {
            return $amount * 0.7;
        } else {
            return $amount;
        }
    }
}

/**
 * Класс с низкой связностью (делает слишком много)
 */
class ProductManager {
    public function updateProduct($id, $data) {
        // 1. Валидация данных
        $this->validateProductData($data);

        // 2. Обновление в БД
        $this->updateInDatabase($id, $data);

        // 3. Кэширование
        $this->updateCache($id, $data);
```

```
        // 4. Отправка уведомлений
        $this->notifySubscribers($id);

        // 5. Логирование
        $this->logUpdate($id);

        // 6. Синхронизация с внешними системами
        $this->syncWithExternalSystems($id);

        return true;
    }

    private function validateProductData($data) { /* ... */ }
    private function updateInDatabase($id, $data) { /* ... */ }
    private function updateCache($id, $data) { /* ... */ }
    private function notifySubscribers($id) { /* ... */ }
    private function logUpdate($id) { /* ... */ }
    private function syncWithExternalSystems($id) { /* ... */ }
}

// Пример использования
echo "=== Запуск обработки заказа ===\n";
$processor = new OrderProcessor(12345);
$result = $processor->processCompleteOrder();
echo "Результат обработки: " . ($result ? "Успешно" : "Неудача") . "\n";

// Демонстрация наследования
$vipCustomer = new VIPCustomer();
echo "Скидка для VIP: " . $vipCustomer->getDiscount(1500) . "\n";
```

Лабораторная работа №10. «Инспекция программного кода на предмет соответствия стандартам кодирования».

Теоретическая часть

1. Введение в инспекцию кода и стандарты кодирования

Инспекция программного кода — это формальный процесс проверки исходного кода с целью выявления дефектов, нарушений стандартов и потенциальных проблем на ранних этапах разработки.

Стандарты кодирования — набор правил и соглашений, определяющих стиль написания кода в проекте. Они включают:

- Форматирование (отступы, переносы строк, пробелы)
- Именование переменных, функций, классов
- Структуру файлов и каталогов
- Рекомендации по безопасности и производительности

2. Преимущества следования стандартам кодирования

1. **Улучшенная читаемость** — единый стиль упрощает понимание кода
2. **Упрощённое сопровождение** — код легче поддерживать и модифицировать
3. **Снижение количества ошибок** — единообразие уменьшает вероятность синтаксических ошибок
4. **Эффективная коллаборация** — разработчики быстрее понимают код друг друга
5. **Автоматизация проверок** — возможность использовать инструменты статического анализа

3. Основные стандарты кодирования для PHP

3.1. PSR (PHP Standards Recommendations)

- **PSR-1**: Basic Coding Standard — базовые правила
- **PSR-12**: Extended Coding Style — расширенные правила форматирования
- **PSR-4**: Autoloader Standard — стандарт автозагрузки классов

3.2. Основные правила PSR-12

php

```
// Пример правильного форматирования по PSR-12
namespace Vendor\Package;

use Vendor\Package\ClassA;
use Vendor\Package\ClassB;
```

```php
class ClassName
{
    private const VERSION = '1.0';
    private $property;

    public function methodName($argument): string
    {
        if ($argument === true) {
            return 'Yes';
        }

        return 'No';
    }
}
```

3.3. Основные правила именования

- **Классы**: CamelCase (MyClassName)
- **Методы/функции**: camelCase (getUserName)
- **Переменные**: camelCase или snake_case (в зависимости от стандарта)
- **Константы**: UPPER_SNAKE_CASE (MAX_SIZE)
- **Пространства имён**: соответствуют структуре каталогов

4. Инструменты для инспекции PHP-кода

1. **PHP_CodeSniffer (phpcs)** — проверка соответствия стандартам
2. **PHP-CS-Fixer** — автоматическое исправление нарушений
3. **PHPStan** — статический анализ
4. **Psalm** — анализ типов и поиск ошибок
5. **PHPMD** — поиск проблем в коде

Практическая часть: Инспекция PHP-кода в VS Code

Задание: Провести инспекцию предоставленного кода, выявить нарушения стандартов PSR-12, исправить их и провести повторную проверку.

1. Установка и настройка инструментов в VS Code

1.1. Установка расширений

1. Откройте VS Code
2. Перейдите в раздел Extensions (Ctrl+Shift+X)
3. Установите расширения:
   - **PHP Intelephense**
   - **PHP Sniffer & Beautifier**
   - **PHP CS Fixer**

1.2. Установка PHP_CodeSniffer через Composer

bash

