МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ
УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

СЕВЕРО-КАВКАЗСКАЯ ГОСУДАРСТВЕННАЯ АКАДЕМИЯ

З.Н. Чагарова

ТЕХНОЛОГИЯ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

Практикум для III-IV курса, обучающихся по специальности
09.02.07 «Информационные системы и программирование»

Черкесск
2025

УДК 004.4
ББК 32.973-4
Ч 12

Рассмотрено на заседании ЦК «Информационные и естественнонаучные дисциплины».

Протокол № 1 от «02» 09. 2024 г.

Рекомендовано к изданию редакционно-издательским советом СКГА

Протокол № 27 от «07» 11. 2024 г.

Ч 12 **Чагарова, З.Н.** Технология разработки программного обеспечения: практикум для III-IV курса, обучающихся по специальности 09.02.07 «Информационные системы и программирование» / З.Н. Чагарова. – Черкесск: БИЦ СКГА, 2025. – 112 с.

По выполнению лабораторных и практических работ по МДК 02.01 Технология разработки программного обеспечения разработаны в соответствии с рабочей программой профессионального модуля и предназначены для приобретения необходимых практических навыков и закрепления теоретических знаний, полученных обучающимися при изучении профессионального модуля, обобщения и систематизации знаний перед экзаменом.

УДК 004.4
ББК 32.973-4

СОДЕРЖАНИЕ

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

Методические рекомендации по выполнению лабораторных и практических работ по дисциплине "Технология разработки программного обеспечения" разработаны в соответствии с учебной программой профессионального модуля. Их цель – обеспечить приобретение необходимых практических навыков и закрепление теоретических знаний, полученных студентами во время изучения данного модуля. Кроме того, эти рекомендации помогают студентам обобщить и систематизировать свои знания перед итоговым экзаменом.

Эти методические рекомендации предназначены для студентов, обучающихся по специальности 09.02.07 "Информационные системы и программирование".

Согласно образовательной программе ПМ.02. Осуществление интеграции программных модулей, практические занятия по курсу "Технология разработки программного обеспечения" распределены между шестым и седьмым семестрами. Это способствует обобщению и систематизации знаний перед экзаменом.

Порядок проведения практической работы

1. Запишите название и цель задания в своей тетради.
2. Выполните основные задачи согласно предоставленным инструкциям.
3. Решите индивидуальные задания.

Рекомендации по оформлению практической работы

Все задания должны выполняться последовательно. Необходимо строго следовать порядку шагов, изложенных в инструкции к практическим работам.

Студентам рекомендуется сохранять результаты выполненных практических работ в специально отведенной личной папке на компьютере или внешних устройствах хранения данных, таких как USB-накопители.

В случае пропуска занятий студенты обязаны самостоятельно изучить материал в свободное время и представить практическую работу вместе с подробными объяснениями её выполнения.

Критерии оценки практической работы:

- Наличие цели работы и выполнение более половины основных заданий (оценка "удовлетворительно").
- Наличие цели работы, выполнение всех основных заданий и более половины дополнительных заданий (оценка "хорошо").
- Наличие цели работы, выполнение всех основных и индивидуальных заданий (оценка "отлично").

Практическое задание №1 «Анализ предметной области»

Цель задания: описать и проанализировать информационную систему (ИС), выделить ключевые компоненты технической и системной инфраструктуры ИС.

Теоретические сведения

Впервые проблемы управления программными проектами возникли в 1960-е – начале 1970-х годов, когда потерпело неудачу множество крупных разработок ПО. Проекты задерживались, программы были ненадёжными, а затраты превышали запланированные в разы. Эти неудачи объяснялись не только недостаточной квалификацией менеджеров и разработчиков. Многие участники проектов имели высокий профессиональный уровень. Однако применяемые методы управления оказались неподходящими для разработки ПО, поскольку основывались на принципах управления техническими проектами.

