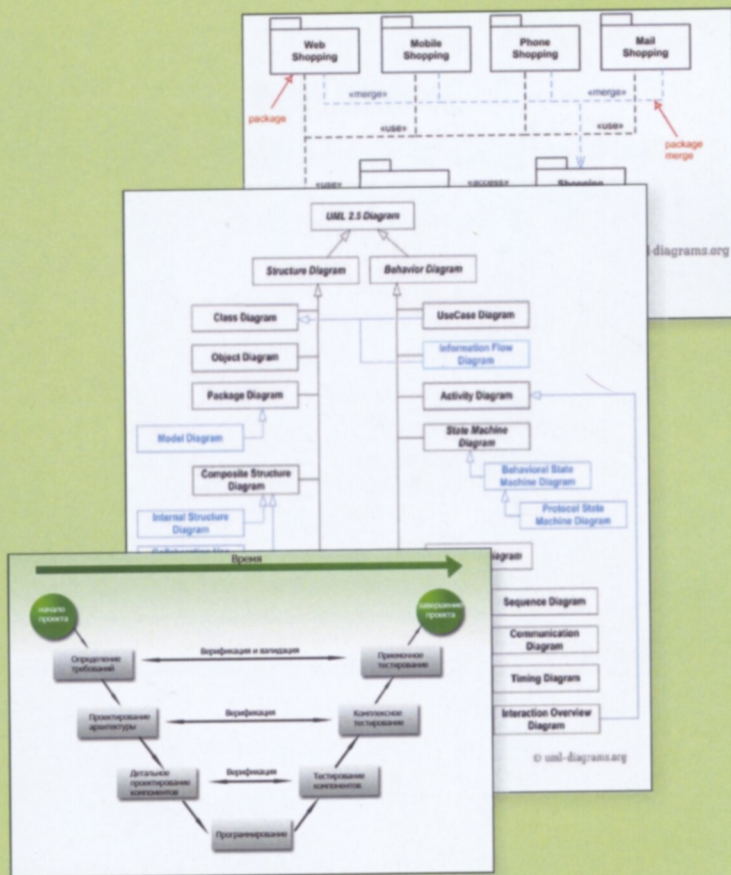


В.К.Волк

ВВЕДЕНИЕ  
В ПРОГРАММНУЮ  
ИНЖЕНЕРИЮ

Министерство образования и науки  
Российской Федерации

федеральное государственное бюджетное образовательное учреждение  
высшего образования  
«Курганский государственный университет»

В.К. Волк

# **ВВЕДЕНИЕ В ПРОГРАММНУЮ ИНЖЕНЕРИЮ**

УЧЕБНОЕ ПОСОБИЕ

Курган 2018

УДК 004.41(075.8)

ББК 32.973я73

В67

#### Рецензенты

кафедра информатики Уральского государственного горного университета (заведующий кафедрой, канд. техн. наук, доцент А.В. Дружинин);

кафедра прикладной информатики и автоматизации бизнес-процессов Шадринского государственного педагогического университета (заведующий кафедрой, канд. физ.-мат. наук, профессор В.Ю. Пирогов).

*Печатается по решению методического совета Курганского государственного университета.*

Научный редактор – канд. техн. наук, доцент Д.И. Дик

**Волк В. К.**

Введение в программную инженерию : учебное пособие. – Курган : Изд-во Курганского гос. ун-та, 2018. – 156 с.

В учебном пособии рассмотрены основные концепции программной инженерии: в первой главе обсуждаются базовые понятия, история и терминология программной инженерии; во второй главе – стандарты и модели жизненного цикла программного продукта, типовая ролевая модель команды программного проекта; основное содержание третьей главы – обзор средств визуального моделирования, используемых при структурном анализе и проектировании систем; четвертая глава содержит введение в язык UML, рассматриваемый как объектно-ориентированное средство графического моделирования и документирования программного проекта; завершающая глава пособия – проектный практикум, содержащий практические задания и примеры разработки UML-моделей на различных этапах выполнения учебного программного проекта.

Пособие предназначено для студентов младших курсов IT-специальностей и может быть рекомендовано широкому кругу читателей для начального ознакомления с проблематикой программной инженерии и технологиями проектирования программного обеспечения.

Рисунков – 54, таблиц – 6.

УДК 004. 41(075.8)

ББК 32.973я73

ISBN 978-5-4217-0452-2

© Курганский  
государственный университет, 2018  
© Волк В.К., 2018

# СОДЕРЖАНИЕ

<b>ПРЕДИСЛОВИЕ</b> .....	5
<b>ГЛАВА 1. ПРОГРАММНАЯ ИНЖЕНЕРИЯ: ИСТОРИЯ И БАЗОВЫЕ ПОНЯТИЯ</b> .....	8
1.1 История становления программной инженерии .....	8
1.1.1 Модульное программирование .....	9
1.1.2 Структурное проектирование и программирование.....	9
1.1.3 Объектно-ориентированный подход .....	11
1.2 Предмет программной инженерии .....	15
1.3 Методология программной инженерии .....	22
1.3.1 Проектирование как процесс преобразования моделей ПО .....	22
1.3.2 Что такое CASE ? .....	25
1.3.3 Свойства хорошей программной системы.....	26
<b>ГЛАВА 2. ПРОЦЕССЫ И МОДЕЛИ ЖИЗНЕННОГО ЦИКЛА</b> <b>ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ</b> .....	29
2.1 Понятие жизненного цикла промышленного изделия .....	29
2.2 Краткая история стандартизации жизненного цикла ПО .....	30
2.3 Терминология программной инженерии .....	31
2.4 Процессы жизненного цикла программного продукта .....	35
2.5 Модели жизненного цикла ПО .....	38
2.5.1 Типовые стадии процесса создания ПО.....	38
2.5.2 Каскадная модель .....	40
2.5.3 V-модель.....	42
2.5.4 Модель формальной разработки.....	44
2.5.5 Эволюционная модель .....	45
2.5.6 Спиральная модель.....	47
2.6 Ролевая модель команды программного проекта .....	49
<b>ГЛАВА 3. ВИЗУАЛЬНОЕ МОДЕЛИРОВАНИЕ В ПРОЕКТИРОВАНИИ</b> .....	54
3.1 Задачи и базовые принципы моделирования программных систем.....	54
3.2 Визуализация при моделировании сложных систем.....	55
3.3 Краткая история развития средств визуального моделирования.....	57
3.3.1 Средства визуализации математических моделей.....	57
3.3.2 Семантические сети .....	58
3.3.3 Диаграммы структурного анализа систем .....	59
<b>ГЛАВА 4. UML – УНИВЕРСАЛЬНЫЙ ЯЗЫК МОДЕЛИРОВАНИЯ</b> .....	64
4.1 История стандартизации языка UML.....	64
4.2 Структура и базовые понятия языка UML .....	65
4.3 Элементы моделей языка UML .....	66
4.4 Диаграммы языка UML .....	70

4.5 Общие правила графической нотации UML-диаграмм .....	72
4.6 UML-диаграмма вариантов использования.....	75
4.6.1 Компоненты UseCase-диаграммы.....	76
4.6.2 Пример UseCase-диаграммы .....	80
4.6.3 Сценарии вариантов использования .....	82
4.7 UML-диаграмма пакетов .....	86
4.8 UML-диаграмма классов .....	90
4.8.1 Классы .....	90
4.8.2 Отношения между классами .....	97
4.8.3 Интерфейсы.....	104
4.9 UML-диаграмма состояний.....	106
4.9.1 Состояния .....	107
4.9.2 Простые переходы.....	109
4.9.3 Составные состояния .....	111
4.9.4 Параллельные переходы.....	113
4.9.5 Переходы в составных состояниях.....	113
<b>ГЛАВА 5. ПРОЕКТНЫЙ ПРАКТИКУМ.....</b>	<b>116</b>
5.1 Общие методические указания.....	116
5.2 Практические задания.....	117
5.3 Содержание семинарских занятий .....	118
5.4 Пример выполнения учебного программного проекта .....	119
5.5 Рекомендуемая тематика учебных программных проектов .....	142
<b>СПИСОК ЛИТЕРАТУРЫ .....</b>	<b>143</b>
<b>ПРИЛОЖЕНИЯ .....</b>	<b>144</b>
ПРИЛОЖЕНИЕ А. ПРОФЕССИОНАЛЬНЫЕ ИТ-СТАНДАРТЫ РФ .....	144
ПРИЛОЖЕНИЕ Б. ОБРАЗОВАТЕЛЬНЫЕ ИТ-СТАНДАРТЫ .....	145
ПРИЛОЖЕНИЕ В. ПЕРЕЧЕНЬ СТАНДАРТОВ КАЧЕСТВА ПО .....	146
ПРИЛОЖЕНИЕ Г. ЗАДАНИЯ НА ВЫПОЛНЕНИЕ УЧЕБНЫХ ПРОЕКТОВ .....	148

## ПРЕДИСЛОВИЕ

Программное обеспечение (ПО) играет важную роль практически во всех отраслях экономики, а также во многих областях повседневной жизни человека. За последние десятилетия область применения ПО существенно расширилась, и масштабы его использования позволяют говорить не только о «цифровой экономике», но и в целом о «цифровом обществе», при этом существенно возрастает и уровень сложности разрабатываемого ПО, а достижение его приемлемой стоимости, разумных сроков разработки и высокого качества и сегодня остается серьезной проблемой.

Проблема высокой технологической сложности программных систем была отмечена еще в конце 60-х годов прошлого века, когда стоимость ПО сравнивалась со стоимостью аппаратного комплекса. При этом прогнозы были неутешительны: они, в частности, показывали, что при сохранении достигнутого к тому времени уровня производительности разработки ПО уже к середине 90-х годов программированием должно было бы заниматься все население планеты.

Именно к этому периоду относят зарождение *программной инженерии (software engineering)*, которая ассоциируется с переходом от программирования, как искусства или высокоинтеллектуального ремесла, к технологиям массового производства ПО, рассматривающим программный продукт как промышленное изделие, к которому следует применять классическое понятие «жизненного цикла». При этом жизненный цикл программного продукта включает программирование лишь как один из многих этапов его производства, наряду с проектированием, испытанием, внедрением, обслуживанием, эксплуатацией, модернизацией и утилизацией.

Сегодня программная инженерия – мощная и динамично развивающаяся индустриальная отрасль со своей технологией, инструментальным обеспечением и системой стандартов качества, а профессия IT-специалиста стала одной из массовых и востребованных на рынке трудовых ресурсов. Внедрение промышленных технологий в сферу программной инженерии привело к необходимости разделения труда: в софтверных компаниях появились должности бизнес-аналитиков и системных архитекторов, менеджеров программных проектов и программных продуктов, разработчиков и тестировщиков, специалистов по эксплуатации и сопровождению программного обеспечения. Были стандартизованы требования к составу тру-

довых функций и квалификации IT-специалистов разных профилей, для их подготовки разработаны и успешно реализуются образовательные программы различных уровней.

Проблематика и методология программной инженерии отражены в многочисленных научных публикациях и монографиях. В ставших уже классическими работах Гради Буча (с соавторами) [3; 13] и Иана Sommerville [11] исчерпывающе рассмотрена тематика инженерии ПО, охватывающая все этапы технологии разработки программных систем. В работах А. Леоненкова [6], Ф. Новикова и Д. Иванова [9] рассмотрена технология проектирования программных систем с использованием языка UML.

Имеются также и учебные издания по тематике программной инженерии, например, работы С.Н. Карпенко [5], В.В. Липаева [7] и А.А. Мирютова [8]. Следует отметить, что все перечисленные учебные пособия (так же, впрочем, как и упомянутые выше и многие другие монографии) рассчитаны на профессионально подготовленного читателя и рекомендованы их авторами системным архитекторам, руководителям программных проектов и опытным программистам, а также магистрантам и аспирантам IT-специальностей.

В предлагаемом учебном пособии автор ставит (и пытается решить) существенно более скромную задачу, а именно – дать студентам 2-го курса представление о профессиональной терминологии, проблематике, методологии и стандартизации в программной инженерии, ознакомить их с корпоративными технологиями проектирования программных систем и с CASE-средствами, поддерживающими процедуры UML-моделирования – то есть заложить основу, необходимую для изучения специальных дисциплин, формирующих базовый набор профессиональных компетенций программных инженеров.

При подготовке учебного пособия автор учитывал личный опыт преподавания основ программной инженерии студентам IT-специальностей. В первой и второй главах учебного пособия рассматриваются концепции и стандарты программной инженерии, при написании этих глав автором активно использовались материалы работ [4; 5; 8; 11]. В третьей главе приведен краткий обзор методов и средств графического моделирования, используемых для структурного анализа систем. Четвертая глава содержит практическое введение в язык UML, примеры и методические рекомендации по моделированию и разработке UML-диаграмм. Материал этой гла-

вы может быть использован для самостоятельного освоения CASE-средств графического моделирования. Пятая глава учебного пособия – это проектный практикум, предусматривающий разработку UML-моделей в процессе реализации учебного программного проекта.

Использование пособия не требует предварительной подготовки в области технологий разработки ПО, но предполагается, что студенты знакомы с основными концепциями объектно-ориентированного программирования и имеют практический опыт разработки несложных компьютерных программ.

Автор выражает благодарность канд. техн. наук Дмитрию Ивановичу Дику, внимательно прочитавшему рукопись и сделавшему ряд профессиональных комментариев и полезных замечаний по ее содержанию, несомненно способствовавших повышению качества учебного пособия.



### ПРОГРАММНАЯ ИНЖЕНЕРИЯ: ИСТОРИЯ И БАЗОВЫЕ ПОНЯТИЯ

*Предпосылки и история: модульное программирование; структурное проектирование и программирование; объектно-ориентированный подход к анализу, проектированию и программированию. Предмет программной инженерии: инженеры и инженерная деятельность; программное обеспечение и программный продукт; программная инженерия и системная инженерия; программная инженерия и информатика; отличия программной инженерии от традиционных инженерных отраслей. Методология программной инженерии.*

#### **1.1 История становления программной инженерии**

Компьютерное программирование как область профессиональной деятельности зародилось относительно недавно – в середине 40-х годов прошлого столетия, когда были созданы первые электронно-вычислительные машины.

Первые компьютерные программы были ориентированы на решение математически и алгоритмически сложных задач, требующих высокой точности и скорости вычислений, а первыми программистами (так же, впрочем, как и первыми пользователями компьютерных программ) были математики, инженеры и научные работники.

50-е и 60-е годы – период бурного развития вычислительной техники и накопления опыта ее применения. В этот период были написаны монографии в области алгоритмизации, созданы языки программирования высокого уровня, разработаны концепции операционных систем. Программирование еще не оформилось в технологию и являлось, скорее, искусством и высокоинтеллектуальным ремеслом<sup>1</sup>.

Программная инженерия ассоциируется с созданием сложных программных комплексов коллективами из десятков и сотен разработчиков. Становление программной инженерии было связано с осознанием профессиональным сообществом ряда технологических и управленческих про-

---

<sup>1</sup> Ряд специалистов, например К. Бюрер [4], считают, что и сегодня программная инженерия все еще остается ремеслом, которому далеко до настоящей инженерной дисциплины, и главная причина такого отставания – в отсутствии базовых принципов и научно-обоснованных методов программной инженерии.

блем: высокой стоимости ПО, сложностью процессов его создания и последующего сопровождения, необходимостью управления процессами разработки и прогнозирования их результатов.

Термин *software engineering* впервые был озвучен в 1968 году на конференции подкомитета НАТО по науке и технике, рассматривавшей проблемы проектирования и поддержки компьютерных программ. Там впервые было официально заявлено о надвигающемся кризисе ПО и необходимости реализации *инженерного подхода* к разработке ПО для преодоления этого кризиса.

Основу инженерного подхода составляет концепция *жизненного цикла ПО* как последовательности стадий планирования, проектирования, производства и эксплуатации, каждая из которых предполагает применение соответствующих *методов, технологий и инструментов*.

### ***1.1.1 Модульное программирование***

С развитием рынка программного обеспечения программирование приняло массовый характер, и разработчиками была отмечена одна важная особенность компьютерных программ и выявлена одна из причин их высокой стоимости: при создании различных программ приходилось разрабатывать одинаковые (или алгоритмически подобные) фрагменты кода, реализующие типовые операции, такие, например, как численное решение нелинейных уравнений, прочностные расчеты элементарных строительных конструкций, расчет заработной платы сотрудников предприятий и т.д.

Технология модульного программирования обеспечивает возможность *повторного использования ранее написанных фрагментов программного кода*, что существенно снижает сроки и стоимость разработки ПО. Главный принцип модульного программирования заключается в выделении таких фрагментов и оформлении их в виде модулей. При этом каждый модуль снабжается интерфейсом – описанием, в котором устанавливаются правила использования модуля и его связи (по данным и управлению) с основной программой.

### ***1.1.2 Структурное проектирование и программирование***

Появление вычислительной техники третьего поколения (интегральные схемы), обеспечившей существенное повышение производительности вычислений, создало предпосылки для создания сложных программных

комплексов, характеризующихся большим объемом программного кода (миллионы строк) и количеством связей между его элементами, большим числом разработчиков (сотни человек) и пользователей (сотни и тысячи), а также длительным временем эксплуатации ПО.

Увеличение объема и усложнение структуры программного кода повышает риск появления ошибок при его разработке, а наличие большого количества участников проекта затрудняет процесс коммуникации между ними и повышает требования к документированию проекта. Длительный срок эксплуатации ПО неизбежно приводит к необходимости модификации программного кода в соответствии с изменяющимися требованиями пользователей.

В этих условиях затраты на внедрение и последующее сопровождение программных комплексов стали превышать стоимость их разработки, а основная причина высокой стоимости сопровождения заключалась в плохой структурированности программ и низком качестве документации, часто не соответствующей программному коду.

В результате был предложен *структурный подход к проектированию и программированию*, который базировался на следующих основных принципах.

*Нисходящее функциональное проектирование*, при котором проектируемая система иерархически декомпозируется на функциональные подсистемы, компоненты, модули и т.д. Разработка системы ведется «сверху вниз», начиная с формирования ее общей архитектуры и решения задач концептуального характера на верхнем уровне иерархии и заканчивая программной реализацией низкоуровневых компонентов.

*Применение специальных языков проектирования* с развитой графической нотацией и поддерживающих эти языки CASE-средств (Computer Aided Soft Engineering – средства автоматизации разработки ПО), обеспечивающих снижение трудоемкости выполнения проектных работ и поддержку процессов формирования и корректировки проектной документации.

*Согласованность процессов проектирования и программирования*: планирование и последовательное документирование всех стадий проекта, поддержка соответствия программного кода проектной документации.

*Дисциплина структурного программирования*:

- использование трех видов языковых конструкций: «последовательность», «ветвление» и «цикл»;

- каждая конструкция должна иметь один вход и один выход (что фактически исключает использование «меток» и операторов *goto*);
- повторяющиеся фрагменты последовательных конструкций должны быть оформлены в виде подпрограмм;
- управляющие конструкции могут быть вложенными друг в друга.

### ***1.1.3 Объектно-ориентированный подход***

Дальнейшее расширение сферы применения ПО (параллельные вычисления, распределенные системы хранения и обработки данных, задачи управления в реальном времени, многопользовательские информационные системы и др.) в условиях динамично развивающегося бизнеса потребовало повышения гибкости и мобильности процессов разработки: заказчик вносил *изменения в требования* к ПО не только на стадии его сопровождения, но и *на стадии проектирования и программирования*. В этих условиях создание программного продукта превращается в его перманентное перепроектирование и перепрограммирование, и потребовалась методология, обеспечивающая возможность оперативного внесения изменений в программу без существенных изменений ранее написанного кода.

В результате была предложена *объектно-ориентированная* методология, в которой модульный и структурный подходы получили дальнейшее развитие.

#### ***Объектная модель***

Традиционный структурный подход предполагает проведение *алгоритмической декомпозиции* проектируемой системы, в результате которой решение задачи представляется в виде последовательности вызовов функций, реализующих определенные алгоритмы. При объектно-ориентированном подходе проводится *объектная декомпозиция* системы, и ее архитектура представляется в виде набора взаимосвязанных объектов, взаимодействующих друг с другом путем передачи сообщений и/или предоставления ресурсов. *«Вместо процессора, беззастенчиво перемалывающего структуры данных, мы получаем сообщество хорошо воспитанных объектов, которые вежливо просят друг друга об услугах»* [9].

Объектно-ориентированный подход базируется на специфической *объектной модели*, в основе которой – понятие *класса*. Класс наделяется свойствами (атрибутами) и поведением (методами), характеризующими

роль этого класса в программной системе. Класс может порождать *объекты* – экземпляры данного класса, для каждого из которых определены *значения атрибутов*. Объекты разных классов могут кооперироваться, устанавливая *связи* друг с другом и получая при этом возможность обмениваться сообщениями или оказывать услуги другим объектам, предоставляя им доступ к своим атрибутам и методам.

В соответствии с объектной моделью программная система представляется как множество взаимосвязанных классов, которые могут объединяться в подсистемы, связанные с другими подсистемами.

Состояние программной системы определяется состояниями всех ее объектов, а функционирование системы рассматривается как процесс ее перехода из начального состояния во все последующие в результате взаимодействия объектов. При этом набор значений атрибутов объекта определяет его текущее состояние, попав в которое объект реализует определенное поведение и в результате переходит в другие состояния.

### ***Базовые принципы объектно-ориентированной методологии***

Объектно-ориентированный подход унаследовал от своих предшественников и частично модифицировал принципы абстрагирования, модульности и иерархичности и ввел в обращение три новых, тесно взаимосвязанных принципа – *инкапсуляция, наследование и полиморфизм*.

***Инкапсуляция*** позволяет объединить в едином классе *данные* (атрибуты), определяющие состояние его объектов, и *методы*, определяющие поведение объектов и изменяющие значения атрибутов, и при этом обеспечивает сокрытие от классов-клиентов информации о внутренней структуре и деталях реализации методов класса-сервера, не влияющей на его внешнее поведение.

Хорошая иллюстрация понятия инкапсуляции приведена в [9] на примере из живой природы. Пусть в модели реального растения определен класс *Растение* с атрибутом *размер*. В реальности можно воздействовать на размер растения, изменяя тепловой режим, интенсивность полива и уровень освещенности, однако вряд ли возможно внешним воздействием заставить растение мгновенно подрасти на один или два метра, так как только самому растению известен механизм изменения своего размера.

Следуя реалиям природы, в модели растения необходимо скрыть от внешних объектов атрибут *размер* класса *Растение* и запретить его явное изменение, но при этом создать в этом классе метод *УправлениеРостом*

(*тепло, вода, свет*), сделав его доступным для объектов классов-клиентов. Заметим, что при таком подходе появляется возможность абстрагироваться и от алгоритма реализации метода, возвращающего значение размера растения в зависимости от значений входных параметров метода.

Инкапсуляция, абстрагируясь от внутреннего содержания классов, упрощает понимание сложных систем в процессе их проектирования, а также позволяет модифицировать содержимое классов-серверов, не затрагивая при этом содержимого классов-клиентов.

**Наследование** – это отношение типа «предок-потомок», позволяющее классам-потомкам (называемым *подчиненными* или *дочерними* классами) наследовать некоторые свойства (атрибуты) и поведение (методы) от своих классов-предков (суперклассов). Наследование порождает иерархию типа «общее – частное», в которой подкласс представляет частный случай своего суперкласса. Суперкласс может иметь множество подклассов, и при этом допускается множественное наследование, когда класс-потомок имеет более одного класса-предка.

Наследование позволяет создавать системы с весьма сложной не-иерархической структурой, которая при этом остается гибкой и легко модифицируемой. Например, для добавления новой функции в систему, находящуюся в стадии проектирования или программной реализации, достаточно дополнить ее объектную модель новым классом-потомком, что (в простых случаях) не потребует внесения изменений ни в класс, являющийся предком дополнительного класса, ни в другие, разработанные ранее, классы-потомки этого суперкласса.

**Полиморфизм** тесно связан с абстрагированием, типизацией и наследованием и позволяет по-разному оперировать одноименными объектами разных подклассов одного суперкласса. При этом объекты, обозначенные полиморфными именами, могут по-своему реагировать на одинаковые операции в зависимости от контекста их выполнения.

Полиморфизм позволяет нескольким классам, являющимся потомками одного суперкласса, иметь одинаково поименованные методы, которые будут иметь разные реализации. С другой стороны, объекты суперкласса могут использовать методы своих подклассов, абстрагируясь от деталей их реализации.

Пусть, например, имеется класс *Window*, представляющий окно графического редактора, и связанный (агрегированный) с ним класс

*GraphicObject*, представляющий геометрические фигуры, отображаемые в окне графического редактора. С суперклассом *GraphicObject* связаны два класса-потомка – *Triangle* и *Circle*, представляющие, соответственно, треугольник и круг.

Класс *Window* содержит метод *redraw()*, выполняющий операцию перерисовки содержимого окна редактора путем вызова метода *paint()* класса *GraphicObject*, который (как "думает" класс *Window*) рисует соответствующую фигуру, однако, метод *paint()* класса *GraphicObject* ничего конкретно не рисует. Так как алгоритмы отображения круга и треугольника различны, каждый из подклассов *Circle* и *Triangle* содержит свой метод рисования, при этом оба эти метода должны иметь такое же имя *paint()*, что и в их суперклассе.

Класс *Window* не знает о существовании классов *Circle* и *Triangle*, и когда работает метод *redraw()* класса *Window*, он знает только, что нужно перерисовать какой-то графический объект, а это обязательно потомок класса *GraphicObject*. Поэтому *redraw()* просто вызывает метод *paint()* для этого объекта, не заботясь больше ни о чем. Механизм полиморфизма позволяет определить нужный метод *paint()* (для круга или треугольника) – туда и передаются данные о конкретном графическом объекте, который должен быть перерисован.

Завершая краткий обзор истории становления программной инженерии, отметим ее основные достижения.

1 Программная инженерия формировалась под давлением роста сложности и стоимости создаваемого ПО, последовательно преодолевая в своем развитии кризисы программирования и вырабатывая при этом эффективные для своего времени принципы, подходы, методы и технологии.

2 Основным достижением программной инженерии является осознание того факта, что программный продукт, являясь по своей природе нематериальным, технологически подобен продуктам материального производства и должен создаваться в результате выполнения нескольких взаимосвязанных этапов, составляющих его жизненный цикл.

3 В процессе развития программной инженерии были сформулированы фундаментальные принципы, выработаны и апробированы методы, созданы технологии и инструментальные средства автоматизированной разработки программных продуктов на основе модульного, структурного

и объектно-ориентированного подходов к проектированию и программированию.

4 Несмотря на то, что программная инженерия достигла определенных успехов, кризис программирования продолжается, и есть опасения, что он принимает хронический характер. По имеющимся статистическим данным до 30% программных проектов закрываются, так и не завершившись, а около 50% проектов завершаются с превышением бюджета и ограниченной функциональностью результатов разработки. Основными причинами неудач программных проектов признаются размытые границы, нереалистичные временные рамки и недофинансирование проектов, а также недостаточное количество квалифицированных исполнителей.

5 Кризисная ситуация в программной инженерии вызвана вполне объективными причинами: развитие средств вычислительной техники и телекоммуникационного оборудования привело к взрывному росту потребности в применении компьютеризированных систем практически во всех отраслях экономики и общественных отношений. В этих условиях потребовались действительно гибкие и постоянно развивающиеся технологии производства программного обеспечения, а также, что немаловажно – потребовалось большое количество квалифицированных программных инженеров, профессионально владеющих такими технологиями.

## ***1.2 Предмет программной инженерии***

В условиях отсутствия единой трактовки понятия «программная инженерия» приведем три альтернативных определения, данные специалистами в этой области и отражающие некоторые ее существенные черты.

Определение 1. Программная инженерия – это установление и использование обоснованных инженерных принципов (методов) для экономного получения ПО, которое надежно и работает на реальных машинах.

Определение 2. Программная инженерия – это та форма инженерии, которая применяет принципы информатики (the computer science) и математики для рентабельного решения проблем ПО.

Определение 3. Программная инженерия – это применение систематического, дисциплинированного, измеряемого подхода к разработке, использованию и сопровождению ПО.



Так чем же занимается программная инженерия? Для того, чтобы получить более детальное представление о предмете программной инженерии, потребуется получить ответы на следующие вопросы:

- 1) что такое *инженерия* (*engineering*) и чем занимаются инженеры?
- 2) что такое *программное обеспечение* (*software*)?
- 3) чем *программная инженерия* отличается от *информатики*?
- 4) в чем отличие *программной инженерии* от других инженерий?
- 5) каковы *методы программной инженерии*?
- 6) что такое *CASE* (*Computer-Aided Software Engineering*)?
- 7) каковы свойства *хорошей компьютерной программы*?

Итак, что такое **инженерия** и чем занимаются инженеры ?

Слово **инженер** (от англ. *engine* – «орудие», «машина») имеет французское происхождение (*ingénieur*) и первоначально использовалось для обозначения военных специалистов по артиллерийскому вооружению. Позднее этим термином стали обозначать широкий круг специалистов, деятельность которых связана с проектированием, производством или эксплуатацией сложных технических устройств. Заметим, что ранее в России вместо французского «*ingénieur*» использовалось русское слово «*розмысл*», недвусмысленно указывающее на интеллектуальный характер инженерной деятельности.

**Инженер – это специалист, который** выполняет практическую работу и **добивается практических результатов**, в отличие, например, от ученого, который может сказать: «*проблема неразрешима в рамках существующих теорий*» – и это будет научный результат, достойный опубликования или защиты научной диссертации.

Для решения задачи инженеры применяют теории, методы и средства, пригодные для решения данной задачи, но они применяют их выборочно и всегда пытаются найти решения даже в тех случаях, когда теорий или методов, соответствующих данной задаче, еще не существует. Такие инженерные методы, возможно, теоретически не обоснованные, но получившие неоднократное подтверждение на практике, в программной инженерии получили название *профессиональных практик* (*best practices*).

Инженер – это специалист, который *«знает, как»* (*know how*) в своей профессиональной области. Сегодня существует множество инженерных профессий, специальностей и специализаций, таких, например, как *инженер-механик*, *инженер-строитель*, *инженер-электрик* или *инженер-*

*программист*. При всем разнообразии инженерных областей существует ряд объединяющих их базовых характеристик, которые и определяют инженерию как таковую.

***Инженерная деятельность неразрывно связана с необходимостью принятия решений*** и выбора подходов, оптимально соответствующих решаемой задаче с учетом существующего контекста, в том числе и с учетом предполагаемых затрат и ожидаемой прибыли.

***Инженеры используют измеримые количественные характеристики*** и при необходимости выдают приближенные решения на основе опыта и эмпирических данных.

***Инженеры используют дисциплинированные процессы*** при реализации проектов и понимают важность эффективной организации работы.

***Инженеры решают широкий спектр задач***, включая исследования, проектирование, производство и тестирование, внедрение, эксплуатацию и управление, а также консультирование и обучение, и при этом ***несут ответственность за результаты их выполнения***.

***Инженеры в своей деятельности используют инструментальные средства***, выбор и использование которых является крайне важным вопросом.

***Инженеры объединяются в профессиональные сообщества*** и способствуют развитию своей отрасли путем разработки и внедрения рекомендаций, аттестационных принципов, стандартов, распространению хорошо зарекомендовавших себя профессиональных практик (*best practices*).

***Инженеры повторно используют полученные ранее результаты***.

Что такое ***программное обеспечение*** ?

***Программное обеспечение*** определяется как ***набор компьютерных программ, связанных с ними данных и документации***.

Программное обеспечение – это не только компьютерная программа, но еще и данные, необходимые для ее корректной работы (например, конфигурационные файлы или база данных), и документация (например, руководство по установке и эксплуатации программы). Поэтому вместо термина ***программное обеспечение*** часто используют термин ***программный продукт***, подчеркивая этим его ориентацию ***на продажу потребителю***.

Различают два типа программных продуктов: ***коробочные*** и ***заказные***, в зависимости от того, для кого они разрабатываются – для конкретного заказчика или для свободного рынка.

**Коробочные продукты** (*shrink-wrapped software* – «упакованное» ПО) ориентированы на большие тиражи и широкий круг потребителей. Постановка задачи и спецификация требований, а также финансирование проекта осуществляется самим разработчиком, он же рискует в случае коммерческой неудачи. Стоимость создания коробочного ПО проецируется на весь его тираж, и поэтому цена одного экземпляра продукта, как правило, невысока.

**Заказные продукты** (*bespoke* – «сделанный на заказ» или *customized products* – «настроенный продукт») разрабатываются в соответствии с требованиями, сформированными конкретным заказчиком, который финансирует проект и рискует тем, что разработчик не сможет реально выполнить все требования в срок при выделенном бюджете. Заказные продукты массово не копируются, и цена экземпляра остается весьма высокой.

Что такое **программная инженерия** ?

**Программная инженерия** – это инженерная отрасль, которая связана со всеми аспектами производства и эксплуатации ПО: от создания внешних спецификаций программного проекта до поддержки системы после передачи ее заказчику вплоть до вывода из эксплуатации.

Программные инженеры работают в условиях ограниченных ресурсов: временных, финансовых и организационных. Программный продукт должен быть создан в установленные сроки, в рамках выделенных средств, оборудования и имеющимся составом исполнителей. Хотя это в первую очередь относится к созданию заказных продуктов, но и при создании коробочных продуктов эти ограничения имеют не меньшее значение, т.к. здесь они диктуются условиями рыночной конкуренции.

Программная инженерия занимается не только техническими вопросами производства ПО (специфицирование требований, проектирование, кодирование, тестирование, документирование и др.), но и управлением программными проектами (планирование, финансирование, управление коллективом разработчиков), и сопровождением ПО в процессе эксплуатации (обучение пользователей, обновление, модификация, мониторинг и администрирование). Кроме того, задачей программной инженерии является разработка средств, методов и теорий для поддержки процесса производства ПО.

## ***Программная инженерия и информатика***

Информатика (*the computer science*) занимается теорией и методологией вычислительных и программных систем, в то время как программная инженерия занимается практическими проблемами создания ПО. Информатика (наряду с дискретной математикой) составляет теоретическую основу программной инженерии, и программный инженер должен знать информатику (так же, как, например, инженер по электронике должен знать физику).

Однако теоретический фундамент программной инженерии составляют не только информатика и математика, так как круг проблем, стоящих перед программными инженерами, значительно шире, чем просто написание программ. Это еще и управление проектами, организация работ в коллективе и т.д. Решение этих проблем требует от программного инженера фундаментальных знаний, далеко выходящих за рамки информатики и математики.

Особо следует отметить еще одну проблему, которую приходится решать программному инженеру – это проблема взаимодействия с заказчиком и пользователями программного продукта на различных стадиях проекта. Качественное решение этой проблемы потребует от программного инженера, как минимум, хорошей эрудиции в различных областях человеческой деятельности – ведь сфера применения программного обеспечения практически ничем не ограничена.

## ***Отличия программной инженерии от других инженерных отраслей***

Жизненный цикл продукта любой инженерии включает типовой набор фаз: *проектирование – создание образца – испытания – производство – эксплуатация – утилизация*. С этой точки зрения программная инженерия ничем не отличается от других инженерных отраслей, и было бы заманчиво максимально использовать в программной инженерии элементы методологии, технологии проектирования и производства, разработанные, апробированные и широко применяемые в других технических областях.

Однако имеются и существенные отличия программной инженерии от других инженерных отраслей, ограничивающие возможности широкого использования накопленного в других областях опыта. Рассмотрим ос-

новные из таких отличий, позволяющие, в частности, получить ответ на вопрос, «почему так велика доля провальных IT-проектов»?

Во-первых, **компьютерная программа не является материальным объектом**, что определяет специфичное распределение стоимости реализации фаз жизненного цикла ПО:

- стоимость *фазы производства* такого объекта *исчезающе мала*, так как производство программы сводится к копированию образца с одного носителя на другие;

- *отсутствует* также и *фаза создания образца*, который формируется компилятором автоматически на завершающем этапе проектирования (если кодирование считать элементом детального проектирования, что очень близко к истине);

- отсюда следует вывод о том, что *стоимость проектирования ПО – это основной элемент его себестоимости*.

Во-вторых, **компьютерная программа – это искусственный объект**, поведение которого не подчиняется объективным законам природы, в отличие, например, от объектов строительной инженерии. Инженер-строитель использует объективные законы строительной механики и может проверить свои технические решения на соответствие этим законам, которые объективны, будут действовать всегда и не зависят от принимаемых им решений.

На первый взгляд у программного инженера также есть типовые, проверенные временем технические решения (например, клиент-серверная архитектура программных комплексов или реляционная модель базы данных), но эти решения определяются уровнем развития вычислительной техники, и с появлением техники с принципиально новыми возможностями программному инженеру придется искать новые решения.

Прямым следствием отсутствия возможности теоретического контроля качества программного проекта является то, что **тестирование продукта – основной (а иногда и единственный) способ убедиться в его работоспособности**. Именно поэтому стоимость тестирования составляет существенную часть стоимости ПО.

Структура затрат на создание ПО существенно зависит от типа ПО и применяемых методов его разработки. Приблизительная схема распреде-

ления затрат между основными фазами проекта (без учета фазы сопровождения):

- **15% – спецификация** (формулировка требований и условий разработки);

- **25% – проектирование** (разработка и верификация проекта);

- **20% – разработка** (кодирование и тестирование компонентов);

- **40% – интеграция** (сборка и тестирование).

Для коробочного ПО характерна более высокая доля стоимости тестирования за счет сокращения (до 5%) доли стоимости спецификации.

Третье существенное отличие заключается в том, что **программная инженерия – молодая дисциплина**, опыт которой насчитывает всего несколько десятков лет. По сравнению с «возрастом» строительной инженерии, насчитывающим не одно тысячелетие, это, конечно, очень мало. Программную инженерию иногда сравнивают с ранней строительной, когда законы строительной механики еще не были известны, и строительные инженеры действовали методом проб и ошибок, накапливая и передавая потомкам свой бесценный опыт.

Завершая обзор отличительных особенностей программной инженерии, следует отметить, что основное ее качественное отличие от других инженерных отраслей определяется нематериальным характером ее основного продукта (программного обеспечения) и дискретной природой его функционирования:

- программная инженерия имеет дело с абстрактными (логическими) объектами, а не с материальными (физическими);

- теоретической основой программной инженерии являются информатика и дискретная математика, а не естественные науки;

- в программной инженерии практически отсутствует фаза производства ее основного продукта, и основной вклад в его стоимость вносят фазы проектирования и программирования;

- фаза сопровождения программного продукта связана с продолжающейся его разработкой (эволюцией), а не с традиционным ремонтом и обслуживанием по причине физического износа;

- программная инженерия, будучи молодой дисциплиной, использует теории, методы и практический опыт, накопленный в сфере материального производства, адаптируя их к своим специфическим условиям.

Итак, *предмет программной инженерии – это процессы создания высококачественного программного обеспечения на всех уровнях – от теории и методологии до технологии и практических приемов.*

Круг вопросов, составляющих предмет программной инженерии, иллюстрируется рисунками 1.1 и 1.2, заимствованными из документации международного образовательного стандарта *IEEE SWEBOOK – Software Engineering Body of Knowledge (Свод знаний по программной инженерии)*.

Российские образовательные и профессиональные стандарты (приложения А и Б) в основном соответствуют требованиям международного стандарта SWEBOOK, схематично представленными на рисунках 1.1 и 1.2.

### **1.3 Методология программной инженерии**

*Методология программной инженерии – это совокупность принципов и способов организации деятельности проектной группы для создания качественного программного продукта при заданных ограничениях на основные ресурсы: время, бюджет, оборудование, количество и профессиональная компетентность исполнителей.*

#### **1.3.1 Проектирование как процесс преобразования моделей ПО**

Методология программной инженерии основана на идее<sup>2</sup> поэтапного преобразования моделей ПО в компьютерную программу – окончательную модель решаемой задачи. Так, на этапе спецификаций создается модель – описание требований, которая далее преобразуется в модель проекта ПО, а модель проекта – в программный код. На каждом этапе преобразования модели она становится более информативной, а степень детализации конечной модели должна быть достаточной для изготовления объекта.

---

<sup>2</sup> Эта идея не нова и заимствована у традиционных инженерных дисциплин – принцип поэтапной обработки больших объемов информации давно применяется в технологиях проектирования сложных технических объектов.

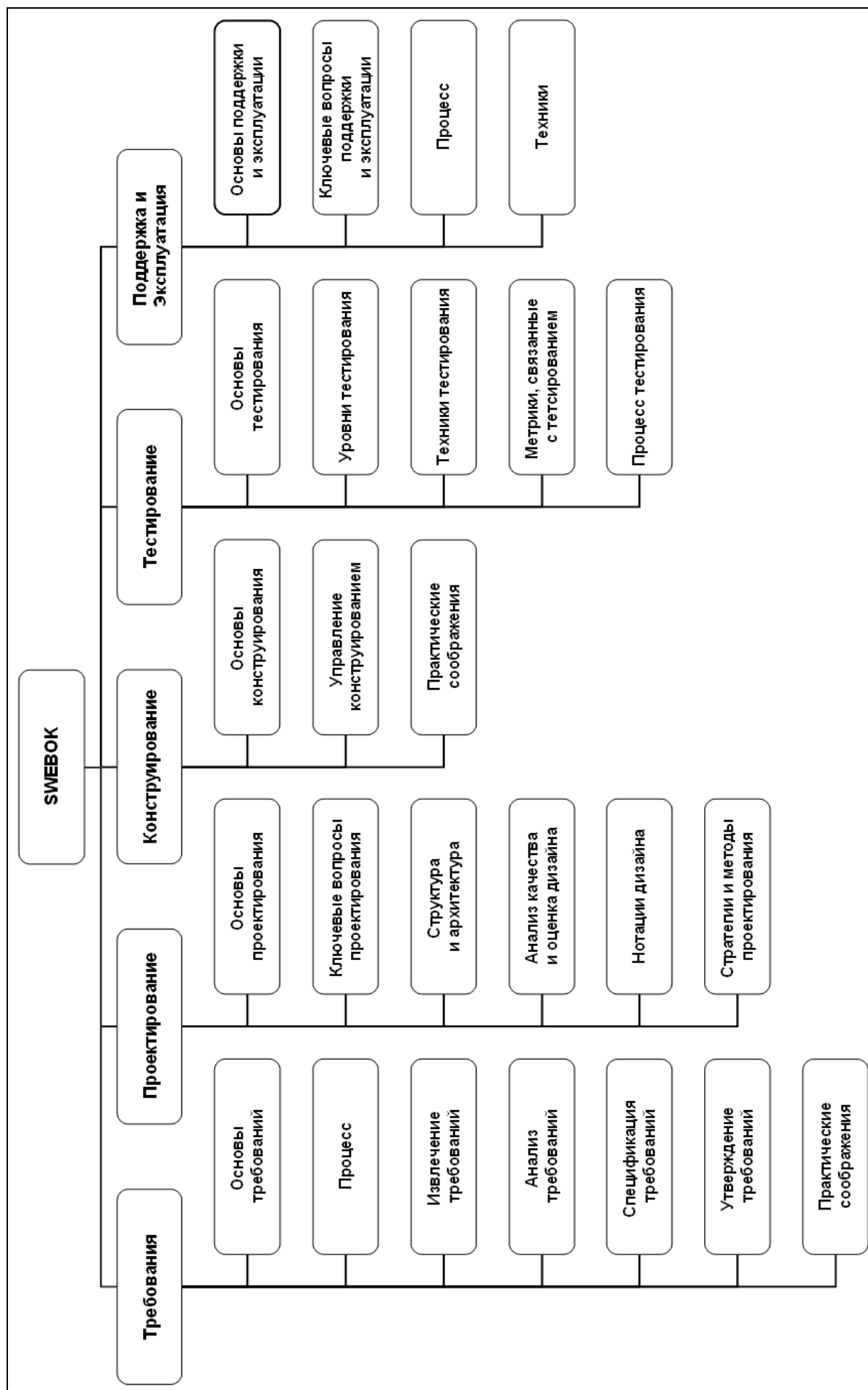


Рисунок 1.1 – SWEBOOK. Часть 1 – Разработка и эксплуатация



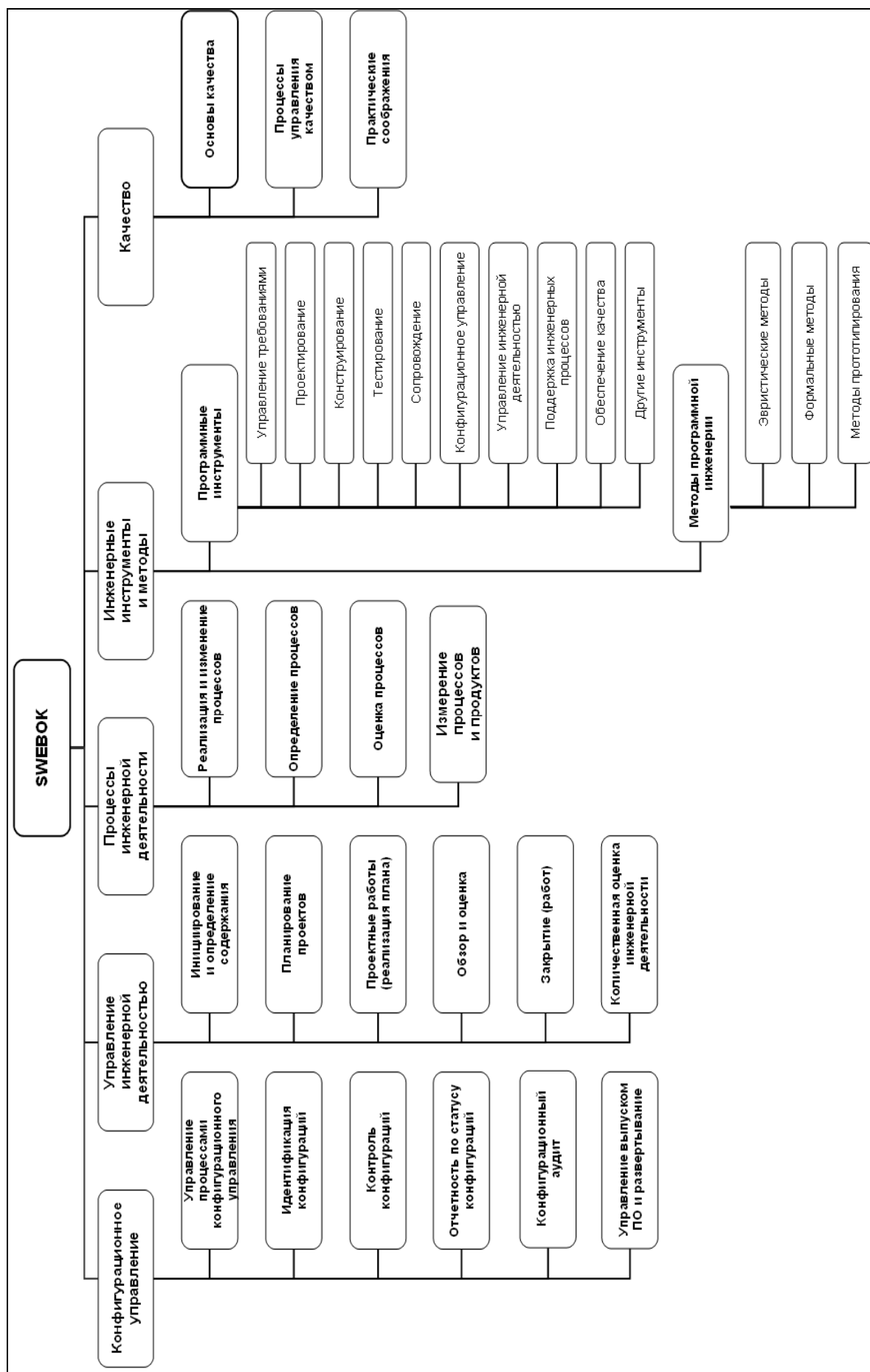


Рисунок 1.2 – SWEBOOK. Часть 2 – Управление, методология, инструменты

Методы моделирования включают следующие компоненты:

- *описание моделей и нотация*, используемая для их описания (например, структурные модели, объектные модели, модели поведения и т.д. с соответствующей графической нотацией);
- *правила и ограничения*, которые надо выполнять при разработке моделей (например, «каждый класс должен иметь уникальное имя»);
- *рекомендации* – эвристики, характеризующие хорошие приемы проектирования с использованием определенного метода (например, «все атрибуты класса должны быть определены до определения его методов»);
- *руководства по применению метода* – описание последовательности работ, которые надо выполнить для построения моделей.