```
# Глобальная установка
composer global require "squizlabs/php_codesniffer=*"

# Проверка установки
phpcs --version
```

1.3. Настройка в VS Code

Добавьте в настройки VS Code (settings.json):

json

```
{
    "phpcs.enable": true,
    "phpcs.standard": "PSR12",
    "phpcs.executablePath":
"C:\\Users\\Username\\AppData\\Roaming\\Composer\\vendor\\bin\\phpcs.bat",
    "phpcbf.enable": true,
    "phpcbf.standard": "PSR12",
    "phpcbf.executablePath":
"C:\\Users\\Username\\AppData\\Roaming\\Composer\\vendor\\bin\\phpcbf.bat"
}
```

2. Исходный код для инспекции

Создайте файл bad_code.php:

php

```
<?php
class user_manager{
private $db_connection;
```

```
private $UserData;
const MAX_LOGIN_ATTEMPTS=5;

function __construct($db){
$this->db_connection=$db;
}

public function authenticateUser($username,$password){
if(!empty($username)&&!empty($password))
{
$sql="SELECT * FROM users WHERE username='".$username."'";
$result=$this->db_connection->query($sql);
if($result->num_rows>0){
$user=$result->fetch_assoc();
if(password_verify($password,$user['password_hash']))
{
if($user['is_active']==1)
{
$this->UserData=$user;
return array('status'=>'success','user'=>$user);
}else{
return array('status'=>'error','message'=>'Account is not active');
}
}else{
return array('status'=>'error','message'=>'Invalid password');
}
}else{
return array('status'=>'error','message'=>'User not found');
}
}else{
return array('status'=>'error','message'=>'Username and password are required');
}
```

```
    }

    function getUserProfile($user_id){
    $sql="SELECT u.*, p.avatar, p.bio FROM users u LEFT JOIN profiles p ON u.id=p.user_id WHERE u.id=".$user_id;
    $result=$this->db_connection->query($sql);
    if($result){
    return $result->fetch_assoc();
    }
    return null;
    }

    public function update_user_email($userId,$newEmail)
    {
    if(filter_var($newEmail,FILTER_VALIDATE_EMAIL)){
    $stmt=$this->db_connection->prepare("UPDATE users SET email=? WHERE id=?");
    $stmt->bind_param("si",$newEmail,$userId);
    if($stmt->execute()){
    return true;
    }
    }
    return false;
    }
    }
    ?>
```

3. Проведение инспекции вручную

3.1. Выявленные нарушения PSR-12

Нарушение	Правило PSR-12	Пример в коде

Нарушение	Правило PSR-12	Пример в коде
Отсутствие пространства имён	PSR-1: Файлы ДОЛЖНЫ объявлять символы (классы, функции, константы) или выполнять побочные эффекты (например, генерировать вывод), но НЕ делать и то, и другое.	Нет namespace
Неправильное имя класса	PSR-1: Имена классов ДОЛЖНЫ быть объявлены в StudlyCaps.	class user_manager должно быть class UserManager
Неправильные имена свойств	PSR-1: Имена свойств ДОЛЖНЫ быть объявлены в $camelCase.	$db_connection, $UserData
Неправильное имя метода	PSR-1: Имена методов ДОЛЖНЫ быть объявлены в camelCase().	update_user_email()
Отсутствие объявления видимости	PSR-12: Все свойства и методы ДОЛЖНЫ объявлять видимость.	function __construct() без public
Неправильный стиль констант	PSR-12: Константы классов ДОЛЖНЫ быть объявлены в верхнем регистре с подчеркиванием в качестве разделителей.	Уже правильно, но для примера оставим
Длинные строки	PSR-12: Длина строки НЕ ДОЛЖНА быть жестко ограничена... но ДОЛЖНА быть мягкое ограничение в 120 символов.	Строки SQL-запросов слишком длинные
Неправильные отступы	PSR-12: Для отступов ДОЛЖНЫ использоваться 4 пробела.	Используются табы или разное количество пробелов

Нарушение	Правило PSR-12	Пример в коде
Отсутствие пробелов вокруг операторов	PSR-12: ДОЛЖЕН быть хотя бы один пробел по обе стороны оператора.	`if(!empty($username)&&!empty($password))`
Неправильное расположение фигурных скобок	PSR-12: Тело каждого класса ДОЛЖНО быть заключено в фигурные скобки... открывающая фигурная скобка ДОЛЖНА находиться на следующей строке.	`class user_manager{` должно быть на новой строке
Смешивание HTML и PHP	PSR-1: Файлы ТОЛЬКО PHP ДОЛЖНЫ заканчиваться тегом `?>` без новой строки.	Лишний `?>` в конце
SQL-инъекция	Не PSR, но критично для безопасности	Конкатенация переменных в SQL

4. Автоматическая проверка с помощью PHP_CodeSniffer

4.1. Запуск проверки через терминал

bash