Руководство программными проектами аналогично управлению техническими проектами, однако разработка ПО существенно отличается от реализации инженерных решений. Основные различия включают:

1. Нематериальность программного продукта. Руководителю строительства видна степень готовности объекта, тогда как состояние разработки ПО оценить сложнее – приходится полагаться на документацию.
2. Отсутствие универсальных стандартов разработки. Процесс создания ПО ещё недостаточно изучен, и чётких методик пока не выработано. Это делает невозможным точное предсказание проблем на различных этапах разработки.
3. Уникальность крупных проектов. Большинство крупных программных проектов уникальны и требуют значительных усилий для минимизации рисков. Постоянные изменения в технологиях делают предыдущий опыт менее значимым.

Управление программными проектами

Несмотря на отсутствие единой методики, основными обязанностями менеджера проекта являются:

- Разработка предложений по созданию ПО.
- Составление плана и графика работ.
- Оценка затрат.
- Контроль над выполнением задач.
- Формирование команды.
- Подготовка отчётов.

Проект может длиться годами, и за это время приоритеты компании могут измениться. Тогда руководство может прекратить разработку или изменить её направление.

Планирование проекта

Правильное планирование — основа успешного завершения проекта. Начальный план включает детальное описание всех этапов, необходимых ресурсов и сроков выполнения. Обычно план содержит такие разделы:

1. Цели и ограничения. Краткий обзор целей проекта и существующих

ограничений (финансовых, временных).

2. Организационная структура. Методы подбора команды и распределения ролей.

3. Риски. Анализ потенциальных угроз и стратегии их снижения.

4. Технические ресурсы. Необходимое оборудование и ПО.

5. Этапы работ. Подробное деление проекта на части с указанием ожидаемых результатов.

6. График выполнения. Определение зависимостей между этапами и времени на их реализацию.

7. Мониторинг. Механизмы отслеживания прогресса и отчетности.

Для повышения эффективности управления проектом устанавливаются контрольные точки (вехи). По завершении каждого этапа составляется отчёт.

Например, на рис. 1 показаны этапы разработки спецификации требований в случае, когда для ее проверки используется прототип системы.

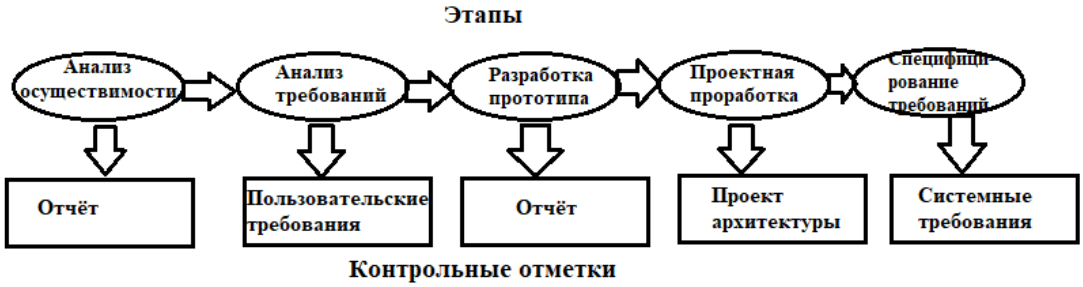


Рисунок 1 – Этапы процесса разработки спецификации

Информация и информационные системы

Информационная система (ИС) – это комплекс методов и инструментов, обеспечивающий полный цикл обработки информации: приём, хранение, передачу и использование. Вне ИС информация остаётся статичной и бесполезной.

ИС состоит из нескольких компонентов:

- Функциональная структура.
- Математическое, техническое и кадровое обеспечение.

Основные задачи корпоративных ИС:

- Улучшить качество управления предприятием.
- Повысить взаимодействие подразделений.
- Гарантировать высокое качество продукции.
- Увеличить экономическую эффективность.
- Создать систему учёта и прогнозирования.

Таким образом, ИС играет ключевую роль в управлении современными предприятиями, обеспечивая автоматизацию процессов и повышение эффективности работы.