Начиная с 70-х годов прошлого века создано достаточно много методов разработки ПО, что, в частности, подтверждает факт отсутствия идеальных методов – каждый из них применим и эффективен только в определенных ситуациях, и при этом методы могут включать элементы различных подходов. Выбор эффективного метода – одна из задач, решаемых программными инженерами.

Для практического использования метода моделирования в процессе проектирования необходим соответствующий этому методу *язык представления моделей* и поддерживающее этот язык CASE-средство. Далее будут рассмотрены элементы одного из языков графического моделирования – *Unified Modeling Language (UML)*.

### **1.3.2 Что такое CASE ?**

*CASE – Computer Aided Software Engineering* – инструментальные программные средства, используемые для поддержки процесса создания ПО. При выборе CASE-средств (приложение В, 22) следует руководствоваться основным принципом: сначала – метод, а затем – CASE-средства, применимые для этого метода<sup>3</sup>.

CASE-средства могут быть классифицированы по нескольким признакам.

---

<sup>3</sup> Следует заметить, что не всегда удастся воспользоваться этой теоретически правильной рекомендацией, так как на выбор методов проектирования оказывает существенное влияние именно набор CASE-средств, которые имеются в распоряжении разработчика или с которыми разработчик хорошо знаком и имеет опыт их применения.

*По уровню применения:*

- средства анализа предметной области проекта и поддержки требований к проектируемой системе (Upper CASE);
- средства проектирования (Middle CASE);
- средства разработки приложений (Low CASE);
- интегрированные CASE-средства, охватывающие несколько этапов создания ПО – от анализа требований до тестирования и выпуска эксплуатационной документации.

*По видам выполняемых работ, например:*

- средства планирования и управления проектом;
- средства конфигурационного управления;
- средства разработки прототипов пользовательских интерфейсов;
- средства проектирования баз данных;
- средства тестирования;
- средства реинжиниринга (например, восстановление ER-моделей по результатам анализа схем баз данных или восстановление диаграмм классов на основе анализа исходных кодов программ).

### ***1.3.3 Свойства хорошей программной системы***

Качество программной системы определяется тем, насколько она соответствует предъявляемым к ней требованиям (и, разумеется, качеством самих требований). Модели качества программных систем (рисунок 1.3) определены соответствующими стандартами (приложение В, 23).

Прежде всего, *хорошая система должна удовлетворять функциональным требованиям* – то есть *делать то, что ожидает от нее заказчик*, однако выполнение *нефункциональных требований* к системе при выполнении программного проекта часто требует существенно больших затрат. Так, например:

- *сопровождаемость* – требует значительных усилий по поддержанию соответствия проекта исходному коду и применения специальных методов создания модифицируемых программ;
- *надежность* – требует дополнительных средств восстановления системы при сбоях, шифрования данных и/или ограничения доступа;
- *уровень производительности* – требует поиска специальных архитектурных решений, выбора каналов передачи данных с требуемыми характеристиками и оптимизации программного кода;

– *удобство использования* – требует создания профессионально понятного пользовательского интерфейса, проектирование которого требует от разработчика более глубокого погружения в предметную область проекта.

### **Контрольные вопросы и задания**

- 1 Определите понятия *«программный продукт»* и *«программное обеспечение»*. Чем отличаются *«коробочные»* программные продукты от *«заказных»*?
- 2 Что понимается под *«жизненным циклом ПО»*?
- 3 Определите понятия *«инженерия»* и *«программная инженерия»*. Что общего и в чем различия между программной инженерией и другими инженерными отраслями?
- 4 Определите основные черты технологий *модульного* и *структурного* программирования, технологии *структурного* проектирования.
- 5 Определите основные черты технологии *объектно-ориентированного* программирования.
- 6 Определите и дайте краткую характеристику основным *нефункциональным требованиям* к программному продукту в соответствии с ГОСТ Р ИСО/МЭК 25010-2015.

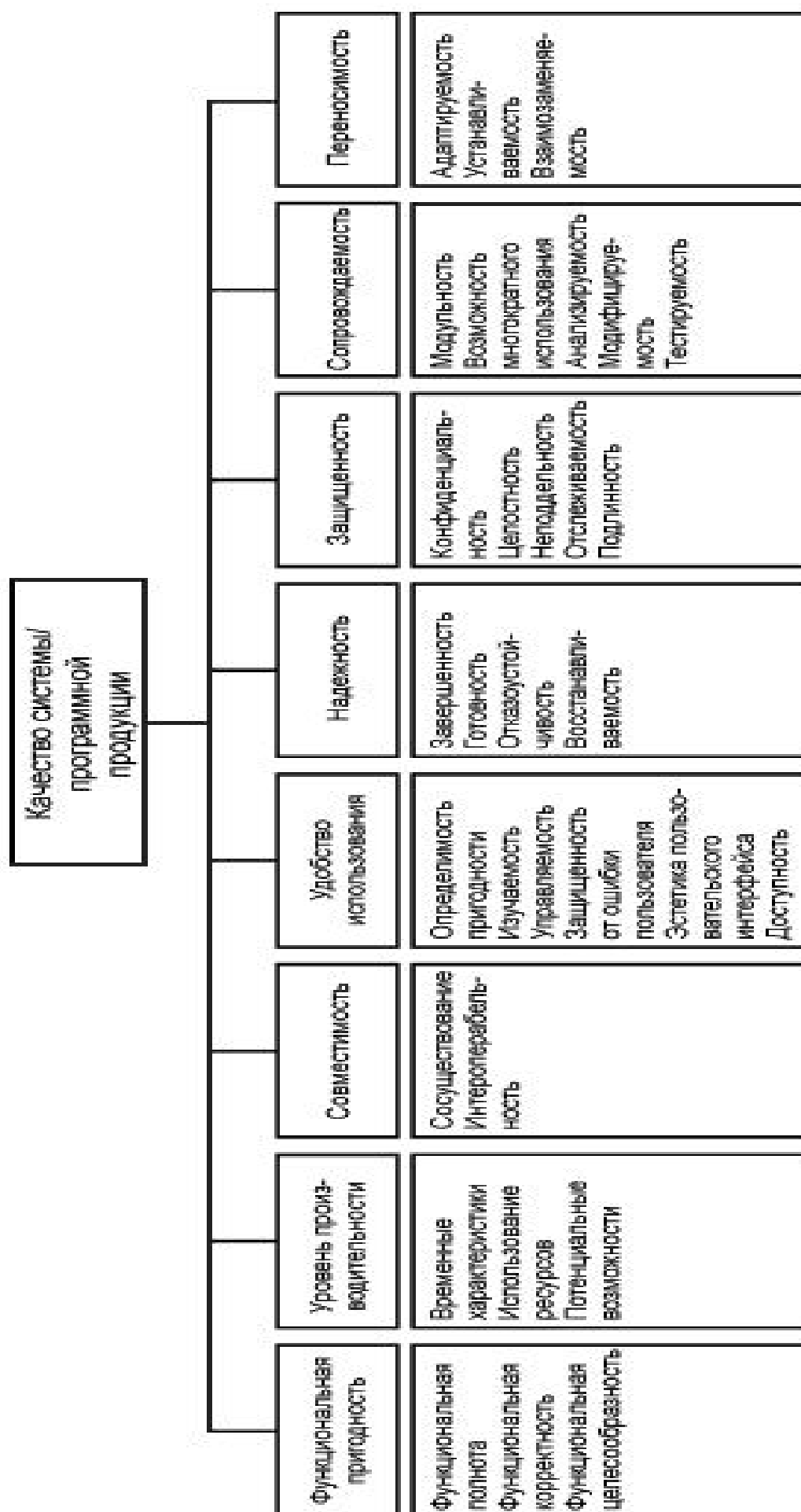


Рисунок 1.3 – Характеристики качества программной системы

### ПРОЦЕССЫ И МОДЕЛИ ЖИЗНЕННОГО ЦИКЛА ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

*Понятие и основные стадии жизненного цикла промышленного изделия; особенности жизненного цикла программного продукта; стандарты жизненного цикла; терминология; участники процессов жизненного цикла; ролевая модель команды программного проекта; понятие модели жизненного цикла программного продукта; каскадная, эволюционная и спиральная модели: их преимущества, недостатки и области эффективного применения.*

#### 2.1 Понятие жизненного цикла промышленного изделия

Методологическую основу любой инженерии составляет понятие ***жизненного цикла (ЖЦ) изделия, рассматриваемого как совокупность взаимосвязанных действий, которые выполняются на протяжении всего существования изделия – от принятия решения о его создании до вывода из эксплуатации и последующей утилизации.***

Жизненный цикл определяется как последовательность этапов (фаз, стадий), каждый из которых имеет иерархическую структуру и состоит из определенного набора технологических процессов, каждый из которых, в свою очередь, состоит из операций, операции – из более мелких работ и т.д.

Обычно к основным этапам ЖЦ промышленного изделия относят: *анализ и формулировку требований – проектирование – изготовление образца – организацию производства – серийное производство – эксплуатацию – ремонт – вывод из эксплуатации.*

Концепция жизненного цикла давно и успешно используется в традиционных инженерных отраслях. Жизненный цикл программного продукта имеет свои специфические особенности:

1) фактическое отсутствие этапов *изготовления образца ПО и серийного производства ПО* (эти этапы сводятся к компиляции и копированию);

2) этап проектирования ПО (включая детальное проектирование, программирование, сборку и тестирование) является наиболее трудоемким и составляет основную долю стоимости производства ПО;

3) традиционный ремонт и обслуживание в процессе эксплуатации, по существу, заменяется процессом сопровождения ПО, основное содержание которого – устранение замеченных при эксплуатации недостатков (ошибок разработчиков) и модификация ПО в соответствии с изменениями требований заказчика.

## **2.2 Краткая история стандартизации жизненного цикла ПО<sup>4</sup>**

Заключая контракт на разработку и поставку заказного ПО потребитель должен быть уверен в том, что разработчик справится с задачей в согласованные сроки и выполнит проект в строгом соответствии с установленными требованиями. Приобретая коробочный программный продукт, покупатель также должен быть уверен в его высоком качестве и соответствии продукта характеристикам, заявленным его производителем и поставщиком.

### ***Что же является гарантией качества программных продуктов?***

*В мировой практике промышленного производства гарантией качества являются стандарты на производство товаров и услуг и сертификация их производителей на соответствие этим стандартам.*

Процесс стандартизации производства и сертификации производителей давно вошел и в программную инженерию, где он составляет основу промышленного производства программных продуктов. Как отмечалось, впервые о необходимости рассматривать разработку ПО с позиций его жизненного цикла было заявлено в 1968 году.

**1985 г.** – в США принят первый стандарт жизненного цикла программных средств для систем военного назначения *DOD-STD-2167 A*. Этим документом регламентированы 8 фаз (этапов) создания сложных программных систем и около 250 типовых обязательных требований к процессам и объектам проектирования на этих этапах.

**1994 г.** – для замены *DOD-STD-2167A* Министерством обороны США принят новый стандарт *MIL-STD-498. Разработка и документирование программного обеспечения*, который предназначался для применения организациями и предприятиями, получающими заказы Министерства обороны США.

**1996 г.** – утверждено подробное руководство «*Применение и рекомендации к стандарту MIL-STD-498*», содержащее рекомендации по

---

<sup>4</sup> Перечень стандартов жизненного цикла ПО приведен в приложении В.

обеспечению и реализации процессов ЖЦ сложных программных систем высокого качества и надежности, функционирующих в реальном времени.

**1995 г.** – принят международный стандарт *IEEE 1074. Процессы жизненного цикла для развития программного обеспечения*. Стандарт охватывает полный жизненный цикл ПО, в котором выделяются шесть крупных базовых процессов.

**1995 г.** – принят международный стандарт *ISO/IEC 12207 – Information Technology – Software Life Cycle Processes*. Стандарт разрабатывался с учетом лучшего мирового опыта на основе перечисленных выше стандартов.

**1998 г.** – выходит новый стандарт *ISO/IEC TR 15504: Information Technology – Software Process Assessment*. Стандарт содержит новую классификацию процессов жизненного цикла ПО, являющуюся развитием стандарта ISO/IEC 12207, и регламентирует процедуры аттестации, определения зрелости и усовершенствования процессов жизненного цикла ПО.

**2000 г.** – на основе стандарта ISO/IEC 12207 принят российский стандарт *ГОСТ 12207 – Процессы жизненного цикла программных средств*.

Следует отметить, что высокая динамичность ИТ-индустрии приводит к быстрому устареванию стандартов, их регулярному переизданию, замене одних стандартов другими, вводу в действие государственных стандартов, создаваемых на основе соответствующих международных, а также созданию корпоративных стандартов, не всегда строго соответствующих государственным.

Все это приводит к терминологической неоднозначности стандартов, часто затрудняющих их практическое внедрение.

## 2.3 Терминология программной инженерии

Таблица 2.1 – Основные термины (по стандарту ISO/IEC 2382-1-93)

Термин	Содержание
1	2
Система (system)	Комплекс, состоящий из взаимодействующих процессов, технических и программных средств и персонала, обладающий возможностью удовлетворять установленным потребностям или целям



Продолжение таблицы 2.1

1	2
Программный продукт (software product)	Одна или несколько компьютерных программ вместе с сопутствующими неэлектронными и немеханическими вспомогательными элементами (документация и данные), поставляемые под одним наименованием для совместного использования
Программная услуга (software service)	Выполнение работ, заданий или обязанностей, связанных с программным продуктом, таких как разработка, сопровождение или эксплуатация
Программный модуль (software unit)	Отдельно компилируемая часть программного кода. Низкоуровневый элемент одного или нескольких программных компонентов
Выпуск (release)	<p>1 Поставляемая версия приложения, которая может включать все или некоторые его части (ISO / IEC 20926: 2003).</p> <p>2 Коллекция новых и / или измененных элементов, которые одновременно тестируются и внедряются (ISO / IEC 20000-1: 2005).</p> <p>3 Конкретная версия элемента конфигурации, доступная для определенной цели (ISO / IEC 12207: 2008).</p> <p>4 Официальное уведомление и распространение утвержденной версии (IEEE Std 828-2005)</p>
Версия (version)	Определенный экземпляр объекта. В результате модификации версии программного продукта появляется новая версия, подвергающаяся управлению конфигурацией
Программно-аппаратное средство (firmware)	Сочетание аппаратных устройств и программного обеспечения, постоянно хранящихся на техническом устройстве. Не может быть изменено только средствами программирования
Непоставляемое изделие (non-deliverable item)	Техническое или программное средство, которое не поставляется по условиям договора, но может быть применено при создании программного продукта

Продолжение таблицы 2.1

1	2
Готовый продукт (off-the-shelf product)	Ранее разработанный и доступный для приобретения продукт, пригодный для использования в поставляемом или модифицированном виде
Модель жизненного цикла (life cycle model)	Структура связанных с жизненным циклом процессов и действий, организуемых в стадии, которые также служат для установления связей и взаимопонимания сторон
Процесс (process)	Совокупность взаимосвязанных или взаимодействующих видов деятельности, преобразующих входы в выходы. Выход процесса – наблюдаемый результат успешного достижения цели процесса (изготовление артефакта, изменение состояния, удовлетворение ограничений)
Заказ (acquisition)	Процесс приобретения системы, программного продукта или программной услуги
Аудит (audit)	Проверка, выполняемая компетентным органом (лицом) с целью обеспечения независимой оценки степени соответствия программных продуктов или процессов установленным требованиям
Оценивание (evaluation)	Систематическое определение степени соответствия объекта установленным критериям
Надзор (monitoring)	Проверка заказчиком или третьей стороной состояния работ, выполняемых поставщиком, и их результатов
Квалификация (qualification)	Процесс демонстрации возможности объекта выполнять установленные квалификационные требования
Квалификационное требование (qualification requirement)	Набор критериев, которые должны быть удовлетворены для того, чтобы квалифицировать программный продукт
Квалификационное испытание (qualification testing)	Испытание (тестирование), проводимое для демонстрации того, что программный продукт удовлетворяет спецификациям и готов для применения в заданном окружении или к интеграции с системой, для которой он предназначен

Продолжение таблицы 2.1

1	2
Верификация (verification)	Совокупность действий по сравнению полученного результата с требуемыми характеристиками для этого результата
Валидация (validation)	Совокупность действий, гарантирующих и обеспечивающих уверенность в том, что система способна реализовать свое предназначение. Подтверждение экспертизой и представлением объективных доказательств того, что конкретные требования к конкретным объектам полностью реализованы
Снятие с эксплуатации (retirement)	Прекращение активной поддержки действующей системы эксплуатирующей или сопровождающей организацией
Заказчик (acquirer)	правообладатель, который приобретает или получает систему, программный продукт или программную услугу от поставщика. Наряду с термином «заказчик» допускается использование терминов «приобретающая сторона», «розничный покупатель», «оптовый покупатель»
Разработчик (developer)	Организация, выполняющая разработку ПО (включая анализ требований, проектирование, программную реализацию, приемочные испытания) в процессе его жизненного цикла
Поставщик (supplier)	Организация, которая заключает соглашение с заказчиком на поставку системы, программного продукта или программной услуги на оговоренных условиях
Оператор (operator)	Лицо или организация, эксплуатирующие систему
Пользователь (user)	Лицо или группа лиц, использующие действующую систему в процессе ее применения
Сопровождающая сторона (maintainer)	Организация, которая выполняет работы по сопровождению ПО в процессе его эксплуатации.

## 2.4 Процессы жизненного цикла программного продукта

Стандарт ISO 12207-2010 вводит понятие *процесса жизненного цикла ПО*, устанавливает иерархическую классификацию процессов и определяет эталонную модель процесса, которая не предлагает конкретных методов или технологий его реализации, но позволяет разработчикам продемонстрировать достижение (или не достижение) цели реализованного процесса, а оценщикам – определить его возможности.

Эталонная модель процесса определяет его наименование, цель реализации, желаемые выходы (результаты) и перечень действий и задач, которые необходимо выполнить для достижения результатов.

На верхнем уровне классификации определены две категории процессов: процессы, представляющие системный контекст и используемые для работы с автономными системами (рисунок 2.1, а), и специальные процессы, используемые в реализации программных компонентов систем (рисунок 2.1, б).

Каждая категория представлена несколькими группами процессов, а каждый процесс группы – несколькими *выходами, видами деятельности и задачами*.

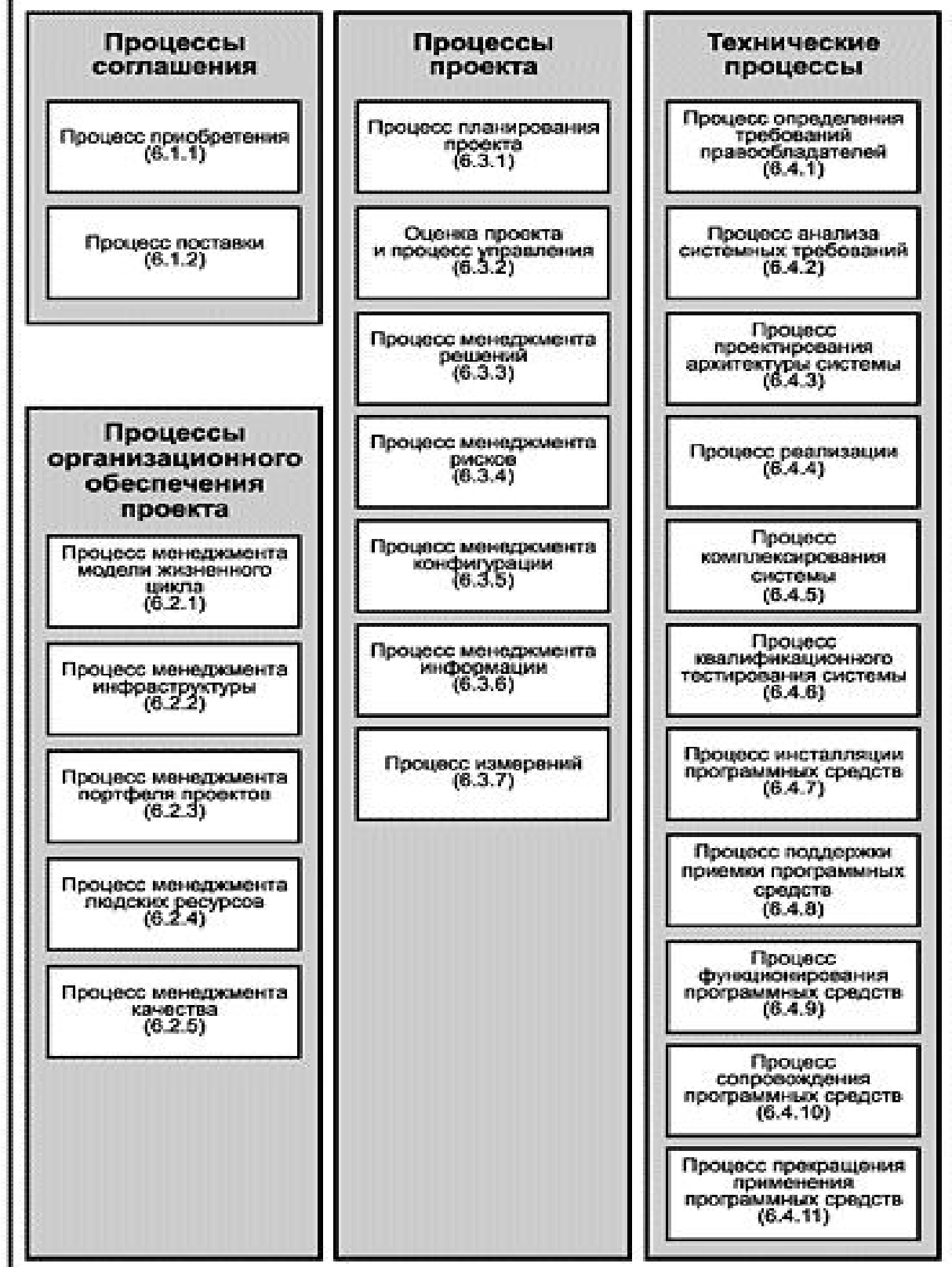
В качестве примера рассмотрим стандартное определение эталонной модели *процесса проектирования архитектуры программных средств* (процесс 7.1.3 на рисунке 2.1, б).

**Цель** – обеспечение проекта для программных средств, которые реализуются и могут быть верифицированы относительно требований.

**Выходы:**

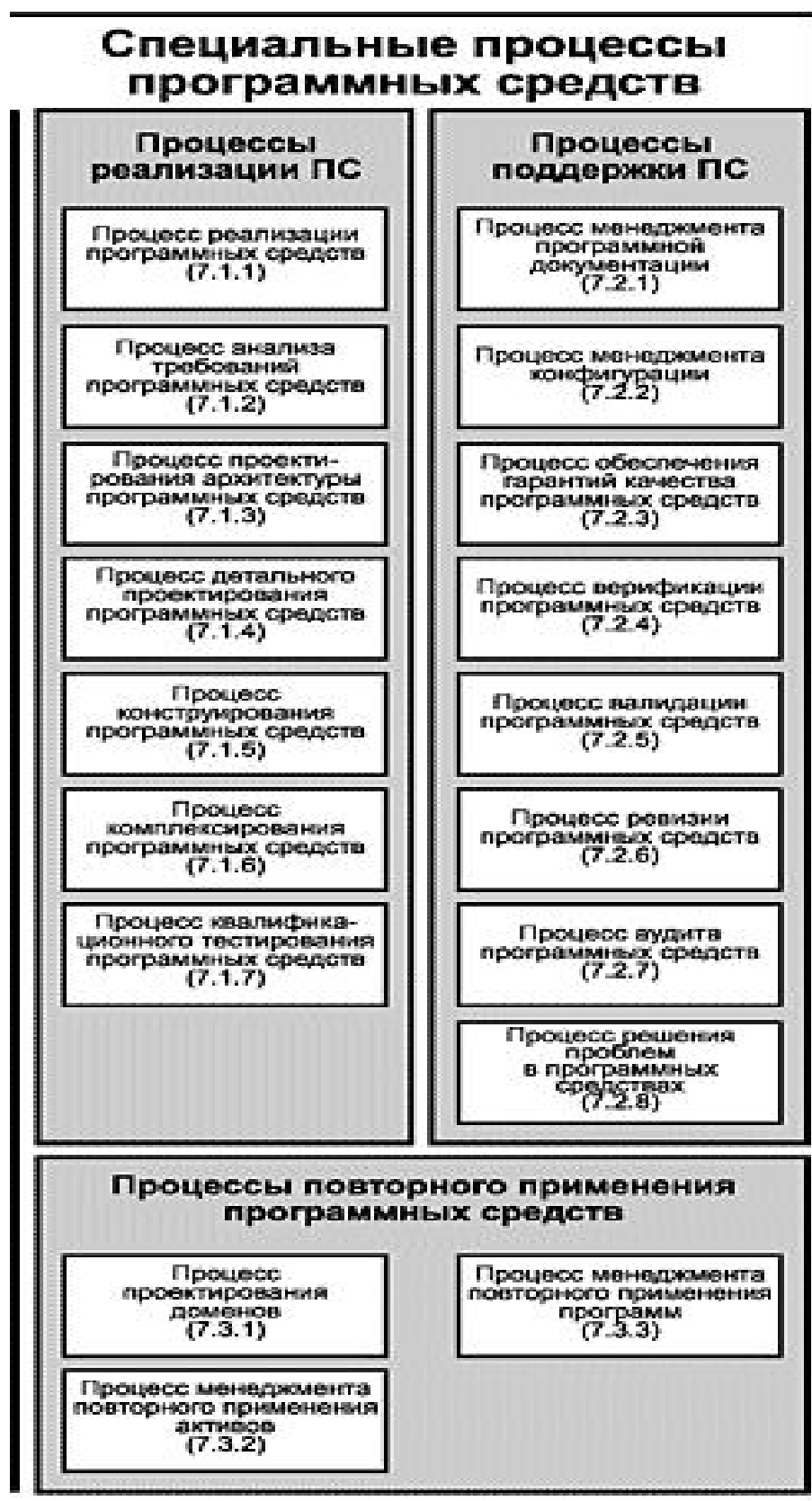
- 1) разрабатывается проект архитектуры программных средств и устанавливается базовая линия, описывающая программные составные части, которые будут реализовывать требования к программным средствам;
- 2) определяются внутренние и внешние интерфейсы каждой программной составной части;
- 3) устанавливаются согласованность и прослеживаемость между требованиями к программным средствам и программным проектом.

## Процессы в контексте системы



а) процессы в контексте системы

Рисунок 2.1(а) – Классификация процессов жизненного цикла ПО



б) специальные процессы программных средств

*Рисунок 2.1(б) – Классификация процессов жизненного цикла ПО*

**Вид деятельности** (для каждой программной составной части согласно проекту архитектуры системы) – **проектирование архитектуры программных средств**.

**Задачи** (для каждого программного элемента):

- 1) преобразование требований к программным составным частям в архитектуру, которая идентифицирует программные компоненты и распределяет между ними все требования к программным составным частям; документирование архитектуры программной составной части;
- 2) разработка и документальное оформление проекта верхнего уровня для внешних интерфейсов программной составной части и интерфейсов между ней и программными компонентами;
- 3) разработка и документальное оформление проекта верхнего уровня для базы данных;
- 4) разработка и документальное оформление предварительных версий пользовательской документации;
- 5) определение и документирование требований к предварительному тестированию и формирование графика работ по комплексированию программных средств;
- 6) оценивание и документальное оформление результатов оценивания архитектуры программной составной части, проектов по интерфейсам и базе данных, учитывая следующие критерии:
  - прослеживаемость к требованиям программной составной части;
  - внешняя согласованность с требованиями программной составной части;
  - внутренняя согласованность между программными компонентами;
  - приспособленность методов проектирования и используемых стандартов;
  - осуществимость детального проектирования;
  - осуществимость функционирования и сопровождения.

## **2.5 Модели жизненного цикла ПО**

### **2.5.1 Типовые стадии процесса создания ПО**

Одним из несомненных достижений стандарта *ISO/IEC 12207* являются разделение понятий *жизненного цикла* и *модели жизненного цикла*. Жизненный цикл ПО в стандарте определяется как *совокупность всех*

процессов развития системы (продукта, услуги), начиная со стадии разработки концепции и заканчивая прекращением применения, а модель жизненного цикла – как вариант организации процессов жизненного цикла, обоснованно выбранный для каждого конкретного случая.

Типовые стадии процесса создания программного продукта иллюстрируются рисунком 2.2:

- **спецификация**: формулирование спецификаций определяет основные требования к ПО (что должна делать система);
- **разработка**: создание ПО в соответствии со спецификациями;
- **аттестация (валидация)**: проверка ПО на соответствие потребностям заказчика;
- **модернизация**: развитие ПО в соответствии с изменившимися потребностями заказчика.

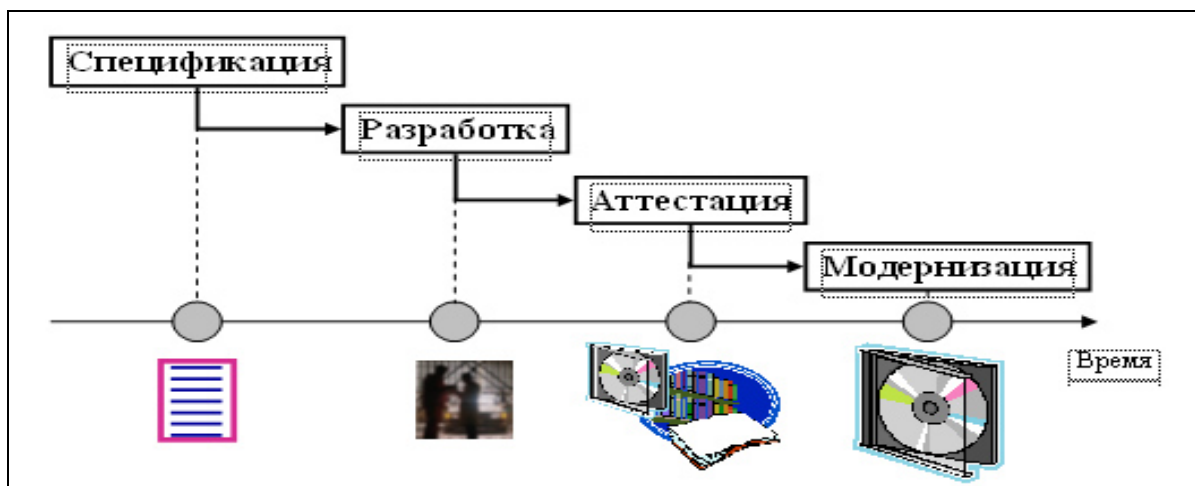


Рисунок 2.2 – Типовые стадии процесса создания ПО

Каждая компания может использовать определенную модель жизненного цикла ПО и организовать процесс его создания так, как ей это представляется разумным, при этом процесс может иметь разную степень формализации.

Так, например, возможен подход «вечером обсудили и обо всем договорились», но этот принцип подходит только для небольших команд разработчиков в достаточно простых проектах.

Возможна другая крайность – каждое действие жестко определено и прописано в описании процесса. В этом случае возникает необходимость длительного предварительного обучения сотрудников работе в рамках этого, безусловно, сложного описания. Подобный подход может быть ре-



комендован для очень больших проектов, выполняемых распределенными коллективами.

Несмотря на естественные отличия в описаниях процессов, в них, как правило, присутствуют все рассмотренные выше стадии.

В программной инженерии существуют хорошо проработанные корпоративные технологии, например, Microsoft Solutions Framework (MSF), Rational Unified Process (RUP), eXtreme Programming (XP) и другие, каждая из которых базируется на определенной модели жизненного цикла ПО.

### 2.5.2 Каскадная модель

Каскадная модель фактически «скопирована» с технологий, применяемых для проектирования сложных технических объектов в традиционных инженерных отраслях. Она наиболее широко распространена и используется на практике при выполнении масштабных программных проектов.

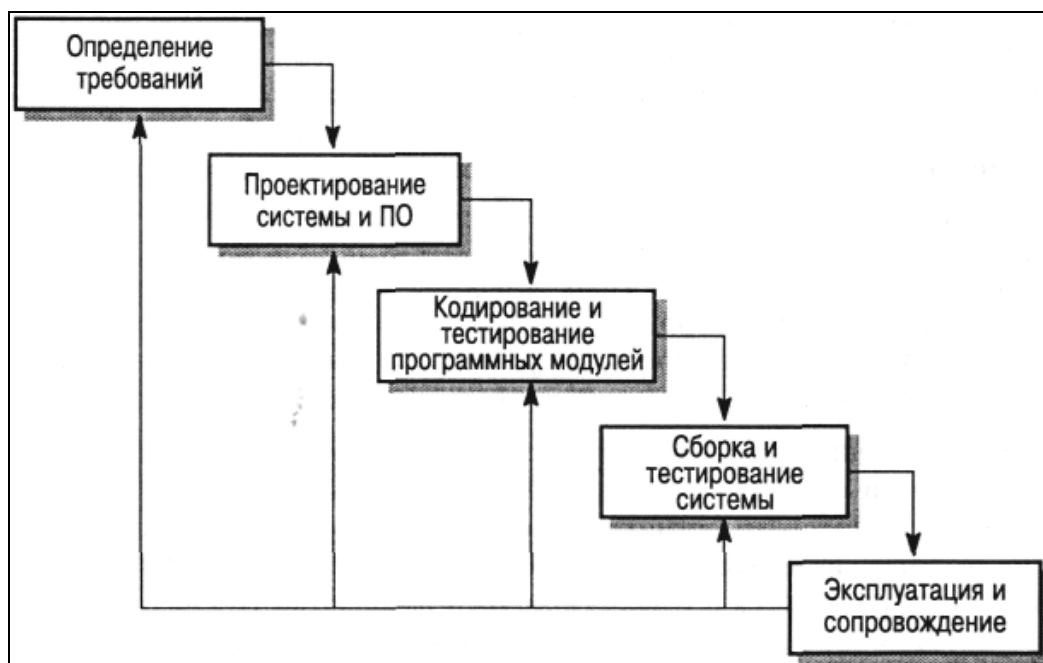


Рисунок 2.3 – Каскадная модель создания ПО

Каскадная модель (рисунок 2.3) представляет жизненный цикл ПО фиксированной последовательностью хронологически упорядоченных фаз (этапов), каждая из которых начинается только после того, как будет завершена и документирована предыдущая фаза.

Начальная фаза – **определение требований**. По результатам анализа предметной области, проводимого совместно с заказчиком ПО, определяются цели, функциональные возможности и ограничения на применение

создаваемой программной системы, формулируются и анализируются функциональные и нефункциональные требования к системе.

**Проектирование системы и программного обеспечения.** Процесс проектирования системы разбивает системные требования на требования, предъявляемые к аппаратным средствам, и требования к программному обеспечению системы. Разрабатывается общая архитектура системы. Проектирование ПО предполагает определение и описание основных программных компонентов и их взаимосвязей.

**Кодирование и тестирование программных модулей.** На этой стадии архитектура ПО реализуется в виде множества программ или программных модулей. Тестирование каждого модуля включает проверку его соответствия требованиям к данному модулю.

**Сборка и тестирование системы.** Отдельные программы и программные модули интегрируются и тестируются в виде целостной системы, определяется соответствие системы своей спецификации.

**Эксплуатация и сопровождение системы.** Обычно это самая длительная фаза жизненного цикла ПО. Система устанавливается, и начинается период ее эксплуатации. Сопровождение системы включает исправление ошибок, которые не были обнаружены на более ранних этапах жизненного цикла, совершенствование системных компонентов и «подгонку» функциональных возможностей системы к новым требованиям.

Ниже перечислены основные черты и области эффективного применения каскадной модели.

1 Каскадная модель создания ПО во многом копирует модели создания крупных технических объектов.

2 Требования определяются на начальном этапе и далее не меняются.

3 Очередной этап начинается лишь тогда, когда закончился предыдущий.

4 Достоинства каскадной модели – в хорошей ее структурированности.

5 Основные недостатки каскадной модели связаны с ее прямолинейностью и отсутствием гибкости: эта модель не учитывает тот факт, что неизменность требований заказчика к системе является мифом – требования менялись, меняются и будут меняться в ходе разработки.

6 К недостаткам каскадной модели можно отнести негибкое разбиение процесса создания ПО на отдельные фиксированные этапы. В этой

модели концептуальные решения принимаются на ранних стадиях проекта, и затем их трудно отменить или изменить.

7 Каскадная модель применяется в условиях, когда требования формализованы достаточно четко и корректно.

8 Вместе с тем каскадная модель хорошо отражает практику создания ПО: технологии создания ПО, основанные на данной модели, используются повсеместно, в частности для разработки систем, входящих в состав больших инженерных проектов.

### 2.5.3 V-модель

V-модель – это разновидность каскадной модели, в которой процесс создания программной системы представляется последовательностью взаимосвязанных фаз разработки и фаз тестирования, как это схематично показано на рисунке 2.4.

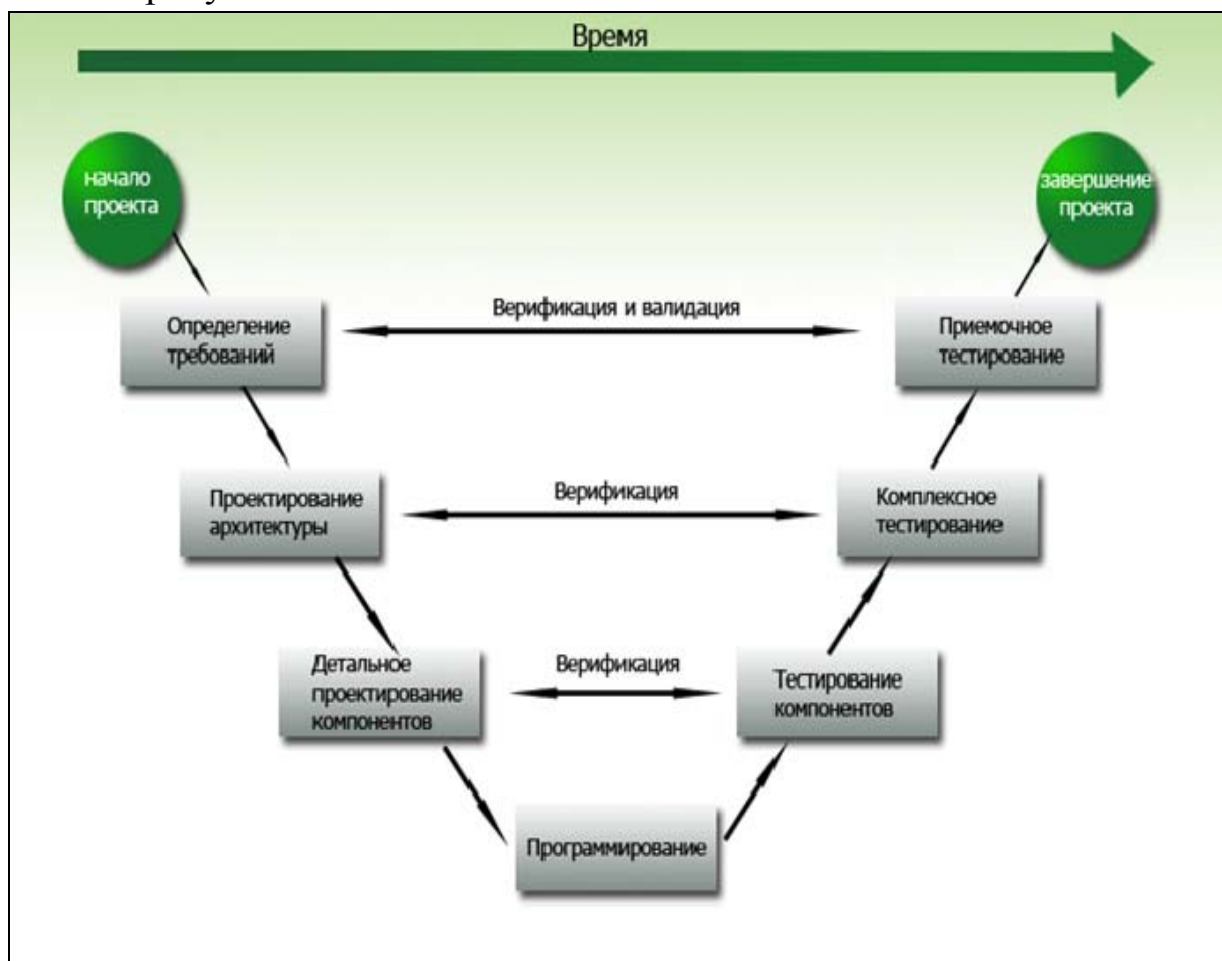


Рисунок 2.4 – V-модель создания ПО

Фазы разработки – от определения требований до программной реализации – размещены «каскадом» на левой стороне буквы «V», реализуя классическую концепцию «нисходящего проектирования». Фазы тестирования размещены на правой стороне буквы «V», обеспечивая процесс «восходящего тестирования» по схеме «от частного – к общему».