```
# Проверка файла на соответствие PSR-12
phpcs --standard=PSR12 bad_code.php

# Проверка с подробным выводом
phpcs --standard=PSR12 --report=full bad_code.php

# Автоматическое исправление некоторых ошибок
phpcbf --standard=PSR12 bad_code.php
```

4.2. Пример вывода PHP_CodeSniffer

text

```
FILE: bad_code.php
----------------------------------------------------------------------
FOUND 18 ERRORS AFFECTING 10 LINES
```

```
----------------------------------------------------------------------
 2 | ERROR | [ ] Missing file doc comment
 3 | ERROR | [ ] Class name "user_manager" is not in PascalCase format
 3 | ERROR | [x] Opening brace of a class must be on the line after the
   |       |     definition
 4 | ERROR | [x] Property name "$db_connection" is not in camelCase format
 5 | ERROR | [ ] Constant name "MAX_LOGIN_ATTEMPTS" must be in uppercase
 7 | ERROR | [ ] Method name "__construct" must not be prefixed with
   |       |     underscores
 7 | ERROR | [x] Missing visibility declaration
 9 | ERROR | [x] Line exceeds 120 characters; contains 158 characters
...
----------------------------------------------------------------------
PHPCBF CAN FIX 12 OF THE 18 ERRORS AUTOMATICALLY
----------------------------------------------------------------------
```

5. Исправленный код после рефакторинга

Создайте файл good_code.php:

php

```php
<?php

declare(strict_types=1);

namespace App\Service;

use PDO;
use PDOException;

/**
 * User Manager Service
 *
 * Handles user authentication and profile management
 */
```

```php
class UserManager
{
    private const MAX_LOGIN_ATTEMPTS = 5;

    /**
     * @var PDO
     */
    private $dbConnection;

    /**
     * @var array
     */
    private $userData;

    /**
     * Constructor
     *
     * @param PDO $dbConnection Database connection
     */
    public function __construct(PDO $dbConnection)
    {
        $this->dbConnection = $dbConnection;
    }

    /**
     * Authenticate user
     *
     * @param string $username Username
     * @param string $password Password
     *
     * @return array Authentication result
     */
```

```php
public function authenticateUser(string $username, string $password): array
{
    if (empty($username) || empty($password)) {
        return [
            'status'  => 'error',
            'message' => 'Username and password are required'
        ];
    }

    $sql = 'SELECT * FROM users WHERE username = :username';
    $stmt = $this->dbConnection->prepare($sql);
    $stmt->execute([':username' => $username]);

    if ($stmt->rowCount() === 0) {
        return [
            'status'  => 'error',
            'message' => 'User not found'
        ];
    }