Задания для практической работы

1. Выберите предметную область
2. Выберите название ИС в рамках предметной области.
3. Определите цель ИС

4. Проведите анализ осуществимости ИС

4.1.Что произойдет с организацией, если система не будет введена в эксплуатацию?

4.2.Какие текущие проблемы существуют в организации и как новая система поможет их решить?

4.3.Каким образом (и будет ли) ИС способствовать целям бизнеса?

4.4.Требует ли разработка ИС технологии, которая до этого раньше не использовалась в организации?

5. Где будет размещена ИС? Кто является пользователем ИС?

6. Комплекс технических средств ИТ

6.1.Какие средства компьютерной техники необходимы для ИС?

6.2.Какие средства коммуникационной техники необходимы для ИС?

6.3.Какие средства организационной техники необходимы для ИС?

6.4.Какие средства оперативной полиграфии необходимы для ИС?

7. Опишите системное ПО ИТ.

Таблица 1. Варианты предметных областей

Предметная область	Сущность задачи
Страховая медицинская компания	Страховая медицинская компания (СМК) заключает договора добровольного медицинского страхования с физическими лицами и соглашения с медицинскими учреждениями для обслуживания застрахованных клиентов. При наступлении страхового случая клиент обращается к своему инспектору, который организует направление пациента в медицинское учреждение. Инспекторы предоставляют отчеты о своей работе в бухгалтерию компании. Бухгалтерия контролирует правильность оплаты договоров, осуществляет платежи медицинским учреждениям за предоставленные услуги, производит необходимые налоговые отчисления и передает отчетность в государственные статистические органы. Помимо покрытия расходов на лечение застрахованных лиц, СМК также берет на себя расходы по лечению возможных осложнений, возникших после основного курса терапии.
Агентство недвижимости	Агентство недвижимости занимается операциями купли-продажи и арендой объектов недвижимости на основе договоров с владельцами. Оно работает с недвижимостью как частных лиц, так и организаций. Владельцы могут располагать несколькими объектами недвижимости одновременно. Потенциальные покупатели или арендаторы имеют право осмотреть объекты перед сделкой. Среди услуг агентства — оценка состояния объекта для определения его рыночной стоимости. По итогам своей работы агентство выплачивает налоги и предоставляет отчетность в соответствующие государственные органы.
Кадровое агентство	Кадровое агентство помогает гражданам найти работу, предлагая вакансии от различных компаний. Данные о безработных гражданах собираются на основе резюме, предоставляемых ими. Предприятия города передают информацию о наличии вакансий, на основе которой агентство формирует предложения для кандидатов. Если кандидат успешно устраивается на предложенную должность, вакансия считается закрытой, а гражданин — трудоустроенным. Кадровое агентство платит налоги и представляет отчетность государственным органам статистики.

Компания по разработке программных продуктов	Компания разрабатывает программное обеспечение по заказу клиентов на основе технических заданий. Утвержденное техническое задание служит основой для планирования объема работ и составления предварительной сметы. Каждый проект курирует назначенный руководитель, который распределяет задачи среди программистов и контролирует соответствие результата требованиям заказчика. Готовый продукт внедряется, клиенту предоставляется обучение, а также осуществляется последующее обслуживание. Компания делает обязательные налоговые выплаты и сдает необходимую отчетность в госорганы.
Туроператор	Туроператоры предлагают своим клиентам организовать туристические или деловые поездки в разные регионы России и зарубежья. Для разработки нового тура проводится анализ рынка туристических направлений, выбираются маршруты. Затем определяются условия тура, бронируются гостиницы и транспорт, формируется культурная или развлекательная программа, устанавливаются даты проведения. За каждым туром закрепляется ответственный сотрудник туроператора, который решает возможные проблемы во время путешествия. Клиенты выбирают туры в офисе оператора и приобретают путевки. После завершения тура клиенты могут оставить отзывы, которые учитываются при улучшении текущих туров или создании новых. Туроператор анализирует отчеты партнеров (например, отелей и гидов), чтобы улучшить качество услуг. Как и другие организации, туроператор совершает налоговые отчисления и сдает отчетность в органы государственной статистики.