Двунаправленные горизонтальные линии, связывающие фазы разработки и фазы тестирования соответствующих уровней, показывают:

- 1) что приёмо-сдаточные испытания должны обеспечить верификацию и валидацию системы с учетом требований к системе в целом;
- 2) что комплексное тестирование должно основываться на требованиях к архитектуре и интерфейсам;
- 3) что компонентное тестирование должно основываться на требованиях к алгоритмам;
- 4) что разработчики и тестировщики должны активно взаимодействовать на соответствующих фазах жизненного цикла программного продукта, как это показано в таблице 2.2.

*Таблица 2.2 – Работы, выполняемые на различных фазах проекта*

Фазы проекта	Выполняемые работы	
	Разработчики	Тестировщики
Определение требований и планирование проекта	Определяют требования к проектируемой программной системе и разрабатывают описание ее поведения. Разрабатывают план программного проекта	Планируют процесс приемо-сдаточных испытаний. Определяют состав приемочных тестов и данные для приемочного тестирования в соответствии со спецификациями требований к системе
Проектирование архитектуры	Разрабатывают архитектурный проект верхнего уровня. Распределяют функциональные требования по компонентам архитектуры проектируемой системы	Планируют процесс комплексного тестирования. Разрабатывают тесты для проверки выполнения системой основных функций
Детальное проектирование компонентов	Для каждого из компонентов архитектурного проекта определяют состав программных модулей, разрабатывают алгоритмы, обеспечивающие выполнение модулями определенных для них функций	Разрабатывают методы тестирования модулей. Разрабатывают тестовые наборы данных для проверки алгоритмов, реализуемых модулями

Продолжение таблицы 2.2

Фазы проекта	Выполняемые работы	
	Разработчики	Тестировщики
Программирование	Разрабатывают, отлаживают и документируют программный код для каждого модуля. Осуществляют сборку компонентов архитектуры системы из программных модулей	Разрабатывают (устанавливают) программное обеспечение, необходимое для проведения тестирования
Тестирование компонентов	Получают результаты тестирования модулей. Исправляют ошибки в программном коде. Документируют процесс внесения изменений в программный код. Осуществляют повторную сборку	Выполняют автономную проверку каждого модуля. Сохраняют результаты в протоколах тестирования
Комплексное тестирование	Получают результаты тестирования компонентов. Исправляют ошибки в сборке компонентов и/или в программном коде модулей. Документируют процесс внесения изменений. Осуществляют повторную сборку	Выполняют тестирование компонентов, включающих модули, успешно протестированные в автономном режиме. Сохраняют результаты в протоколах тестирования
Приемочное тестирование	Получают результаты тестирования. Исправляют ошибки. При необходимости осуществляют повторную сборку и интеграцию системы.	Выполняют проверку функционирования системы, полностью интегрированной в аппаратную среду в соответствии со спецификацией требований.

#### 2.5.4 Модель формальной разработки

Эта модель также может считаться разновидностью каскадной модели, но построена на основе формальных математических преобразований системной спецификации в исполняемую программу (рисунок 2.5).

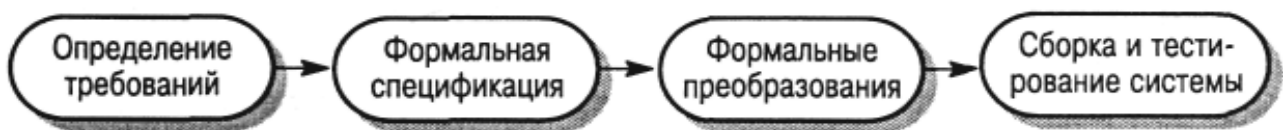


Рисунок 2.5 – Модель формальной разработки ПО

Отличия модели формальной разработки от каскадной модели:

- 1) спецификация системных требований имеет вид детализированной формальной спецификации, записанной с помощью специальной математической нотации;
- 2) процессы проектирования, кодирования и тестирования программных модулей заменяются процессом, в котором спецификация путем формальных преобразований трансформируется в исполняемую программу.

В процессе преобразования формальное математическое представление системы последовательно трансформируется в программный код, постепенно все более детализированный. Преобразования выполняются математически корректно и проверка соответствия спецификации и программы не требуется.

Наиболее известным примером метода формальных преобразований является метод «чистой комнаты» (*Cleanroom*), разработанный компанией IBM. Этот метод предполагает пошаговую разработку ПО, когда на каждом шаге применяются формальные преобразования, что позволяет отказаться от тестирования отдельных программных модулей, а тестирование всей системы происходит после ее сборки.

Модель формальных преобразований применяется для разработки систем, которые должны удовлетворять строгим требованиям надежности, безотказности и безопасности, так как они гарантируют соответствие созданных систем их спецификациям. Эта модель не нашла широкого применения – основная причина заключается в том, что функционирование большинства систем с трудом поддается описанию методом формальных спецификаций.

### **2.5.5 Эволюционная модель**

Отличительной чертой эволюционной модели является то, что специфицирование, то есть выработка требований к программному продукту, не выполняется единовременно на продукт в целом, а постепенно уточняется по результатам реализации процессов разработки и валидации различных его версий, как это показано на рисунке 2.6.

Начальная версия программного продукта (после ее специфицирования, разработки и валидации) передается на испытание пользователям, а затем дорабатывается с учетом результатов испытаний. Полученная про-

межуточная версия также передается пользователям для испытаний, затем снова дорабатывается – и так несколько раз, пока не будет получена конечная версия продукта. Таким образом, программный продукт эволюционирует в процессе его разработки, постепенно наращивая свою функциональность, и в своей конечной версии наиболее полно соответствует требованиям пользователей.

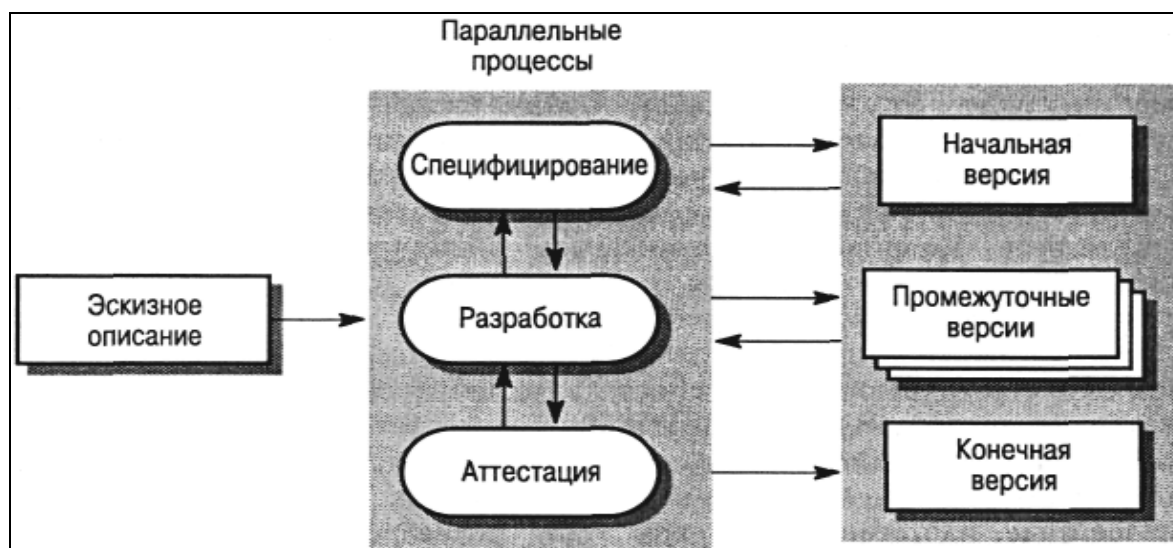


Рисунок 2.6 – Эволюционная модель разработки ПО

При этом сами пользователи фактически становятся участниками программного проекта и получают возможность постепенного освоения разрабатываемого ПО в процессе испытаний его промежуточных версий, что облегчает процесс ввода в эксплуатацию программного продукта.

Существуют два метода (подхода) реализации эволюционной модели.

*Метод пробных разработок* применяется в условиях, когда заказчик не готов сразу определить полный состав требований к ПО, необходимых для разработки конечной версии продукта. В рамках этого подхода начальная версия системы включает наиболее очевидные и/или хорошо специфицированные компоненты системы. Далее система эволюционирует (дорабатывается) путем добавления в нее новых компонентов по мере разработки их спецификаций и согласования требований с заказчиком.

*Метод прототипирования.* В данном случае прототип – это действующий программный модуль, реализующий отдельные функциональные требования к проектируемой системе. Метод прототипирования применяется в условиях, когда заказчик готов определить полный набор требований к проектируемой системе, но разработчик не готов сразу предло-

жить решения для реализации той части требований, которая сформулирована противоречиво или недостаточно четко. При таком подходе начальные версии продукта включают прототипы, создаваемые для экспериментирования с той частью требований заказчика, которые сформированы нечетко или с внутренними противоречиями.

Достоинства эволюционной модели очевидны: ее применение позволяет сократить сроки разработки программного продукта и дает возможность разработчику постепенно уточнять спецификацию отдельных его компонентов по мере того, как заказчик осознает и сформулирует требования к их функционированию.

Недостатки эволюционных моделей – это обратная сторона их достоинств:

- *многие этапы программного проекта оказываются плохо документированными*, так как при большом количестве версий системы и ограниченных сроках разработки документирование каждой версии системы становится экономически не выгодным;
- *система часто получается плохо структурированной* из-за частых изменений требований и постепенного наращивания функциональных компонентов и модулей системы;
- для реализации эволюционных моделей *требуются специальные средства и технологии разработки ПО*, что вызвано необходимостью поддержки многоверсионности программного продукта.

Эволюционные модели наиболее приемлемы для разработки небольших (до 100 000 строк кода) и средних (до 500 000 строк кода) программных систем с относительно коротким сроком жизни. На больших и долгоживущих системах слишком заметно проявляются недостатки этого подхода.

### **2.5.6 Спиральная модель**

Спиральная модель относится к категории *гибридных итерационных моделей* и является дальнейшим развитием эволюционного подхода. В отличие от рассмотренных ранее эволюционных моделей, где процесс создания ПО представлен последовательностью отдельных его этапов с возможностью их многократного повторения, здесь процесс разработки представлен в виде спирали (рисунок 2.7).



Спираль программного проекта раскручивается от центра к периферии, при этом каждый виток спирали соответствует одной его стадии (итерации): внутренний виток – это стадия принятия решения о создании ПО, на следующем витке определяются системные требования, далее следует стадия проектирования, затем стадия реализации и т.д.

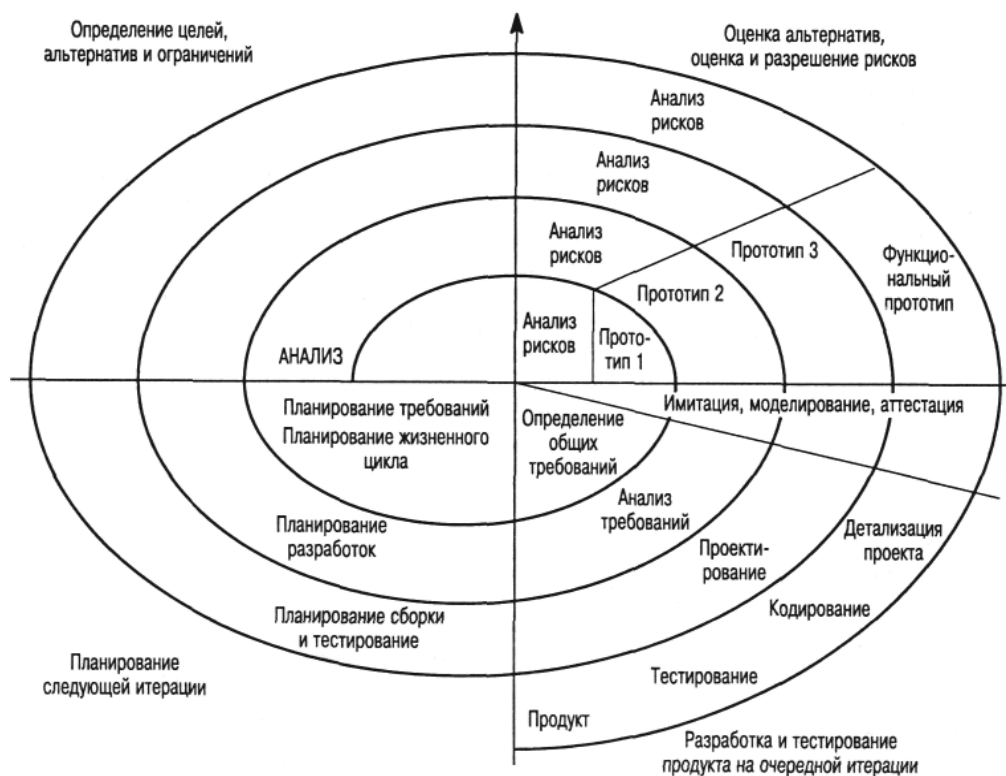


Рисунок 2.7 – Спиральная модель разработки ПО

Первая итерация (внутренний виток спирали) начинается с тщательной проработки целей системы, ее эксплуатационных показателей и функциональных возможностей. Формируется множество альтернативных путей достижения этих показателей, оценивается стоимость каждого из них.

На следующей стадии спиральной модели производится более детальный анализ рисков, сопутствующих альтернативным вариантам, затем – прототипирование, имитационное моделирование и т.п. С учетом полученных оценок рисков выбирается тот или иной подход к разработке компонентов, далее он реализуется, затем осуществляется планирование следующего этапа процесса создания ПО.

Спиральная модель может включать в себя любые другие модели и не предусматривает наличия фиксированных этапов на каждом ее витке. Например, на одном витке спирали может использоваться прототипирование

для более четкого определения требований, но на следующем витке может применяться каскадная модель или метод формальных преобразований.

Каждый виток спирали разбит на четыре сектора.

**Сектор 1 – определение целей.** Определяются цели и устанавливаются ограничения для каждой итерации проекта, уточняются планы производства компонентов. В обязательном порядке прогнозируются проектные риски (например, риск превышения сроков или риск превышения стоимости проекта) и планируются альтернативные стратегии разработки.

**Сектор 2 – оценка и разрешение рисков.** Проводится детальный анализ проектных рисков, планируются мероприятия для их разрешения. Например, если существует риск неправильного определения системных требований, планируется разработка соответствующего прототипа.

**Сектор 3 – разработка и тестирование.** После оценки рисков выбирается модель процесса создания системы. Например, если доминируют риски, связанные с разработкой интерфейсов, наиболее подходящей будет эволюционная модель с прототипированием; если же основные риски связаны с соответствием системы её спецификациям, скорее всего, следует применить модель формальных преобразований; если основные риски определены как ошибки, которые могут проявиться на этапе сборки системы, может быть применена каскадная модель.

**Сектор 4 – планирование.** Производится оценка результатов выполнения текущей стадии проекта и принимается решение о том, начинать ли следующий виток спирали. Если принимается решение о продолжении проекта, разрабатывается план реализации его следующей стадии.

Существенным отличием спиральной модели от других моделей является требование точного оценивания проектных рисков на каждом витке спирали и выработки способов их разрешения. Оценка и разрешение рисков – важный элемент системы управления программным проектом.

## **2.6 Ролевая модель команды программного проекта**

Ведущую роль в реализации программного проекта играет правильная организация команды разработчиков: *определение ролевой модели команды; подбор исполнителей на каждую роль; организация их взаимодействия в процессе выполнения проекта.* Состав команды определяется особенностями проекта, применяемыми технологиями и инструментальными средствами, опытом и уровнем профессиональной подготовки коллектива.

Классический вариант состава команды включает следующие роли: *менеджер проекта, проектировщик, разработчик, тестировщик, инженер по качеству, технический писатель, технолог, инженерный психолог, инженер по маркетингу.*

*Менеджер проекта* – главное действующее лицо, обладающее знаниями и навыками, необходимыми для успешного управления проектом. Его основные функции:

- *подбор и управление кадрами;*
- *подготовка и исполнение плана проекта;*
- *руководство командой;*
- *обеспечение связи между подразделениями;*
- *обеспечение готовности продукта.*

*Проектировщик* занимается проектированием архитектуры высокого уровня и осуществляет контроль ее выполнения. В небольших командах функции проектировщика обычно распределяются между менеджером проекта и разработчиками, в больших проектах эти функции может исполнять отдельная группа или целый отдел.

Основные функции проектировщика:

- *анализ требований;*
- *разработка архитектуры и основных интерфейсов;*
- *участие в планировании проекта;*
- *контроль выполнения проекта;*
- *участие в подборе кадров.*

*Разработчик* отвечает за непосредственное создание конечного продукта. Помимо собственно программирования (кодирования) в его функции входит:

- *контроль архитектурных и технических спецификаций продукта;*
- *подбор и контроль использования технологических инструментов и стандартов;*
- *диагностика и разрешение всех технических проблем;*
- *контроль за работой тестировщиков, технологов, разработчиков документации;*
- *мониторинг состояния продукта (ведение списка обнаруженных ошибок).*

*Тестировщик* отвечает за удовлетворение требований к продукту (функциональных и нефункциональных). Функции тестировщика:

- *составление плана тестирования*, который является элементом плана проекта и составляется до начала реализации проекта, при этом время, отводимое на тестирование, может быть сопоставимо с временем разработки;
- *контроль выполнения плана тестирования*: поддержка общедоступной для всех членов команды проекта базы данных зарегистрированных ошибок, в которой фиксируется детальная информация об ошибках (описание ошибки и состояния среды, статус ошибки, приоритет, кто разрешает, проблемы);
- *разработка тестов* – самая трудоемкая составляющая работы тестировщика, так как тестирование должно обеспечить полную проверку функциональности при всех режимах работы продукта;
- *автоматизация тестирования* включает автоматизацию составления тестов, автоматизацию процесса запуска тестов и автоматизацию обработки результатов тестирования; в командах крупных проектов может быть введена отдельная роль – инженер по автоматизации;
- *выбор стандартов, метрик и инструментов* для организации процесса тестирования;
- *организация бета-тестирования*: тестирование почти готового продукта внешними тестерами (пользователями – в случае разработки коробочного продукта).

*Инженер по качеству*. В небольших проектах его функции возлагаются на тестировщика, однако функции инженера по качеству гораздо шире – он обеспечивает *качество процесса разработки* (контроль соответствия стандартам качества *основных процессов* жизненного цикла) и *качество организации* (контроль соответствия стандартам качества *организационных процессов* жизненного цикла).

Основные функции инженера по качеству:

- *составление плана качества*, который имеет долговременный характер и включает мероприятия по повышению качества на всех уровнях; план тестирования – оперативная составляющая плана качества;

- *описание процессов*: формализация процессов, определение метрик процессов, влияющих на качество продукта;
- *оценка процессов* включает регистрацию хода выполнения процессов и оценку значений установленных метрик процессов с целью выявления их слабых мест и выработки рекомендаций по улучшению;
- *улучшение процессов*: переопределение процесса, автоматизация части работ, обучение персонала.

*Технический писатель* – разработчик проектной, программной и эксплуатационной документации как части программного продукта. Технический писатель все время работает с продуктом (его готовыми версиями) и, выступая от имени пользователя, видит все недочеты и несоответствия.

Функции технического писателя:

- *разработка плана документирования*, включающего состав, сроки подготовки и порядок тестирования документов;
- *выбор и разработка стандартов* и шаблонов подготовки документов;
- *выбор средств автоматизации документирования*;
- *разработка документации*;
- *организация тестирования документации*;
- *участие в тестировании продукта*.

*Технолог разработки ПО* обеспечивает выполнение следующих функций:

- *поддержка модели жизненного цикла* программного продукта: создание служб и структур по поддержке работоспособности принятой модели жизненного цикла;
- *создание и сопровождение среды сборки* продукта на завершающих этапах разработки (или гораздо раньше, как, например, при использовании модели прототипирования, когда сборка проводится практически ежедневно);
- *создание и сопровождение процедуры установки* версий продукта;
- *управление исходными текстами*: сопровождение и администрирование системы управления версиями исходных текстов.

*Инженерный психолог* и *инженер по маркетингу* – эти роли присутствуют, как правило, в командах, формируемых для создания коробочных программных продуктов (ориентированных на массового потребителя).

Рольевые модели команд, создаваемых под конкретные программные проекты, могут быть самыми разнообразными и определяются масштабом проекта, используемой методологией и корпоративными стандартами разработки ПО.

### Контрольные вопросы и задания

1 Прокомментируйте понятия *«жизненный цикл ПО»*, *«стадия разработки»* и *«модель жизненного цикла ПО»*.

2 Проанализируйте и прокомментируйте эталонные модели процесса анализа требований программных средств и процесса детального проектирования программных средств, определенных стандартом ISO 12207-2010.

3 Опишите основные черты, преимущества, недостатки и области эффективного применения *каскадной модели*.

4 Опишите основные черты V-модели. Перечислите функции разработчиков и тестировщиков на фазе проекта *«Детальное проектирование компонентов программной системы»*.

5 Дайте общую характеристику *эволюционной модели*. В каких условиях и за счет чего применение этой модели может привести к сокращению сроков выполнения программного проекта ?

6 В чем отличие метода *пробных разработок* от метода *прототипирования* ? Назовите условия применения этих методов.

7 Опишите основные черты *спиральной модели*. Почему спиральную модель относят к категории *гибридных итерационных* моделей? Чем отличается эта модель от каскадной и эволюционной моделей?

8 Используя соответствующие профессиональные стандарты (приложение Б), определите состав *обобщенных трудовых функций* и требования к *квалификационным уровням* IT-специалистов, работающих по профессиям *«программист»*, *«специалист по тестированию»*, *«архитектор ПО»*.

9 Дайте краткую характеристику типовой *ролевой модели* команды программного проекта. Перечислите основные функции *разработчика*, *тестировщика*, *технолога* и *инженера по качеству*.

## ВИЗУАЛЬНОЕ МОДЕЛИРОВАНИЕ В ПРОЕКТИРОВАНИИ

*Проектирование как процесс преобразования информационных моделей; проектные модели: назначение и базовые принципы использования; уровни абстрагирования при моделировании; диаграммы структурного анализа: SADT и ERD.*

### 3.1 Задачи и базовые принципы моделирования программных систем

Модель – это упрощенное представление какого-либо объекта или явления реального мира. Проектные модели строят для того, чтобы исследовать, лучше понять и документировать проектируемую систему. Моделирование позволяет решать проектировщикам следующие основные задачи:

- 1) определение структуры системы как множества взаимосвязанных компонентов;
- 2) определение поведения системы в различных ситуациях;
- 3) визуализация системы в некоторых её состояниях;
- 4) получение шаблона для разработки системы;
- 5) документирование принимаемых проектных решений.

При создании моделей используют ряд базовых принципов, основные из которых – ***абстрагирование***, ***иерархичность*** и ***многомодельность***.

*Принцип абстрагирования* является одним из основных принципов построения моделей сложных систем. Согласно этому принципу модель должна абстрагироваться от второстепенных деталей моделируемого объекта, чтобы чрезмерно не усложнять процесс его анализа и исследования.

*Принцип иерархичности* предписывает рассматривать процесс построения модели сложной системы на разных уровнях ее детализации и, соответственно, на разных стадиях проекта. При этом исходная модель, представляющая систему на самом высоком уровне абстракции, может не содержать многих деталей моделируемой системы и используется на начальном этапе ее проектирования. На последующих стадиях проекта разрабатываются более детализированные и менее абстрактные модели.

*Принцип многомодельности* базируется на следующем утверждении: «никакая единственная модель не может с достаточной степенью адекватности описывать различные аспекты устройства и функционирования сложной системы». Нет идеальных моделей: наилучшее решение заключается в использовании нескольких моделей при разработке сложной системы, отражающих различные аспекты ее структуры или поведения.

Принципы иерархичности и многомодельности иллюстрируются рисунком 3.1, представляющим множество типов моделей, описывающих некоторую систему на трех иерархических уровнях – концептуальном, логическом и физическом (более детальном). При этом каждый уровень представлен моделями двух типов: статические модели, описывающие структуру системы, и динамические модели, описывающие ее поведение в процессе функционирования.

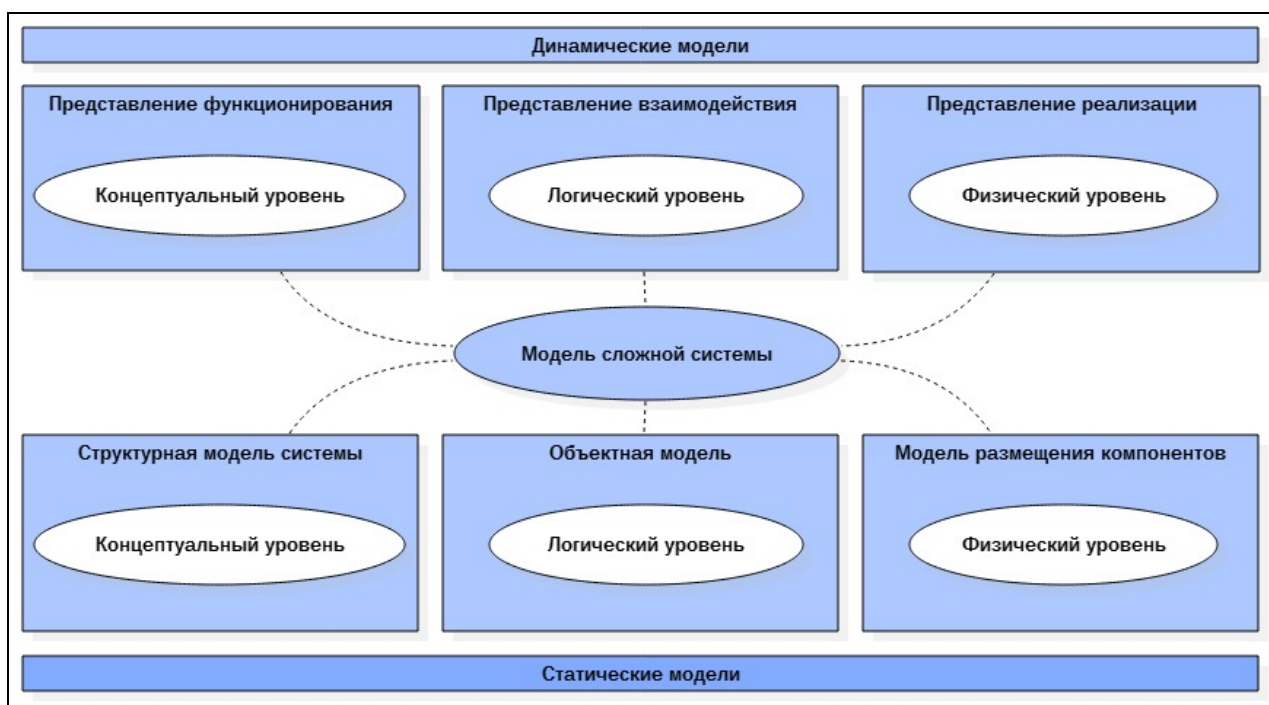


Рисунок 3.1 – Модели сложной системы

### 3.2 Визуализация при моделировании сложных систем

Проектирование, по существу – это производство проектной документации, содержащей описание проектируемого объекта. Процесс проектирования – это информационный процесс, то есть процесс преобразования информации о проектируемом (еще не реализованном) объекте. В этом смысле процесс проектирования объекта можно считать процессом



формирования и последовательного преобразования его информационных моделей.

На разных стадиях проекта информация об объекте представлена с разной степенью детализации: на начальных стадиях используются более абстрактные модели, описывающие требования к функционированию объекта; модели, используемые на завершающих стадиях проекта, существенно более детальны и содержат информацию, достаточную для изготовления образца объекта.

При выполнении крупных проектов сложность моделируемого объекта и, соответственно, объем информации, обрабатываемой проектировщиками, слишком велики для адекватного восприятия проекта одним человеком. По существу, речь идет о биологическом ограничении возможностей мозга человека по хранению и скорости переработки больших объемов информации.

Один из методов преодоления таких ограничений заключается в декомпозиции проектируемого объекта на множество взаимосвязанных компонентов, каждый из которых, естественно, оказывается проще проекта в целом. При этом разработка отдельных компонентов крупного проекта, как правило, поручается различным исполнителям (или группам исполнителей), что частично решает проблему «борьбы со сложностью проекта», но создает ряд дополнительных проблем, связанных с интеграцией компонентов системы и организацией согласованной работы нескольких групп исполнителей проекта.

В этих условиях становится очевидной необходимость визуального моделирования проектируемых объектов, суть которого – в графическом отображении обсуждаемых и принимаемых проектных решений. Визуальное моделирование направлено на достижение следующих целей:

- 1) визуализация упрощает понимание проекта всеми его участниками, включая и представителей заказчика;
- 2) визуализация помогает согласовать со всеми участниками проекта профессиональную терминологию;
- 3) визуализация делает обсуждение конструктивным и понятным.

Важным достоинством визуальных графических моделей является возможность образного закрепления содержательного смысла отдельных

понятий, что существенно упрощает процесс общения между участниками проекта.

Для практического использования средств графической визуализации в процессе анализа и проектирования должен быть создан специальный язык, поддерживающий некоторую графическую нотацию для описания объектов и процессов предметной области.

Система поддержки графических моделей программного проекта должна включать собственно средства визуального моделирования, а также средства контроля непротиворечивости моделей, их преобразования и хранения для многократного использования.

### 3.3 Краткая история развития средств визуального моделирования

#### 3.3.1 Средства визуализации математических моделей

*Теория множеств.* Одна из наиболее известных систем графических символов – это язык диаграмм (рисунок 3.2), предложенный английским логиком Джоном Венном для иллюстрации основных теоретико-множественных операций.

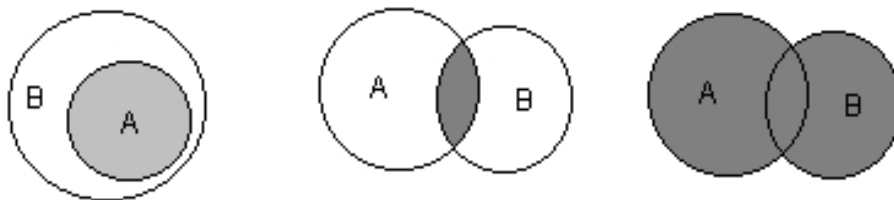


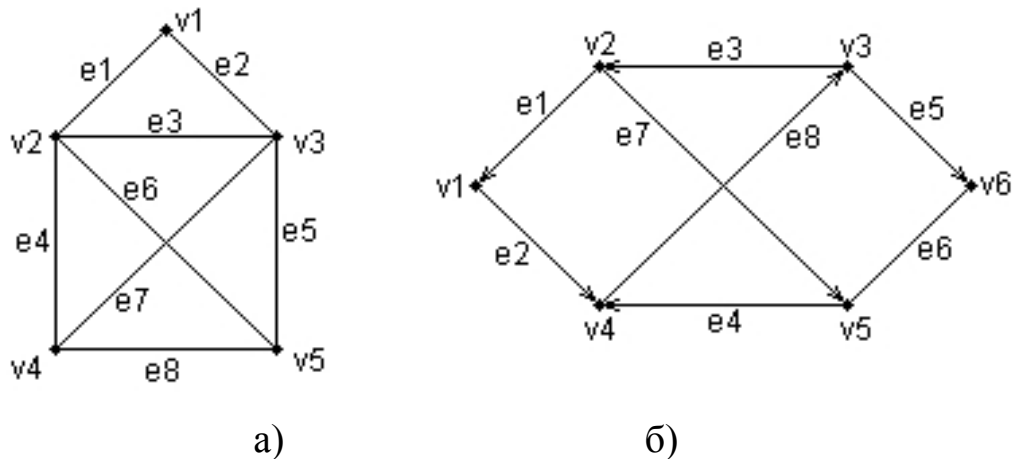
Рисунок 3.2 – Диаграммы Венна

Вряд ли стоит оспаривать тот факт, что графические образы, используемые для обозначений отношений включения, пересечения и объединения множеств, гораздо более информативны и, что еще важнее, гораздо легче запоминаются по сравнению с соответствующими математическими формулами.

*Теория графов.* Другой пример заимствован из теории графов. Формально граф задается множеством  $G=(V, E)$ , где  $V=\{v_1, v_2, \dots, v_n\}$  – множество вершин графа, а  $E=\{e_1, e_2, \dots, e_m\}$  – множество ребер, связывающих вершины графа. На каждом графе должно быть задано некоторое бинарное отношение связности  $PG$ , состоящее из всех пар вида  $(v_i, v_j)$ , где  $v_i, v_j \in V$ . При этом пара  $(v_i, v_j)$  принадлежит отношению  $PG$  в том и только в

том случае, если вершины  $v_i$  и  $v_j$  соединяются в графе  $G$  некоторым ребром  $e_k \in E$ .

Возможно, для математика такое представление графовой модели является естественным и понятным, однако инженеру явно больше понравится (и легко запомнится) визуальная модель, примеры представления которой приведены на рисунке 3.3: вершины графа изображаются точками, а ребра – отрезками линий со стрелками, указывающими направленность связи между вершинами.



а) неориентированный граф;      б) ориентированный граф

*Рисунок 3.3 – Примеры визуального представления графовых моделей*

### 3.3.2 Семантические сети

Семантические сети как средство моделирования первоначально создавались в рамках разработки языков и графических средств для представления знаний и в последующем были конкретизированы и успешно использованы при построении концептуальных моделей баз данных.

Под семантической сетью понимают некоторый граф  $G=(V, E)$ , в котором множество вершин  $V$  и множество ребер  $E$  разделены на отдельные типы, обладающие специальной семантикой, характерной для той или иной предметной области. Например, множество вершин может соответствовать объектам рассматриваемой предметной области, а множество ребер – различным видам связей между объектами.

На рисунке 3.4 приведен пример графического представления фрагмента простейшей семантической сети, с помощью которой моделируются отношения между понятиями некоторой предметной области.

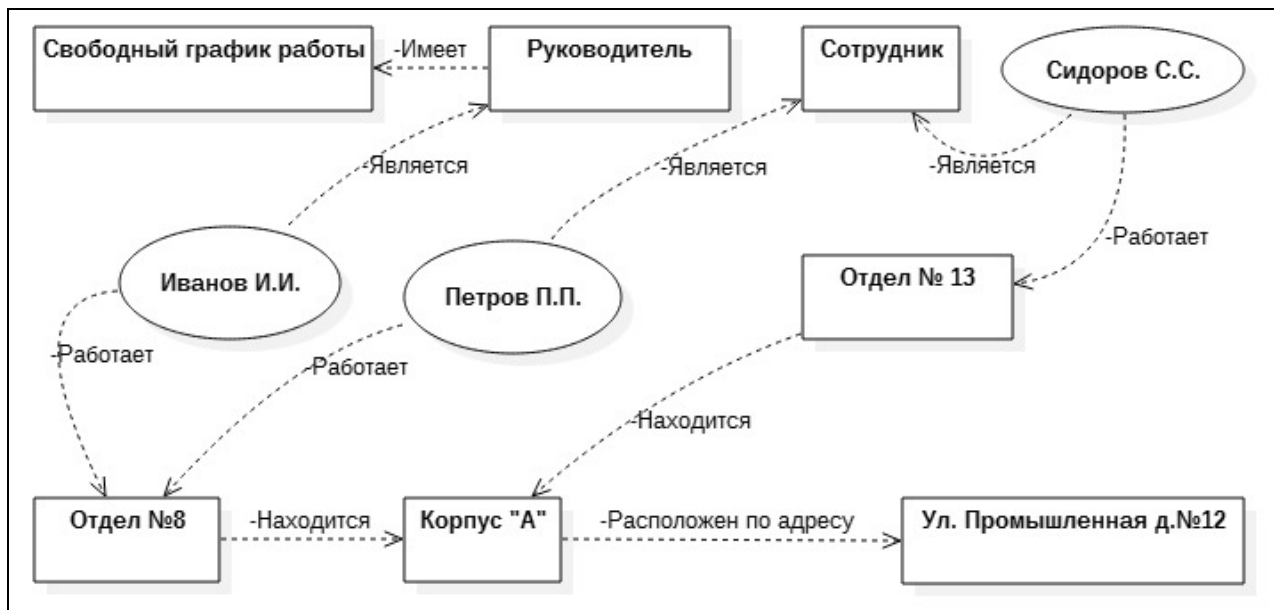


Рисунок 3.4 – Фрагмент семантической сети

### 3.3.3 Диаграммы структурного анализа систем

Структурный анализ использует «нисходящий» метод исследования, начиная с наиболее общего описания системы с последующей детализацией отдельных аспектов ее поведения и функционирования. При этом общая модель системы представляется в виде иерархической структуры, которая отражает различные уровни абстракции с ограниченным числом компонентов на каждом из уровней.

Для проведения структурного анализа программных систем разработано несколько видов диаграмм (моделей), основными являются диаграммы *функционального моделирования* **SADT** (Structured Analysis and Design Technique) и диаграммы «сущность – связь» **ERD** (Entity-Relationship Diagrams).

#### *Диаграммы функционального моделирования*

Начало разработки диаграмм функционального моделирования относится к середине 1960-х годов, когда была предложена специальная *техника структурного анализа и проектирования* SADT. Эта техника использовалась в рамках программы ICAM (Integrated Computer Aided Manufacturing), целью которой было повышение эффективности промышленного производства за счет применения компьютерных технологий.

В рамках программы ICAM была разработана система графических нотаций, используемых для документирования и моделирования бизнес-

процессов, производственных процессов, процессов управления ресурсами. В 1993 году на базе этих нотаций был создан стандарт правительства США, который послужил основой для реализации первых CASE-средств, поддерживающих методологию функционального моделирования.

Функциональная модель SADT (рисунок 3.5) позволяет наглядно представить деятельность системы в целом и деятельность отдельных ее подсистем во взаимодействии с другими подсистемами.

*Деятельность* представляет собой некоторый процесс, который имеет фиксированную цель и приводит к конечному результату. На диаграммах деятельность изображается прямоугольником, который называется *блоком*.

*Стрелки* на диаграммах соединяют блоки деятельности и служат для обозначения переноса информации между ними. Модель поддерживает четыре вида стрелок: *I (input)* – вход; *C (control)* – управление или ограничения на выполнение операций процесса; *O (output)* – выход или результат процесса; *M (mechanism)* – механизм, который реализуют данный процесс.

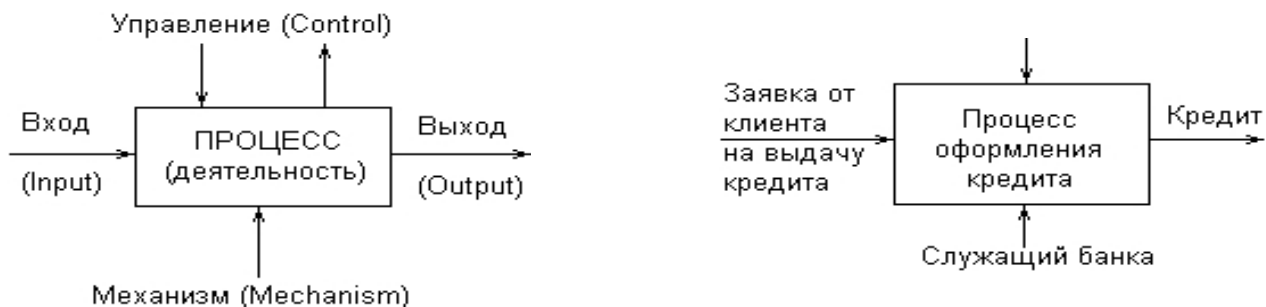


Рисунок 3.5 – Примеры SADT-диаграмм

Важной особенностью методологии SADT является последовательная детализация модели системы по мере ее разработки: построение модели начинается с представления всей системы в виде простейшей диаграммы, состоящей из одного блока процесса и стрелок, служащих для изображения основных видов взаимодействия с внешними по отношению к системе объектами. Исходный процесс, представляющий всю систему как единое целое, подлежит дальнейшей декомпозиции на следующем этапе проектирования; частные процессы, полученные в результате декомпозиции исходного процесса, также декомпозируются – и так до некоторого разумного предела, пока уровень детальности рассмотрения свойств про-

цесса не окажется достаточным для его реализации. В конечном итоге модель SADT представляет собой множество иерархически взаимосвязанных диаграмм, которые представляют сложную систему отдельными ее составными частями.

### *Диаграммы «сущность – связь»*

ER-диаграммы предназначены для графического представления моделей данных программных систем и используются на начальных стадиях проектов баз данных – при разработке концептуальных моделей предметной области. Основными компонентами ER-модели являются *сущность* (entity) и *связь* (relationship), а также *атрибуты* (свойства) сущностей и связей.

Под *сущностью* понимается абстракция множества реальных объектов (или процессов) предметной области, имеющих одинаковый набор свойств (атрибутов), существенных в контексте проектируемой программной системы. При этом каждый реальный объект может быть представлен экземпляром одной сущности, а среди экземпляров одной сущности не должно быть дубликатов. И сущности, и их атрибуты – это именованные объекты модели, каждому из которых должно быть присвоено уникальное имя (идентификатор).

Для обозначения сущностей на ER-диаграммах используют набор стандартных графических элементов – прямоугольников с дополнительными элементами оформления, внутри которых записывают имена сущностей.

Под *связью* понимают абстракцию некоторого отношения (ассоциации) между реальными объектами предметной области, существенного в контексте проектируемой программной системы. Программная реализация связей в базе данных позволяет находить экземпляры одной сущности по их связям с экземплярами других сущностей. Связь, так же как и сущность, является именованным объектом ER-модели и тоже может иметь свои собственные атрибуты.

Связь на ER-диаграммах обозначается соединяющей сущности линией, в разрыве линии помещается ромб, внутри которого записывается имя связи. Связь отображает не только семантику отношения между сущностями, но и дополнительные характеристики связи: *арность*, определяемую количеством сущностей, участвующих в связи, и *кратность*, определяемая количеством экземпляров сущностей, участвующих в связи.

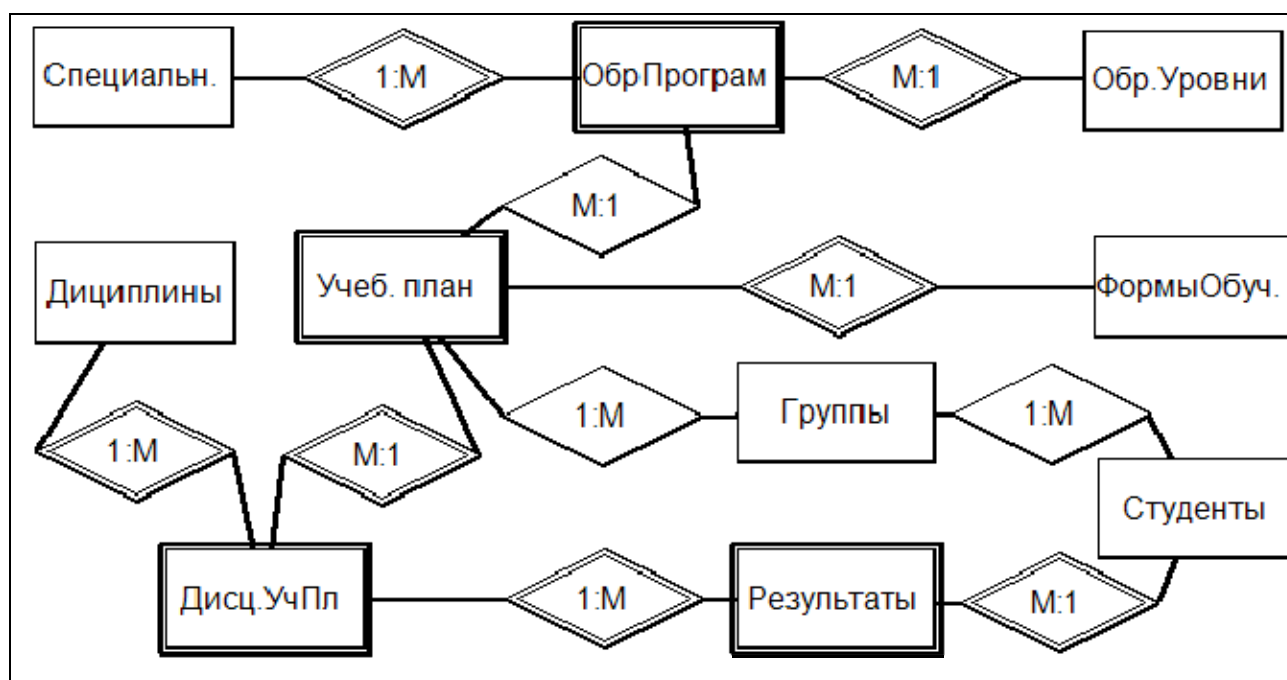


Рисунок 3.6 – ER-диаграмма «Учебный процесс»

ER-диаграмма, приведенная на рисунке 3.6, представляет упрощенную структурную модель концептуального уровня, описывающую учебный процесс, реализуемый в высшем учебном заведении. Связи между сущностями модели позволяют, в частности, получить перечень студенческих групп по специальностям и образовательным уровням, перечни дисциплин учебных планов и результаты успеваемости студентов по учебным дисциплинам.