    $user = $stmt->fetch(PDO::FETCH_ASSOC);

    if (!password_verify($password, $user['password_hash'])) {
        return [
            'status'  => 'error',
            'message' => 'Invalid password'
        ];
    }

    if ($user['is_active'] !== 1) {
        return [
            'status'  => 'error',
```

```
            'message' => 'Account is not active'
        ];
    }

    $this->userData = $user;

    return [
        'status' => 'success',
        'user'   => $user
    ];
}

/**
 * Get user profile
 *
 * @param int $userId User ID
 *
 * @return array|null User profile or null if not found
 */
public function getUserProfile(int $userId): ?array
{
    $sql = '
        SELECT
            u.*,
            p.avatar,
            p.bio
        FROM users u
        LEFT JOIN profiles p ON u.id = p.user_id
        WHERE u.id = :userId
    ';

    $stmt = $this->dbConnection->prepare($sql);
```

```
        $stmt->execute([':userId' => $userId]);

        if ($stmt->rowCount() === 0) {
            return null;
        }

        return $stmt->fetch(PDO::FETCH_ASSOC);
    }

    /**
     * Update user email
     *
     * @param int    $userId   User ID
     * @param string $newEmail New email address
     *
     * @return bool True if successful, false otherwise
     */
    public function updateUserEmail(int $userId, string $newEmail): bool
    {
        if (!filter_var($newEmail, FILTER_VALIDATE_EMAIL)) {
            return false;
        }

        $sql = 'UPDATE users SET email = :email WHERE id = :userId';
        $stmt = $this->dbConnection->prepare($sql);

        return $stmt->execute([
            ':email'  => $newEmail,
            ':userId' => $userId
        ]);
    }
}
```

6. Сравнительный анализ

Таблица улучшений:

Аспект	Было	Стало	Улучшение
Имя класса	`user_manager`	`UserManager`	Соответствие StudlyCaps
Имена методов	`update_user_email`	`updateUserEmail`	Соответствие camelCase
Видимость методов	Не указана	Указана (`public`, `private`)	Явное объявление
Безопасность SQL	Конкатенация	Подготовленные выражения	Защита от SQL-инъекций
Длина строк	До 158 символов	Макс. 120 символов	Улучшенная читаемость
Комментарии	Отсутствуют	Документирующие блоки	Улучшенная документация
Пространство имён	Отсутствует	`App\Service`	Правильная организация

7. Проверка исправленного кода

bash

```
# Проверка исправленного кода
phpcs --standard=PSR12 good_code.php

# Ожидаемый результат
No sniffs registered for provided standard

# Или при использовании полного пути
phpcs --standard=/path/to/PSR12/ruleset.xml good_code.php
```

8. Настройка автоматической проверки в проекте

Создание конфигурационного файла phpcs.xml:

xml

```xml
<?xml version="1.0"?>
<ruleset name="PHP_CodeSniffer">
    <description>Code standards for our project</description>

    <!-- Check all PHP files in src directory -->
    <file>src</file>

    <!-- Exclude vendor directory -->
    <exclude-pattern>vendor/*</exclude-pattern>

    <!-- Use PSR-12 standard -->
    <rule ref="PSR12"/>

    <!-- Additional custom rules -->
    <rule ref="Generic.Files.LineLength">
        <properties>
            <property name="lineLimit" value="120"/>
            <property name="absoluteLineLimit" value="0"/>
        </properties>
    </rule>

    <!-- Require strict types declaration -->
    <rule ref="SlevomatCodingStandard.TypeHints.DeclareStrictTypes">
        <properties>
            <property name="spacesCountAroundEqualsSign" value="0"/>
        </properties>
    </rule>
</ruleset>
```

9. Интеграция с Git (pre-commit hook)

Создайте файл `.git/hooks/pre-commit`:

bash

```
#!/bin/bash

echo "Running PHP CodeSniffer..."