Практическая работа №2 «Разработка и оформление технического задания»

Цель работы: приобретение навыков разработки технического задания на программный продукт, ознакомиться с правилами написания технического задания

Теоретические сведения

Техническое задание (ТЗ, техзадание) — это исходный документ, который используется для проектирования различных объектов, таких как сооружения, промышленные комплексы, технические устройства (например, приборы, машины, системы управления), информационные системы, стандарты, а также для проведения научно-исследовательских работ (НИР).

В техническом задании содержатся ключевые технические требования, которые предъявляются к проектируемым объектам, изделиям или услугам, а также исходные данные, необходимые для дальнейшей разработки. ТЗ включает описание назначения объекта, области его применения, этапов разработки проектной, конструкторской, технологической, программной и другой документации, её состава, сроков выполнения работ, а также особых требований, связанных с особенностями самого объекта или условиями его эксплуатации.

Обычно техническое задание составляется на основании проведённого предварительного анализа, включая результаты исследовательской работы, расчетов и моделирования.

Типовые требования к составу и содержанию технического задания приведены в таблице 1.

Таблица 1. Состав и содержание технического задания (ГОСТ 34.602- 89)

№ пп	Раздел	- Содержание
1	Общие сведения	- полное наименование системы и ее условное обозначение - шифр темы или шифр (номер) договора; - наименование предприятий разработчика и заказчика системы, их реквизиты - перечень документов, на основании которых создается ИС - плановые сроки начала и окончания работ - сведения об источниках и порядке финансирования работ - порядок оформления и предъявления заказчику - результатов работ по созданию системы, ее частей и отдельных средств
2	Назначение и цели создания (развития) системы	- вид автоматизируемой деятельности - перечень объектов, на которых предполагается использование системы - наименования и требуемые значения технических, технологических, производственно-экономических и др. показателей объекта, которые должны быть достигнуты при - внедрении ИС

3	Характеристика объектов автоматизации	краткие сведения об объекте автоматизации - сведения об условиях эксплуатации и характеристиках окружающей среды
4	Требования к системе	- Требования к системе в целом: - требования к структуре и функционированию системы (перечень подсистем, уровни иерархии, степень централизации, способы информационного обмена, режимы функционирования, взаимодействие со смежными системами, перспективы развития системы) - требования к персоналу(численность пользователей, квалификация, режим работы, порядок подготовки) - показатели назначения (степень приспособляемости системы к изменениям процессов управления и значений параметров) - требования к надежности, безопасности, эргономике, транспортабельности, эксплуатации, техническому обслуживанию и ремонту, защите и сохранности - информации, защите от внешних воздействий, к патентной чистоте, по стандартизации и унификации - Требования к функциям (по подсистемам) : - перечень подлежащих автоматизации задач - временной регламент реализации каждой функции - требования к качеству реализации каждой функции, к форме представления выходной информации, характеристики точности, достоверности выдачи результатов - перечень и критерии отказов Требования к видам обеспечения: - математическому (состав и область применения мат. моделей и методов, типовых и разрабатываемых алгоритмов) - информационному (состав, структура и организация данных, обмен данными между компонентами системы, информационная совместимость со смежными системами, используемые классификаторы, СУБД, контроль данных и ведение информационных массивов, процедуры придания юридической силы выходным документам) - лингвистическому (языки программирования, языки взаимодействия пользователей с системой, системы кодирования, языки ввода- вывода) - программному (независимость программных средств от платформы, качество программных средств и способы его контроля, использование фондов алгоритмов и программ) - техническому - метрологическому - организационному (структура и функции эксплуатирующих подразделений, защита от ошибочных действий персонала) методическому(состав нормативно-технической документации