Завершая обзор средств визуального моделирования, поддерживающих методологию структурного анализа систем, можно сделать следующие выводы.

1 Исследования в области структурного анализа сложных систем позволили выработать базовые концепции визуального моделирования, создать и апробировать в CASE-системах графическую нотацию для отображения моделей.

2 В настоящее время нотации диаграмм структурного анализа используются в ряде CASE-средств для построения информационных моделей систем обработки данных.

3 Возможности средств визуального моделирования, разработанных в поддержку методологии структурного анализа, оказались сильно ограниченными при переходе к объектно-ориентированным технологиям разработки сложных систем.

4 Основные недостатки этой методологии и поддерживающих ее графических нотаций:

- отсутствие средств представления сложных алгоритмов обработки данных;
- недостаточная развитость средств отображения временных характеристик процессов и потоков данных;
- отсутствие явных средств объектно-ориентированного представления моделей сложных систем.

5 Многие идеи визуального моделирования и элементы графических нотаций моделей структурного анализа были эффективно использованы при разработке UML – языка моделирования, поддерживающего методологию объектно-ориентированного анализа и проектирования сложных систем.

### **Контрольные вопросы и задания**

1 Для чего используют модели сложных систем при их проектировании? Перечислите основные задачи, решаемые проектировщиками систем с помощью моделирования.

2 Прокомментируйте принципы абстрагирования, иерархичности и многомодельности, используемые при разработке моделей сложных систем.

3 Какие цели преследует визуальное моделирование систем ?

4 В какой области знаний используются диаграммы Венна? Приведите примеры.

5 Смоделируйте с помощью семантической сети отношения между понятиями «Факультет», «Специальность», «Студент», «Учебная дисциплина», «Кафедра» и «Преподаватель» предметной области «Учебный процесс».

6 Расшифруйте сокращенные названия диаграмм *SADT* и *ERD*. Для чего используются диаграммы перечисленных типов ?

7 Перечислите компоненты ER-модели, дайте им определения. Предложите свой вариант ER-модели системы учета книжного фонда публичной библиотеки.



**UML – УНИВЕРСАЛЬНЫЙ ЯЗЫК МОДЕЛИРОВАНИЯ**

*История стандартизации средств объектно-ориентированного моделирования; базовые понятия языка UML; классификация UML-моделей: структурные модели, модели поведения и модели взаимодействия; обзор UML-диаграмм: диаграммы вариантов использования; диаграммы пакетов; диаграммы классов; диаграммы состояний.*

**4.1 История стандартизации языка UML**

Переход от методологии структурного анализа систем к объектно-ориентированным технологиям их проектирования и программирования потребовал создания новых методов, языков и CASE-средств моделирования, при разработке которых были частично использованы многие идеи структурного анализа, рассмотренные в предыдущей главе.

Первые языки объектно-ориентированного моделирования стали появляться в конце 1970-х, к середине 90-х годов их число возросло до 50, причем ни один из языков не удовлетворял всем требованиям, предъявляемым к построению моделей сложных систем, что, естественно, создавало серьезные затруднения для разработчиков CASE-средств и препятствовало широкому распространению объектно-ориентированного подхода.

*Начало работ по унификации* объектно-ориентированных методов относят к 1994 году, когда компании *Rational Software Corporation* и *Objectory AB* совместно сформулировали следующие требования к языку моделирования:

- язык должен быть универсальным, то есть позволять моделировать с использованием объектно-ориентированных понятий не только программные системы, но и более широкие классы систем и бизнес-приложений;
- язык должен явным образом обеспечивать взаимосвязь между базовыми понятиями моделей концептуального и физического уровней;
- язык должен обеспечивать масштабируемость моделей, что является важной особенностью сложных многоцелевых систем;
- язык должен быть понятен и аналитикам, и программистам, а также должен поддерживаться специальными инструментальными средствами.

В 1995 году к разработке UML подключается консорциум *OMG* (*Object Management Group*), и этот год принято считать годом начала разработки промышленных стандартов в области языков объектно-ориентированного моделирования систем.

В 1996 году вышло первое описание языка UML версии 0.9, имевшее статус *запроса предложений*. В этом же году был учрежден консорциум *партнеров UML*, в который вошли такие компании, как *DEC, HP, i-Logix, Intellicorp, IBM, ICON Computing, MCI Systemhouse, Microsoft, Oracle, Rational Software*.

В 1997 году опубликовано описание языка UML версии 1.0, и в этом же году принята в качестве стандарта *OMG* версия UML 1.1.

В 1998 году компания *Rational Software Corporation* выпустила на рынок первое CASE-средство ***Rational Rose 98***, реализованное на базе UML.

В период с 1999 по 2004 годы были выпущены версии UML 1.3, 1.4, 1.5 и 2.0, а в 2005 году был принят международный стандарт *ISO/IEC 19501:2005 Information technology – Open Distributed Processing – Unified Modeling Language (UML)*.

Полный перечень версий UML доступен на официальном интернет-ресурсе [14], последняя (к моменту написания учебного пособия) версия UML 2.5.1 опубликована в декабре 2017 года.

## 4.2 Структура и базовые понятия языка UML

UML не является языком программирования – он служит средством для решения задач объектно-ориентированного моделирования систем. Основное назначение языка UML – *визуальное моделирование и документирование моделей сложных систем различного назначения*.

Спецификация UML-моделей не зависит от конкретных языков программирования, программная поддержка конструкций UML осуществляется специальными инструментальными CASE-средствами.

Язык UML состоит из двух взаимосвязанных и дополняющих друг друга частей: *семантики языка*, определяющей абстрактный синтаксис и семантику понятий объектного моделирования, и *графической нотации* для визуального представления элементов языка на UML-диаграммах.

Семантика языка определяет две категории объектных моделей, описывающих систему в терминах объектно-ориентированного подхода:

структурные (статические) модели и модели поведения (динамические). Структурные модели описывают состав и взаимосвязи компонентов системы в терминах *классов, интерфейсов, атрибутов и отношений*. Модели поведения описывают функционирование объектов системы в терминах *методов, взаимодействия и сотрудничества* между ними, а также процесс *изменения состояний* системы и отдельных ее компонентов.

### 4.3 Элементы моделей языка UML

Для описания моделей язык UML использует более двухсот элементов, организованных в логические блоки – так называемые *пакеты*. Пакет – это своеобразный «контейнер», предназначенный для группировки различных элементов UML-модели, в том числе и самих пакетов.

Для графического изображения пакетов на диаграммах применяется специальный графический символ (рисунок 4.1), внутри которого может записываться информация, относящаяся к данному пакету.

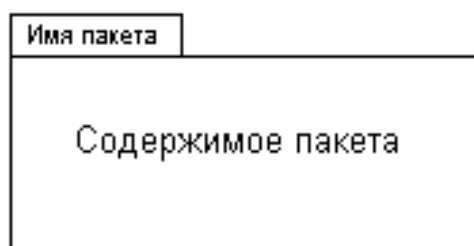
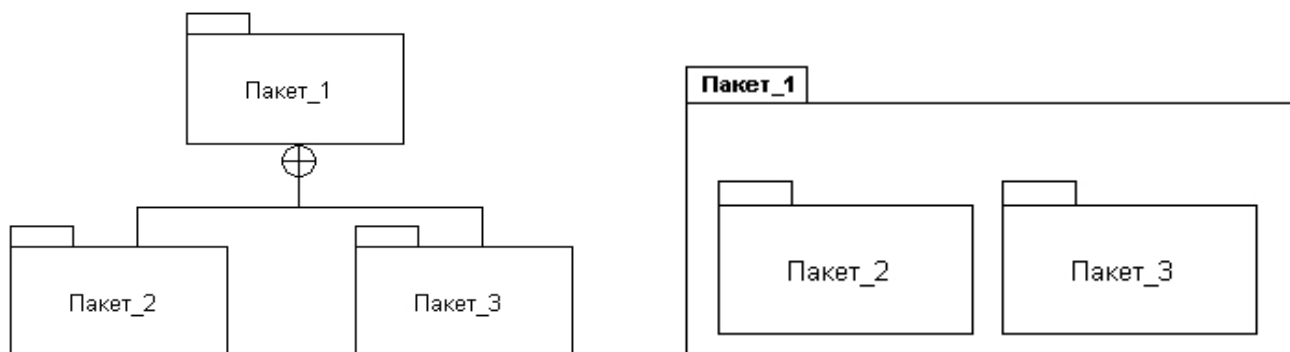


Рисунок 4.1 – Графическое изображение пакета в языке UML

Имя пакета должно быть уникальным, перед именем допускается указывать так называемый *стереотип* – служебное слово из предопределенного набора, например *facade*, *framework*, *stub* или *topLevel*. Содержимое пакета – это имена входящих в него элементов и (возможно) их свойства, определяющие, например, видимость элементов за пределами пакета.

Между пакетами может быть задано *иерархическое отношение вложенности*: подчиненные пакеты («*подпакеты*») могут быть вложены в другие пакеты («*метапакеты*») – в этом случае все элементы подпакета будут принадлежать метапакету. Два возможных варианта графического представления вложенности пакетов на UML-диаграммах иллюстрируются рисунком 4.2.



*Рисунок 4.2 – Графическое изображение вложенности пакетов*

Множество элементов моделей языка UML распределены по иерархически организованным пакетам. На верхнем (метамодельном) уровне язык содержит один метапакет (собственно, язык) и три подпакета (рисунок 4.3), каждый из которых, в свою очередь, также выступает в роли метапакета и содержит определенный набор подпакетов (рисунок 4.4).



*Рисунок 4.3 – Основные пакеты метамодели языка UML*



*Рисунок 4.4 – Подпакеты метапакета «Основные элементы»*

Подпакет «Элементы ядра» является наиболее фундаментальным – он описывает базовые понятия языка UML (рисунок 4.5).

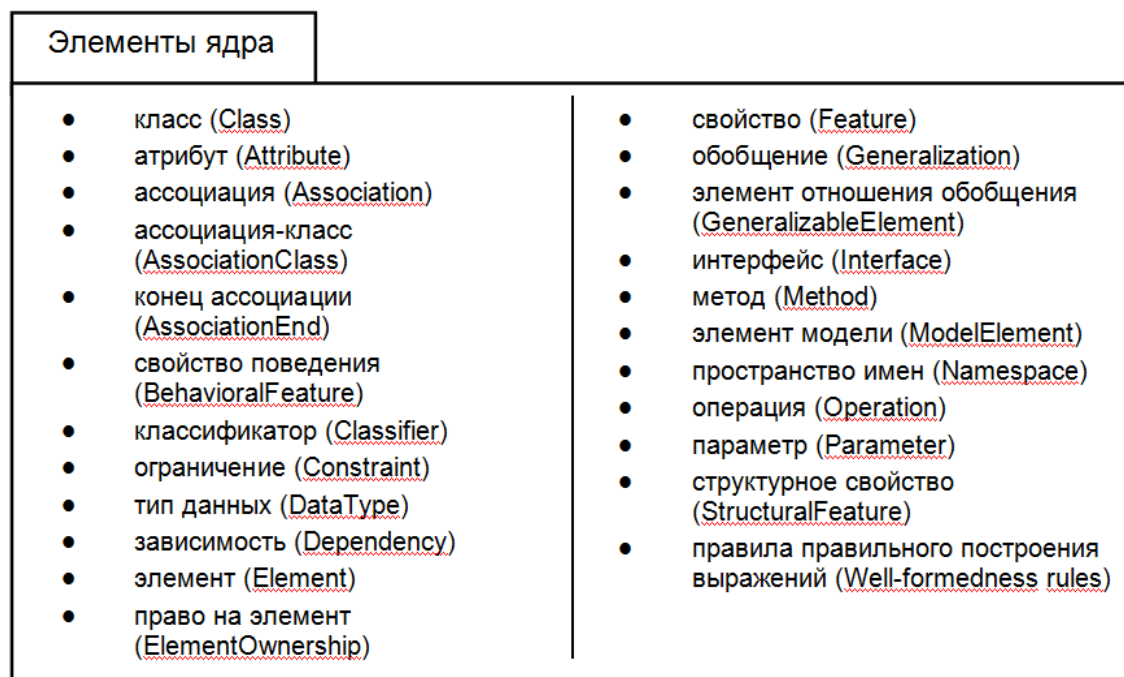


Рисунок 4.5 – Структура пакета «Элементы ядра»

Другим важнейшим компонентом языка UML является пакет «Элементы поведения» (рисунок 4.3), который специфицирует поведение динамических элементов и состоит из четырех подпакетов (рисунок 4.6).



Рисунок 4.6 – Структура пакета «Элементы поведения»

Пакет «Общее поведение» специфицирует семантику всех динамических элементов, включенных в другие подпакеты метапакета «Элементы поведения». Пакет включает следующие элементы:

- Объект (Object);
- Действие (Action);
- Последовательность действий (ActionSequence);
- Аргумент (Argument);
- Экземпляр (Instance);
- Исключение (Exception);

- Связь (Link);
- Сигнал (Signal);
- Значение данных (DataValue);
- Связь атрибутов (AttributeLink);
- Действие вызова (CallAction);
- Действие создания (CreateAction);
- Действие уничтожения (DestroyAction).

Пакет «*Кооперации*» включает элементы, используемые для описания *взаимодействия* между элементами модели с точки зрения классификаторов и ассоциаций. В пакет входят следующие элементы, используемые при построении диаграмм кооперации:

- Кооперация (Collaboration);
- Взаимодействие (Interaction);
- Сообщение (Message);
- Роль ассоциации (AssociationRole);
- Роль классификатора (ClassifierRole);
- Роль конца ассоциации (AssociationEndRole).

Пакет «*Варианты использования*» включает элементы модели, используемые для первоначального определения поведения (функциональности) моделируемой системы без детализации ее внутренней структуры. В пакет входят следующие элементы:

- Актор (Actor);
- Вариант использования (UseCase);
- Расширение (Extension);
- Точка расширения (ExtensionPoint);
- Включение (Include);
- Экземпляр варианта использования (UseCaseInstance).

Пакет «*Автоматы*» содержит множество понятий, которые необходимы для представления поведения системы в виде *переходов* из одних *состояний* в другие под воздействием разнообразных *событий*, при этом *состояние* может использоваться для моделирования процессов выполнения некоторой деятельности. *Автоматы* используются для моделирования поведения экземпляров классов, а также для спецификации взаимодействий между сущностями, такими как *кооперации*.

В пакет включены следующие элементы:

- Автомат (StateMachine);

- Состояние (State);
- Простое состояние (SimpleState);
- Составное состояние (CompositeState);
- Псевдосостояние (PseudoState);
- Конечное состояние (FinalState);
- Событие (Event);
- Переход (Transition).

## 4.4 Диаграммы языка UML

UML-диаграммы предназначены для визуализации моделей и представлены тремя категориями: структурные диаграммы, диаграммы поведения и диаграммы взаимодействия (рисунок 4.8). Каждая из диаграмм (вспомним принцип многомодельности) детализирует и конкретизирует в терминах языка UML определенное представление о модели сложной системы.

### Структурные диаграммы

*Диаграмма пакетов (Package Diagram)* представляет результаты функциональной декомпозиции проектируемой системы на концептуальном уровне.

*Диаграмма классов (Class Diagram)* представляет результаты объектной декомпозиции системы в стиле ООАП: показывает классы, их атрибуты и методы, связи между классами.

*Диаграмма компонентов (Component Diagram)* показывает компоненты и связи между ними.

*Диаграмма развертывания (Deployment Diagram)* показывает, как ПО размещается на элементах аппаратного комплекса.

### Диаграммы поведения

*Диаграмма вариантов использования (UseCase Diagram)* показывает работу системы с точки зрения пользователей.

*Диаграмма состояний (Statechart Diagram или Statemachine Diagram)* представляет поведение системы, как процесс смены ее состояний, управляемый внешними и внутренними событиями.

*Диаграмма деятельности (Activity Diagram)* – это, по существу, упрощенная диаграмма состояний. Основное назначение диаграммы – представление бизнес-процессов и алгоритмов реализации методов классов.

## Диаграммы взаимодействия

Диаграмма кооперации (*Collaboration Diagram*) показывает структурную организацию участвующих во взаимодействии объектов (в более поздних версиях UML заменена на *Communication Diagram*, рисунок 4.8).

Диаграмма последовательности (*Sequence Diagram*) показывает хронологическую упорядоченность событий.

Диаграмма вариантов использования является исходной для построения всех остальных диаграмм и представляет наиболее общую концептуальную модель сложной системы.

Диаграмма классов представляет модель логического уровня, отражающую статические аспекты структуры моделируемой системы. Этот же (логический) уровень модели представляют и диаграммы поведения, но они, в отличие от диаграмм классов, отражают динамические аспекты функционирования системы.

Диаграмма компонентов и диаграмма развертывания служат для представления компонентов физической модели и используются для моделирования системы на уровне ее реализации.

Таким образом, интегрированная модель сложной системы в нотации UML (ранних версий) представляется в виде совокупности указанных выше диаграмм, как это показано на рисунке 4.7.



Рисунок 4.7 – Интегрированная модель системы в нотации UML-диаграмм



## 4.5 Общие правила графической нотации UML-диаграмм

UML-диаграмма представляет собой граф: вершины графа – это графические символы, представляющие элементы соответствующих моделей, а дуги графа обозначают связи между этими элементами. Форма графических символов и соединяющих их линий определяется типом диаграммы, размеры и расположение элементов на диаграмме, как правило, не имеют значения.

Внутри основных графических символов может размещаться текст, определяющий наименование и свойства элемента модели или уточняющий его семантику, а также другие графические символы.

Дуги графа могут сопровождаться текстами и могут иметь на своих концах специальные значки (пиктограммы), например, стрелки различных видов.

Пиктограммы могут размещаться как внутри основных графических символов, так и вне их на свободном поле диаграммы, и представляют собой графические фигуры фиксированной формы и размера, внутри которых не допускается размещать текст и дополнительные символы (например, пиктограммой  $\oplus$  обозначено отношение вложенности пакетов на рисунках 4.2, 4.3, 4.4 и 4.6).

При построении UML-диаграмм следует придерживаться следующих основных рекомендаций.

1 Каждая диаграмма должна служить законченным представлением соответствующего фрагмента моделируемой предметной области. Отсутствие существенных элементов на диаграмме служит признаком неполноты модели и может потребовать ее последующей доработки.

2 Все элементы диаграммы должны принадлежать одному концептуальному уровню представления модели (рисунок 4.7). При этом отдельные фрагменты диаграммы могут детализироваться на других диаграммах этого же типа, образуя, таким образом, иерархию вложенных диаграмм.

3 Необходимо стремиться к явному указанию свойств всех элементов диаграмм, несмотря на то, что язык UML в ряде случаев допускает использование значений по умолчанию.

4 Не следует перегружать диаграммы текстовой информацией – визуализация модели является наиболее эффективной, если она содержит минимум поясняющего текста.

5 Состав диаграмм, используемых в конкретном программном проекте, не является строго фиксированным и определяется на стадии технического задания в зависимости от специфики проекта.

6 Таким образом, модель системы на языке UML представляет собой пакет диаграмм различных типов, степень детализации которых на нижних уровнях иерархии должна быть достаточной для последующей генерации программного кода, реализующего проект.

Ранние версии языка UML включали 9 различных диаграмм, и практически с каждой новой версией их число увеличивается. В версии UML 2.5, выпущенной в 2015 году, представлено уже 14 основных и 8 дополнительных диаграмм (рисунок 4.8), описывающих структуру и поведение систем.

Далее в этой главе будут рассмотрены UML-диаграммы четырех типов, используемые в подавляющем большинстве программных проектов.

*Диаграмма вариантов использования*, разрабатываемая на начальной стадии проекта для описания функционального поведения системы.

*Диаграмма пакетов*, используемая для представления результатов структурной декомпозиции проектируемой системы на концептуальном уровне.

*Диаграмма классов*, представляющая результаты объектной декомпозиции проектируемой системы на логическом уровне.

*Диаграмма состояний*, описывающая динамические аспекты поведения системы в форме смены ее состояний под воздействием внешних и внутренних событий.

Модель системы, представляемая этими четырьмя диаграммами, позволяет проектировщику ответить на следующие базовые вопросы:

- 1) *что делает система ? (UseCase Diagram)*
- 2) *как она устроена ? (Package Diagram и Class Diagram)*
- 3) *как она работает ? (Statemachine Diagram)*

Детальное описание остальных диаграмм языка UML и технологий их использования приведено в [6] и [14].

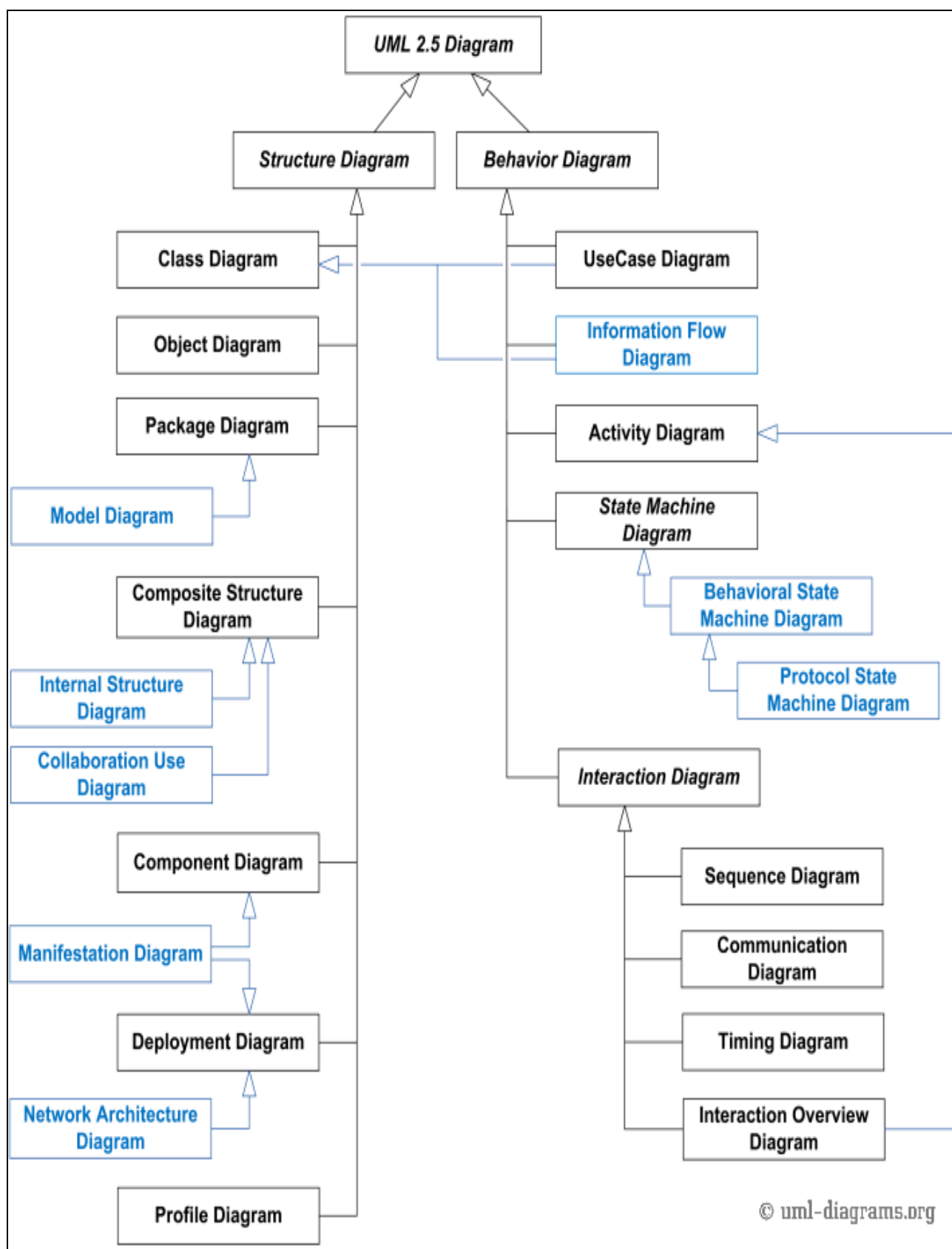


Рисунок 4.8 – Состав UML-диаграмм (версия UML 2.5)

## 4.6 UML-диаграмма вариантов использования

*Разработка требований* – обязательный начальный этап программного проекта, целью которого является определение свойств и характеристик проектируемой системы, необходимых для удовлетворения потребностей пользователей и других заинтересованных лиц.

Сформулировать требования к сложной системе не так легко – в большинстве случаев заказчик может перечислить только приблизительный набор функций, выполняемых системой. При этом формулировки заказчика часто будут непонятны большинству разработчиков ПО, а терминология разработчиков будет непонятной специалистам в предметной области.

Чтобы проектируемая система была действительно полезной, важно, чтобы она соответствовала реальным потребностям организаций, которые часто отличаются от «пожеланий», непосредственно выражаемых пользователями. *«Заказчику надо дать не то, что он просит, а то, что ему надо»* – и в этой шутке (приписываемой советскому кибернетику, академику В.М. Глушкову) – только доля шутки.

Для выявления этих потребностей, а также для выяснения смысла высказанных заказчиком требований проводится *моделирование бизнес-процессов*, которым занимаются *системные аналитики*, которые передают полученные ими знания другим членам команды проекта, сформулировав их на языке графических моделей и текстовых документов.

Для моделирования бизнес-процессов в свое время были разработаны (и до сих пор успешно используются) диаграммы структурного анализа. Язык UML для этих целей предлагает *диаграмму вариантов использования*, называемую также *диаграммой прецедентов*.

*Диаграмма вариантов использования* – это форма представления концептуальной модели проектируемой системы, которая описывает ее функциональное назначение и будет детализирована и преобразована в модели логического и физического уровней на следующих стадиях программного проекта.

Разработка *UseCase*-диаграммы преследует следующие цели:

- 1) определить общие границы и контекст моделируемой предметной области на начальных этапах проектирования системы;

- 2) сформулировать общие требования к функциональному поведению системы;
- 3) разработать исходную концептуальную модель системы для ее последующей детализации моделями логического и физического уровней;
- 4) подготовить исходную документацию для взаимодействия разработчиков системы с ее заказчиками и пользователями.

#### **4.6.1 Компоненты UseCase-диаграммы**

UseCase-диаграмма использует три типа основных компонентов: собственно *вариант использования (use case)*, *действующее лицо* или *актор (actor)* и *связь* или *отношение (relationship)*, а также дополнительные компоненты двух типов: *интерфейсы (interface)* и *примечания (notes)* – произвольные текстовые комментарии разработчика, имеющие отношение к другим компонентам UseCase-диаграммы.

**Актор** – это абстракция любой сущности, взаимодействующей с системой извне, его свойства и внутренняя структура в контексте данной диаграммы не рассматриваются. Актором может быть группа пользователей, техническое устройство, программа или любая другая система, которая служит источником воздействия на моделируемую систему и/или потребляет информацию, генерируемой системой.

Актор представляет определенную *роль*, включающую множество экземпляров субъектов, одинаково взаимодействующих с системой, то есть связанных с одинаковыми *вариантами использования*. Актеры могут быть связаны друг с другом *отношениями обобщения*, в этом случае считается, что актер-потомок наследует все связи с вариантами использования, имеющимися у его предка, и при этом может участвовать в дополнительных связях. Допускается множественное наследование, когда у одного актора имеется более одного предка.

Не следует путать понятия «актор» и «пользователь»: один и тот же пользователь может быть экземпляром нескольких *ролей*, то есть представлять в системе нескольких акторов.

Актеры взаимодействуют с вариантами использования посредством передачи им запросов и получения от них соответствующих сервисов. Такое взаимодействие выражается посредством *ассоциативных связей*. Кро-

ме этого, с актерами могут быть связаны интерфейсы, уточняющие способы такого взаимодействия. Актор представляется на диаграмме вершиной графа и изображается в виде схематичного человечка, помеченного именем соответствующей *роли*.

**Варианты использования** – это результат функциональной декомпозиции проектируемой системы, они служат для описания сервисов, которые система предоставляет актерам. Каждый вариант использования определяет некоторый сценарий – законченную последовательность действий, совершаемых системой при взаимодействии с актором или с другими вариантами использования, причем детали реализации такого взаимодействия в диаграмме не раскрываются.

Вариант использования представляется на UseCase-диаграмме вершиной графа и изображается в виде овала, внутри которого записывается имя. Варианты использования могут быть связаны с одним или несколькими актерами, другими вариантами использования и интерфейсами. Варианты использования, непосредственно связанные (ассоциированные) с актерами, называются *базовыми*, варианты использования, связанные с актерами косвенно, через их связи с базовыми вариантами использования, называются *подчиненными*.

Вариант использования имеет следующие *атрибуты* (все они, кроме имени, не указываются на диаграмме, но приводятся в сопроводительной документации – в описании *сценария варианта использования*):

- *имя*, кратко определяющее его назначение;
- *описание*, кратко поясняющее предоставляемый им сервис;
- *тип* (базовый или подчиненный);
- *частота* выполнения;
- *предусловия* и *постусловия*;
- *основной сценарий* работы, описывающий типичный ход событий, когда не возникает альтернативных ситуаций;
- *альтернативные сценарии* (в том числе, используемые в исключительных ситуациях);
- *ассоциированные акторы* (необязательно);
- *включаемые варианты использования* (необязательно);
- *расширяемые варианты использования* (необязательно);

- *статус* (необязательно), например: «в разработке», «готов к проверке», «в процессе проверки», «подтвержден», «отвергнут».

**Интерфейсы** (interface) в UseCase-диаграммах используются для конкретизации *операций, обеспечивающих выполнение вариантов использования*. Интерфейс представляется вершиной графа и изображается в виде графического символа – маленького круга, соединенного линией с тем вариантом использования, который его поддерживает. Рядом с символом интерфейса записывается его имя, например, *датчик, видеокамера, запрос к базе данных* или *устройство подачи звукового сигнала*. Интерфейсы не могут иметь ни атрибутов, ни состояний – они могут содержать только наименования операций.

**Связи** (relationships) используются в UseCase-диаграммах для обозначения различных отношений между компонентами модели. Связи отображаются дугами графа – линиями определенного вида, попарно соединяющими другие компоненты диаграммы (акторов, варианты использования и интерфейсы).

UseCase-диаграммы могут использовать один из пяти видов связей, каждый из которых соответствует определенному типу отношения между компонентами модели: *ассоциация* (association relationship), *обобщение* (generalization relationship), *включение* (include relationship), *расширение* (extend relationship) или *зависимость* (dependency relationship).

Отношение **ассоциации** является элементом многих UML-диаграмм и обозначается сплошной линией, которая может иметь *имя ассоциации* и специальные знаки на *концах ассоциации*, обозначающие ее *направленность* или *кратность* (multiplicity). В рассматриваемой диаграмме ассоциация обозначает взаимодействие *актера и варианта использования*, а кратности концов ассоциации определяют количество экземпляров этих элементов, участвующих в связи. Для обозначения кратности ассоциации можно использовать целые неотрицательные числа, множества чисел, например, «3; 10; 12», их диапазоны, например, «1..12», или символ «\*», обозначающий *любое целое неотрицательное число* (значение по умолчанию).

Отношение **обобщения** также является элементом многих UML-диаграмм и обозначает связь типа «предок – потомок», указывая на то, что *потомок* является специальным случаем своего *предка*. В рассматриваемой диаграмме отношение обобщения может быть установлено или между парой акторов, или между парой вариантов использования и обозначается

сплошной линией со стрелкой в виде незакрашенного треугольника, направленной от *потомка* к *предку*.

**Отношения включения** и **расширения** являются направленными бинарными отношениями между базовым и подчиненным вариантами использования. При этом функциональное поведение, заданное для подчиненного варианта использования, дополняет поведение базового варианта, а выполнение подчиненного варианта всегда происходит по инициативе базового.

Подчиненный вариант использования предоставляет базовому варианту некоторое инкапсулированное поведение, детали реализации которого скрыты от базового варианта. Поведение базового варианта использования зависит только от результатов выполнения подчиненного ему варианта, но не зависит ни от его структуры, ни от способа его реализации.

Один вариант использования может быть подчинен нескольким базовым вариантам и при этом может сам выступать в роли базового варианта по отношению к другим, подчиненным ему вариантам использования.

Отличие между этими двумя типами отношений заключается в способе вызова подчиненного варианта из базового.

Если между парой вариантов использования задано **отношение включения**, это означает, что подчиненный вариант **безусловно** включен во все экземпляры базового варианта, а его поведение выделено из поведения базового варианта лишь по той причине, что оно включается в несколько различных базовых вариантов. Графически отношение включения обозначается *пунктирной линией со стрелкой*, направленной от базового варианта использования к подчиненному (включаемому) и помеченной специальным *стереотипом* – ключевым словом «include».

**Отношение расширения** тоже определяет взаимосвязь между базовым и подчиненным вариантами использования, но в этом отношении подчиненный вариант *расширяет* поведение базового варианта только в альтернативных (в том числе – исключительных) ситуациях. В описании сценария базового варианта могут указываться *точки расширения*, в которых будет проверяться выполнение соответствующего условия. Если для каких-то экземпляров базового варианта это условие выполняется, поведение базового варианта будет расширено поведением подчиненного варианта. Отношение расширения также обозначается *пунктирной линией со стрелкой*, но, в отличие от отношения включения, стрелка здесь направ-



лена *от подчиненного варианта* использования к *базовому варианту* и помечена стереотипом «*extend*».

Отношение **зависимости** используется во многих UML-диаграммах для описания ситуации, в которой изменение одного элемента модели, называемого *независимым* элементом или *источником зависимости*, может потребовать изменения другого, *зависимого* от него элемента, называемого *клиентом зависимости*. В рассматриваемой диаграмме отношение зависимости может быть установлено между парой вариантов использования, в которой поведение одного варианта использования (*клиента зависимости*) зависит от свойств или поведения другого варианта (*источника зависимости*).

Отношение зависимости изображается на диаграмме пунктирной линией со стрелкой, направленной *от клиента к источнику зависимости*, но эта линия, в отличие от визуально похожих на нее линий *включения* и *расширения*, может не помечаться ключевыми словами-стереотипами. Более детально отношение зависимости будет рассмотрено далее в разделе 4.8 учебного пособия.

#### 4.6.2 Пример UseCase-диаграммы

На рисунке 4.9 приведен пример UseCase-диаграммы, разработанной в рамках проекта автоматизированной системы формирования расписания учебных занятий.

В проектируемой системе определены три основных роли акторов: *Студенты*, *Кафедра* и *Учебный отдел*, ассоциированные с базовым вариантом использования *Поиск и просмотр*, с которым связаны *отношением расширения* три подчиненных варианта использования: *Расписание преподавателей*, *Расписание групп*, и *Свободные аудитории* (очевидно, в реализации базового варианта использования будет разработано пользовательское меню с альтернативным выбором соответствующих операций просмотра расписания).

Роли акторов *Преподаватели* и *Заведующий кафедрой* связаны *отношением обобщения* с ролью *Кафедра*, и это означает, что членам (экземплярам) всех этих ролей будет доступен сервис *Поиск и просмотр*, но актер *Заведующий кафедрой* дополнительно ассоциирован с вариантом использования *Распределение дисциплин*, а актер *Преподаватели* – с вариантом использования *Формирование рекомендаций*.

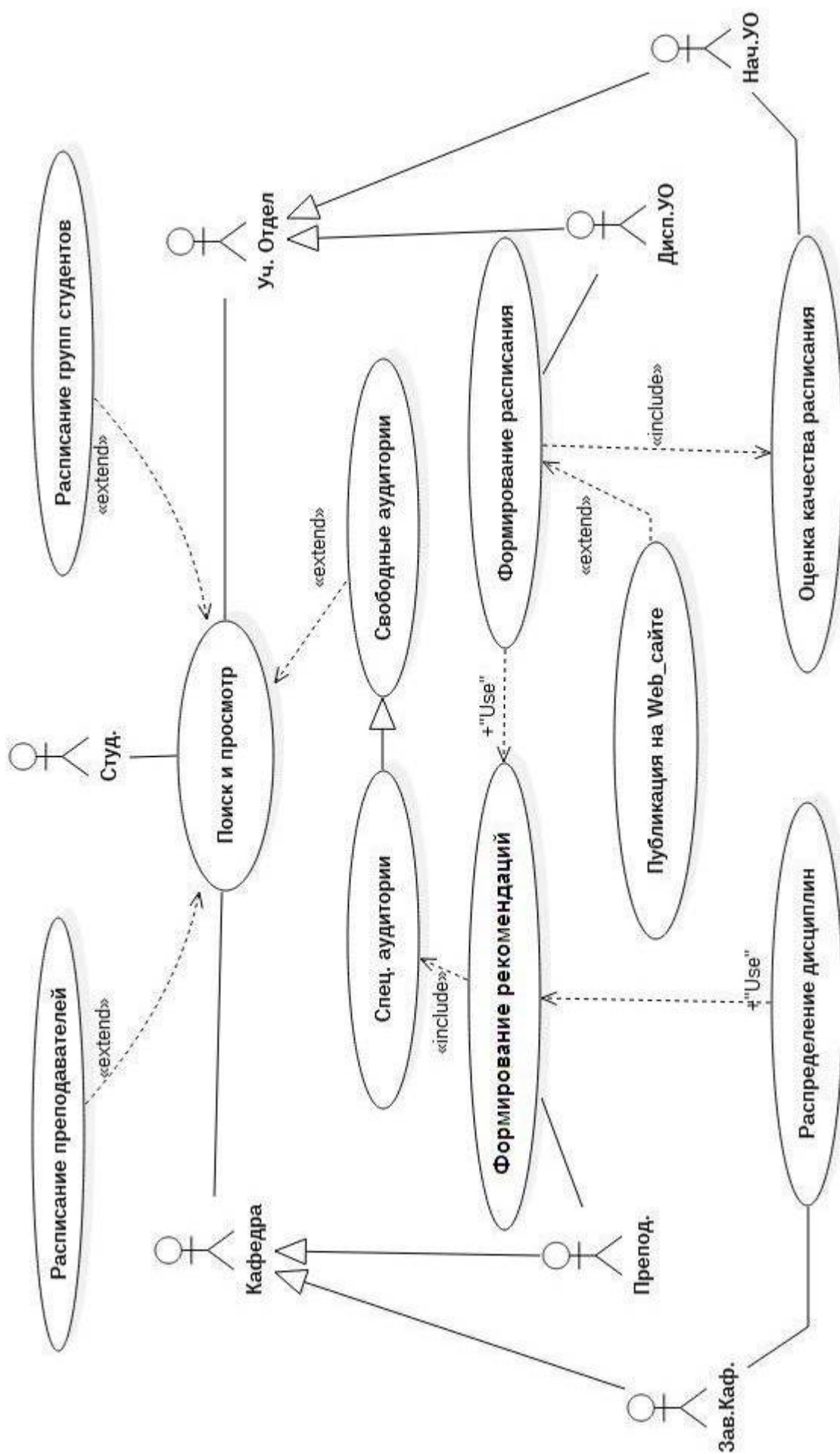


Рисунок 4.9 – UseCase-диаграмма системы «Расписание учебных занятий»

С ролью *Учебный отдел* связаны отношением обобщения роли *Начальник учебного отдела* и *Диспетчер учебного отдела*, которые унаследовали от своего родителя право Поиска и просмотра расписания, но дополнительно ассоциированы с другими вариантами использования.

*Начальник учебного отдела* ассоциирован с вариантом *Оценка качества расписания*, который в данном отношении является базовым вариантом.

*Диспетчер учебного отдела* ассоциирован с базовым вариантом использования *Формирование расписания*, в который включены подчиненные варианты использования *Свободные аудитории* и *Оценка качества расписания*.

Как видно из диаграммы, вариант использования *Оценка качества расписания* в одном отношении является базовым, а в другом – подчиненным, а вариант использования *Свободные аудитории* подчинен двум различным базовым вариантам использования.

Вариант использования *Формирование расписания* связан отношением зависимости (в качестве клиента зависимостей) с вариантом *Формирования рекомендаций* (который выступает в качестве источника зависимости), что соответствует реальной технологии составления расписания: изменения поведения вариантов использования – источников зависимости (например, перераспределение учебных дисциплин между преподавателями или изменение рекомендаций по времени проведения учебных занятий) естественным образом должны привести и к изменениям в поведении варианта – клиента этих зависимостей (то есть – к изменению результатов формирования расписания).

#### **4.6.3 Сценарии вариантов использования**

В ряде случаев, когда изобразительных средств UseCase-диаграммы оказывается недостаточно, она может быть дополнена текстовыми *сценариями вариантов использования*, уточняющими детали поведения системы.

Текстовый сценарий должен дополнять, а не заменять собой UseCase-диаграмму, он незаменим в качестве инструмента аналитика при выявлении требований к системе на стадии технического задания и может быть полезен на последующих стадиях проекта – например, при разработке диаграмм классов или прототипов пользовательских интерфейсов.

Как правило, единый сценарий разрабатывается для каждого базового и всех подчиненных ему вариантов использования и оформляется в со-

ответствии с шаблоном (таблица 4.1), состоящим из четырех разделов: «Главный раздел», разделы «Типичный ход событий», «Расширения» и «Примечания».

**Главный раздел** представляет собой заголовок варианта использования и содержит основные его атрибуты: наименование; тип (по отношению к другим связанным вариантам), перечень ассоциированных с ним *акторов*; назначение; краткое описание функционального поведения; ссылки на другие (связанные) варианты использования.

**Раздел «Типичный ход событий»** содержит описание хода событий, которое приводит к успешному выполнению варианта использования. Рекомендуется оформлять этот раздел сценария в форме бинарной таблицы, один из столбцов которой содержит описание действий *акторов* (и, при их наличии, описание исключительных ситуаций, возникающих в результате этих действий), а другой – описание откликов системы на эти действия. При этом все действия акторов и отклики системы последовательно нумеруются. Отдельно нумеруются также все *исключения*, представленные на диаграмме подчиненными вариантами, связанными с базовым отношением типа *extend*.