PHP_FILES=$(git diff --cached --name-only --diff-filter=ACM | grep "\.php$")

if [ -z "$PHP_FILES" ]; then
    echo "No PHP files to check"
    exit 0
fi

PASS=true

for FILE in $PHP_FILES
do
    phpcs --standard=PSR12 "$FILE"

    if [ $? != 0 ]; then
        PASS=false
    fi
done

if ! $PASS; then
    echo "PHP CodeSniffer found violations. Commit aborted."
    echo "Run 'phpcbf' to fix some violations automatically."
    exit 1
fi

exit 0
```

10. Практическое задание для студентов

Задание 1: Проведите инспекцию следующего кода:

php

```
<?php
function calculate_price($quantity, $price_per_unit,$discount_percentage=0){
$subtotal=$quantity*$price_per_unit;
if($discount_percentage>0){
$discount_amount=$subtotal*($discount_percentage/100);
$total=$subtotal-$discount_amount;
}else{
$total=$subtotal;
}
return round($total,2);
}
?>
```

Задание 2: Напишите скрипт, который:

1. Принимает путь к PHP-файлу
2. Запускает phpcs для проверки
3. Сохраняет отчёт в файл
4. Выводит статистику по ошибкам

Задание 3: Создайте собственный стандарт кодирования на основе PSR-12 с дополнительными правилами:

- Максимальная длина строки: 100 символов
- Обязательное использование strict_types
- Запрет на var_dump() в production-коде

11. Выводы

1. **Стандарты кодирования** — необходимый элемент профессиональной разработки
2. **Инспекция кода** должна быть регулярным процессом, а не разовой акцией
3. **Автоматизированные инструменты** существенно упрощают проверку соответствия стандартам
4. **Единый стиль кода** улучшает читаемость и снижает количество ошибок

5. **Интеграция проверок** в процесс разработки (CI/CD) обеспечивает постоянное качество кода

12. Контрольные вопросы

1. Какие преимущества даёт следование стандартам кодирования?
2. В чём разница между PSR-1 и PSR-12?
3. Какие инструменты можно использовать для автоматической проверки PHP-кода?
4. Почему подготовленные выражения в SQL предпочтительнее конкатенации?
5. Как организовать автоматическую проверку кода перед коммитом в Git?

13. Дополнительные ресурсы

1. Официальная документация PSR
2. PHP_CodeSniffer на GitHub
3. PHP-CS-Fixer
4. Стандарты кодирования Symfony

СПИСОК ИСПОЛЬЗУЕМОЙ ЛИТЕРАТУРЫ

Основная литература

1. Рудаков А.В. Технология разработки программных продуктов: учебное издание / Рудаков А.В. - Москва : Академия, 2024. - 208 с. (Специальности среднего профессионального образования). - URL: https://academia-moscow.ru - Режим доступа: Электронная библиотека «Academiamoscow». - Текст : электронный.

Дополнительная литература

1. Казанский, А. А. Объектно-ориентированное программирование. Visual Basic : учебник для среднего профессионального образования / А. А. Казанский. — 2-е изд. — Москва : Издательство Юрайт, 2025. — 295 с. — (Профессиональное образование). — ISBN 978-5-534-21384-3. — Текст : электронный // Образовательная платформа Юрайт [сайт]. — URL: https://urait.ru/bcode/569868.

2. Казанский, А. А. Программирование на visual c# 2013.: учебное пособие для СПО / Казанский А. А.. – Москва : Юрайт, 2020. – 191 с. – ISBN 978-5-534-02721-1. – URL: https://urait.ru/book/programmirovaniena-visual-c-2013-452454. – Текст : электронный.

ПРИЛОЖЕНИЕ 1. Предметная область для анализа и ее описание

№	Предметная область	Описание предметной области
1	Предметная область «операции с недвижимостью»	Администрация агентства недвижимости заказала разработку информационной системы для отдела работы с клиентами. Система предназначена для обработки данных о квартирах, которые покупает и продает агентство, расценках на квартиры, расценках на оказываемые услуги, о покупателях и совершенных сделках. Система должна выдавать
2	Предметная область «медицинские услуги»	Руководство частной медицинской клиники заказало разработку информационной системы для административной группы. Система предназначена для обработки данных о клиентах, врачах, о перечне медицинских услуг (с расценками и описанием). Система должна выдавать информацию по запросу
3	Предметная область управляющего рекламным агентством	Руководство рекламного агентства заказало разработку информационной системы для отдела работы с клиентами. Система предназначена для обработки данных о клиентах, о продукции, предоставляемых услугах, стоимости пакета заказываемой рекламы и медиа-план для заказчика. Система должна выдавать отчеты по запросу