5	Состав и содержание работ по созданию системы	- перечень стадий и этапов работ - сроки исполнения - состав организаций – исполнителей работ - вид и порядок экспертизы технической документации - программа обеспечения надежности - программа метрологического обеспечения
6	Порядок контроля и приемки системы	- виды, состав, объем и методы испытаний системы - общие требования к приемке работ по стадиям - статус приемной комиссии
7	Требования к составу и содержанию работ по подготовке объекта автоматизации к вводу системы в действие	- преобразование входной информации к машиночитаемому виду - изменения в объекте автоматизации - сроки и порядок комплектования и обучения персонала
8	Требования к документированию	- перечень подлежащих разработке документов - перечень документов на машинных носителях
9	Источники разработки	- документы и информационные материалы, на основании которых разрабатывается ТЗ и система

Порядок разработки технического задания

Разработка технического задания выполняется в следующей последовательности. Прежде всего, устанавливают набор выполняемых функций, а также перечень и характеристики исходных данных.

Затем определяют перечень результатов, их характеристики и способы представления.

Далее уточняют среду функционирования программною обеспечения: конкретную комплектацию и параметры технических средств.

В случаях, когда разрабатываемое программное обеспеченно собирает и хранит некоторую информацию или включается в управление каким-либо техническим процессом, необходимо также четко регламентировать действия программы в случае сбоев оборудования и энергоснабжения.

1. Общие положения

1.1 Техническое задание оформляют в соответствии с ГОСТ 19.106-78 на листах формата А4 и А3 по ГОСТ 2.301-68, как правило, без заполнения полей листа. Номера листов (страниц) проставляют в верхней части листа над текстом.

1.2 Лист утверждения и титульный лист оформляют в соответствии с ГОСТ 19.104-78. Информационную часть (аннотацию и содержание), лист регистрации изменений допускается и в документ не включать.

1.3 Для внесения изменений и дополнений в техническое задание на последующих стадиях разработки программы или программного изделия выпускают дополнение к нему. Согласование и утверждение дополнения к техническому заданию проводят в том же порядке, который установлен для технического задания.

1.4. Техническое задание должно содержать следующие разделы:

- введение;
- наименование и область применения;
- основание для разработки;
- назначение разработки;
- технические требования к программе или программному изделию;
- технико-экономические показатели;
- стадии и этапы разработки;
- порядок контроля и приемки;
- приложения.

В зависимости от особенностей программы или программного изделия допускается уточнять содержание разделов, вводить новые разделы или объединять отдельные из них. При необходимости допускается в техническое задание включать приложения.

2. Содержание разделов

2.1 *Введение* должно включать краткую характеристику области применения программы или программного продукта, а также объекта, в котором предполагается их использовать. Основное назначение введения – продемонстрировать актуальность данной разработки и показать, какое место эта разработка занимает в ряду подобных.

2.2 В разделе *«Наименование и область применения»* указывают наименование, краткую характеристику области применения программы или программного изделия и объекта, в котором используют программу или программное изделие.

2.3 В разделе *«Основание для разработки»* должны быть указаны:

- документ (документы), на основании которых ведется разработка. Таким документом может служить план, приказ, договор и т.п.;
- организация, утвердившая этот документ, и дата его утверждения;
- наименование и (или) условное обозначение темы разработки.

2.4 В разделе *«Назначение разработки»* должно быть указано функциональное и эксплуатационное назначение программы или программного изделия.

2.5 Раздел *«Технические требования к программе или программному изделию»* должен содержать следующие подразделы:

- требования к функциональным характеристикам;
- требования к надежности;
- условия эксплуатации;
- требования к составу и параметрам технических средств;
- требования к информационной и программной совместимости;
- требования к маркировке и упаковке;
- требования к транспортированию и хранению;
- специальные требования.

В подразделе *«Требования к функциональным характеристикам»* должны быть указаны требования к составу выполняемых функций, организации входных и выходных данных, временным характеристикам и т. п.

2.5.2 В подразделе *«Требования к надежности»* должны быть указаны требования к обеспечению надежного функционирования (обеспечение устойчивого функционирования, контроль входной и выходной информации, время восстановления после отказа и т. п.).