Таблица 4.1 – Пример оформления сценария варианта использования

Сценарий варианта использования «Формирование расписания»	
<b>Главный раздел</b>	
Наименование	Формирование расписания
Тип	Базовый
Акторы	Диспетчер учебного отдела
Цель	Формирование расписания учебных занятий
Краткое описание	Последовательно просматриваются рекомендации по составлению расписания, сформированные кафедрами, определяется время проведения учебных занятий, выбираются подходящие аудитории (из числа свободных) и производится количественная оценка текущей версии расписания по заданным критериям. Если оценка версии расписания оказывается удовлетворительной, расписание размещается на Web-сайте, если нет – производится генерация очередной версии. При многократной отрицательной оценке версий расписания производится корректировка исходных рекомендаций, сформированных кафедрами

Связанные варианты использования	Включаемые: <i>Свободные аудитории</i> <i>Оценка качества расписания</i> Расширяющие: <i>Формирование рекомендаций</i> <i>Публикация на Web-сайте</i>
Раздел « <b>Типичный ход событий</b> »	
Действия акторов	Отклик системы
<p>1 Активизирует рабочее окно клиентского приложения.</p> <p>3 Выбирает очередную кафедру из предложенного списка.</p> <p><u>Исключение 1:</u> <i>Список кафедр исчерпан.</i></p> <p>5 Выбирает очередную дисциплину из предложенного списка.</p> <p><u>Исключение 2:</u> <i>Список дисциплин исчерпан.</i></p> <p>9 Выбирает очередной вариант из предложенного перечня и резервирует время и место проведения занятий.</p> <p><u>Исключение 3:</u> <i>Получена отрицательная оценка качества очередной версии расписания.</i></p> <p><u>Исключение 4:</u> <i>Получена отрицательная оценка качества всех альтернативных версий расписания.</i></p>	<p>2 Визуализирует список кафедр.</p> <p>4 Визуализирует список дисциплин, обеспечиваемых кафедрой (по группам студентов).</p> <p>6 Визуализирует информацию о преподавателях и рекомендациях (ограничениях) о времени и месте (специализированные аудитории) проведения занятий по дисциплине.</p> <p>7 Определяет количество учебных занятий в неделю в соответствии с учебным планом.</p> <p>8 Определяет и визуализирует допустимые варианты расписания занятий по дисциплине (с учетом наличия свободных аудиторий и ограничений по времени работы преподавателя).</p> <p>10 Оценивает качество сформированной версии расписания по заданным критериям.</p> <p>11 Размещает сформированное расписание на Web-сайте.</p> <p>12 Завершает сеанс работы с приложением.</p>
Раздел « <b>Расширения</b> »	
Действия акторов	Отклик системы
<u>Исключение 1:</u> <i>Список кафедр исчерпан.</i>	
	13 Переход к п.12
<u>Исключение 2:</u> <i>Список дисциплин исчерпан.</i>	
	14 Переход к п.3
<u>Исключение 3:</u> <i>Получена отрицательная оценка качества очередной версии расписания.</i>	
	15 Переход к п.9

<i>Исключение 4: Получена отрицательная оценка качества всех альтернативных версий расписания.</i>	
	16 Выполняет процедуру корректировки рекомендаций и ограничений по составлению расписания, сформированных кафедрой. 17 Переход к п.6

**Раздел «Расширения»** содержит перечень и краткие описания всех альтернативных ситуаций. Оформляется в виде таблицы аналогично разделу «Типичный ход событий».

**Раздел «Примечания»** может содержать комментарии к сценарию или описываемому им фрагменту UseCase-диаграммы.

Таблица 4.1 представляет упрощенный пример сценария варианта использования *Формирование расписания* для UseCase-диаграммы, приведенной на рисунке 4.9.

### Контрольные вопросы и задания

1 Перечислите четыре основные задачи, которые решает разработчик программной системы, используя *UseCase*-диаграммы ?

2 Перечислите основные компоненты *UseCase*-диаграмм, укажите назначение каждого из них и приведите примеры их условных обозначений на диаграмме.

3 Перечислите основные и опциональные атрибуты варианта использования, указываемые в спецификации *UseCase*-диаграммы.

4 Определите понятие «интерфейс» как компонент *UseCase*-диаграммы.

5 Определите понятие «связь (отношение)» как компонент *UseCase*-диаграммы.

6 Создайте свой вариант *UseCase*-диаграммы, приведенной на рисунке 4.9. Дополните диаграмму элементом типа *интерфейс* для обозначения операции доступа к базе данных «Учебные планы», выполняемой в рамках варианта использования *Формирование расписания*.

7 Дополните *отношения ассоциации* акторов с вариантами использования этой же *UseCase*-диаграммы параметрами «имя связи» и «кратность связи», обоснуйте значения параметра кратности.

8 Оформите (по шаблону) сценарий варианта использования *Поиск и просмотр UseCase*-диаграммы, приведенной на рисунке 4.9.

9 Создайте UseCase-диаграмму системы доступа клиента банка к своим банковским счетам с целью снятия наличных денег или пополнения своих счетов через банкомат.

10 Создайте UseCase-диаграмму системы доступа клиента банка к своему банковскому счету с целью перевода безналичных денежных средств со своего счета на другие счета через банкомат.

## 4.7 UML-диаграмма пакетов

*Пакет (package)* – одно из базовых понятий языка UML, используемых при построении различных моделей для группировки семантически связанных друг с другом элементов. Диаграмма пакетов (*Package Diagram*) является универсальным инструментом и может использоваться для описания структурных моделей различных типов, в том числе и для представления результатов функциональной декомпозиции проектируемой системы на концептуальном уровне.

Пакет – это именованный элемент диаграммы, допускается отношение вложенности между пакетами, как это показано на рисунке 4.2.


Возможно использование специального пакета типа *подсистема (Subsystem)* – в его обозначении присутствует специальный символ  (рисунок 4.11).

Диаграмма пакетов представляется множеством элементов типа *пакет (package)*, связанных друг с другом пунктирными линиями со стрелками, каждая из которых обозначает некоторое *направленное отношение зависимости* между пакетами, как это показано на рисунке 4.10 [14], иллюстрирующем проект системы продаж интернет-магазина.

Отношение может быть содержательно поименовано и должно быть помечено одним из следующих *стереотипов* (зарезервированных ключевых слов, заключенных в кавычки):

- «*use*» – *отношение использования (Usage)* – одна из категорий *отношения зависимости (dependency)*<sup>5</sup> между пакетом – *клиентом* зависимости (*client*) и пакетом – *источником* или *поставщиком* зависимости (*sup-*

---

<sup>5</sup> Отношение зависимости используется в различных UML-диаграммах и более детально рассматривается в следующем разделе учебного пособия.

plier). Стрелка должна быть направлена от клиента к поставщику зависимости, что обозначает ситуацию, когда пакету – клиенту зависимости требуется для своего определения или реализации использование другого пакета – поставщика зависимости;

- **«merge»** – *отношение слияния*, когда к содержимому пакета – получателя добавляется содержимое целевого пакета, при этом стрелка на диаграмме должна быть направлена от пакета – получателя к целевому пакету<sup>6</sup>;

- **«import»** – импорт элементов, принадлежащих одному пакету (импортируемому) в состав элементов другого (импортирующего) пакета, при этом импортированные элементы станут общедоступными (*public*) за пределами импортирующего пакета (в дополнение к его собственным общедоступным элементам). Стрелка на диаграмме должна быть направлена от импортирующего пакета к импортированному;

- **«access»** – это разновидность отношения «import» (*private import*). Обозначает, что область видимости импортированных элементов будет ограничена импортирующим пакетом.

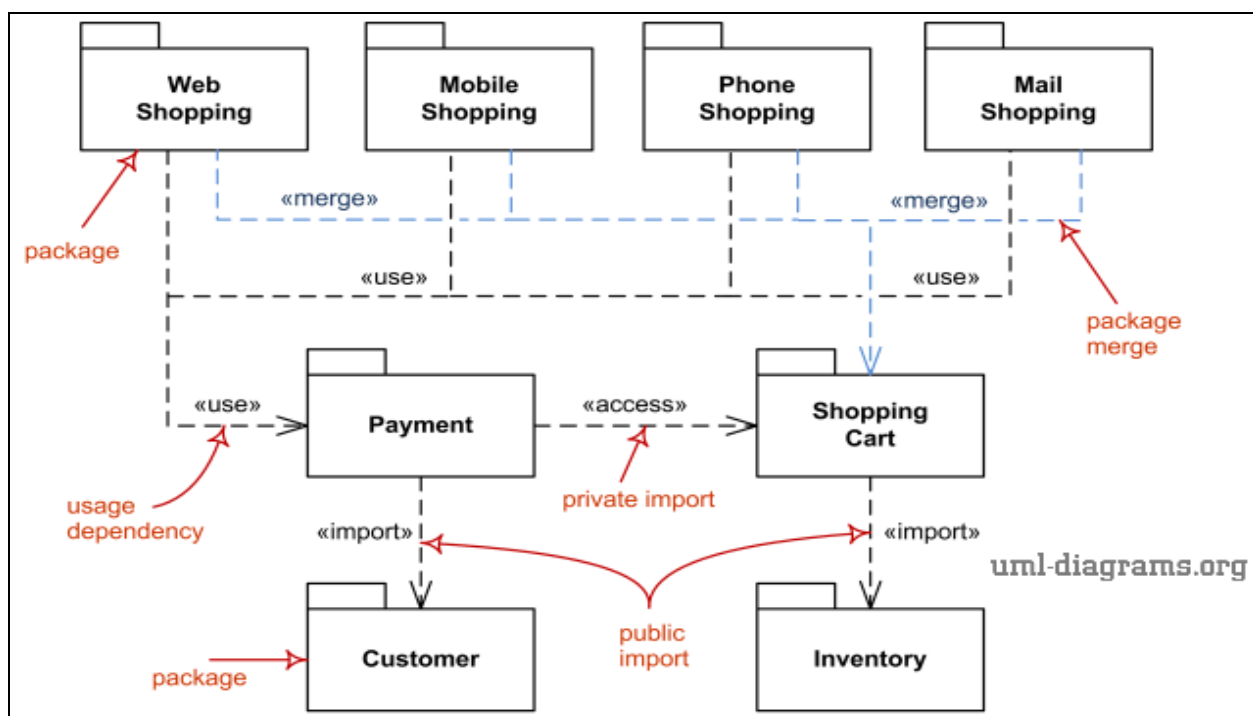


Рисунок 4.10 – Диаграмма пакетов проекта «Интернет-продажи»

<sup>6</sup> Отношение слияния в определенном смысле подобно отношению обобщения: исходный элемент концептуально добавляет характеристики целевого элемента к своим собственным характеристикам.



Четыре пакета *\*Shopping*, реализующие различные способы оформления заказов на покупку товаров, связаны отношением слияния «merge» с целевым пакетом *Shopping Cart*, представляющим товары, отобранные покупателями в их персональные «корзины». Это означает, что любой заказ, независимо от способа его оформления, содержит информацию о включенных в него товарах.

Эти же четыре пакета в качестве *клиентов зависимости* связаны отношением использования «use» с пакетом *Payment*, представляющим платежную систему магазина и выступающим в этом отношении в качестве *поставщика зависимости*. Это может, например, означать, что для реализации процессов исполнения заказов потребуется использование элементов платежной системы.

Отношениями типа «import» связаны на диаграмме пары пакетов *Shopping Cart* – – → *Inventory* и *Payment* – – → *Customer*, следовательно:

- информация о складских запасах и свойствах товаров (элементы пакета *Inventory*) доступна в пакете *Shopping Cart* наряду с его собственными элементами, представляющими товары, включенные в покупательские «корзины»;
- информация о покупателях (элементы пакета *Customer*) доступна в пакете *Payment* наряду с его собственными элементами, представляющими платежи;
- вся импортированная информация становится общедоступной, то есть видимой за пределами импортирующих пакетов – в пакетах *\*Shopping*.

Пакеты *Payment* и *Shopping Cart* связаны отношением «access», которое является частным случаем отношения «import» и ограничивает видимость элементов пакета *Shopping Cart* исключительно импортирующим пакетом *Payment*.

На рисунке 4.11 приведен пример диаграммы пакетов, разработанной для автоматизированной системы планирования учебной работы, фрагмент UseCase-диаграммы которой был рассмотрен ранее (рисунк 4.9).

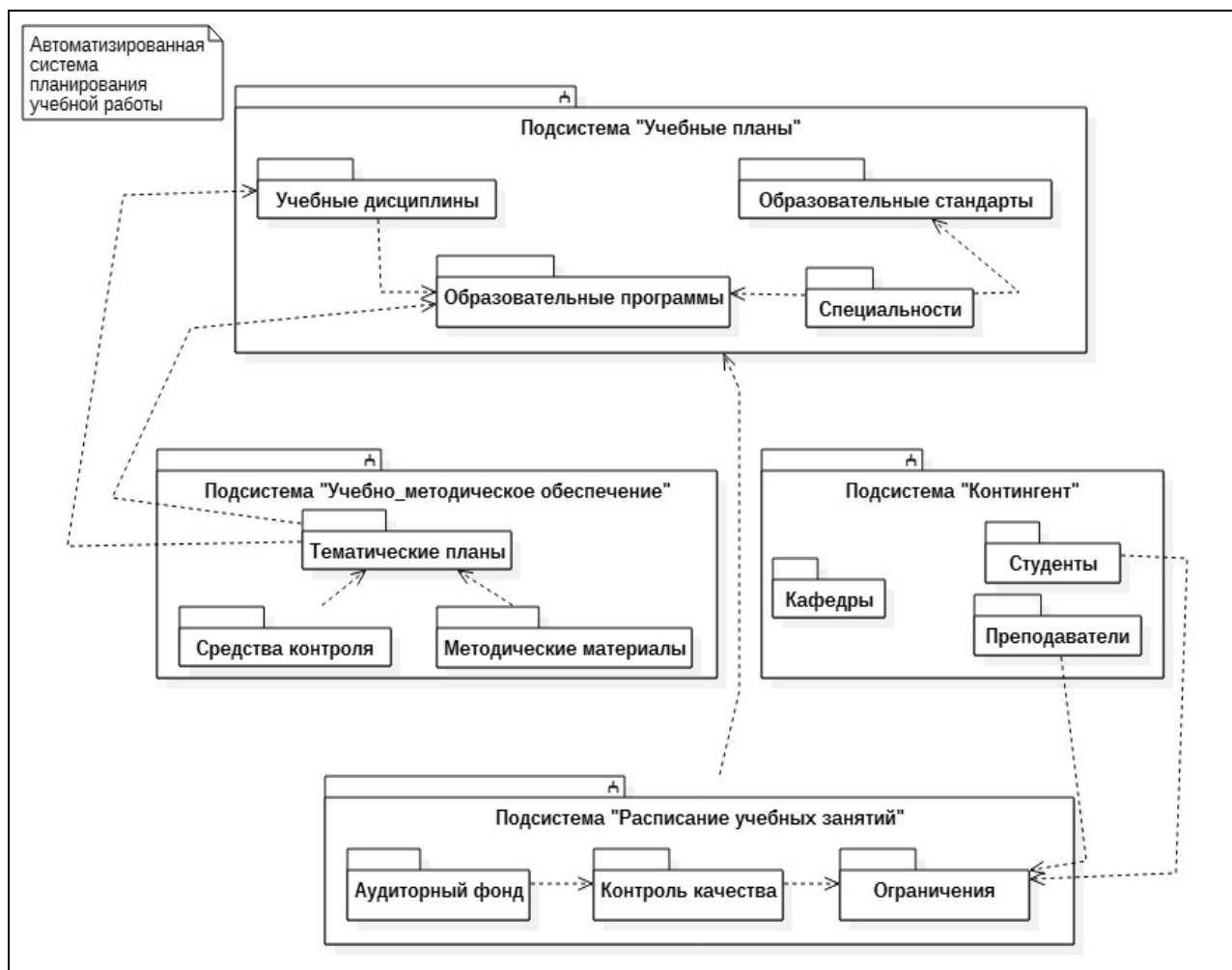


Рисунок 4.11 – Пример (незавершенной) диаграммы пакетов

### Контрольные вопросы и задания

1 Какие виды отношений между пакетами обозначаются стереотипами «*use*», «*merge*», «*import*» и «*access*»? Как обозначаются эти отношения на диаграммах пакетов?

2 Прокомментируйте применение стереотипов для обозначения отношений между пакетами на диаграмме, приведенной на рисунке 4.10. Оцените правильность использования отношений типов «*import*» и «*access*».

3 Создайте свою версию *Package*-диаграммы, приведенной на рисунке 4.11, и дополните обозначения отношений между пакетами соответствующими стереотипами.

## 4.8 UML-диаграмма классов

Рассмотренные выше диаграммы пакетов и вариантов использования разрабатываются на начальной стадии проекта и служат для графической визуализации концептуальной модели проектируемой системы, отражающей представления пользователей о ее основных функциях и распределении этих функций между подсистемами.

Следующий этап проектирования системы связан с проведением ее объектной декомпозиции, результаты которой представляют статические модели концептуального и логического уровней, описывающие структуру системы в терминах объектно-ориентированного подхода. Для графического представления таких моделей используется диаграмма классов, которая, в определенном смысле, является дальнейшей детализацией модели системы, представленной диаграммами пакетов и вариантов использования.

В зависимости от требуемой степени детализации описания структуры системы (или для удобства графической визуализации структурных взаимосвязей) могут разрабатываться диаграммы классов различных уровней, объединяемые в пакеты: диаграмма для системы в целом, диаграммы для отдельных ее подсистем, представленных на Package-диаграммах.

Диаграмма классов отображает статическую структурную модель: она описывает состав и взаимосвязи объектов системы, их свойства и поведение, но не содержит информации о временных аспектах их функционирования. Диаграмма представляется в виде графа, вершинами которого являются элементы типа *класс*, связанные структурными *отношениями* различных типов.

### 4.8.1 Классы

Класс (class) как элемент модели – это абстракция множества экземпляров некоторой сущности предметной области, обладающих одинаковой структурой, поведением и отношениями с экземплярами других сущностей. Класс порождает объекты, каждый из которых является абстракцией экземпляра соответствующей сущности предметной области.

Диаграмма классов концептуального уровня представлена так называемыми *концептуальными классами* или *бизнес-классами*, используемыми для моделирования сущностей (entity) предметной области. Такие

классы описывают свойства (атрибуты) сущностей и обычно не содержат методов.

Структурные модели логического уровня (модели проектирования или реализации) оперируют понятием *программного класса*, который содержит как атрибуты, так и методы. Программный класс является, по существу, описанием структурированного типа и используется для создания объектов программы.

На диаграмме класс изображается в виде прямоугольника, который дополнительно может быть разделен горизонтальными линиями на секции (рисунок 4.12), в которых могут указываться имя класса, его атрибуты, определяющие свойства объектов данного класса, и методы, определяющие их поведение.

Обязательным элементом обозначения класса является только его имя. По мере детализации диаграммы описания классов дополняются секциями атрибутов и операций, а иногда и дополнительной четвертой секцией, в которой приводится информация справочного характера или явно указываются исключительные ситуации. Даже если секция атрибутов или операций является пустой, в обозначении класса она выделяется горизонтальной линией.

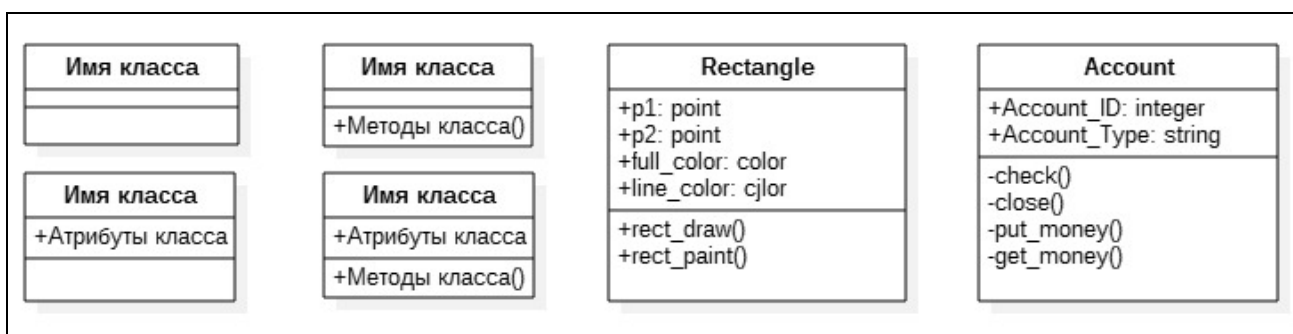


Рисунок 4.12 – Варианты изображения классов на диаграмме классов

Класс может не иметь экземпляров (объектов) – в этом случае он называется *абстрактным классом*, а для обозначения его имени используется наклонный шрифт (*курсив*).

**Имя класса** должно быть уникальным в пределах множества диаграмм классов, входящих в один пакет, оно записывается полужирным шрифтом и должно начинаться с заглавной буквы. Рекомендуется в качестве имен классов использовать имена существительные, записанные без пробелов.

В качестве префикса имени класса может использоваться имя пакета, к которому принадлежит этот класс – в этом случае имя пакета отделяется от имени класса специальным символом «::» – двойным двоеточием.

Секция имени класса дополнительно к собственно имени может содержать так называемый «стереотип класса» – одно из зарезервированных ключевых слов, определяющих некоторые специфические особенности данного класса. Стереотип записывается непосредственно над именем класса, заключается в парные кавычки и должен начинаться с заглавной буквы.

UML версии 2.5 определяет набор стандартных и нестандартных (дополнительных) стереотипов классов:

- стереотип «Auxiliary» определяет вспомогательный класс, который поддерживает другой, более фундаментальный класс, как правило, путем реализации вторичной бизнес-логики или потока управления; на диаграмме классов вспомогательный класс связывается с основным классом *отношением зависимости*;

- стереотип «Focus» определяет основную бизнес-логику или поток управления для одного или нескольких классов;

- стереотип «Implementation class» определяет реализацию класса на каком-то языке программирования;

- стереотипом «Type» обозначается домен объектов вместе с операциями, применимыми к объектам, без определения физической реализации этих объектов;

- стереотип «Utility» определяет класс, имеющий только статические атрибуты и операции; как правило, такой класс не имеет экземпляров;

- стереотипом «Entity» обозначается *пассивный класс*, не содержащий методов; объекты такого класса представляют свойства сущностей предметной области и не могут самостоятельно инициировать взаимодействия с другими объектами; в реализации такие классы могут соответствовать, например, таблицам реляционной базы данных.

**Атрибуты** (свойства) класса записываются во второй (сверху) секции в соответствии со следующими синтаксическими правилами.

- 1 Каждый атрибут записывается в отдельной строке.

2 Если строка атрибута подчеркнута, это означает, что соответствующий атрибут является *общим*, то есть значение этого атрибута будет одинаковым для всех создаваемых объектов данного класса.

3 Строка описания атрибута состоит из следующих последовательно записанных параметров атрибута (из которых обязательным является только один параметр – **имя** атрибута):

- *квантор видимости атрибута* – обозначается одним из четырех символов: «+» (*public*) обозначает *общедоступный* атрибут, то есть видимый из любого класса пакета, в котором определена диаграмма; «#» (*protected*) – *защищенный* атрибут, область видимости которого ограничена только данным классом и его подклассами; «-» (*private*) – *закрытый* атрибут, область видимости которого ограничена только данным классом; «~» (*package*) – *пакетный* атрибут, доступен только для классов того пакета, в котором определен класс – владелец атрибута; если квантор видимости атрибута опущен, то это означает только то, что видимость атрибута не задана;

- *имя атрибута* – текстовая строка, используемая в качестве идентификатора атрибута; должна быть уникальной в пределах класса;

- *кратность атрибута* характеризует общее количество экземпляров атрибута, входящих в состав класса. Кратность записывается в квадратных скобках в виде строки из цифр, разделенных запятыми или многоточием, как это показано в следующих примерах: [3] – кратность атрибута строго равна числу 3; [1, 3, 12, 22] – одно из указанных в скобках целых чисел; [1 .. 5, 7, 9 .. 12] – любое целое число из диапазонов от 1 до 5 и от 9 до 12 (включая границы диапазонов), а также число 7; [7 .. \*] – любое целое число, большее или равное 7; [\*] – любое целое неотрицательное число;

- *тип атрибута* указывается строкой текста, имеющей осмысленное значение в пределах пакета или модели, к которым относится рассматриваемый класс (например, String, Boolean или Color); перед описателем типа атрибута ставится знак «:» (двоеточие);

- *исходное значение* – определяет начальное значение атрибута, которое он получит (по умолчанию) в момент создания экземпляра класса. Если исходное значение атрибута опущено, то на момент создания экземпляра класса значение атрибута будет неопределенным; перед описателем исходного значения атрибута ставится знак равенства «=».

- {*Строка-свойство*}, заключенная в фигурные скобки, служит для указания значений атрибута, которые фиксируются для всех экземпляров данного класса: это значение атрибут получит в момент создания экземпляра класса, и оно не может быть переопределено в программе при работе с этим объектом.

### *Примеры описаний атрибутов класса*

Атрибуты класса «**Геометрические\_Объекты**»:

1) атрибут **цвет:Color=(255,0,0)**

- имя атрибута – «**цвет**»;
- тип атрибута – «**Color**»;
- начальное значение – **(255,0,0)**, что в RGB-модели соответствует чистому красному цвету;
- факт подчеркивания строки атрибута указывает на то, что все экземпляры этого класса будут иметь такое же значение данного атрибута (т.е. все создаваемые объекты будут красного цвета);

2) атрибут **форма:Многоугольник=прямоугольник**

- имя атрибута – «**форма**»;
- тип атрибута – «**Многоугольник**»;
- начальное значение – «**прямоугольник**», то есть каждый вновь создаваемый экземпляр этого класса будет иметь форму прямоугольника.

Атрибуты класса «**Сотрудники**»:

1) атрибут **имя\_сотрудника[1..2]:String="Новый\_Сотрудник"**

- имя атрибута – «**имя\_сотрудника**» кратностью 1 или 2;
- тип атрибута – «**String**»;
- начальное значение – «**Новый\_Сотрудник**», то есть при создании нового экземпляра класса (то есть при приеме на работу нового сотрудника) атрибут **имя\_сотрудника** этого класса получит начальное значение «**Новый\_Сотрудник**»;

2) атрибут **заработная\_плата:Money=\$500** (каждому вновь принятому на работу сотруднику назначается 500-долларовая зарплата, размер которой в будущем может быть изменен);

3) атрибут {**заработная\_плата:Money=\$500**} (строка-свойство, заключенная в фигурные скобки, может означать фиксированную заработную плату для всех экземпляров класса).

**Операция** (*operation*), или **метод** (*method*) класса представляет собой некоторый сервис, предоставляемый каждым экземпляром (объектом) этого класса по требованию его клиентов – других объектов, в том числе и объектов этого же класса. Совокупность всех операций класса характеризует функциональный аспект его поведения.

Спецификации операций класса записываются в третьей сверху секции графического символа класса по следующим синтаксическим правилам.

1 Спецификация каждой операции записывается в отдельной строке.

2 Если область действия операции распространяется на все объекты класса, спецификация операции подчеркивается; по умолчанию (если строка не подчеркнута) под областью действия операции понимается объект класса.

3 Спецификация операции, заданная у класса верхнего уровня (предка), наследуется всеми потомками данного класса. Если в некотором классе-потомке данная операция не выполняется (что указывает на абстрактный характер этой операции для данного класса), то для указания на абстрактный характер операции в *строке-свойстве* этой операции записывается **{abstract}**.

4 Строка операции имеет стандартный формат: **<квантор видимости> <имя операции> (список параметров): <выражение типа возвращаемого значения> {список свойств}**:

- **квантор видимости**, как и в случае с атрибутами класса, может принимать одно из четырех возможных значений и обозначается соответствующими символами: «+» – *public*, «#» – *protected*, «-» – *private*, «~» – *package*;

- **имя операции** – это идентифицирующая операцию строка текста, которая должна быть уникальной в пределах данного класса. Имя операции является единственным обязательным элементом обозначения операции. Имя операции должно начинаться со строчной (малой) буквы и не должно содержать пробелов. Пара скобок после имени операции является обязательным элементом спецификации (даже при пустом списке параметров);

- **список параметров** – это заключенный в скобки перечень разделенных запятой формальных параметров, каждый из которых может быть представлен в следующем формате: **<вид параметра>**



**<имя параметра> : <выражение типа> = <значение параметра по умолчанию>:**

- **вид параметра** (direction) – одно из ключевых слов: **in**, **out** или **inout** (по умолчанию – **in**);
- **имя параметра** – идентификатор соответствующего формального параметра, записывается с прописной буквы;
- **выражение типа** – спецификация типа значения параметра, зависящая от конкретного языка программирования;
- **значение по умолчанию** – выражение для значения формального параметра, синтаксис которого зависит от конкретного языка программирования;

- **выражение типа возвращаемого значения** указывает на тип данных значения, возвращаемого объектом после выполнения операции;

- **строка-свойство** заключается в фигурные скобки и служит для указания значений свойств, которые могут быть применены к данному элементу, например: **{query}** (запрос) обозначает пассивную операцию, которая не изменяет состояния системы; **{sequential}** обозначает *последовательную* операцию, при реализации которой необходимо обеспечить ее единственное выполнение в системе; **{concurrent}** обозначает *параллельную* операцию, допускающую одновременное выполнение с несколькими другими операциями; **{guarded}** обозначает *охраняемую* операцию, все обращения к которой должны быть строго упорядочены во времени с целью сохранения целостности объектов данного класса.

Ниже приведены примеры спецификаций операций классов.

**Пример 1.** При выполнении следующей операции на экране монитора будет изображена прямоугольная область синего цвета:

**+нарисовать(форма: Многоугольник = прямоугольник, цвет\_заливки: Color = (0, 0, 255)).**

Обозначена общедоступная (+) операция «нарисовать» с двумя входными (**in** – по умолчанию) параметрами: «форма» типа «Многоугольник» с начальным значением «прямоугольник», и «цвет заливки» типа **Color** с начальным значением **(0,0,255)**.

**Пример 2.** Результатом выполнения следующей операции является некоторое число, записанное в принятом денежном формате:

**-запросить\_счет\_клиента(номер\_счета:integer):Currency**

Обозначена закрытая (-) операция по установлению наличия средств на текущем счете клиента банка. При этом аргументом данной операции является номер счета клиента, который записывается в виде целого числа.

#### **4.8.2 Отношения между классами**

*Отношения* между классами отражают взаимосвязи между их экземплярами (объектами) и обозначаются на диаграмме линиями, соединяющими классы. Всего определено 3 базовых типа отношений между классами, для каждого из которых предусмотрено соответствующее графическое обозначение:

- 1) отношение *ассоциации* (association relationship):
  - *агрегация* (aggregation);
  - *композиция* (composition);
- 2) отношение *обобщения* (generalization relationship);
- 3) отношение *зависимости* (dependency relationship).

**Отношение ассоциации** – это наиболее общий случай взаимосвязи между классами. В определенном смысле все остальные типы отношений являются частными случаями отношения ассоциации, выделенными в отдельные категории ввиду их важности с точки зрения описания моделируемой системы. Классы, связанные отношением ассоциации, называются *ролями ассоциации*.

Важнейшая характеристика ассоциации – её *арность*, определяемая количеством классов, участвующих в ассоциации. Простейшая ассоциация – бинарная, в которой участвуют ровно два класса. Если в ассоциации участвуют три класса, она называется *тернарной*, если больше – *тетрарной*, *пентарной* и т.д., в общем случае – *n*-арной ассоциацией.

Фактически в ассоциации участвуют объекты – *экземпляры* ассоциированных классов, поэтому допустимой является и *унарная ассоциация*, описывающая взаимосвязь между различными объектами одного и того же класса. Каждый экземпляр *n*-арной ассоциации представляет собой *n*-арный кортеж значений объектов из соответствующих классов, при этом

один и тот же объект (экземпляр класса) может участвовать в нескольких экземплярах одной и той же ассоциации.

Отношение ассоциации обозначается на диаграмме сплошной линией (со стрелками или без них), соединяющей ассоциированные классы. Линия ассоциации может быть поименована и помечена специальными символами, в её разрыве может изображаться ромб, концы ассоциации могут быть помечены именами ролей и/или кратностью ассоциированных классов. Если имя ассоциации задано, оно записывается с заглавной буквы. Ромб в разрыве линии бинарной ассоциации, как правило, не изображается.

Концы ассоциации помечены параметрами *кратности*, указывающими на количество объектов (экземпляров классов), участвующих в одном экземпляре ассоциации, и могут быть помечены символами «агрегации» или «композиции».

***Агрегация и композиция*** – частные случаи отношения ассоциации.

*Агрегация* между классами представляет системные взаимосвязи типа «целое – часть» и описывает декомпозицию сложной системы на *составные части*. Такая декомпозиция системы представляет собой иерархию ее составных компонентов, однако эта иерархия принципиально отличается от иерархии, порождаемой отношением обобщения.

Отличие заключается в том, что *части системы* являются вполне самостоятельными сущностями и *не обязаны наследовать ее свойства и поведение*. Более того, части целого обладают своими собственными атрибутами и операциями, которые существенно отличаются от атрибутов и операций целого. Примером *отношения агрегации* может служить иерархия:

[*Автомобиль: Двигатель, Шасси, Кабина, Кузов*]

Сравните это с отношением обобщения:

[*Автомобиль: Легковой, Грузовой, Автобус*]

Пример графического обозначения *отношения агрегации* показан на рисунке 4.13: это отношение изображается на диаграмме сплошной линией, на одном из концов которой (соединенным с классом-агрегатом, представляющим «целое») помещается *не закрашенный внутри ромб*.

*Композиция* является частным случаем агрегации и используется для выделения специальной формы взаимосвязи между «целым» и его «частями», при которой части не могут существовать в отрыве от целого и уничтожаются при его уничтожении.

Пример отношения композиции показан на рисунке 4.14. Такое отношение установлено между классами *Окно\_интерфейса\_программы* (целое) и классами, описывающими составляющие (части) окна: *Строка\_заголовка*, *Полоса\_прокрутки*, *Главное\_меню*, *Рабочая\_область*.

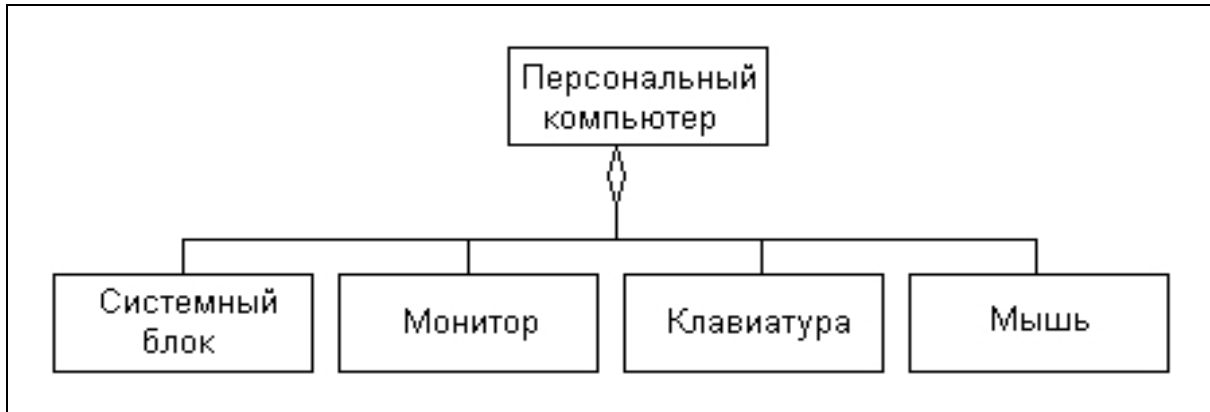


Рисунок 4.13 – Отношение агрегации

Отношение композиции изображается на диаграмме сплошной линией, на одном из концов которой (соединенным с классом-композицией, представляющим «целое») помещается закрашенный внутри ромб.

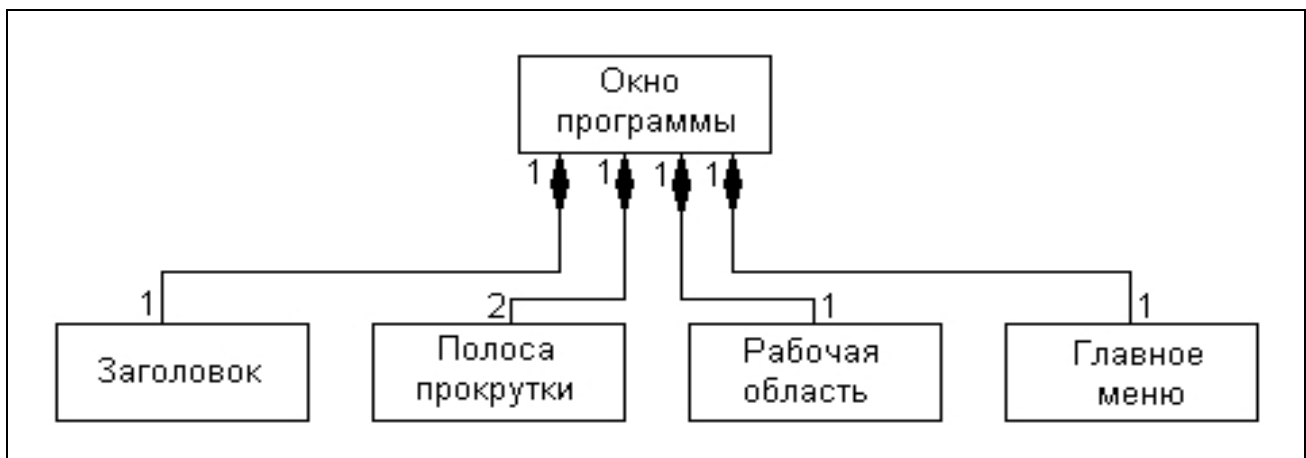


Рисунок 4.14 – Отношение композиции

Рисунок 4.14 иллюстрирует также возможность указания на диаграмме дополнительных параметров отношения композиции, в частности, параметра кратности связей. Например, кратность «1» связи с классом «*Рабочая\_область*» указывает на тот факт, что некоторое приложение будет обрабатывать (и отображать в рабочей области окна) одновременно только один документ.

**Отношение обобщения** устанавливается между *общим элементом* (предком) и *частным элементом* (потомком). Применительно к диаграм-

ме классов такое отношение описывает иерархию наследования классов, при этом предполагается, что класс-потомок (подкласс) обладает всеми свойствами и поведением класса-предка (суперкласса), но также может иметь и свои собственные свойства и поведение, которые отсутствуют у класса-предка.

Класс-предок может иметь несколько связанных с ним классов-потомков: в этом случае множество классов, связанных отношением обобщения, образуют древовидную структуру, корнем которой является класс-предок. Допускается также и множественное наследование, когда класс может быть потомком более, чем одного предка.

На диаграммах отношение обобщения обозначается сплошной линией с треугольной незакрашенной внутри стрелкой, направленной от класса-потомка к классу-предку. Рядом со стрелкой обобщения на диаграмме может быть помещен текст, указывающий на некоторые дополнительные свойства этого отношения, касающиеся всех подклассов данного отношения. При этом *если текст записан в фигурных скобках, его следует рассматривать как ограничение* (рисунок 4.15).

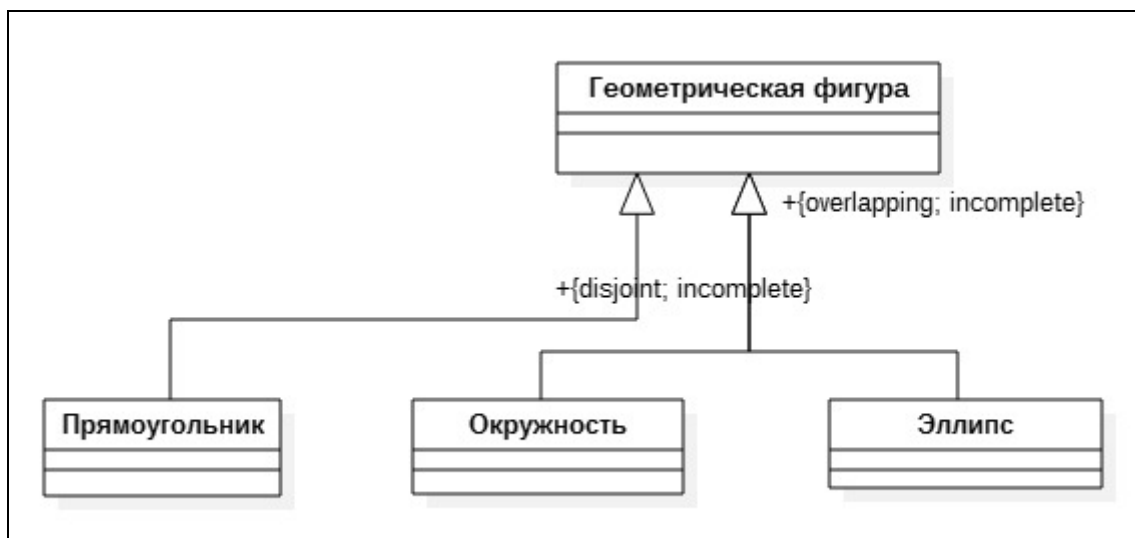


Рисунок 4.15 – Отношение обобщения с использованием строк-ограничений

В качестве таких ограничений язык UML допускает использовать следующие ключевые слова:

- **{complete}** означает, что в данном отношении обобщения специфицированы все без исключения классы-потомки, и других классов-потомков у данного класса-предка быть не может; например, класс **Кли-**

ент\_банка является предком для трех классов: **Физическое\_лицо**, **Юридическое\_лицо** и **Индивидуальный\_предприниматель**, и других классов-потомков он не имеет;

- **{incomplete}** означает случай, противоположный первому: на диаграмме указаны не все классы-потомки, и впоследствии можно пополнить их перечень, не изменяя уже построенную диаграмму;
- **{disjoint}** означает, что классы-потомки не могут содержать объектов, одновременно являющихся экземплярами двух или более классов;
- **{overlapping}** означает, что отдельные экземпляры классов-потомков могут принадлежать одновременно нескольким классам: например, класс **Многоугольник** является классом-предком для классов **Прямоугольник** и **Ромб**, и при этом существует отдельный класс **Квадрат**, экземпляры которого одновременно являются объектами первых двух классов.

На рисунке 4.15 ограничение **{incomplete}** указывает на тот факт, что кроме прямоугольника, окружности и эллипса существуют и другие типы геометрических фигур, ограничение **{disjoint}** – на то, что прямоугольники не могут одновременно являться окружностями и эллипсами (и наоборот), а ограничение **{overlapping}** – на то, что некоторые экземпляры класса **Эллипс** могут одновременно принадлежать и классу **Окружность**.

**Отношение зависимости** (рассмотренное ранее применительно к диаграммам пакетов и вариантов использования) указывает на некоторое семантическое отношение между двумя классами, которое не является отношением ассоциации или обобщения, но при этом семантика одного класса («независимого») дополняется семантикой другого, зависимого от него класса. Независимый класс называется *источником* (или *поставщиком*) *зависимости* (*supplier*), а зависимый от него класс – *клиентом зависимости* (*client*).

Отношение зависимости изображается на диаграмме пунктирной линией со стрелкой на одном из ее концов, направленной от класса-клиента к классу-источнику зависимости. Линия зависимости может помечаться стандартным ключевым словом (стереотипом), заключенным в кавычки, и/или индивидуальным содержательным наименованием.

На рисунке 4.16 приведена классификация [14] отношений зависимости, используемых в UML-диаграммах различных типов, с указанием для каждого их них соответствующего графического символа и стереотипа.