№	Предметная область	Описание предметной области
4	Система учета заказов и их выполнение в мебельном салоне	Администрация компании по производству и продаже мебели, заказала разработку информационной системы для отдела работы с клиентами. Система предназначена для обработки данных о клиентах, о товарах (характеристика товара, возможный материал изготовления), услугах, о учете заказов. Система должна выдавать отчеты по
5	Разработка автоматизированной системы заказов по каталогу	Администрация торговой компании заказала разработку информационной системы заказов товаров по каталогам. Система предназначена для обработки данных о клиентах, товарах в каталогах, сроках поставок и дополнительных услугах, оказываемых фирмой. Система должна выдавать
6	Предметная область продавца-консультанта магазина «оптика»	Администрация магазина «оптика» заказала разработку ИС для отдела работы с покупателем. Система предназначена для обработки данных о клиенте, о материалах, учет заказов. Система должна выдавать отчеты по запросу продавца- консультанта магазина: расчеты с
7	Предметная область «расписание для спорткомплекса»	Администрация спорткомплекса заказала разработку ИС для организации своей работы. Система предназначена для обработки данных о кол-ве человек в группе, виде занятий, учет помещений, фамилии тренеров.

№	Предметная область	Описание предметной области
8	Предметная область администратора ресторана	Администрация ресторана заказала разработку ИС. Система предназначена для обработки данных о местах и площадях залов, информация о заказах на места, предварительный заказ блюд. Система должна выдавать отчеты по запросу
9	Система организации чемпионата по определенному виду спорта	Администрация города заказала разработку ИС для спортивного комитета. Система предназначена для обработки данных о стадионах, о спортсменах, тренерах, а также о времени проведения игр. Система должна выдавать отчеты по запросу члена комитета: расписание игр
10	Расчеты с поставщиками	Система должна содержать информацию о расчете с поставщиками продукции включая данные: о дате поставки и о самом поставщике, а также информацию о поставляемых изделиях. Выходная информация: документы по расчету с поставщиками, перечень имеющихся в наличии
11	Предметная область менеджера авто-сервиса	Администрация службы автосервиса заказала разработку информационной системы для отдела работы с клиентами. Система предназначена для обработки данных о комплектующих, о заказах на комплектующие, расценках по оказываемым услугам, о машинах и их обслуживании. Система должна выдавать отчеты по запросу менеджера

№	Предметная область	Описание предметной области
12	Предметная область «страхование населения»	Руководство страховой компании заказало разработку информационной системы для отдела работы с клиентами. Система предназначена для обработки данных о видах страховок, их стоимость, о совершенных сделках, о клиентах, сроках действия страховки. Система должна выдавать отчеты по запросу менеджера: прайс-лист по видам страховки,
13	Предметная область управляющего рекламным агентством	Руководство рекламного агентства заказало разработку информационной системы для отдела работы с клиентами. Система предназначена для обработки данных о клиентах, о продукции, предоставляемых услугах, стоимости пакета заказываемой рекламы и медиа-план для заказчика. Система должна выдавать отчеты по запросу менеджера: перечень изготавливаемо рекламной
14	Предметная область оператора агентства по трудоустройству	Администрация агентства по трудоустройству заказала разработку информационной системы для отдела по работе с клиентами. Система предназначена для обработки данных о специалистах, стоящих на учете, фирмах, где требуются специалисты, и требованиях, которые к специалистам предъявляются. Кроме того в системе должны обрабатываться данные об услугах предоставляемых агентством. Система должна выдавать отчеты по запросу менеджера: бланк

№	Предметная область	Описание предметной области