2.5.3 В подразделе *«Условия эксплуатации»* должны быть указаны условия эксплуатации, при которых должны обеспечиваться заданные характеристики, а также вид обслуживания, необходимое количество и квалификация персонала.

2.5.4В подразделе *«Требования к составу и параметрам технических средств»* указывают необходимый состав технических средств с указанием их технических характеристик.

2.5.5 В подразделе *«Требования к информационной и программной совместимости»* должны быть указаны требования к информационным структурам на входе и выходе и методам решения, исходным кодам, языкам программирования. При необходимости должна обеспечиваться защита информации и программ.

2.5.6 В подразделе *«Требования к маркировке и упаковке»* в общем случае указывают требования к маркировке программного изделия, варианты и способы упаковки.

2.5.7 В подразделе *«Требования к транспортированию и хранению»* должны быть указаны для программного изделия условия транспортирования, места хранения, условия хранения, условия складирования, сроки хранения в различных условиях.

2.5.8 В разделе *«Технико-экономические показатели»* должны быть указаны: ориентировочная экономическая эффективность, предполагаемая годовая потребность, экономические преимущества разработки по сравнению с лучшими отечественными и зарубежными образцами или аналогами.

2.6 В разделе *«Стадии и этапы разработки»* устанавливают необходимые стадии разработки, этапы и содержание работ (перечень программных документов, которые должны быть разработаны, согласованы и утверждены).

2.7 В разделе *«Порядок контроля и приемки»* должны быть указаны виды испытаний и общие требования к приемке работы.

2.8 В приложениях к техническому заданию при необходимости приводят:

- Перечень научно-исследовательских и других работ, обосновывающих разработку;
- схемы алгоритмов, таблицы, описания, обоснования, расчеты и другие документы, которые могут быть использованы при разработке;
- другие источники разработки.

В случаях, если какие-либо требования, предусмотренные техническим заданием, заказчик не предъявляет, следует в соответствующем месте указать «Требования не предъявляются».

Задания для практической работы

1. Разработать техническое задание по варианту выбранному в практической работе №1
2. Оформить отчет

Порядок выполнения отчета по практической работе

1. Разработать техническое задание на программный продукт
2. Оформить работу в соответствии с ГОСТ 19.106-78.

При оформлении использовать MS Office.

3. Сдать и защитить работу

Практическое занятие №3 «Построение архитектуры программного продукта»

Цель занятия:

Целью данного практического занятия является развитие навыков студентов в построении формальных моделей программного обеспечения и определении спецификаций на их основе. Также важно освоить принципы проектирования программного обеспечения, чтобы иметь возможность создавать качественные продукты, соответствующие требованиям заказчика.

Теоретические сведения

Этап эскизного проектирования

На этапе эскизного проектирования создаются предварительные проектные решения для всей системы и её отдельных компонентов. Данный процесс включает в себя несколько важных шагов:

1. Определение ключевых функций системы: Необходимо чётко определить основные функции, которые должна выполнять система.
2. Анализ целей и эффектов внедрения подсистем: Важно проанализировать цели каждой подсистемы и возможные эффекты от их внедрения.
3. Разделение задач на комплексы и отдельные элементы: Следует разделить общие задачи на более мелкие части для упрощения реализации.
4. Концептуализация информационной базы и её структуры: Определяются данные, которые будут храниться в системе, и способы их организации.
5. Планирование системы управления базой данных: Выбирается подходящая СУБД и определяются механизмы взаимодействия с ней.
6. Выбор вычислительного оборудования и других технических средств: Подбираются необходимые аппаратные ресурсы для эффективной работы системы.
7. Определение характеристик основного программного обеспечения: Устанавливаются требования к программному обеспечению, которое будет использоваться в системе.

Результаты эскизного проектирования

По завершении этапа эскизного проектирования создаётся документация, содержащая все принятые решения. Эта документация служит основой для дальнейших работ над проектом. Однако этот этап можно пропустить, если ключевые решения были приняты заранее.

Создание эскизного проекта программы

Основные этапы

Эскизный проект помогает создать общее представление о будущей программе. В него входят следующие разделы:

1. Общие сведения о проекте:

Краткое описание целей и задач проекта.