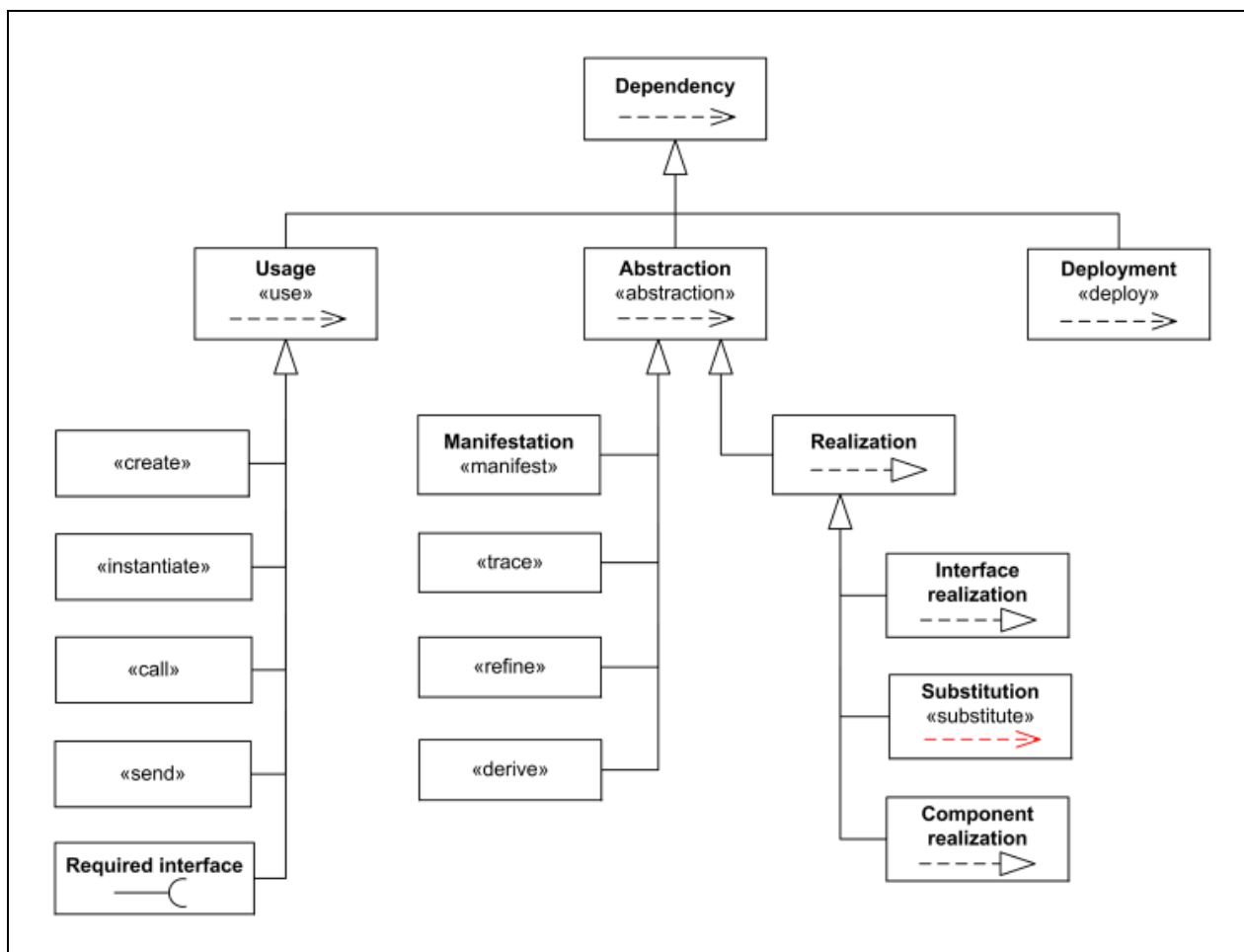


Рисунок 4.16 – Классификация отношений зависимости

Отношения группы **Abstraction** используют для обозначения зависимостей между элементами модели, представляющими одну и ту же сущность на разных уровнях абстракции или с разных точек зрения:

- стереотип «**manifest**» (проявление) используют в диаграммах развертывания (*deployment diagrams*) для обозначения физической интерпретации (rendering) элемента модели соответствующим артефактом;
- стереотип «**trace**» определяет «межмодельную зависимость» и используется, в основном, для отслеживания изменений в элементах, представляющих одну и ту же сущность в моделях разных уровней;
- стереотип «**refine**» (усовершенствование, детализация) определяет уточняющую зависимость и используется для определения взаимосвязи между элементами модели на разных семантических уровнях, таких как анализ, проектирование и реализация;
- стереотип «**derive**» обозначает возможность вычисления значения клиента зависимости по значению поставщика зависимости: например, от-

ношение вида [age] – – → [birth date], помеченное стереотипом «derive», означает, что возраст может быть вычислен по заданной дате рождения.

Отношения подгруппы **Realization** обозначают специализированные абстрактные зависимости между двумя элементами, один из которых (*поставщик зависимости*) представляет собой спецификацию, а другой (*клиент зависимости*) – реализацию этой спецификации. Реализация может использоваться для поэтапного уточнения, оптимизации или трансформации моделей.

Отношение **Deployment** (развертывание) отражает взаимосвязь между концептуальным или физическим *целевым элементом* моделируемой системы (*Deployment Target*) и присвоенными ему *артефактами* (*Deployment Artifacts*) – информационными ресурсами, которые используются этим целевым элементом или создаются при его разработке. Таковыми артефактами могут быть файлы UML-моделей, исходные коды программных компонентов, исполнимые файлы программ, базы данных и отдельные их компоненты, текстовые документы, почтовые сообщения и иные информационные ресурсы.

Наиболее актуальным для диаграмм классов являются отношения зависимости группы **Usage** (*использование*), которые применяются в случаях, когда одному элементу диаграммы (клиенту зависимости) требуется для своего определения или реализации использование другого элемента (источника зависимости). Если способ такого использования не уточняется, на диаграмме указывается стереотип «use», если требуется детализация – используются стереотипы «create», «instantiate», «call», «send», или «required interface».

Стереотипы «create» (создать) и «instantiate» (создать экземпляр) означают одно и то же: класс-клиент зависимости создает экземпляры класса-поставщика зависимости.

Стереотип «call» применяется для обозначения отношения зависимости между операциями, одна из которых (источник зависимости) запрашивает (invoke) вызов другой (целевой) операции (клиент зависимости).

Стереотип «send» обозначает отношение зависимости между операцией и сигналом и указывает на то, что операция-источник отправляет целевой сигнал.

Стереотип «required interface» определяет зависимость между классом и соответствующим интерфейсом, требуемым для выполнения классом своих функций (рисунок 4.17).



### 4.8.3 Интерфейсы

Интерфейс является элементом UseCase-диаграммы, однако, при построении диаграммы классов отдельные интерфейсы могут уточняться, и в этом случае для их изображения используется специальный графический символ – прямоугольник класса со стереотипом «interface» (рисунок 4.17), в котором секция атрибутов может отсутствовать. В левой части рисунка приведено обозначение интерфейса – источника сигнала, а в правой – обозначение интерфейса реализации.

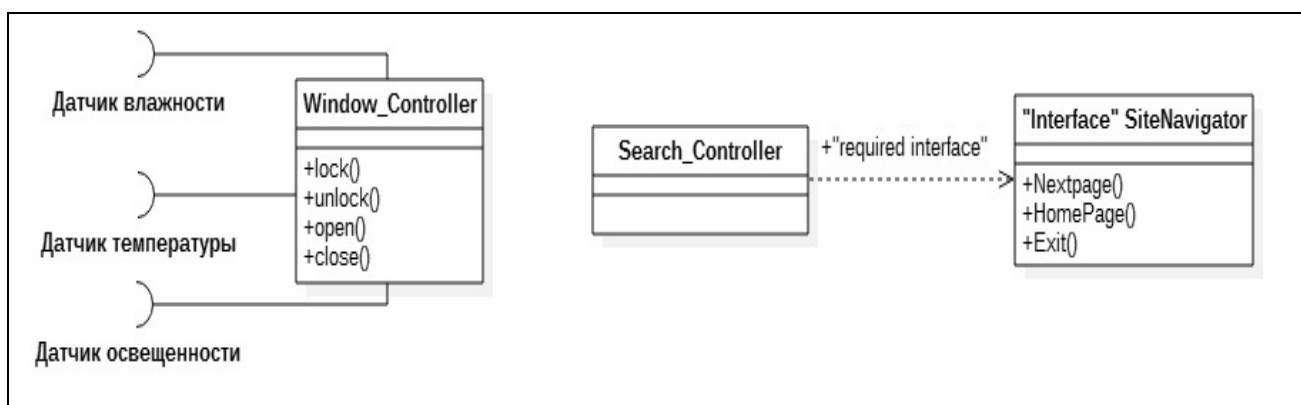


Рисунок 4.17 – Обозначение интерфейсов на диаграмме классов

На рисунке 4.18 [14] приведен упрощенный вариант диаграммы классов, разработанной в рамках проекта автоматизированной библиотечной системы. Рисунок иллюстрирует применение элементов графической нотации, используемых для обозначений классов и отношений между ними.

Диаграмма классов завершает процесс объектной декомпозиции системы на логическом уровне и отображает взаимосвязи структурного (статического) характера между ее компонентами, не зависящие от времени или реакции системы на внешние события.

Если проектируемая система по своей природе статична или ее поведение тривиально, проект продолжается разработкой статических моделей более низких уровней, для представления которых могут использоваться UML- диаграммы компонентов (*Component Diagram*) и диаграммы развертывания (*Deployment Diagram*).

Для динамических систем, поведение которых является существенным аспектом их функционирования, на следующей стадии проекта разрабатываются UML-диаграммы, представляющие динамические модели системы различных уровней.

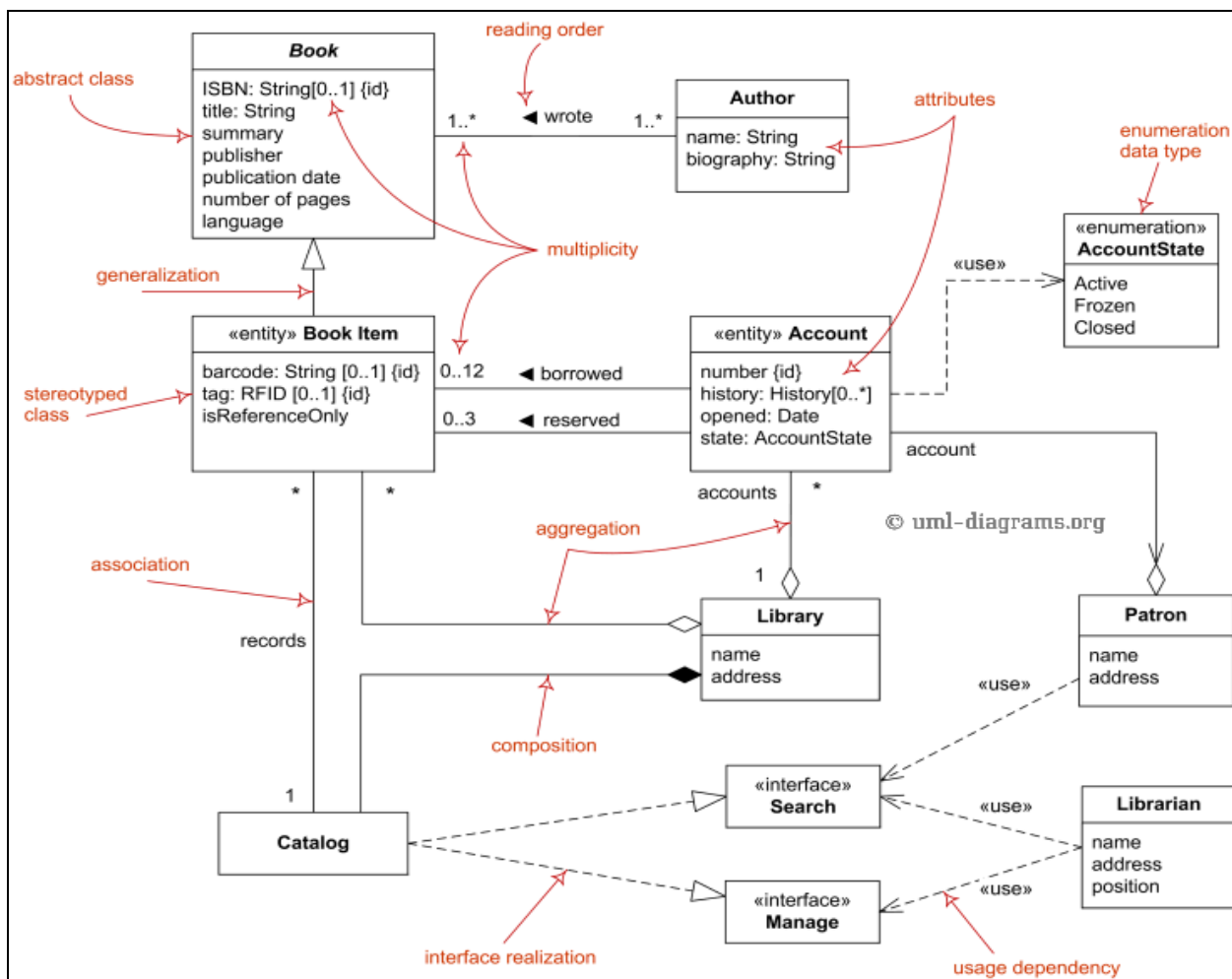


Рисунок 4.18 – Диаграмма классов библиотечной системы

### Контрольные вопросы и задания

- 1 Определите понятие *Класса* в языке UML. Какие из секций графического символа класса на UML-диаграммах являются обязательными?
- 2 Понятия и форматы спецификации *атрибута* и *метода класса*. Как обозначается область видимости атрибутов и выходные параметры методов?
- 3 Отношение *ассоциации* между классами: понятие, характеристики, обозначение на UML-диаграммах. Приведите примеры обозначений кратности концов ассоциации.
- 4 Отношения *агрегации* и *композиции* между классами: понятие, характеристики, обозначение на UML-диаграммах.
- 5 Отношение *обобщения* между классами: понятие, характеристики, обозначение на UML-диаграммах.
- 6 Отношение *зависимости* между классами: понятие, характеристики, обозначение на UML-диаграммах. Приведите примеры *отношений использования*.

7 Проведите анализ диаграммы классов, приведенной на рисунке 4.18:

– какие элементы определяют состав читателей и состав библиотечного фонда каждой из библиотек?

– что означает отношение агрегации между классами «**Account**» и «**Library**»?

– может ли (в этой модели) один читатель пользоваться фондами нескольких библиотек?

– какие ограничения на количество книг, выдаваемых читателю, накладывает модель?

– доработайте диаграмму классов так, чтобы она обеспечивала возможность поиска книг по тематическому каталогу.

## 4.9 UML-диаграмма состояний

Диаграмма состояний (*State Machine* или *State Chart Diagram*) представляет динамическую модель логического уровня и чаще всего используется для описания поведения экземпляров классов (объектов). Возможно также использование таких диаграмм для спецификаций функциональности вариантов использования и для представления алгоритмов реализации методов классов.

Диаграмма состояний описывает поведение элемента проектируемой системы как последовательность *переходов* (*transition*) этого элемента из одних *состояний* (*state*) в другие состояния под воздействием различных *событий* (*event*), как внутренних, так и внешних. С каждым состоянием связана определенная *деятельность* (*activity*), завершение которой является внутренним событием, на которое может реагировать система.

Несмотря на то, что диаграмма состояний представляет динамическую модель, время (в качестве параметра модели) в этой диаграмме явно не учитывается: длительность нахождения объекта в любом состоянии считается несущественной и при этом предполагается, что переход объекта из состояния в состояние происходит мгновенно<sup>7</sup>.

При построении диаграммы состояний время учитывается лишь косвенно: во-первых, предполагается, что последовательность изменения со-

---

<sup>7</sup> Для представления временных аспектов поведения проектируемой системы в языке UML используется другая диаграмма – диаграмма *последовательности* (*sequence diagram*), которая в данном учебном пособии не рассматривается.

стояний упорядочена во времени, и каждое последующее состояние всегда наступает позже предшествующего ему состояния; во-вторых, время нахождения системы в некотором состоянии считается существенно большим по сравнению с временем ее перехода в другое состояние; в-третьих, если переходы из нескольких *параллельных* состояний *соединяются* в один переход, то считается, что момент срабатывания этого перехода совпадает с моментом срабатывания последнего из соединяемых переходов.

Диаграмма состояний – это ориентированный граф (рисунок 4.19, [14]), в узлах которого изображаются графические символы, представляющие *состояния*, а дуги графа обозначают *переходы* между состояниями. Переход обозначается на диаграмме сплошной линией со стрелкой, соединяющей *исходное состояние* с *целевым состоянием*.

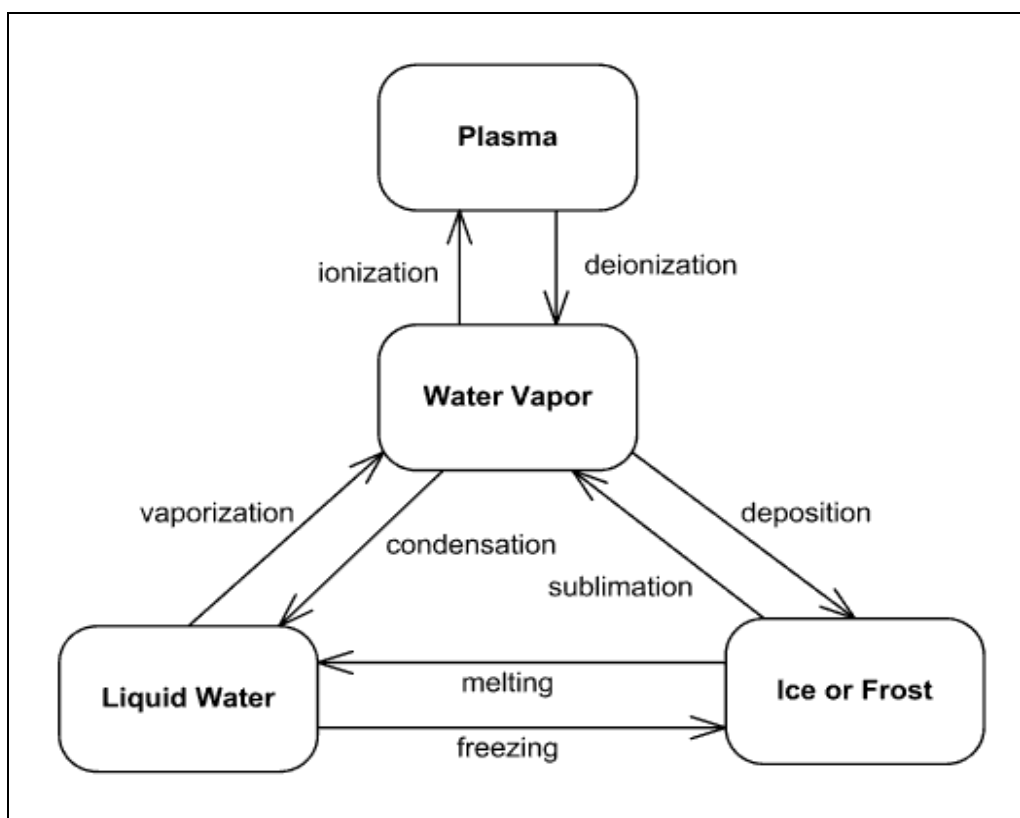


Рисунок 4.19 – Диаграмма смены агрегатных состояний воды

### 4.9.1 Состояния

Для обозначения состояния используется графический символ – прямоугольник со скругленными углами, разделенный горизонтальными линиями на несколько (до четырех) секций: в верхней секции записывается имя состояния, а в остальных – описание действий, выполняемых объектом, оказавшимся в этом состоянии, или внутренних переходов для

составных состояний. На начальных этапах разработки модели допускается и более простое обозначение состояния, в котором присутствует только одна секция – имя состояния.

Имя состояния – это короткий текст, отражающий основное содержание деятельности объекта, находящегося в этом состоянии. Согласно правилам UML, «**Имя\_состояния**» *всегда записывается с заглавной буквы*.

Если визуализация некоторого состояния затрудняет общее восприятие диаграммы из-за сложного поведения объекта (или если детали поведения объекта еще не определены разработчиком модели на данной стадии проекта), допускается показывать это состояние как *скрытое составное состояние*, не раскрывающее деталей его деятельности. В этом случае в правом нижнем углу графического символа отображается специальная пиктограмма (☐—☐), указывающая на то, что в дальнейшем для этого состояния будет изображена отдельная диаграмма (рисунок 4.20).



Рисунок 4.20 – Пример обозначения состояния

Нижняя секция символа состояния содержит список внутренних действий, каждое из которых записывается отдельной строкой вида:

#### ***метка-действия / выражение-действия***

**Метка действия** идентифицирует событие, которое запускает выполнение деятельности, определенной **выражением действия**.

В языке UML предусмотрены несколько фиксированных имен меток действия:

- **entry** – эта метка указывает на действие, которое будет выполняться в момент входа в данное состояние (входное действие);
- **exit** – эта метка указывает на действие, которое выполняется в момент выхода из данного состояния (выходное действие);
- **do** – эта метка специфицирует деятельность, которая выполняется в течение всего времени, пока объект находится в данном состоянии, или до тех пор, пока не закончится деятельность, специфицированная следующим за этой меткой выражением действия.

Существуют два особых состояния, называемых *псевдосостояниями*: это *начальное* (*initial*) и *конечное* (*final*) состояния, для обозначения которых используются специальные символы (рисунок 4.21).

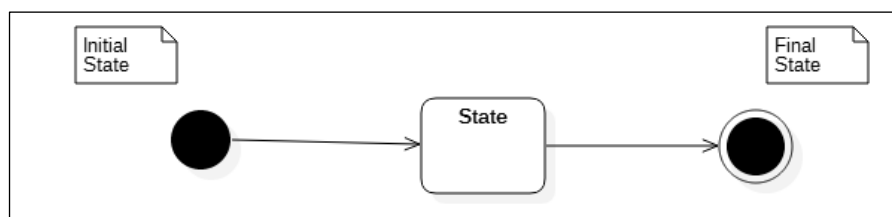


Рисунок 4.21 – Обозначения начального и конечного состояний

Начальное и конечное состояния не содержат никакой деятельности и используются исключительно для указания на диаграмме областей, в которых, соответственно, начинается и заканчивается жизненный цикл моделируемого объекта. Начальное состояние является самым первым из всех исходных состояний, а конечное – самым последним из всех целевых.

#### 4.9.2 Простые переходы

В результате срабатывания *простого перехода* (*simple transition*) моделируемый объект переходит из *исходного* состояния в *целевое*. Срабатывание перехода обусловлено *событиями* (*event*) двух категорий: событие первого типа связано с окончанием выполнения деятельности (*do activity*), заданной для исходного состояния; событие второго типа – это *внешнее событие*, происходящее асинхронно с внутренней деятельностью данного состояния, например, прием объектом сообщения от других объектов системы или поступление сигнала из внешней среды.

Если переход вызван таким внешним событием, он называется *триггерным переходом*. Срабатывание триггерного перехода может зависеть не только от наступления связанного с ним внешнего события, но и от выполнения так называемого *сторожевого условия* (*guard condition*), в качестве которого может быть использовано любое корректное логическое выражение. Объект перейдет в целевое состояние только в том случае, если произошло указанное внешнее событие и (AND) при этом сторожевое условие приняло значение «истина».

Событие может инициировать множество альтернативных триггерных переходов из одного исходного состояния в несколько целевых состояний. При этом с каждым из альтернативных триггерных переходов

связано свое сторожевое условие, но только одно из них может принимать истинное значение.

Для графического изображения альтернативных переходов может использоваться специальный символ ***choice*** (выбор) в виде ромба, как это показано на рисунке 4.22.

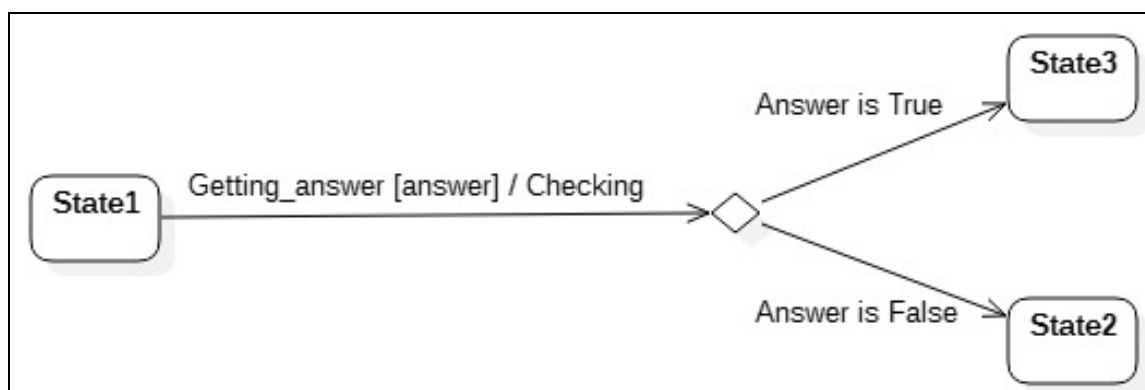


Рисунок 4.22 – Пример обозначения альтернативных переходов

Триггерный переход должен быть помечен на диаграмме строкой текста, в которой описываются: *имя события*, инициирующего данный переход; *сторожевое условие*, которое будет проверяться перед срабатыванием перехода; *действие*, которое будет выполняться в момент срабатывания перехода.

Строка описания триггерного перехода имеет следующий формат, в котором единственным обязательным элементом является *имя события*:  
*имя\_события* [(список параметров)] [[сторожевое условие]] [/действие].

*Сторожевое условие*, при его наличии, всегда записывается в прямых скобках после описания события; предполагается, что определение истинности условия происходит в момент наступления соответствующего события.

*Выражение действия (action expression)*, при его наличии:

- записывается последним в строке и отделяется от предшествующих элементов символом «/»;
- выполняется только в случае, если переход срабатывает, и до начала любых действий в целевом состоянии;
- представляет собой простую операцию и не может быть прервано никаким другим действием до тех пор, пока не будет выполнено;
- может оказывать влияние как на сам объект, так и на его окружение;
- может содержать список отдельных действий, разделенных символом «;» и выполняемых строго в порядке их записи;

- чаще всего записывается на языке программирования, который предполагается использовать для реализации модели.

### **4.9.3 Составные состояния**

Составное состояние (*composite state*) состоит из других, вложенных в него состояний, называемых *подсостояниями* (*substates*) по отношению к составному состоянию. На диаграмме вложенные состояния изображаются внутри символа составного состояния.

Составное состояние может содержать несколько параллельных подсостояний, каждое из которых может быть представлено одним или более последовательными подсостояниями. При этом подсостояние также может быть составным, то есть может содержать другие вложенные в него параллельные и/или последовательные подсостояния, без каких-либо формальных ограничений на количество уровней вложенности.

#### ***Последовательные подсостояния***

Поведение объекта, оказавшегося в составном состоянии после срабатывания соответствующего перехода, представляется последовательной сменой его подсостояний, начиная от начального и заканчивая конечным подсостояниями. В каждом составном состоянии может быть только одно начальное и только одно конечное *последовательные* состояния.

На рисунке 4.24 показано составное состояние, содержащее параллельные и последовательные подсостояния. Каждое из вложенных параллельных подсостояний может состоять из нескольких последовательных подсостояний.

#### ***Параллельные подсостояния***

Если в поведении объекта объективно присутствуют признаки параллелизма, следует явно специфицировать все параллельные состояния и "вложить" их в одно или несколько составных состояний. Такие составные состояния будут последовательно связаны с другими состояниями, но внутри их будут преднамеренно нарушаться условия последовательности переходов.

На диаграмме для каждого параллельного подсостояния выделяется горизонтальная область (рисунок 4.23), визуальное отделяемая от других областей пунктирной линией. Переход моделируемого объекта в составное состояние означает его одновременный переход в *начальное подсостояние* каждого из вложенных параллельных подсостояний. Деятель-



ность всего составного состояния будет считаться завершенной только после завершения деятельности всех параллельных подсостояний (то есть только когда моделируемый объект перейдет в *конечное подсостояние* каждого из вложенных параллельных состояний).

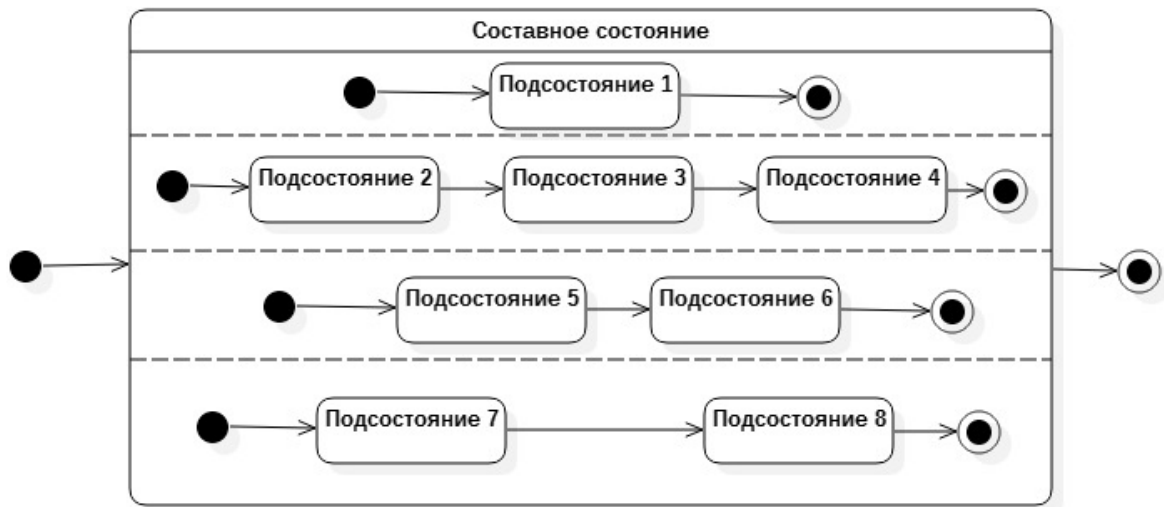


Рисунок 4.23 – Составное состояние

### ***Исторические составные состояния (history state)***

Исторические составные состояния используются для моделирования ситуаций, когда объект многократно оказывается в одном и том же состоянии, и при этом при каждом очередном переходе объекта в это состояние требуется учитывать ранее выполненную часть деятельности.

При первом переходе в составное состояние история ранее выполненных последовательных состояний пуста, и переход осуществляется в начальное подсостояние.

Если срабатывает триггерный переход из одного из промежуточных последовательных подсостояний, историческое состояние запоминает его в своей истории и при следующем переходе в это составное состояние активизирует запомненное промежуточное подсостояние, минуя начальное и все выполненные ранее последовательные подсостояния.

Используются две разновидности исторического состояния: *неглубокое (shallow history state)* и *глубокое (deep history state)*. *Неглубокое историческое состояние* запоминает историю только одного с ним уровня вложенности, а *глубокое* запоминает в своей истории все подсостояния любого уровня вложенности. Историческое состояние теряет свою историю при достижении конечного подсостояния.

Неглубокое историческое состояние обозначается специальным символом: буквой ***H***, помещенной внутри небольшой окружности. В обозначении глубокого исторического состояния к букве ***H*** добавляется символ «\*».

#### 4.9.4 Параллельные переходы

Параллельные переходы позволяют синхронизировать несколько параллельных процессов или разделить процесс на несколько параллельных.

Графически параллельный переход изображается жирной чертой (рисунок 4.24) с входящими в нее и выходящими линиями переходов.

Переход типа «соединение» (*join*) соединяет несколько исходных состояний с одним целевым и срабатывает только после срабатывания последнего из входящих в него переходов.

Переход типа «ветвление» (*fork*) соединяет одно исходное состояние с несколькими целевыми, при этом переходы во все целевые состояния срабатывают одновременно в момент срабатывания перехода из исходного состояния.

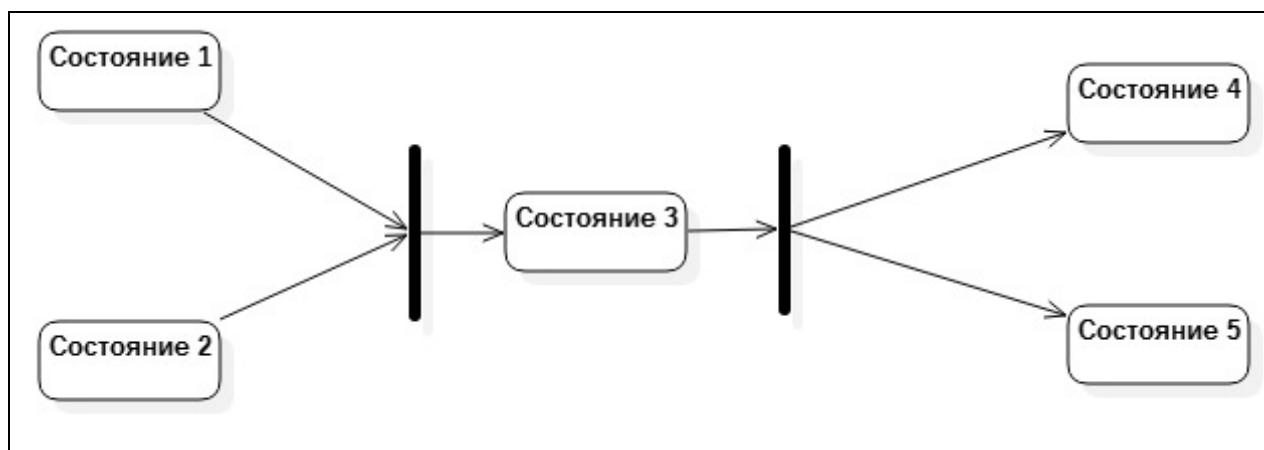


Рисунок 4.24 – Параллельные переходы

#### 4.9.5 Переходы в составных состояниях

На рисунке 4.25 приведен фрагмент диаграммы состояний, иллюстрирующий правила обозначения переходов между составными состояниями и между подсостояниями одного составного состояния.

Переход «***b***», стрелка которого соединена с границей составного состояния, обозначает переход в составное состояние, что соответствует переходу в начальное состояние каждого из подсостояний, входящих в составное состояние (на рисунке – это подсостояние «*SubState 5.2*»).

Переход «*e*», выходящий из составного состояния, относится ко всему составному состоянию и срабатывает при достижении последним из параллельных подсостояний своего конечного подсостояния.

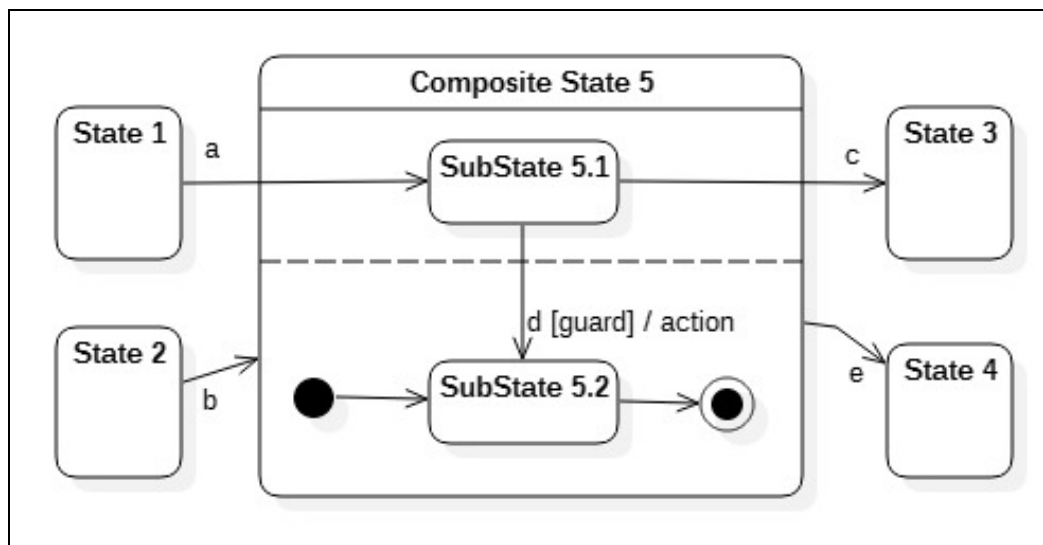


Рисунок 4.25 – Переходы в составных состояниях

Допускаются переходы от внешних исходных состояний непосредственно к вложенным целевым состояниям («*a*»), а также переходы от вложенных исходных подсостояний к внешним целевым состояниям, в обход границы составного состояния (переход «*c*»). Возможны также и переходы между последовательными подсостояниями разных параллельных подсостояний: на рисунке такой триггерный переход обозначен, как «*d*».

На рисунках 4.26 и 4.27 приведены две диаграммы состояний [14], иллюстрирующие различные возможности использования их графических элементов.

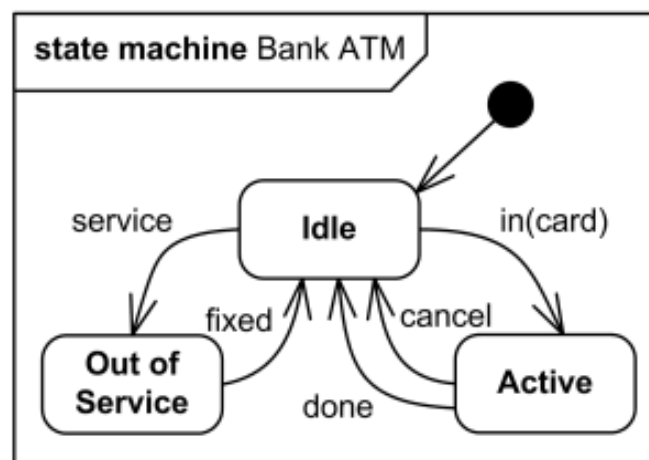


Рисунок 4.26 – Упрощенная диаграмма состояний банкомата

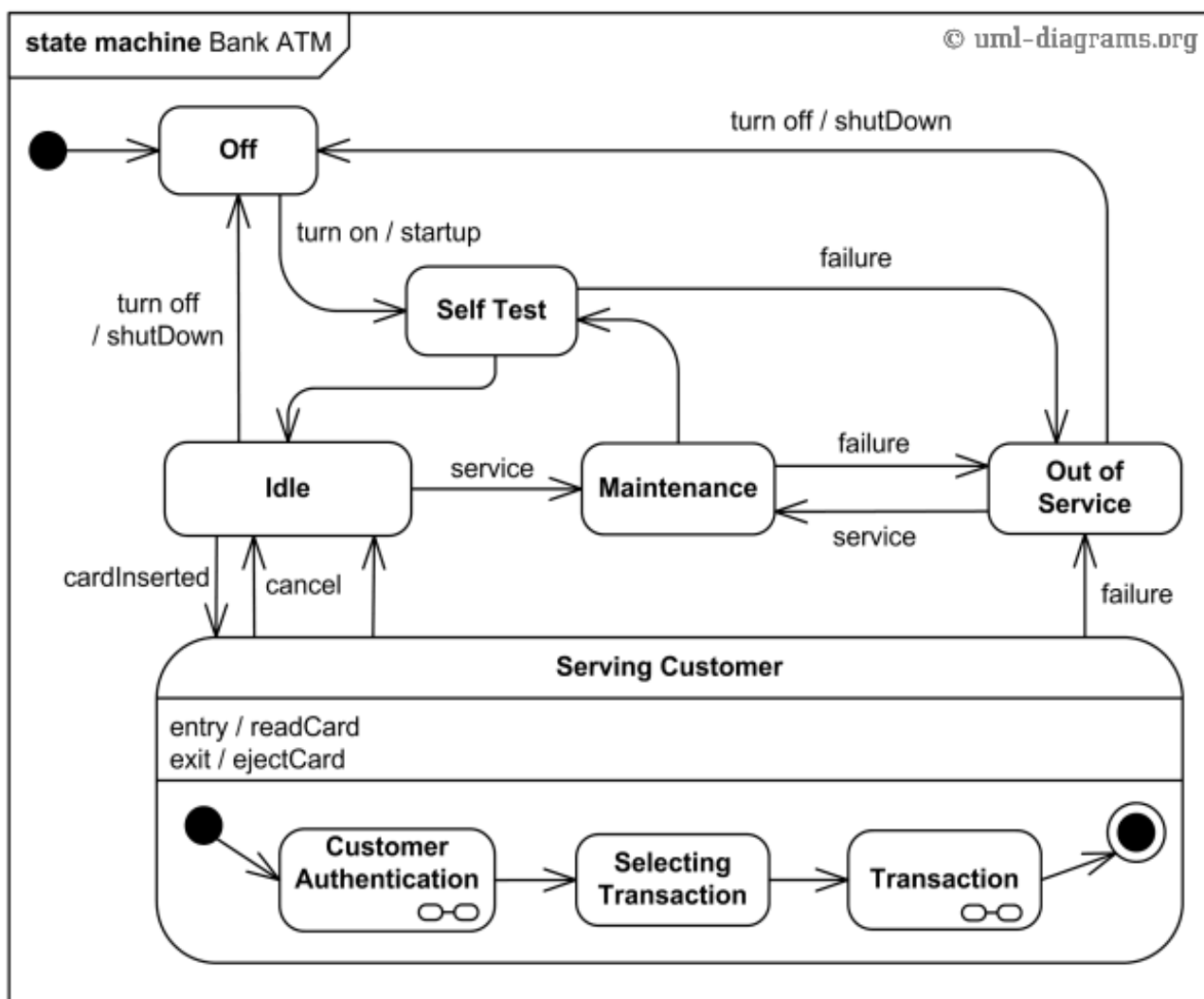


Рисунок 4.27 – Диаграмма состояний банкомата

### Контрольные вопросы и задания

- 1 Что означают метки действия **entry**, **exit** и **do**?
- 2 Как обозначаются на диаграммах переходы типа **fork**, **join** и **choice**?
- 3 Определите понятие «*историческое состояние*». Для чего используются и как обозначаются на диаграммах такие состояния?
- 4 Прокомментируйте диаграммы, приведенные на рисунках 4.25 – 4.27. Создайте свою версию – аналог одной из этих диаграмм.

## ПРОЕКТНЫЙ ПРАКТИКУМ

## 5.1 Общие методические указания

**Основная цель** проектного практикума – ознакомление с технологией объектно-ориентированного проектирования программных систем и получение практического опыта использования UML-ориентированных CASE-средств в процессе коллективного выполнения учебных программных проектов.

**Структура и содержание.** Программный проект выполняется командой разработчиков (2 – 3 человека) в соответствии с утвержденной темой проекта (раздел 5.5) и персональными заданиями, полученными каждым из членов команды. В составе проекта выполняются его начальные этапы, на которых последовательно разрабатываются UML-модели концептуального и логического уровней: диаграмма вариантов использования, диаграмма пакетов, диаграмма классов и диаграмма состояний.

**Проектная документация** оформляется в форме единого (для каждой команды) технического отчета, включающего титульный лист, введение, проектную часть, заключение и список используемых источников.

*Титульный лист* отчета содержит наименование кафедры и изучаемой дисциплины, формулировку темы учебного программного проекта и список его исполнителей.

*Введение* содержит общее описание предметной области, в которой будет функционировать проектируемая система, требования к ее функциональным характеристикам, классификацию пользователей.

*Проектная часть* включает четыре раздела, содержащие описание методов и результатов выполнения соответствующих стадий проекта.

В *заключении* приводится анализ результатов проектирования и формулируются предложения по развитию проекта.

**Защита проекта** проводится в форме собеседования по материалу представленного отчета. В процессе защиты оценивается полнота и качество выполнения практических заданий каждым из участников команды проекта, грамотность использования инструментальных средств, правильность и обоснованность выводов по результатам работы, качество оформления отчета.

**Программное обеспечение.** Проект выполняется в среде одной из CASE-систем, поддерживающих язык UML. Решение о выборе такой системы студенты принимают самостоятельно, рекомендуемая система – StarUML™ (любая из свободно распространяемых версий).

## 5.2 Практические задания

Учебный программный проект содержит четыре практических задания, объединенные общей темой и предусматривающие последовательную разработку четырех видов UML-диаграмм.

Каждое задание реализует одну из стадий проекта и требует освоения и применения соответствующих CASE-средств.

**Задание №1** реализует стадию технического задания (ТЗ), на которой разрабатывается терминологический словарь предметной области и уточняются требования к функционированию проектируемой системы. Результатом выполнения этого задания является концептуальная модель поведения системы, представленная обобщенной *UseCase-диаграммой* системы. Эта диаграмма составляет основу программного проекта и будет детализирована при выполнении последующих заданий.

**Задание №2** реализует стадию эскизного проекта (ЭП) и предусматривает проведение функциональной декомпозиции проектируемой системы и разработку *диаграммы пакетов* системы, а также *UseCase-диаграмм* и *сценариев вариантов использования* для отдельных ее подсистем.

**Задание №3** соответствует стадии технического проекта (ТП), на которой производится объектная декомпозиция проектируемой системы и разрабатываются ее структурные (статические) модели логического уровня. Результаты выполнения задания – множество *UML-диаграмм классов*, представляющих структуру системы в целом, структуру и взаимосвязи отдельных ее компонентов, разработанных на стадии эскизного проекта.

**Задание №4** также соответствует стадии технического проекта и продолжает детализацию тех компонентов структурной модели, динамический аспект функционирования которых является существенным. В результате выполнения задания формируются динамические модели логического уровня, представленные множеством *UML-диаграмм состояний*, которые могут использоваться для описания поведения объектов классов.

### **5.3 Содержание семинарских занятий**

Основная часть работы над проектом выполняется студентами самостоятельно. На семинарских занятиях рассматривается учебный пример выполнения и документирования программного проекта, обсуждаются методика и технология разработки моделей, заслушиваются сообщения разработчиков и анализируются представленные ими результаты выполнения этапов проекта.

#### **Занятие №1. Подготовительный этап:**

- рассмотрение примера выполнения программного проекта;
- формирование команд разработчиков;
- согласование тем проектов;
- согласование графика выполнения проектов;
- согласование требований к проектной документации.

#### **Занятие №2. Стадия ТЗ:**

- формирование терминологического словаря предметной области;
- обсуждение требований к функциональным характеристикам проектируемой системы;
- разработка UseCase-диаграммы системы.

#### **Занятие №3. Стадия ЭП:**

- разработка программной архитектуры системы (UML-диаграмма пакетов);
- разработка требований к функциональным характеристикам программных компонентов;
- разработка UseCase-диаграмм и сценариев вариантов использования для компонентов системы;
- совместный анализ результатов реализации 1-й и 2-й стадий проекта;
- согласование персональных заданий для членов команд проекта.