15	Система исследования товарного рынка (товар на выбор)	Администрация предприятия заказала разработку информационной системы для отдела маркетинга. Система предназначена для обработки данных о продажах товара за определенный промежуток времени (по подразделениям), ценах на этот же товар у конкурентов, статистике об альтернативных товарах, взаимозаменяющих элементах и т.п.. Система должна выдавать отчеты по запросу менеджера: отчет о динамике продаж с графическим
16	Система учета заказов и их выполнение в строительной фирме (ремонт квартир)	Администрация строительной компании, занимающейся ремонтом квартир, заказала разработку информационной системы для отдела работы с клиентами. Система предназначена для обработки данных о клиентах и перечне услуг, а также учете заказов, используемом материале и учет затрат по заказам. Кроме того, в системе должна храниться база фотографий с образцами ремонта и в целом отремонтированных квартир. Система должна выдавать отчеты по запросу менеджера: прайс-лист
17	Предметная область оператора отделения связи (подписка на издания)	Руководство отделения связи федеральной почтовой службы заказало разработку информационной системы для отдела оформления подписки на периодические издания. Система предназначена для обработки данных о клиентах, изданиях, каталогах со стоимостью подписки (по разделам и тематике), а также услугах, оказываемых подписчикам. Система должна выдавать отчеты по запросу менеджера:

№	Предметная область	Описание предметной области
18	Предметная область «система подсчета голосов в избирательных компаниях»	Администрация города заказала разработку ИС для избиркома. Система предназначена для обработки данных об избирателях, о кандидатах, информация об избирательных участках. Система должна выдавать отчеты по запросу члена комиссии: бланк голосования, формирование итоговых протоколов по участкам, округам и городу.
19	Предметная область администратора ателье мод	Администрация ателье мод заказала разработку ИС. Система предназначена для обработки данных о клиентах, сроки выполнения заказов, информация об исполнителях, перечень услуг и их стоимость, учет затрат и заказов. Система должна выдавать отчеты по запросу администратора ателье мод: прайс-лист на
20	Обработка оборотных ведомостей	Система должна хранить и обновлять оборотные ведомости по материалам. Для различных материалов содержатся данные о цене, количестве и сумме. По цене и количеству необходимо иметь остатки на начало года или месяца, поступления от подотчетного лица и с центрального склада, выдачи

№	Предметная область	Описание предметной области
21	Предметная область бухгалтера расчетчика (задача начисления з/платы)	Система должна содержать информацию об учете заработной платы сотрудников предприятия. Для каждого лица в базе должны содержаться данные о профессии, должности, начислениях заработной платы, премиях, начислениях по больничному листу, задолженностям по выплатам на начало месяца. система должна содержать для каждого сотрудника информацию об удержании, включая налоги, алименты и сумму к выдаче. Выходная информация: ведомость на получение
22	Предметная область оператора кинотеатра (на примере кинотеатра	Администрация кинотеатра заказала разработку ИС. Система предназначена для обработки данных о времени проведения сеансов, включая названия фильмов, о количестве мест в зале, о ценах. Система должна выдавать отчеты по запросу оператора кинотеатра: расписание сеансов со
23	Предметная область «Расчет стоимости ПК из комплектующих »	Администрация магазина заказала разработку ИС. Система предназначена для расчета суммарной стоимости компьютера при известных ценах на комплектующие. Система должна обеспечивать продавцу выбор комплектующих из списка (с известными ценами). При расчете итоговой стоимости должна учитываться стоимость доставки (если она необходима). Так же система должна запрашивать ФИО и адрес клиента. Система должна выводить квитанцию на оплату с

№	Предметная область	Описание предметной области
24	Предметная область «Организация мероприятий и конференций»	Компания-организатор мероприятий заказала разработку ИС для автоматизации процесса подготовки и проведения конференций. Система должна позволять администраторам создавать мероприятия, устанавливать даты, определять бюджет и максимальное количество участников. Участники должны иметь возможность регистрироваться на мероприятие, выбирать интересующие их секции и мастер-классы из предложенного расписания. Система должна автоматически формировать индивидуальные программы для каждого участника, учитывать диетические предпочтения при формировании заявок на кейтеринг, а также генерировать