2. Пояснительная записка:

Подробное изложение идей и концепций, лежащих в основе разработки.

3. Организационная структура компании-потребителя:

Описание организационной структуры компании-заказчика, чтобы учесть особенности её функционирования.

4. Структурная схема технических средств:

Графическое изображение используемых технических ресурсов и их взаимосвязей.

5. Функциональная схема работы системы:

Детализированное описание процессов и операций, выполняемых системой.

6. Схема автоматизации процесса:

Показано, какие процессы подлежат автоматизации и каким образом это будет реализовано.

Данный документ позволяет заказчику оценить реализуемость предложенной идеи и её потенциальные выгоды.

Разработка спецификаций

Спецификации описывают функциональные и технические характеристики разрабатываемого программного обеспечения. Они содержат:

1. Диаграммы потоков данных (DFD):

Представляют потоки данных внутри системы и между внешними объектами.

2. Диаграммы сущностей и связей (ERD):

Моделируют отношения между различными сущностями в базе данных.

3. Диаграммы перехода состояний (STD):

Отображают изменения состояния системы в зависимости от внешних воздействий.

4. Словари терминов:

Содержат определения специальных терминов, используемых в проекте.

5. Алгоритмы обработки данных:

Описывают пошаговые инструкции для выполнения определённых операций.

Эти инструменты способствуют лучшему пониманию работы системы и минимизации ошибок на ранних этапах проектирования.

Технический проект

Технический проект детализирует методы и технологии, применяемые при разработке системы. Он включает:

1. Описание выполняемых функций:

Подробное описание функциональности системы.

2. Характеристики документов:

Указание типов документов, обрабатываемых системой.

3. Информационные базы данных:

Описание структуры баз данных и хранимых данных.

4. Связи между данными:

Уточняются связи между элементами данных.

5. Программное обеспечение и оборудование:

Перечень необходимых программных и аппаратных средств.

Также в техническом проекте оцениваются показатели надёжности системы и приводится список требуемого оборудования.

При разработке технического проекта оформляются:

- ведомость технического проекта. Общая информация по проекту;
- пояснительная записка к техническому проекту. Вводная информация, позволяющая ее потребителю быстро освоить данные по конкретному проекту;
- описание систем классификации и кодирования;
- перечень входных данных. Перечень информации, которая используется как входящий поток и служит источником накопления;
- перечень выходных данных (документов). Перечень информации, которая используется для анализа накопленных данных;
- описание используемого программного обеспечения. Перечень программного обеспечения и СУБД, которые планируется использовать для создания информационной системы;
- описание используемых технических средств. Перечень аппаратных средств, на которых планируется работа проектируемого программного продукта;
- проектная оценка надежности системы. Экспертная оценки надежности с выявлением наиболее благополучных участков программной системы и ее узких мест;
- ведомость оборудования и материалов. Перечень оборудования и материалов, которые потребуются в ходе реализации проекта.

Структурная схема.

Структурной называют схему, отражающую состав и взаимодействие по управлению частями разрабатываемого программного обеспечения. Структурная схема определяется архитектурой разрабатываемого ПО.

Функциональная схема.

Функциональная схема – это схема взаимодействия компонентов программного обеспечения с описанием информационных потоков, состава данных в потоках и указанием используемых файлов и устройств.

Разработка алгоритмов.

Метод пошаговой детализации реализует нисходящий подход к программированию и предполагает пошаговую разработку алгоритма.

Структурные карты.

Методика структурных карт используется на этапе проектирования ПО для того, чтобы продемонстрировать, каким образом программный продукт выполняет системные требования. Структурные карты Константайна предназначены для описания отношений между модулями.

Техника структурных карт Джексона основана на методе структурного программирования Джексона, который выявляет соответствие между структурой потоков данных и структурой программы. Основное внимание в

методе сконцентрировано на соответствии входных и выходных потоков данных.