#### **Занятие №4. Стадия ТП:**

- объектная декомпозиция системы: разработка UML-диаграмм классов компонентов системы;
- совместный анализ результатов реализации 3-й стадии проекта.

#### **Занятие №5. Стадия ТП:**

- разработка компонентов динамической UML-модели (UML-диаграммы состояний для объектов классов);
- совместный анализ результатов реализации 4-й стадии проекта.

#### **Занятие №6. Защита проектов.**

## 5.4 Пример выполнения учебного программного проекта

### 1 Стадия «Техническое задание»

#### 1.1 Общая постановка задачи

Наименование разработки: *Автоматизированная система контроля знаний студентов по результатам тестирования.*

Назначение системы – автоматизация процессов тестирования студентов по разделам (темам) учебных дисциплин.

Область применения – тестирование по классической схеме: выбор правильных ответов из множества предложенных вариантов (в текстовых и/или графических форматах).

Масштаб развертывания системы – локальная сеть кафедры.

Таблица 5.1 – Пользователи и бизнес-процессы предметной области проекта

Пользователи	Основные бизнес-процессы
Администратор системы тестирования  Преподаватели кафедры  Кураторы студенческих групп  Студенты, изучающие дисциплины кафедры	<b>Администрирование системы:</b> <ul style="list-style-type: none"><li>■ управление правами доступа пользователей к компонентам системы;</li><li>■ поддержка справочников системы в актуальном состоянии:<ul style="list-style-type: none"><li>– учебные планы;</li><li>– состав преподавателей кафедры;</li><li>– контингент студентов;</li></ul></li><li>■ формирование локальных компонентов системы по заявкам преподавателей;</li><li>■ Архивирование данных о результатах тестирования.</li></ul> <b>Планирование:</b> <ul style="list-style-type: none"><li>■ формирование тематического плана изучения дисциплины;</li><li>■ формирование плана контрольного тестирования;</li><li>■ разработка тестов по разделам (темам) дисциплины.</li></ul> <b>Тестирование:</b> <ul style="list-style-type: none"><li>■ проведение сеансов пробного тестирования:<ul style="list-style-type: none"><li>– генерация заданий;</li><li>– оценивание;</li><li>– формирование журнала пробного тестирования;</li></ul></li></ul>



	<ul style="list-style-type: none"> <li>▪ проведение сеансов контрольного тестирования: <ul style="list-style-type: none"> <li>– генерация заданий;</li> <li>– оценивание;</li> <li>– формирование журнала контрольного тестирования;</li> <li>– формирование протокола контрольного тестирования;</li> <li>– сохранение результатов тестирования.</li> </ul> </li> </ul> <p><b>Мониторинг:</b></p> <ul style="list-style-type: none"> <li>▪ просмотр планов и результатов контрольного тестирования;</li> <li>▪ формирование рейтинговых списков студентов;</li> </ul> <p>подготовка аналитической и отчетной документации.</p>
--	---

## 1.2 Разработка терминологического словаря предметной области

Таблица 5.2 – Терминологический словарь

№	Термин	Применение	Комментарии
1	2	3	4
1	Специальность	Наименование и код специальности, направления подготовки или дополнительной образовательной программы, по которой обучаются студенты, проходящие тестирование	<p>Как правило, используются полное и краткое наименования (например: <i>Программная инженерия</i>; <i>ПрИнж</i>), и числовой код (например, 09.03.04), в котором представлены:</p> <ul style="list-style-type: none"> <li>• группа специальностей (09 – Информатика и вычислительная техника);</li> <li>• собственно специальность группы (04 – Программная инженерия);</li> <li>• образовательный уровень (03 – бакалавриат)</li> </ul>
2	Образовательный уровень	Наименование и код уровня высшего образования (согласно действующим нормативным документам)	<p>Код уровня – второй элемент кода специальности:</p> <p>03 – бакалавриат;  04 – магистратура;  05 – специалитет;  06 – аспирантура</p>

Продолжение таблицы 5.2

1	2	3	4
3	Форма обучения	Очная, заочная, вечерняя, дистанционная, экстернат, и т.д.	Как правило, первая буква наименования формы обучения предшествует номеру группы в наименовании группы
4	Студент	Студент, стажер или слушатель курсов повышения квалификации	Изучает дисциплину, включенную в систему тестирования кафедры
5	Группа	Группа студентов одного курса, специальности и формы обучения	Изучает дисциплину, включенную в систему тестирования
6	Дисциплина	Учебная дисциплина, обеспечиваемая кафедрой и включенная в систему тестирования	Допускается совпадение наименований дисциплин для разных специальностей. Дисциплина может изучаться в одном или нескольких семестрах
7	Учебный план	Выборка из учебного плана специальности, включающая только те дисциплины, по которым предполагается проводить тестирование на соответствующей кафедре	Содержит информацию о специальности, образовательном уровне, форме обучения и семестрах, в которых изучаются дисциплины
8	Тема	Раздел дисциплины, изучаемый в одном семестре, по которому предусмотрен контроль знаний в форме тестирования	Допускается наличие одинаковых тем в разных дисциплинах
9	Тематический план	Упорядоченная последовательность тем (разделов) одной дисциплины, изучаемых в одном семестре	Для каждой позиции тематического плана (темы) устанавливается «Максимальная оценка в баллах»

Продолжение таблицы 5.2

1	2	3	4
10	Задание	Один вопрос теста и несколько вариантов ответов на него	Среди предложенных вариантов ответов, как минимум, один вариант (и, как максимум, все варианты) должен быть правильным
11	Тест	Множество заданий, связанных с одной позицией тематического плана (темой дисциплины)	Каждый тест отнесен к одному уровню сложности и одному типу
12	Тип теста	Определяет область его применения, например: пробное тестирование, текущий контроль, рубежный контроль, итоговый контроль	Тип теста определяет преподаватель
13	Уровень сложности теста	Например: «начальный», «базовый», «средний», «высокий»	Уровень сложности теста определяет преподаватель
14	План тестирования	Расписание сеансов контрольного тестирования по дисциплине в течение семестра	Формируется преподавателем для каждой группы
15	Сеанс тестирования	Процесс пробного или контрольного тестирования одного студента по одной или нескольким темам дисциплины	Сеансы пробного тестирования инициируются студентами и проводятся индивидуально вне расписания
16	Журнал регистрации сеансов тестирования	Множество записей обо всех сеансах тестирования (как пробных, так и контрольных), датированных и хронологически упорядоченных в течение семестра	В каждой локальной БД поддерживается локальный журнал. Журнал не содержит информации о результатах тестирования

Продолжение таблицы 5.2

1	2	3	4
17	Протокол тестирования	Множество записей обо всех сеансах контрольного тестирования, датированных и хронологически упорядоченных в течение семестра. В каждой локальной БД поддерживается локальный протокол тестирования	Содержит индивидуальные результаты тестирования студентов по каждому заданию теста и интегральные показатели по каждому тесту – процент правильных ответов и количество набранных баллов
18	Аттестационная ведомость	Документ установленной табличной формы, содержащий наименование дисциплины, фамилию преподавателя и список студентов группы с их балльными оценками по дисциплине, суммированными по результатам текущего и рубежного контроля	Документ подготовлен к печати и/или к выгрузке в текстовый редактор или электронную таблицу. Обновляется 2-3 раза в семестре в соответствии с графиком проведения рубежного контроля
19	Рейтинг-лист по дисциплине	Документ табличной формы, содержащий список студентов группы с их балльными оценками по дисциплине, отсортированный в порядке убывания суммы баллов	Обновляется оперативно по факту завершения очередного сеанса контрольного тестирования. Может быть запрошен пользователем в произвольный момент времени
20	Рейтинг-лист за семестр	То же суммарно по всем дисциплинам	
21	Аналитический отчет	Документ произвольной формы, содержащий детализированную или статистическую информацию о результатах тестирования студентов.	Формируется по запросу преподавателя.

### *1.3 Общие требования к проектируемой системе*

#### **Архитектурные и эксплуатационные требования**

- Масштаб развертывания системы – локальная сеть кафедры.
- Приложение: двухуровневая клиент-серверная архитектура, клиентские приложения пользователей развернуты на рабочих станциях локальной сети.
- Хранилище данных: централизованное, состоит из множества баз данных, управляемых единым сервером баз данных.
- Требования безопасности – регистрация и аутентификация пользователей, разграничение прав их доступа к ресурсам системы.

#### **Требования к функциональным характеристикам**

- *Поддержка справочников* системы в актуальном состоянии: формирование и редактирование списков студенческих групп, учебных планов специальностей, преподавание дисциплин которых обеспечивает кафедра.
- *Управление доступом* пользователей к компонентам и информационным ресурсам системы.
- *Планирование*: тематические планы изучения дисциплин, планы контрольного тестирования по дисциплинам.
- *Подготовка тестов*: формирование, редактирование и классификация заданий тестов по уровням сложности и разделам (темам) учебных дисциплин.
- *Пробное тестирование*: проведение студентами сеансов самотестирования без регистрации индивидуальных результатов.
- *Контрольное тестирование*: проведение сеансов тестирования в соответствии с тематическими планами изучения дисциплин с регистрацией индивидуальных результатов.
- *Анализ*: обработка, визуализация и документальное оформление результатов контрольного тестирования (в т. ч. подготовка аттестационных ведомостей и рейтинг-листов в соответствии с требованиями балльно-рейтинговой системы оценки знаний).
- *Архивирование*: создание архивных копий баз данных по окончании учебного периода с возможностью просмотра сохраненной в них информации.

### 1.4 Категории пользователей проектируемой системы

Схема взаимодействия пользователей системы с ее функциональными модулями, реализующими основные бизнес-процессы (таблица 5.1), приведена на обобщенной диаграмме вариантов использования (рисунок 5.1).

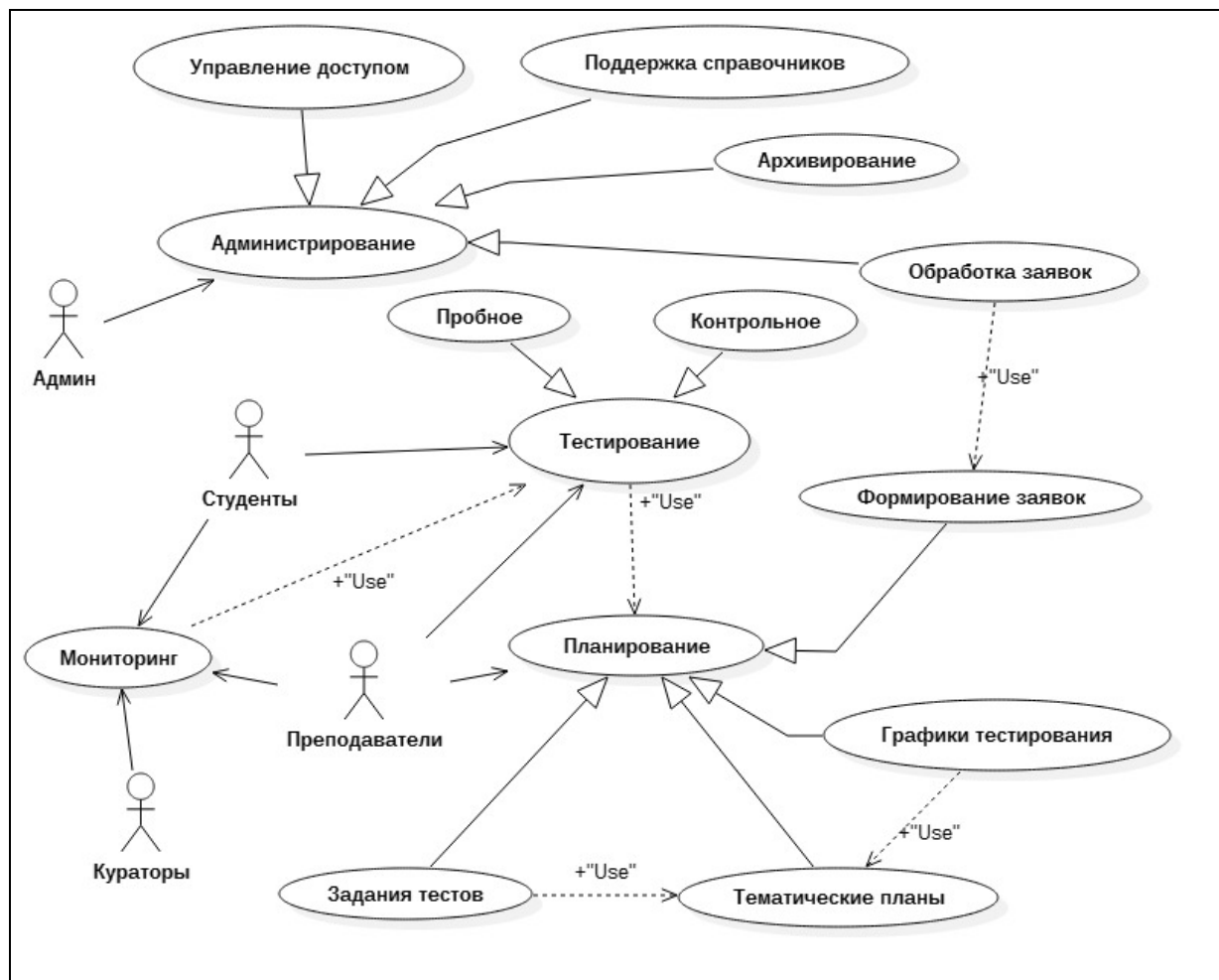


Рисунок 5.1 – Обобщенная UML-диаграмма вариантов использования

**Администратор** выполняет функции поддержки справочников в актуальном состоянии, управления доступом пользователей к компонентам системы и создания архивных копий баз данных. Осуществляет мониторинг исполнения заявок, поступающих от пользователей группы *Преподаватель*.

**Преподаватель** выполняет функции планирования, проведения сеансов контрольного тестирования и анализа результатов тестирования. Получает доступ к локальным модулям системы (в соответствии с перечнем дисциплин, закрепленных за преподавателем) с правами формирования и редактирования тематических планов изучения дисциплин и планов контрольного тестирования, формирования тестовых заданий и анализа результатов контрольного тестирования. Дополнительно получает право

размещения заявок администратору на формирование и актуализацию баз данных локальных модулей системы.

**Куратор** студенческой группы выполняет функцию анализа результатов тестирования студентов курируемых групп. Получает права просмотра тематических планов дисциплин, графиков контрольного тестирования и рейтинговых показателей своих групп.

**Студент** выполняет функции проведения индивидуальных сеансов пробного тестирования, участия в групповых сеансах контрольного тестирования и анализа результатов тестирования. Получает доступ к соответствующим локальным модулям системы с правами просмотра тематических планов дисциплин, проведения индивидуальных сеансов пробного тестирования, просмотра графиков контрольного тестирования и просмотра своих личных результатов и рейтинговых показателей своей группы.

## **2 Стадия «Эскизный проект»**

В соответствии с обобщенной диаграммой вариантов использования (рисунок 5.1) производится функциональная декомпозиция проектируемой системы. Результаты декомпозиции представляются *диаграммой пакетов* системы (рисунок 5.2), отражающей ее программную архитектуру, и множеством *диаграмм вариантов использования* (рисунки 5.3 и 5.4), детализирующих процессы функционирования отдельных ее подсистем.

### **2.1 Разработка программной архитектуры системы**

Программная архитектура системы включает две основные подсистемы: *хранилище данных* и *подсистему обработки данных* (рисунок 5.2).

**Хранилище данных** включает множество баз данных, управляемых единым сервером.

База данных «**Учебные планы**» используется для хранения информации о дисциплинах учебных планов различных специальностей, по которым предполагается проведение тестирования студентов.

База данных «**Студенты**» используется для хранения информации о контингенте студентов различных специальностей, изучающих в текущем учебном периоде дисциплины, включенные в систему тестирования кафедры.

База данных «**Преподаватели**» используется для хранения информации о преподавателях кафедры и преподаваемых ими учебных дисциплинах, включенных в систему тестирования.

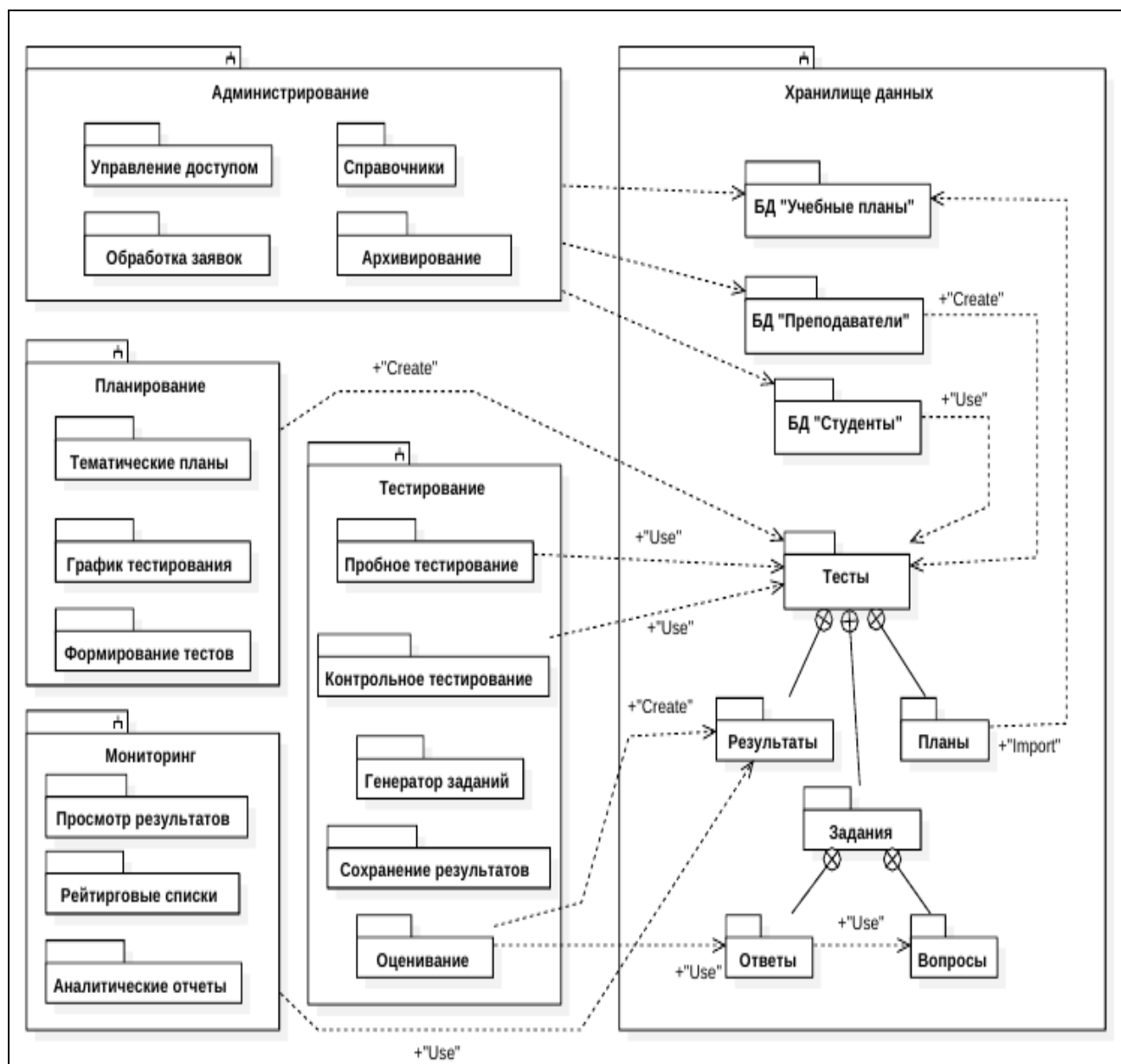


Рисунок 5.2 – UML-диаграмма пакетов

Множество локальных баз данных «**Тесты**» единой структуры, каждая из которых содержит данные об одной дисциплине: тематический план, множество тестовых заданий и результаты тестирования студентов, изучающих эту дисциплину в текущем семестре.

Базы данных «Учебные планы» и «Студенты» администрируются пользователем «Администратор» и обновляются по заявкам преподавателей.

Каждая из локальных баз данных «Тесты» администрируется соответствующим пользователем категории «Преподаватель» и ежегодно обновляется путем импорта актуальной информации из баз данных «Учебные планы» и «Студенты».



**Подсистема обработки данных** включает четыре взаимосвязанных модуля, обеспечивающих требуемый набор функций проектируемой системы.

Функции модуля **«Администрирование»**:

- поддержка справочников системы в актуальном состоянии;
- обработка заявок, поступивших от преподавателей;
- создание (по шаблону) локальных баз данных «Тесты»;
- ежегодное архивирование данных;
- регистрация пользователей и управление доступом.

Функции модуля **«Планирование»**:

- редактирование тематического плана изучения дисциплины;
- формирование плана контрольного тестирования по дисциплине, соответствующего тематическому плану её изучения;
- формирование, редактирование и классификация тестов;
- формирование и редактирование заданий тестов;
- сохранение результатов планирования в локальной базе данных.

Функции модуля **«Тестирование»**:

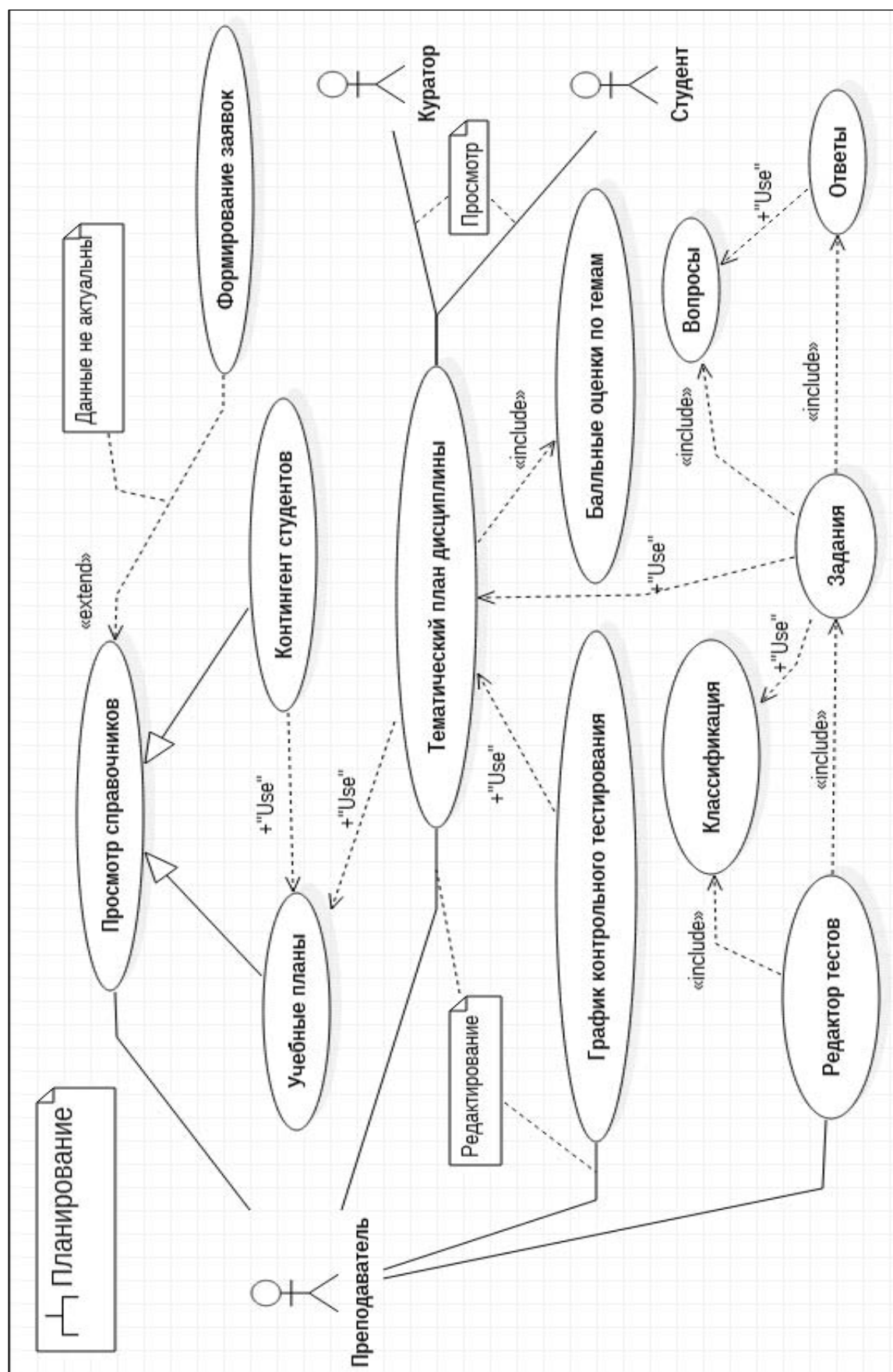
- регистрация и сопровождение сеансов тестирования;
- генерация индивидуальных тестов (случайная выборка заданий требуемого уровня сложности, соответствующих теме дисциплины);
- формирование журнала регистрации сеансов тестирования;
- оценивание результатов тестирования;
- формирование индивидуальных протоколов сеансов контрольного тестирования.

Функции модуля **«Мониторинг»**:

- просмотр планов и результатов тестирования;
- подготовка аттестационных ведомостей установленных форм;
- подготовка аналитических отчетов (рейтинг-листов и пр.);
- выгрузка и печать ведомостей и аналитических отчетов.

## *2.2 Разработка диаграмм вариантов использования компонентов*

В качестве примеров детализации процессов функционирования подсистем ниже приведены диаграммы вариантов использования для компонентов, включенных в пакеты **«Планирование»** и **«Тестирование»** (рисунки 5.3 и 5.4), и сценарий базового варианта использования **«Пробное тестирование»** (таблица 5.3).



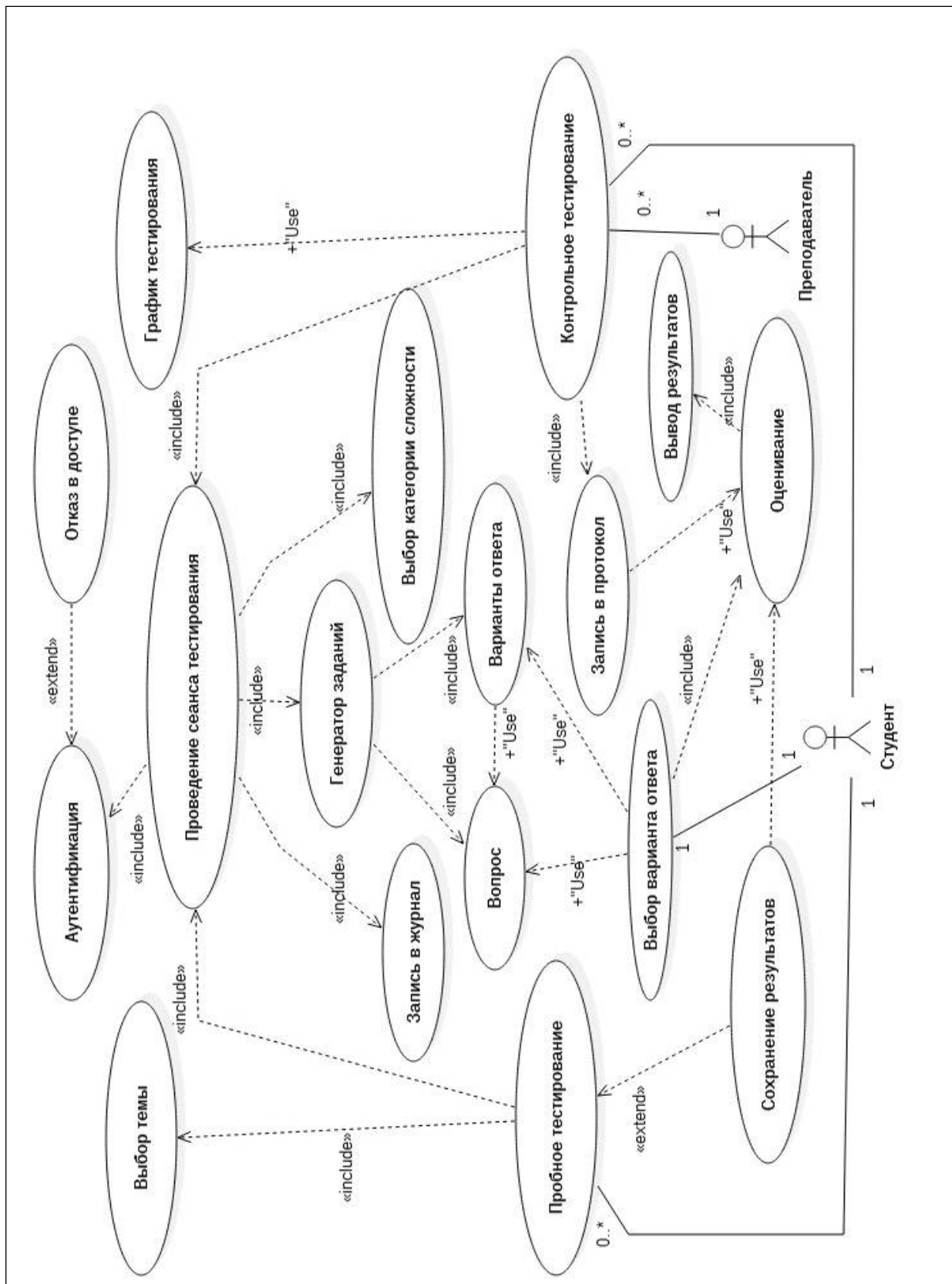


Рисунок 5.4 – UseCase -диаграмма подсистемы «Тестирование»

Таблица 5.3 – Сценарий варианта использования «Пробное тестирование»

Главный раздел	
Наименование	<i>Пробное тестирование</i>
Тип	Базовый
Акторы	<i>Студент</i>
Цель	Реализация процесса проведения сеанса пробного тестирования
Краткое описание	Студент проходит штатную процедуру идентификации и аутентификации, выбирает тему (из тематического плана изучения дисциплины), затем выбирает уровень сложности заданий теста и запускает сеанс тестирования. В течение сеанса система последовательно генерирует и отображает на экране вопросы с вариантами ответов, принимает идентификаторы выбранных студентом ответов и производит оценку правильности выбора. В произвольный момент времени студент может завершить сеанс тестирования. По окончании сеанса тестирования система отображает на экране результаты тестирования и регистрирует в журнале следующие параметры: <i>идентификатор студента, время начала и окончания сеанса тестирования, идентификаторы темы и уровня сложности заданий, количество выполненных заданий</i> . По желанию студента все параметры сеанса и результаты тестирования могут быть сохранены
Связанные варианты использования	Включаемые: « <i>Проведение сеанса тестирования</i> » и « <i>Выбор темы</i> ». Расширяющие: « <i>Сохранение результатов</i> » и « <i>Отказ в доступе</i> »
Раздел «Типичный ход событий»	
Действия акторов	Отклик системы
1 Активизирует рабочее окно клиентского приложения.	2 Запрашивает ID пользователя и пароль.
3 Вводит запрашиваемые данные. <i>Исключение 1:</i> отказ в доступе. 6 Выбирает тему. 8 Выбирает уровень сложности.	4 Выполняет процедуру аутентификации. 5 Предлагает меню выбора тем дисциплины. 7 Предлагает меню выбора уровня сложности заданий. 9 Выводит на экран очередное задание: вопрос и варианты ответа.

10 Выбирает вариант ответа. <i>Исключение 2:</i> выбрана команда «Завершение сеанса».	11 Производит оценку правильности ответа. 12 Временно сохраняет результаты оценивания. 13 Осуществляет переход к п.9.
Раздел « <i>Исключения</i> »	
Действия акторов	Отклик системы
<i>Исключение 1:</i> отказ в доступе.	
	14 Завершает сеанс пробного тестирования.
<i>Исключение 2:</i> выбрана команда «Завершение сеанса».	
15 Отказывается от сохранения. <i>Исключение 3:</i> Принимает предложение о сохранении результатов.	16 Записывает в журнал параметры сеанса пробного тестирования. 17 Отображает на экране результаты сеанса пробного тестирования. 18 Предлагает пользователю сохранить результаты пробного тестирования (по умолчанию – « <i>Не сохранять</i> »)). 19 Осуществляет переход к п.14.
<i>Исключение 3:</i> Пользователь принимает предложение о сохранении результатов.	
21 Вводит путь к файлу.	20 Запрашивает путь к файлу. 22 Записывает результаты тестирования в файл. 23 Осуществляет переход к п.14.

### 3 Стадия «Технический проект»

На стадии технического проекта проводится объектная декомпозиция системы, результаты которой представляются UML-диаграммами классов ее компонентов.

#### 3.1 Разработка диаграмм концептуальных классов

На рисунках 5.5, 5.6 и 5.7 приведены диаграммы концептуальных (пассивных) классов для пакетов подсистемы хранения данных (рисунок 5.2). Имена таких классов снабжены стереотипом «Entity» (сущность), у каждого из них имеется раздел атрибутов и отсутствует раздел операций. По существу, такие диаграммы представляют концептуальные модели «сущность – связь» компонентов подсистемы хранения данных и могут ис-

пользоваться для реализации схем баз данных на последующих стадиях проекта.

Диаграмма концептуальных классов пакета «БД Студенты» представляет структурную модель, описывающую контингент студентов (рисунок 5.5).

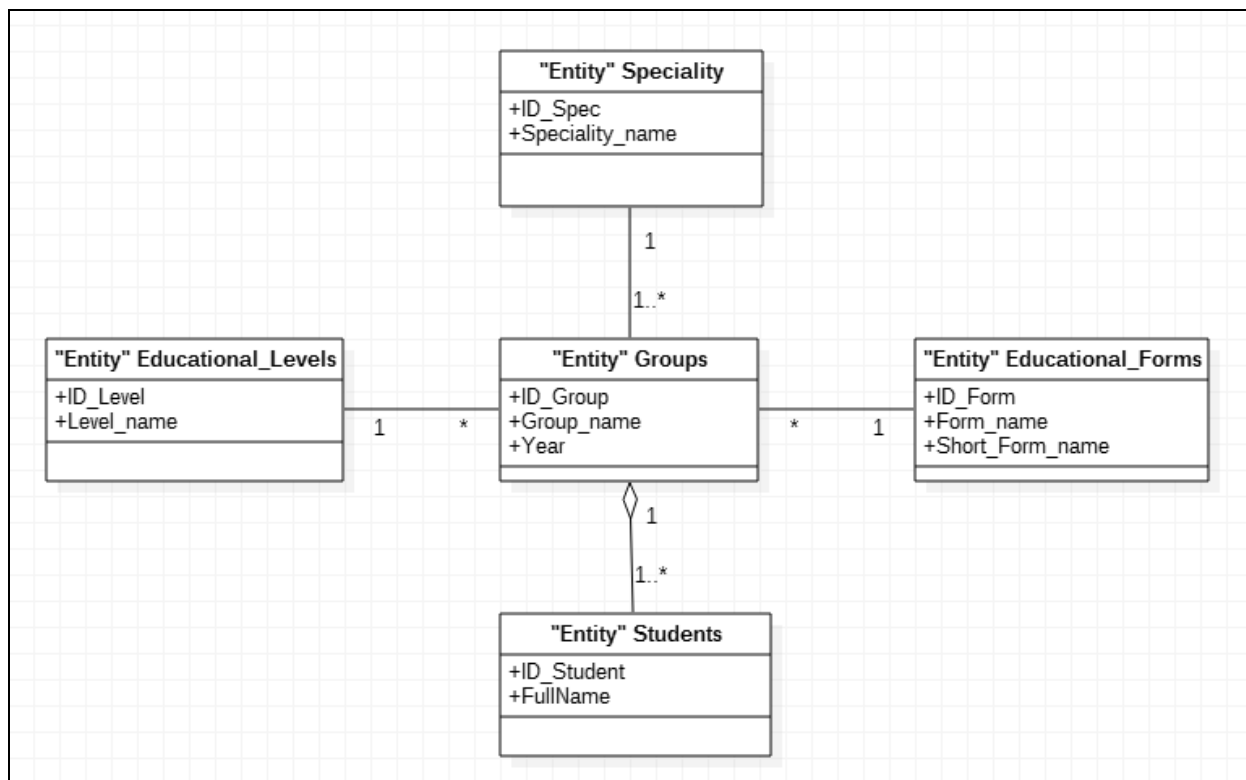


Рисунок 5.5 – Диаграмма концептуальных классов для пакета «БД Студенты»

Объекты класса «*Students*» связаны отношением агрегации с объектами класса «*Groups*», в котором присутствует открытый атрибут «*Year*», что позволяет не только определить списочный состав каждой из групп, но также, например, получить списки студентов по годам обучения (курсам).

Объекты класса «*Groups*» связаны отношениями ассоциации с объектами «*Speciality*», «*Educational\_Levels*» и «*Educational\_Forms*», что позволяет определить для каждого студента его принадлежность к определенной специальности, образовательному уровню и форме обучения.

На рисунке 5.6 представлена объединенная диаграмма концептуальных классов для пакетов «БД Студенты» и «БД Учебные планы», дополняющая предыдущую диаграмму информацией о структуре учебных планов.

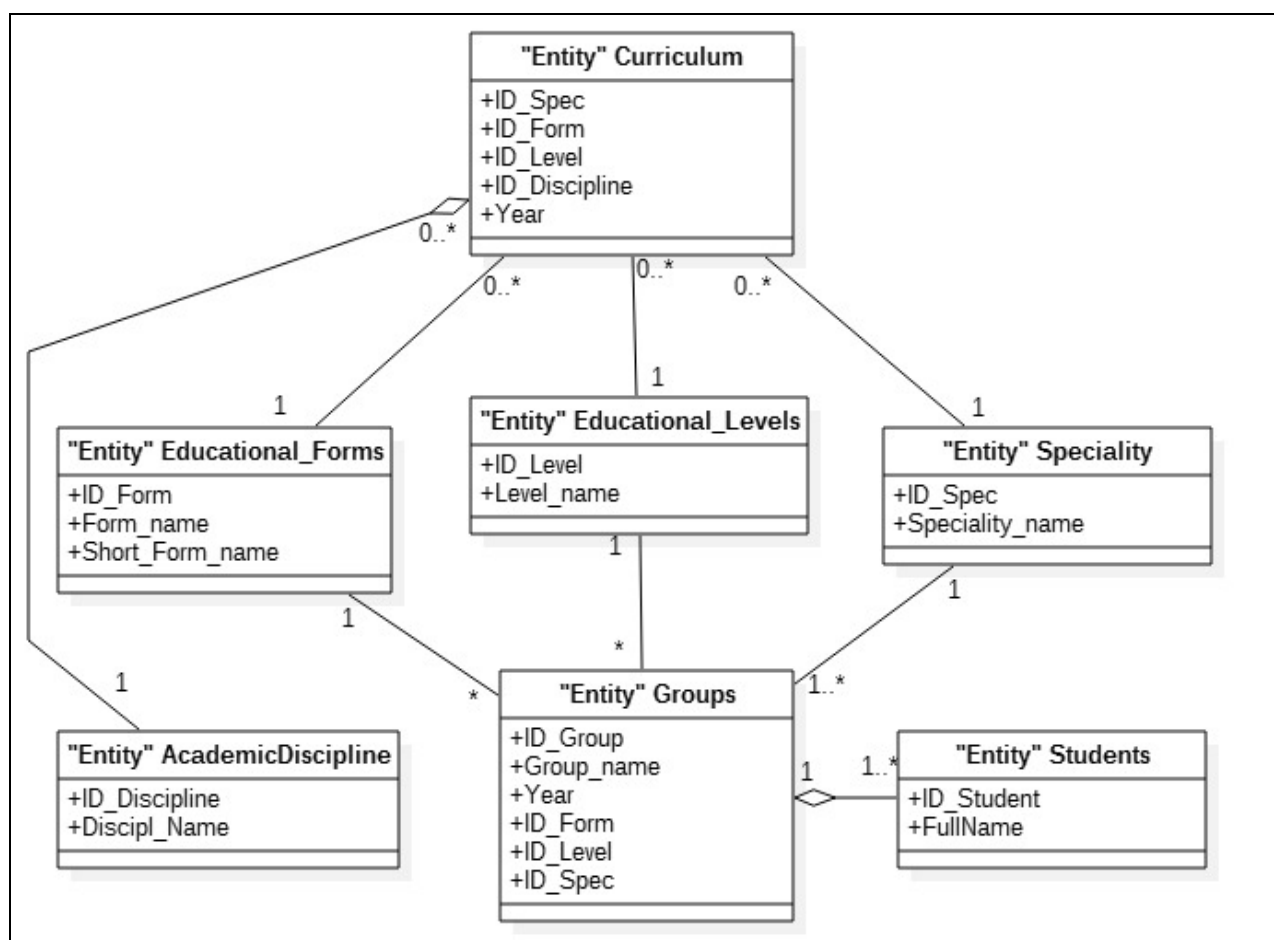


Рисунок 5.6 – Объединенная диаграмма концептуальных классов для пакетов «БД Студенты» и «БД Учебные планы»

Множество экземпляров класса «*AcademicDiscipline*» представляет справочник наименований учебных дисциплин, а каждый экземпляр класса «*Curriculum*» представляет одну из дисциплин учебного плана, сформированного для определенной специальности, формы и уровня обучения. Для каждой дисциплины одного учебного плана определен атрибут «*Year*» – учебный год, в котором она изучается, и могут быть определены и другие атрибуты, существенные в рамках выполняемого проекта.

Модель, представленная этой диаграммой, позволяет, в частности, определить для каждого студента перечень учебных дисциплин, изучаемых им в каждом учебном году. Простое дополнение модели позволит детализировать распределение дисциплин не только по годам, но и по семестрам.

На рисунке 5.7 приведена диаграмма концептуальных классов для пакета «Тесты».

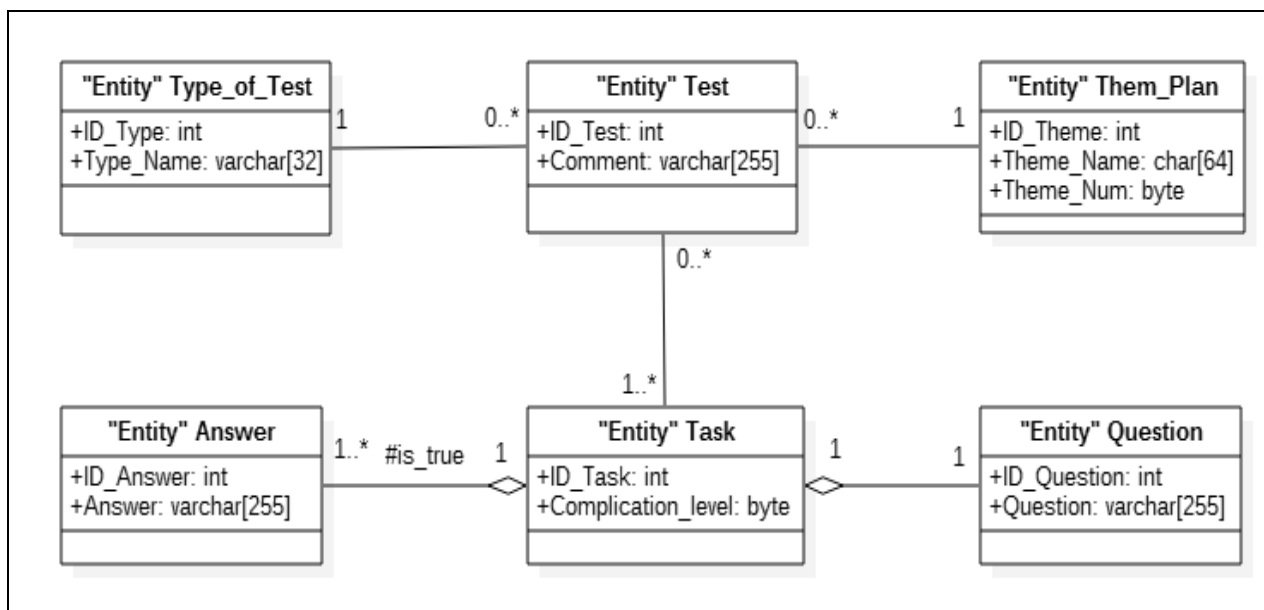


Рисунок 5.7 – Диаграмма концептуальных классов пакета «Тесты»

Объекты класса «*Them\_Plan*» представляют отдельные темы изучаемой дисциплины, по каждой из которых может быть подготовлено (в рамках подсистемы «Планирование») множество тестов – экземпляров класса «*Test*». Класс «*Type\_of\_Test*» представляют множество типов тестов (например, «пробный», «контрольный», «рубежный», «экзаменационный» или какой-либо иной), с каждым из которых может быть ассоциировано множество различных тестов.