25	Предметная область «Учет и обслуживание книжного фонда библиотеки»	Центральная городская библиотека заказала разработку ИС для учета книжного фонда и автоматизации процессов обслуживания читателей. Система должна обеспечивать каталогизацию новых поступлений с возможностью поиска по автору, названию, жанру и ключевым словам. Читатели должны иметь возможность бронировать книги онлайн, продлевать срок пользования и просматривать историю взятых книг. Для библиотекарей система должна предусматривать учет выданных и возвращенных книг, формирование напоминаний о просроченных возвратах, автоматическое резервирование книг в очередь на

№	Предметная область	Описание предметной области
26	Предметная область «Управление задачами в IT-проектах»	IT-компания заказала разработку внутренней ИС для управления задачами в рамках проектов разработки ПО. Система должна позволять менеджерам проектов создавать проекты, разбивать их на задачи, назначать исполнителей и устанавливать сроки выполнения. Разработчики должны иметь возможность менять статус задач, добавлять комментарии, прикреплять файлы и вести учет затраченного времени. Система должна автоматически строить диаграммы Ганта, вычислять загрузку сотрудников, формировать отчеты о
27	Предметная область «Мониторинг состояния здоровья пациентов»	Медицинский центр заказал разработку ИС для дистанционного мониторинга состояния пациентов с хроническими заболеваниями. Система должна получать данные с медицинских устройств пациентов (глюкометры, тонометры, фитнес-трекеры) и сохранять их в электронных медицинских картах. Врачи должны иметь доступ к истории показателей, устанавливать целевые диапазоны и получать автоматические оповещения при выходе показателей за пределы нормы. Пациенты должны видеть графики своих

№	Предметная область	Описание предметной области
28	Предметная область «Учет аренды помещений в бизнес-центре»	Управляющая компания бизнес-центра заказала разработку ИС для учета аренды офисных помещений. Система должна вести базу данных арендаторов с историей платежей, автоматически формировать счета на оплату аренды и коммунальных услуг, отслеживать просроченные платежи и начислять пени. Дополнительно система должна управлять бронированием переговорных
29	Предметная область «Подбор персонала и управние вакансиями»	Кадровое агентство заказало разработку ИС для автоматизации процесса подбора персонала. Система должна позволять размещать вакансии на различных платформах, собирать и структурировать резюме кандидатов, автоматически определять соответствие кандидатов требованиям вакансий. Рекрутеры должны иметь возможность планировать собеседования, вести заметки по кандидатам и формировать предложения о работе. Система также должна строить аналитику по эффективности
30	Своя область	Требования к описанию предметной области приведены ниже

Шаблон для формулирования требований к любой предметной области:

1. **Основные пользователи и их роли**
2. **Ключевые бизнес-процессы**
3. **Требуемые функции системы**
4. **Входные данные системы**
5. **Выходные данные/отчеты**
6. **Интеграции с внешними системами**
7. **Требования к безопасности**
8. **Ограничения и условия**

Пример заполнения для области «Расчет стоимости ПК из комплектующих»:

1. **Пользователи:** Продавец, Администратор, Клиент
2. **Процессы:** Подбор комплектующих → Расчет стоимости → Оформление заказа → Формирование квитанции
3. **Функции:** Выбор из каталога, расчет итога, добавление доставки, ввод данных клиента
4. **Входные данные:** Цены на комплектующие, выбор пользователя, ФИО и адрес клиента
5. **Выходные данные:** Квитанция с детализацией и итоговой суммой
6. **Интеграции:** База данных товаров, система печати
7. **Безопасность:** Доступ по ролям, защита данных клиентов
8. **Ограничения:** Поддержка только наличного расчета, работа в offline-режиме

Составитель
Витвицкий Максим Николаевич

Методические указания по выполнению самостоятельной работы
для студентов очной формы обучения
по направлению специальности
09.02.07 «Информационные системы и программирование»

Публикуется в авторской редакции