Задания тестов представляются объектами класса «*Task*», ассоциированного с классом «*Test*»: в одном тесте может быть несколько заданий, при этом одно и то же задание может быть включено более, чем в один тест.

Задание представляет собой агрегат, состоящий из одного вопроса (экземпляр класса «*Question*») и нескольких вариантов ответа (экземпляров класса «*Answer*»). Некоторые из вариантов ответа на вопрос являются правильными, что отмечено на диаграмме защищенным атрибутом «*#is\_true*» отношения агрегации между классами «*Task*» и «*Answer*».

### 3.2 Разработка диаграмм программных классов

На рисунке 5.8 представлена диаграмма классов подсистемы «Тестирование», разработанная в соответствии с диаграммой пакетов (рисунок 5.2) и диаграммой вариантов использования (рисунок 5.4) этой подсистемы. В отличие от рассмотренных ранее диаграмм концептуальных



классов, эта диаграмма содержит так называемые *программные классы*, в которых определены не только атрибуты, но и методы – операции классов.

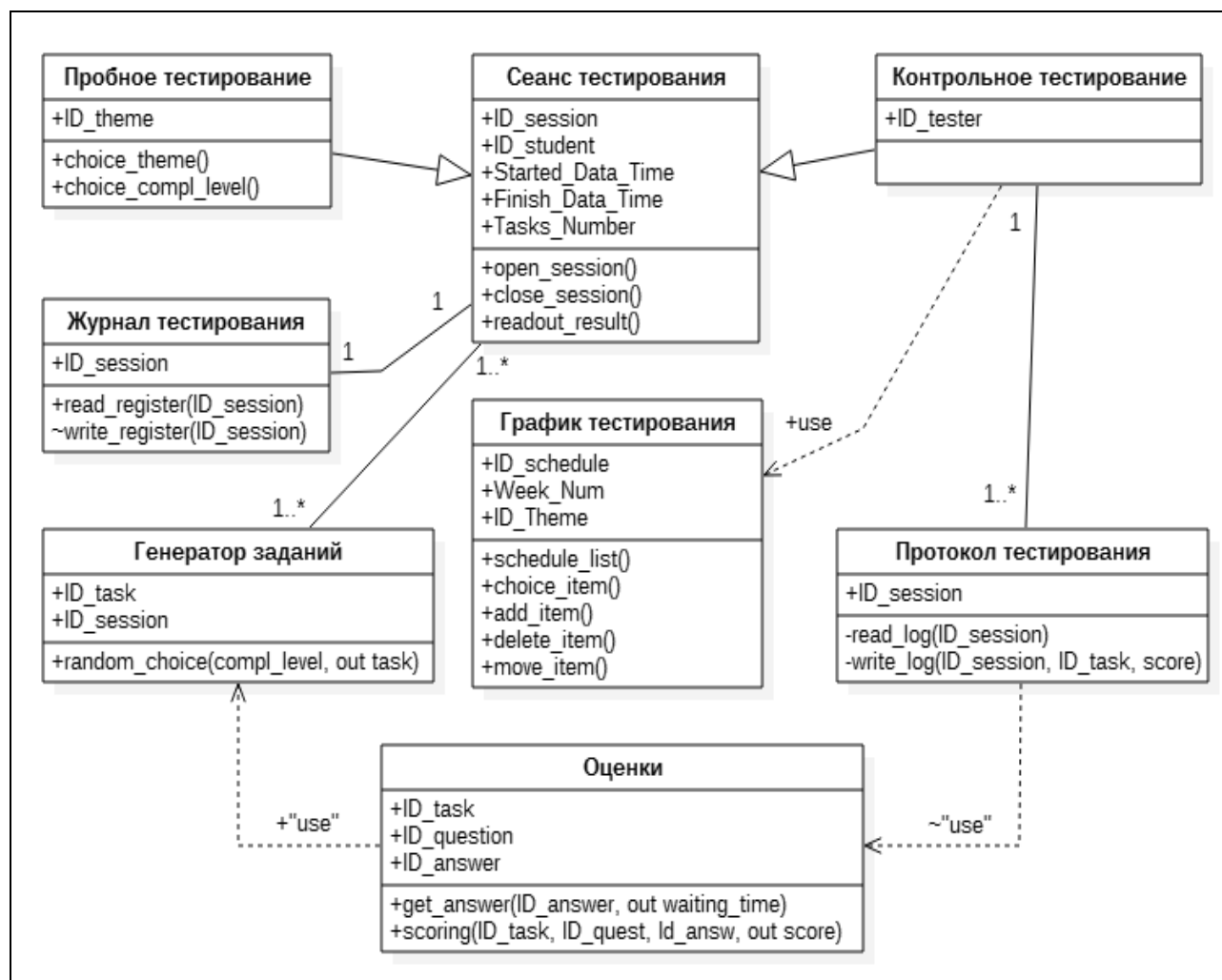


Рисунок 5.8 – Диаграмма программных классов подсистемы «Тестирование»

Каждый объект класса «Сеанс тестирования» представляет один сеанс тестирования (как пробного, так и контрольного) одного студента. Общедоступные атрибуты этого класса определяют время начала и окончания сеанса и количество заданий, выполненных в течение сеанса. Методы класса позволяют открывать и закрывать сеансы тестирования, а также выводить информацию о результатах тестирования на экран монитора.

Класс связан отношением ассоциации с классом «Журнал тестирования», в котором определен метод «~write\_register()», область видимости которого ограничена только классами пакета «Тестирование». Метод обеспечивает запись в журнал информации о каждом завершенном сеансе

тестирования, как пробном, так и контрольном. Кратность ассоциации «один-к-одному» указывает на то, что для каждого сеанса тестирования создается одна журнальная запись, в которой сохраняются значения атрибутов класса «*Сеанс тестирования*», но не сохраняется информация о результатах тестирования. В этом же классе определен общедоступный метод «*+read\_register()*», обеспечивающий возможность просмотра журнала.

Дочерний класс «*Пробное тестирование*» характеризуется собственным атрибутом, определяющим тему дисциплины, и двумя собственными методами, обеспечивающими возможность выбора (из предлагаемого «меню») темы дисциплины и уровня сложности заданий теста.

Дочерний класс «*Контрольное тестирование*» содержит собственный атрибут, идентифицирующий преподавателя, проводящего сеанс тестирования, и использует («*use*») в качестве источника зависимости класс «*График тестирования*», общедоступные методы которого позволяют, в частности, просматривать график тестирования и выбирать тему дисциплины для сеанса контрольного тестирования.

Класс «*Сеанс тестирования*» связан отношением ассоциации с классом «*Генератор заданий*», метод которого производит случайную выборку заданий заданного уровня сложности. Кратность ассоциации «многие-ко-многим» указывает на то, что в одном сеансе может быть более одного задания, и при этом одно и то же задание может быть выбрано в нескольких сеансах.

Класс «*Генератор заданий*» связан отношением зависимости (в качестве источника) с классом «*Оценки*», методы которого позволяют получить (от студента) ответ на очередное задание и оценить полученный ответ.

В свою очередь, класс «*Оценки*» связан отношением зависимости (в качестве клиента) с классом «*Протокол тестирования*», один из методов которого сохраняет идентификатор задания и полученную оценку. Класс «*Протокол тестирования*» связан с классом «*Контрольное тестирование*» отношением ассоциации кратности «многие-к-одному» – это отражает тот факт, что для каждого сеанса контрольного тестирования в протокол заносится информация о всех заданиях и оценках, полученных студентом за их выполнение.

Рисунок 5.9 иллюстрирует еще один пример диаграммы программных классов, построенной для пакета «*Формирование тестов*» подсистемы «*Планирование*» (рисунок 5.2).

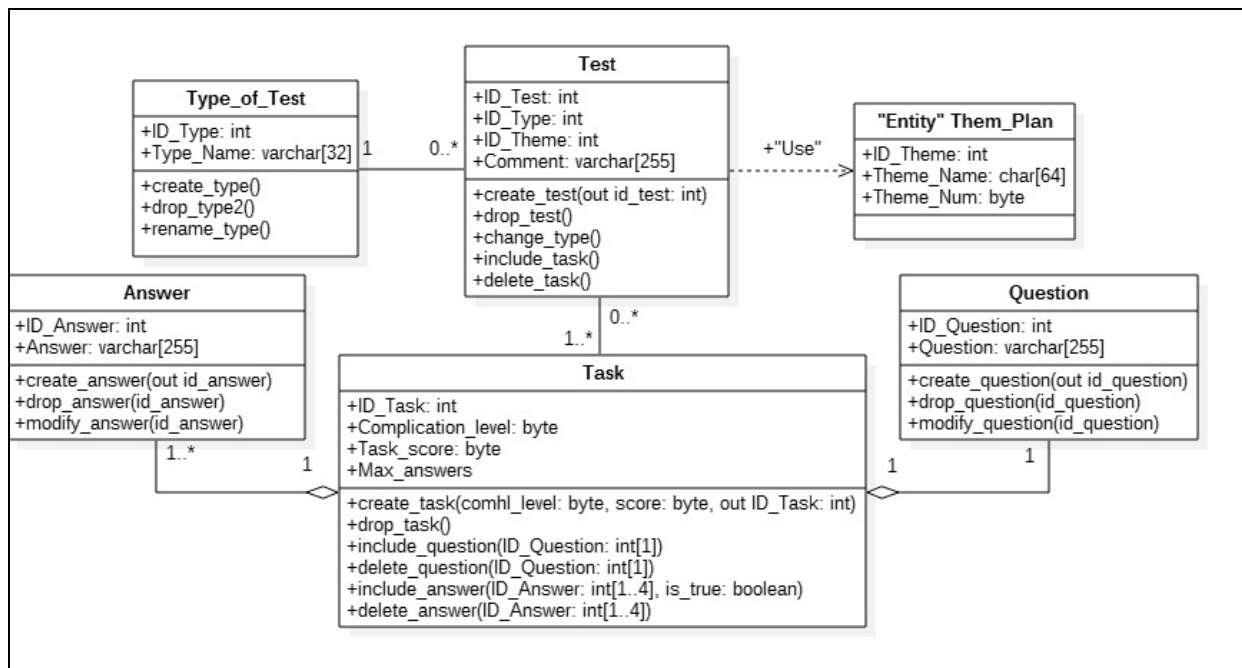


Рисунок 5.9 – Диаграмма классов пакета «Формирование тестов»

Методы класса «*Test*» позволяют создавать и уничтожать объекты класса, изменять уровень сложности заданий теста, а также включать и удалять задания теста.

Методы класса «*Task*» позволяют создавать объекты класса (задания), присваивая им уровень сложности и максимальную балльную оценку, а также включать в задания вопросы и соответствующие им ответы, присваивая некоторым из них статус «правильного ответа».

### 3.3 Разработка диаграмм состояний

Рассмотренные выше диаграммы классов представляют статические модели компонентов проектируемой системы и отражают результаты её объектной декомпозиции. Результаты статического моделирования, полученные на данной стадии проекта, уже позволяют приступить к более детальному проектированию программной системы, например, к конструированию схемы базы данных и к разработке некоторых программных компонентов.

Однако в ряде случаев такая статическая модель оказывается малоинформативной и недостаточной для программной реализации компонентов, динамический аспект функционирования которых является существенным.

Например, диаграмма классов, представленная на рисунке 5.9, не учитывает большой длительности и, как следствие, многосессионности

процесса формирования тестов – в ней явно недостает информации о состоянии системы в момент прерывания очередного сеанса планирования, необходимой для эффективного продолжения процесса при открытии следующего сеанса.

Еще более яркий пример – диаграмма классов подсистемы «Тестирование» (рисунок 5.8), реализация которой потребует дополнительной информации, необходимой для моделирования реакции системы на внутренние и внешние события, такие, например, как «истекло время ожидания ответа», «ответ выбран», «аварийное прерывание сеанса» или «сеанс штатно завершен».

В качестве примера реализации динамической модели логического уровня на рисунках 5.10 и 5.11 приведены упрощенные диаграммы состояний для пакета «Тестирование», разработанные в соответствии с диаграммой классов этого пакета (рисунок 5.8).

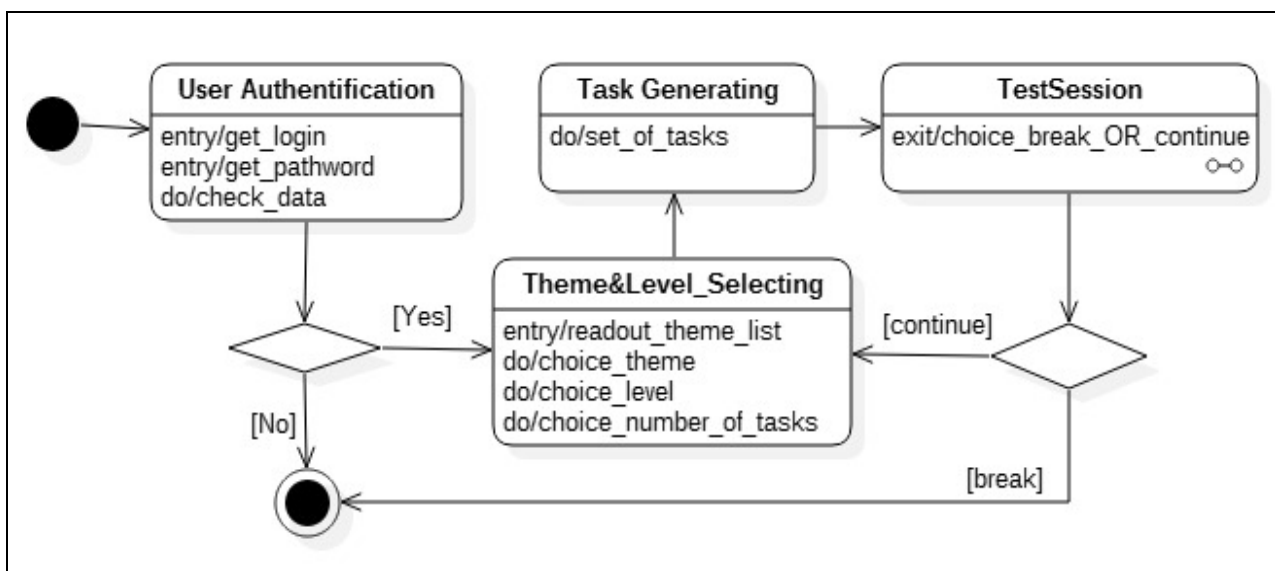


Рисунок 5.10 – Диаграмма состояний пакета «Тестирование»

Первое из целевых состояний «*User Authentication*» завершится альтернативным переходом: в случае, если аутентификация пользователя завершена успешно, он получает доступ к системе тестирования, которая переходит в следующее целевое состояние «*Theme&Level\_Selecting*», в противном случае – сразу переходит в конечное состояние.

В состоянии «*Theme&Level\_Selecting*» выполняется входное (entry) действие – вывод на экран меню выбора темы дисциплины и последующие (do) действия, связанные с ожиданием ввода и приемом результатов выбора темы, уровня сложности и количества заданий. Из этого состояния осуществ-

ляется переход в целевое состояние «*Task Generating*», в котором выполняется деятельность по формированию множества заданий, соответствующих выбранной теме и уровню сложности, и далее – переход в целевое состояние «*Test Sessions*», в котором реализуется сеанс тестирования. Это состояние показано на диаграмме как «скрытое состояние», детали которого отображены на отдельной диаграмме состояний (приведена ниже на рисунке 5.11).

Состояние «*Test Sessions*» завершается выходным (exit) действием, в результате которого пользователь (тестируемый студент) выбирает следующий шаг взаимодействия с системой (продолжение и завершение работы), и осуществляется альтернативный переход системы либо в состояние «*Theme&Level\_Selecting*» (для запуска следующего сеанса тестирования), либо в конечное состояние.

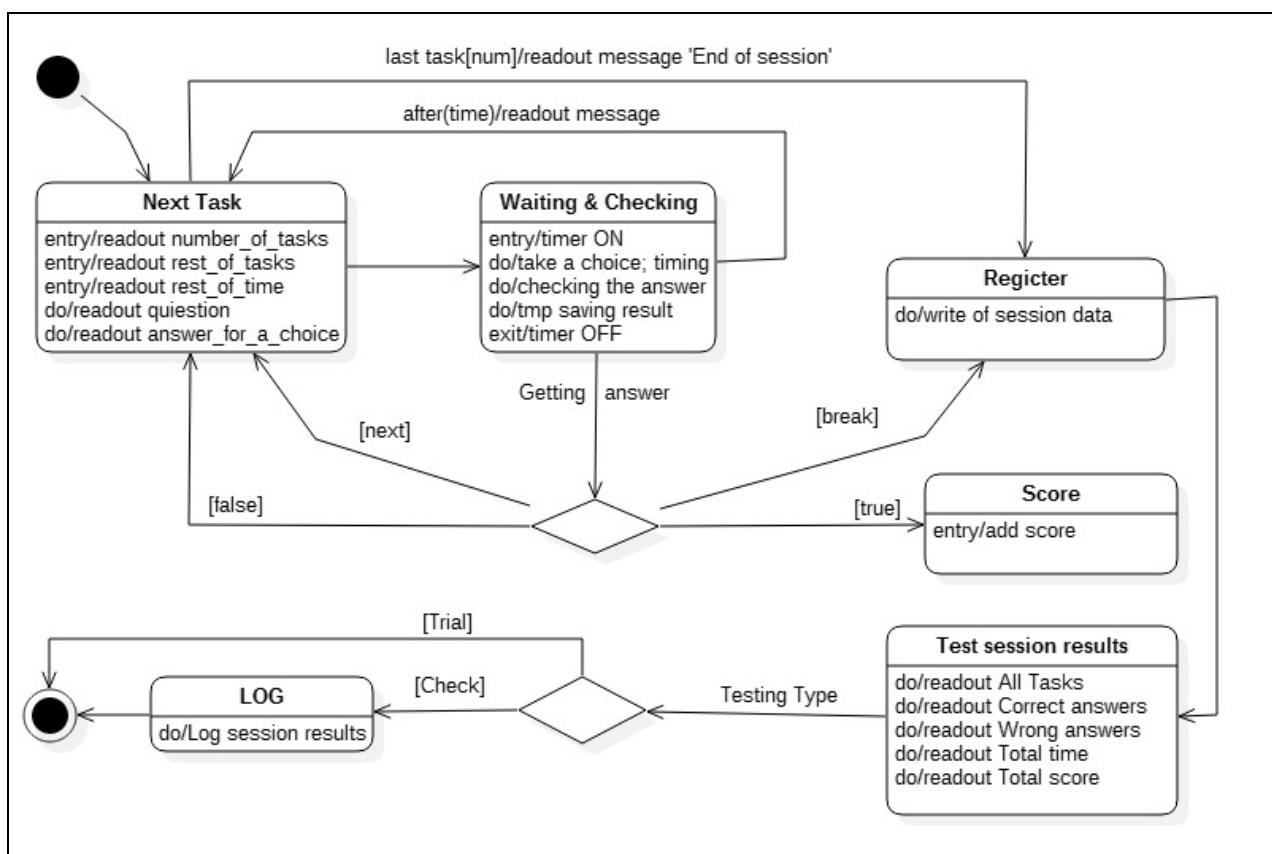


Рисунок 5.11 – Диаграмма состояний «Сеанс тестирования»

Входные действия состояния «Next Task» отображают на терминале тестируемого порядковый номер очередного задания, количество оставшихся заданий и оставшийся лимит времени, после чего выводится очередной вопрос и соответствующие ему варианты ответов и осуществляется переход в состояние «Waiting and Checking», в котором включается

таймер, ожидается ввод выбранного ответа, ответ проверяется, и временно сохраняются результаты проверки, после чего таймер выключается.

Если время ожидания ответа превышает заданное значение (*time*), срабатывает безальтернативный триггерный переход в состояние «*Next Task*», в противном случае – срабатывает альтернативный переход, управляемый сторожевым условием «*Getting Answer*»:

- если получен правильный ответ [*true*] – переход в состояние «*Score*», увеличивающее счетчик баллов, и далее – в состояние «*Next Task*»;
- при неправильном ответе [*false*] – сразу переход в состояние «*Next Task*»;
- если получен ответ [*Next*] (тестируемый принял решение перейти к следующему заданию, не выполнив предыдущего) – переход в состояние «*Next Task*»;
- если получен ответ [*Break*] (тестируемый принял решение прервать сеанс) – переход в состояние «*Register*».

Если в состоянии «*Next Task*» список заданий исчерпан, срабатывает триггерный переход (инициируемый событием «*Last\_task*») в целевое состояние «*Register*», в котором формируется очередная запись в журнал сеансов тестирования, и далее – переход в состояние «*Test Session Results*», действия которого выводят на терминал тестируемого основные результаты сеанса.

По завершению всех действий состояния «*Test Session Results*» срабатывает альтернативный переход, управляемый условием «*Testing Type*»: если проводился сеанс контрольного тестирования [*check*], осуществляется переход в состояние «*LOG*», действие которого сохраняет результаты в протоколе тестирования, и далее срабатывает переход в конечное состояние; если проводился сеанс пробного тестирования [*trial*], его результаты не сохраняются в протоколе тестирования, а сразу осуществляется переход в конечное состояние.

#### **4 Обзор результатов программного проекта**

Подводя итоги выполнения трех стадий учебного программного проекта, перечислим основные из полученных результатов.

На стадии *технического задания* определены назначение и область применения проектируемой системы, разработан терминологический словарь предметной области, определен состав пользователей и сформулиро-

ваны основные функциональные требования, разработана обобщенная UML-диаграмма вариантов использования.

На стадии *эскизного проекта* разрабатывались концептуальные модели проектируемой системы: проведена ее декомпозиция, результаты которой представлены диаграммой пакетов, отражающей ее программную архитектуру, и множеством диаграмм и сценариев вариантов использования, детализирующих процессы функционирования отдельных подсистем.

На стадии *технического проекта* разрабатывались структурные и динамические модели логического уровня, детализирующие концептуальное представление о системе, выработанное на предшествующих стадиях проекта.

Разработаны диаграммы концептуальных классов для компонентов системы, обеспечивающих хранение данных, и диаграммы программных классов для компонентов системы, обеспечивающих обработку данных.

Для компонентов, динамические аспекты функционирования которых оказались существенными, разработаны диаграммы состояний, отражающие реакцию системы на внутренние и внешние события.

Таким образом, получен набор артефактов проекта, включающий UML-диаграммы и их описания, минимально достаточный для программной реализации компонентов проектируемой системы на завершающей стадии проекта.

## **5.5 Рекомендуемая тематика учебных программных проектов**

Предлагаемые варианты тем проектов (приложение Г) сгруппированы по пяти тематическим категориям, каждая из которых соответствует определенной предметной области, в которой будет функционировать проектируемая программная система.

Команда проекта может выбрать для реализации один из предложенных вариантов, может внести корректировки в выбранный вариант или предложить свою тему проекта. В любом случае тема проекта и основное содержание проектируемой системы должны быть согласованы с преподавателем.

## СПИСОК ЛИТЕРАТУРЫ

- 1 Бек К. Экстремальное программирование: разработка через тестирование. Библиотека программиста. – Санкт-Петербург : Питер, 2003. – 224 с. : ил.
- 2 Брауде Э. Технология разработки программного обеспечения / пер. с англ. – Санкт-Петербург : Питер, 2004.
- 3 Буч Г. и др. Объектно-ориентированный анализ и проектирование с примерами приложений на C++. – 3-е изд. / пер. с англ. – Москва : Издательский дом «Вильямс», 2010. – 720 с.
- 4 Бюрер К. От ремесла к науке: поиск основных принципов разработки программного обеспечения. URL: <http://www.therationaledge.com/bio/buhrer.html/>
- 5 Карпенко С. Н. Введение в программную инженерию. Учебно-методические материалы по программе повышения квалификации «Информационные технологии и компьютерное моделирование в прикладной математике». – Нижний Новгород, 2007. – 103 с.
- 6 Леоненков А. Самоучитель UML. – 2-е изд. – Санкт-Петербург : БХВ-Петербург, 2004. – 418 с.
- 7 Липаев В. В. Программная инженерия. Методологические основы : учебник. – Москва : ТЕИС, 2006. – 608 с.
- 8 Мирютов А. А. Учебное пособие по дисциплине «Проектирование программного обеспечения» магистерской программы «Технология разработки программных систем» направления «Информатика и вычислительная техника». – Томск : Изд-во ТУСУР, 2007. – 115 с.
- 9 Новиков Ф. А., Иванов Д. Ю. Моделирование на UML. URL: <http://www.book.uml3.ru>
- 10 Рекомендации по преподаванию программной инженерии и информатики в университетах (Software engineering 2004: curriculum guidelines for undergraduate degree programs in software engineering) / пер. с англ. – Москва : ИНТУИТ.РУ «Интернет-университет информационных технологий», 2007. URL: [www.intuit.ru](http://www.intuit.ru).
- 11 Соммервилл И. Инженерия программного обеспечения. – 6-е изд. / пер. с англ. – Москва : Издательский дом «Вильямс», 2002. – 624 с. : ил.
- 12 Фатрелл Р. Т., Шафер Д. Ф., Шафер Л. И. Управление программными проектами: достижение оптимального качества при минимуме затрат / пер. с англ. – Москва : Издательский дом «Вильямс», 2003. – 1136 с.
- 13 Якобсон А., Буч Г., Рамбо Дж. Унифицированный процесс разработки программного обеспечения / пер. с англ. – Санкт-Петербург : Питер, 2002. – 496 с. : ил.
- 14 UML 2.5 Diagrams Overview. URL: <https://www.uml-diagrams.org/>



# ПРИЛОЖЕНИЯ

## Приложение А

### Профессиональные IT-стандарты РФ

Область профессиональной деятельности: 06 – Связь, информационные и коммуникационные технологии				
Код	Вид профессиональной деятельности	Наименование профессионального стандарта	Приказ Минтруда России	
			№	Дата
06.001	Разработка программного обеспечения (ПО)	Программист	679н	18.11.2013
06.003	Создание и сопровождение архитектуры программных средств	Архитектор ПО	228н	11.04.2014
06.004	Оценка качества разрабатываемого программного обеспечения	Специалист по тестированию в области информационных технологий (ИТ)	225н	11.04.2014
06.011	Поддержание эффективной работы баз данных, обеспечивающих функционирование информационных систем (ИС)	Администратор баз данных	647н	17.09.2014
06.012	Предпринимательская деятельность в области ИТ	Менеджер продуктов в области ИТ	915н	20.11.2014
06.013	Создание и управление информационными ресурсами	Специалист по информационным ресурсам	629н	08.09.2014
06.014	ИТ в экономике и государственном управлении.	Менеджер по ИТ	716н	13.10.2014
06.015	Создание и поддержка ИС в экономике.	Специалист по ИС	896н	18.11.2014
06.016	Менеджмент проектов в области ИТ	Руководитель проектов в области ИТ	893н	18.11.2014
06.017	Менеджмент проектов в области ИТ	Руководитель разработки ПО	645н	17.09.2014
06.019	Разработка технической документации и методического обеспечения продукции в сфере ИТ	Технический писатель	612н	08.09.2014
06.022	Проектно-исследовательская деятельность в области ИТ	Системный аналитик	809н	28.10.2014
06.026	Администрирование информационно-коммуникационных систем (ИКС)	Системный администратор ИКС	684н	05.10.2015
06.027	Администрирование сетевых устройств ИКС	Специалист по администрированию сетевых устройств ИКС	686н	05.10.2015

## Приложение Б

### Образовательные IT-стандарты

#### 1 Федеральные государственные образовательные стандарты РФ

Группа направлений подготовки: 09 – Информатика и вычислительная техника				
Код	Образовательный уровень	Направление подготовки	Приказ Минобрнауки России	
			№	Дата
09.03.01	Бакалавриат	Информатика и вычислительная техника	929	19.09.2017
09.03.02		Информационные системы и технологии	926	19.09.2017
09.03.03		Прикладная информатика	922	19.09.2017
09.03.04		Программная инженерия	920	19.09.2017
09.04.01	Магистратура	Информатика и вычислительная техника	918	19.09.2017
09.04.02		Информационные системы и технологии	917	19.09.2017
09.04.03		Прикладная информатика	916	19.09.2017
09.04.04		Программная инженерия	932	19.09.2017

#### 2 Международные образовательные стандарты

Guide to the Software Engineering Body of Knowledge (SWEBOK), IEEE 2004 Version – Руководство к Своду Знаний по Программной Инженерии.

Software Engineering 2004. Curriculum Guidelines for Undergraduate Degree Programs in Software Engineering – Рекомендации по преподаванию программной инженерии.

**Перечень стандартов качества ПО**

- 1 CMMI – Capability Maturity Model Integration for Product and Process Development – Интегрированная модель оценивания зрелости продуктов и процессов разработки программных средств.
- 2 ISO 15288:2002. Системная инженерия. Процессы жизненного цикла систем.
- 3 ISO 12207:1995. (ГОСТ Р–1999). ИТ. Процессы жизненного цикла программных средств.
- 4 ISO 15504:1-9:1998. ТО. Оценка и аттестация зрелости процессов жизненного цикла программных средств. Ч. 1. Основные понятия и вводное руководство. Ч. 2. Эталонная модель процессов и их зрелости. Ч. 3. Проведение аттестации. Ч. 4. Руководство по проведению аттестации. Ч. 5. Модель аттестации и руководство по показателям. Ч. 6. Руководство по компетентности аттестаторов. Ч. 7. Руководство по применению при усовершенствовании процессов. Ч. 8. Руководство по применению при определении зрелости процессов поставщика. Ч. 9. Словарь.
- 5 ISO 15504:1-5:2003-2006. ИТ. Процесс аттестации. Ч. 1. Концепция и словарь. Ч. 2. Выполнение аттестации. Ч. 3. Руководство по производству аттестации. Ч. 4. Руководство пользователей для процессов усовершенствования и определения зрелости процессов. Ч. 5. Образец модели процессов аттестации.
- 6 ISO 9000:2000. (ГОСТ Р-2001). Система менеджмента (административного управления) качества. Основы и словарь.
- 7 ISO 9126:1991. (ГОСТ-1993). ИТ. Оценка программного продукта. Характеристики качества и руководство по их применению.
- 8 ISO 14598-1-6:1998-2000. Оценивание программного продукта. Ч. 1. Общий обзор. Ч. 2. Планирование и управление. Ч. 3. Процессы для разработчиков. Ч. 4. Процессы для покупателей. Ч. 5. Процессы для оценщиков. Ч.6. Документирование и оценивание модулей.
- 9 ISO 9126-1-4:2002. ИТ. ТО. Качество программных средств: Ч. 1. Модель качества. Ч. 2. Внешние метрики. Ч. 3. Внутренние метрики. Ч. 4. Метрики качества в использовании.

- 10 ISO 25000:2005. ТО. Руководство для применения новой серии стандартов по качеству программных средств на базе обобщения стандартов ISO 9126:1-4:2002 и ISO 14598:1-6:1998-2000.
- 11 ISO 15408-1-3:1999. (ГОСТ Р-2002). Методы и средства обеспечения безопасности. Критерии оценки безопасности информационных технологий. Ч. 1. Введение и общая модель. Ч. 2. Защита функциональных требований. Ч. 3. Защита требований к качеству.
- 12 ISO 12119:1994. (ГОСТ Р-2000). ИТ. Требования к качеству и тестирование.
- 13 ISO 14764:1999. (ГОСТ Р-2002). ИТ. Сопровождение программных средств.
- 14 ISO 15846:1998. ТО. Процессы жизненного цикла программных средств. Конфигурационное управление программными средствами.
- 15 ISO 16085:2004. Характеристики процессов управления рисками при разработке, применении и сопровождении программных средств.
- 16 ISO 6592:2000. ОН. Руководство по документации для вычислительных систем.
- 17 ISO 9294:1990. (ГОСТ-1993). ТО. ИТ. Руководство по управлению документированием программного обеспечения.
- 18 ISO 9127:1990. (ГОСТ-1993). ТО. ИТ. Руководство по управлению документированием программного обеспечения.
- 19 ISO 15910:1999. (ГОСТ Р-2002). ИТ. Пользовательская документация программных средств.
- 20 ISO 18019:2004. ИТ. Руководство по разработке пользовательской документации на прикладные программные средства для офисов, бизнеса и профессиональных применений.
- 21 РД 50-34.698-90. Методические указания. Информационная технология. Автоматизированные системы. Требования к содержанию документов.
- 22 ISO 14102:1995. Оценка и выбор CASE-средств.
- 23 ГОСТ Р ИСО/МЭК 25010-2015 Информационные технологии (ИТ). Системная и программная инженерия. Требования и оценка качества систем и программного обеспечения (SQuaRE). Модели качества систем и программных продуктов.

**Задания  
на выполнение учебных проектов**

Тематическая категория №1: «*Торгово-складской учет*»

**Вариант №1.1 «Интернет-магазин»**

*Пользователи:*

- клиенты («гость», «покупатель», «постоянный клиент»);
- кладовщики;
- сотрудники отдела продаж, службы доставки и отдела маркетинга.

*Основные бизнес-процессы:*

- прайс-лист (обновление и публикация; просмотр и поиск товара);
- управление заказами (дисконтные карты; формирование заказа; доставка товара покупателям; мониторинг и контроль исполнения заказов);
- складской учет (регистрация поставки и отгрузки товаров);
- анализ качества исполнения заказов;
- анализ объемов продаж (по категориям товаров; по сотрудникам).

**Вариант №1.2 «Магазин по продаже компьютерной и оргтехники»**

*Пользователи:*

- клиенты («гость», «покупатель», «постоянный клиент»);
- кладовщики;
- сотрудники отделов продаж, маркетинга и доставки.

*Основные бизнес-процессы:*

- прайс-лист (обновление и публикация; просмотр и поиск товара);
- управление заказами (формирование заказа; подбор комплектующих с учетом совместимости);
- складской учет (регистрация поставки и отгрузки товаров);
- анализ объемов продаж (по категориям товаров; по сотрудникам).

**Вариант №1.3 «Автосалон»**

*Пользователи:*

- клиенты («гость», «покупатель», «постоянный клиент»);
- сотрудники отдела продаж;
- сотрудники отдела маркетинга.

*Основные бизнес-процессы:*

- прайс-лист (обновление и публикация; поиск автомобиля по маркам, моделям, ценам);
- управление продажами (продажи автомобилей с пробегом; подбор комплектации и продажа новых автомобилей; оформление договоров с клиентами);
- Trade IN (продажа нового автомобиля с одновременной покупкой автомобиля клиента).

Тематическая категория №2: **«Управление процессами»**

**Вариант №2.1 «Управление проектами»**

*Пользователи:*

- руководитель;
- руководители отделов;
- менеджеры проектов;
- исполнители.

*Основные бизнес-процессы:*

- кадровый учет (штатное расписание отделов);
- формирование проекта (структуризация проекта; разработка графика выполнения работ);
- распределение работ (закрепление заданий проектов за отделами; анализ загруженности исполнителей; распределение работ между исполнителями; формирование индивидуальных планов работы исполнителей);
- мониторинг и контроль исполнения (визуализация планов и графиков; регистрация выполнения работ; система напоминаний);
- формирование аналитической и отчетной документации.

**Вариант №2.2 «Интернет-провайдер»**

*Пользователи:*

- клиенты;
- руководитель;
- сотрудники call-центра и службы технической поддержки.

*Основные бизнес-процессы:*

- прайс-лист (перечень услуг; тарифные планы; акции и скидки);

- кадровый учет (штатное расписание call-центра и службы технической поддержки);
- просмотр информации о клиентах (адреса, заключенные договоры, установленное оборудование);
- управление договорами (заключение, приостановка и расторжение договоров; изменение перечня услуг и тарифных планов);
- управление работами (прием заявок от клиентов; просмотр списка заявок; распределение работ между специалистами);
- мониторинг и контроль исполнения (визуализация планов и графиков; регистрация выполнения работ; система напоминаний исполнителям);
- формирование аналитической и отчетной документации.

### **Вариант №2.3 «Компьютер-сервис»**

*Пользователи:*

- клиенты;
- руководитель;
- сотрудники (приемщик заявок, мастер по техническому обслуживанию и ремонту, специалист по программному обеспечению).

*Основные бизнес-процессы:*

- прайс-лист (перечень услуг);
- кадровый учет;
- просмотр информации о клиентах (адреса, история выполненных работ);
- управление работами (прием заявок от клиентов; прием/доставка оборудования; распределение работ между специалистами);
- мониторинг и контроль исполнения (визуализация планов и графиков; регистрация выполнения работ; система напоминаний исполнителям);
- формирование аналитической и отчетной документации.

Тематическая категория №3: **«Библиотечные системы»**

### **Вариант №3.1 «Абонемент публичной библиотеки»**

*Пользователи:*

- клиенты (гость, зарегистрированный читатель);
- сотрудники (руководитель; библиотекарь; менеджер).

*Основные бизнес-процессы:*

- регистрация читателей;
- управление библиотечным фондом (регистрация поступлений и списаний экземпляров книг; формирование и оперативное обновление тематического каталога);
- поиск книг (по названию, автору и году издания; по тематическому каталогу; по типу носителя);
- резервирование книг по заявкам читателей;
- регистрация выдачи/возврата книг;
- анализ читательской популярности книг (по тематическим категориям; по авторам книг; по категориям читателей).

### **Вариант №3.2 «Читальный зал периодических изданий»**

*Пользователи:*

- клиенты (гость, зарегистрированный читатель);
- сотрудники (руководитель; библиотекарь; менеджер).

*Основные бизнес-процессы:*

- регистрация читателей;
- управление библиотечным фондом (регистрация поступлений и списаний экземпляров выпусков периодических изданий; формирование и оперативное обновление тематического каталога);
- поиск (по тематическому каталогу; по названию, году и номеру выпуска; по авторам и названиям статей);
- предварительный просмотр (оглавлений выпусков журнала, аннотаций опубликованных статей);
- регистрация выдачи/возврата.

### **Вариант №3.3 «Видеотека»**

*Пользователи:*

- клиенты (гость, зарегистрированный пользователь);
- сотрудники (менеджер, администратор сервиса).

*Основные бизнес-процессы:*

- регистрация пользователей;
- управление фондом видеотеки (регистрация поступлений; формирование и оперативное обновление тематического каталога);



- поиск и предварительный просмотр (по категориям видеопродукции; по названию и году выпуска; по авторам и исполнителям; по тематическому каталогу; по типу носителя и формату записи);
- просмотр и скачивание файлов (истории пользователей; платежи; выдача прав доступа к файлам);
- формирование аналитической и отчетной документации (финансовая отчетность; анализ популярности видеопродуктов по тематическим категориям; по авторам и исполнителям; по категориям пользователей).

Тематическая категория №4: **«Физкультура, спорт и здоровье»**

#### **Вариант №4.1 «Командные спортивные соревнования»**

(вид спорта – по выбору разработчиков)

*Пользователи:*

- клиенты (гость, зарегистрированный болельщик, спортивный аналитик);
- сотрудники (менеджер; администратор сервиса).

*Основные бизнес-процессы:*

- регистрация пользователей;
- управление спортивной лигой (регистрация спортивных команд; регистрация спортивных арен; формирование структуры чемпионата);
- планирование соревнований (формирование графика проведения матчей; использование спортивных арен);
- оперативный учет результатов матчей (индивидуальные результаты игроков; командные результаты; формирование текущих рейтингов команд и игроков);
- аналитическая и отчетная документация (история рейтингов команд и игроков по сезонам).

#### **Вариант №4.2 «Физкультурно-оздоровительный комплекс»**

*Пользователи:*

- клиенты (гость, зарегистрированный клиент, постоянный клиент);
- сотрудники (менеджер по работе с персоналом; менеджер по работе с клиентами).

*Основные бизнес-процессы:*

- прайс-лист (формирование; просмотр списков секций и групп клиентов);

- кадровый учет (прием/увольнение тренерского состава; распределение тренеров по группам клиентов);
- регистрация клиентов (запись в группы; контроль посещений занятий; платежи);
- расписание занятий групп (анализ загруженности спортивных залов; формирование и просмотр расписаний проведения групповых занятий);
- аналитическая и отчетная документация (финансовая отчетность; популярность услуг по категориям клиентов).

### **Вариант №4.3 «Регистратура поликлиники»**

#### *Пользователи:*

- клиенты (гость, пациент);
- сотрудники (регистратор; врач; сотрудник отдела медицинской статистики).

#### *Основные бизнес-процессы:*

- кадровый учет (структура поликлиники; штатное расписание; прием/увольнение/перевод персонала);
- регистрация пациентов (создание амбулаторной карты; хронические заболевания);
- расписание работы (участковых врачей и фельдшеров; врачей-специалистов; лабораторий и процедурных кабинетов);
- оперативный учет приема пациентов (запись на прием к врачам; результаты лабораторных анализов; услуги процедурных кабинетов);
- аналитическая и отчетная документация (объемы оказания медицинских услуг; аналитические отчеты (за период времени) по группам заболеваний, по полу и возрасту пациентов).

### Тематическая категория №5: «**Управление образованием**»

### **Вариант №5.1 «Электронный дневник школьника»**

#### *Пользователи:*

- клиенты (ученики; родители или официальные представители учеников);
- сотрудники (завуч; учителя; классные руководители; администратор сервиса).

#### *Основные бизнес-процессы:*

- управление кадрами (состав классов по параллелям; зачисление/перевод/отчисление учащихся; прием/увольнение учителей; закрепление за классами учителей-предметников и классных руководителей);
- планирование учебного процесса (тематические планы учебных предметов; планы самостоятельных и контрольных работ);
- оперативный учет успеваемости учащихся (текущие оценки по темам предметов; оценки за самостоятельные и контрольные работы; итоговые оценки за учебный период);
- планирование и учет внеучебной работы (олимпиады по предметам; научно-технические конкурсы; творческие конкурсы; прочие мероприятия);
- аналитическая и отчетная документация (рейтинговые списки учащихся по классам, предметам, параллелям по учебной и внеучебной работе);
- аналитические отчеты за учебный период;
- архивирование и резервное копирование.

## **Вариант №5.2 «Расписание учебных занятий»**

### *Пользователи:*

- гости;
- студенты;
- сотрудники (диспетчеры; заведующие кафедрами; преподаватели, администратор сервиса).

### *Основные бизнес-процессы:*

- управление справочниками (факультеты и кафедры; специальности, формы обучения и образовательные уровни; корпуса и аудитории);
- управление кадрами (профессорско-преподавательский состав; зачисление/перевод/отчисление студентов);
- планирование учебного процесса (учебные планы специальностей; дисциплины и специализированные аудитории кафедр; составление и корректировка расписаний проведения учебных занятий);
- публикация расписаний на Web-ресурсе;
- поиск и просмотр расписаний (по группам, преподавателям, аудиториям);

- поиск групп и преподавателей по времени проведения учебных занятий;
- поиск свободных аудиторий;
- архивирование и резервное копирование.

### **Вариант №5.3 «Центр повышения квалификации специалистов»**

#### *Пользователи:*

- гости;
- слушатели;
- сотрудники (руководитель; менеджеры образовательных программ; преподаватели; администратор сервиса).

#### *Основные бизнес-процессы:*

- управление образовательными программами (категории программ; специальности, формы, сроки и стоимость обучения; формирование учебно-тематических планов; учебно-методическое обеспечение; формирование расписания групповых занятий);
- публикация на Web-ресурсе Центра;
- управление кадрами и бухгалтерский учет (профессорско-преподавательский состав; формирование групп слушателей; прием платежей от слушателей; оплата труда сотрудников);
- оперативный учет результатов обучения (контроль посещения учебных занятий; контроль выполнения контрольных заданий; регистрация результатов сдачи зачетов и экзаменов; регистрация выдачи документов (сертификатов, дипломов и пр.);
- архивирование и резервное копирование.

Учебное издание

Волк Владимир Константинович

**ВВЕДЕНИЕ**  
**В ПРОГРАММНУЮ ИНЖЕНЕРИЮ**  
Учебное пособие

Редактор Л.П. Чукомина

---

Подписано в печать 28.05.18  
Печать цифровая  
Заказ № 105

Формат 60×84 1/16  
Усл. печ. л. 9,75  
Тираж 100

Бумага 80 г/м<sup>2</sup>  
Уч.-изд. л. 9,75

---

Библиотечно-издательский центр КГУ.  
640020, г. Курган, ул. Советская, 63/4.  
Курганский государственный университет.