Задания для практической работы Задание №1

1. На основе технического задания из практической работы №2 выполнить анализ функциональных и эксплуатационных требований к программному продукту.
2. Определить основные технические решения и занести результаты в документ, называемый «Эскизным проектом».
3. Определить диаграммы потоков данных для решаемой задачи.
4. Определить диаграммы «сущность-связь», если программный продукт содержит базу данных.
5. Добавить словарь терминов.
6. Оформить результаты, используя MS Office или MS Visio в виде эскизного проекта.
7. Сдать и защитить работу.

Порядок выполнения отчета по практической работе.

Отчет по лабораторной работе должен состоять из:

1. Постановки задачи.
2. Документа «Эскизный проект», содержащего:
- выбор метода решения и языка программирования;
- спецификации процессов;
- все полученные диаграммы;
- словарь терминов.

Задание №2

1. Разработать функциональную схему программного продукта.
2. Представить структурную схему в виде структурных карт Константайна.
3. Представить структурную схему в виде структурных карт Джексона.
4. Оформить результаты, используя MS Office или MS Visio в виде технического проекта.
5. Сдать и защитить работу.

Порядок выполнения отчета по практической работе.

Отчет по практической работе должен состоять из:

1. Структурной схемы программного продукта.
2. Функциональной схемы.
3. Алгоритма программы.
4. Структурной карты Константайна.
5. Структурной карты Джексона.
6. Законченного технического проекта программного модуля.

Защита отчета по практической работе заключается в предъявлении преподавателю полученных результатов (на экране монитора), демонстрации полученных навыков и ответах на вопросы преподавателя.

Практическая работа №4
«Изучение работы в системе контроля версий»

Целью работы является изучение порядка работы с системой контроля версий GIT. Результатом практической работы является отчет, в котором должны быть приведено описание созданного репозитария, демонстрация приемов работы с ним.

Для выполнения лабораторной работы № 4 студент должен изучить приведенный ниже теоретический материал. Отчет сдается в распечатанном и электронном (файл Word) видах.

Изучение работы в системе контроля версий.

Git – это набор консольных утилит, которые отслеживают и фиксируют изменения в файлах.С его помощью можно откатиться на более старую версию вашего проекта, сравнивать, анализировать, сливать изменения и многое другое. Этот процесс называется контролем версий. Существуют различные системы для контроля версий. Вы, возможно, о них слышали: SVN, Mercurial, Perforce, CVS, Bitkeeperи другие.

Git является распределенным, то есть не зависит от одного центрального сервера, на котором хранятся файлы. Вместо этого он работает полностью локально, сохраняя данные в папках на жестком диске, которые называются репозиторием. Тем не менее, можно хранить копию репозитория онлайн, это сильно облегчает работу над одним проектом для нескольких людей. Для этого используются сайты вроде github и bitbucket.

– Установка.

Установить git на свою машину очень просто:

Windows – мы рекомендуем git for windows, так как он содержит и клиент с графическим интерфейсом, и эмулятор bash.

– Настройка.

После установки git, нужно добавить немного настроек. Есть довольно много опций, с которыми можно играть, но мы настроим самые важные: наше имя пользователя и адрес электронной почты. Откройте терминал и запустите команды:

git config --global user.name «My Name»

git config --global user.email myEmail@example.com

Теперь каждое действие будет отмечено именем и почтой. Таким образом, пользователи всегда будут в курсе, кто отвечает за какие изменения – это вносит порядок.

– Создание нового репозитория.

Как было отмечено ранее, git хранит свои файлы и историю прямо в папке проекта. Чтобы создать новый репозиторий, нужно открыть терминал, зайти в папку нашего проекта и выполнить команду init. Это включит приложение в этой конкретной папке и создаст скрытую директорию .git, где будет храниться история репозитория и настройки.

Создайте на рабочем столе папку под названием git_exercise.

Для этого в окне терминала введите:

$ mkdir Desktop/git_exercise/

$ cd Desktop/git_exercise/

$ git init

Командная строка должна вернуть что-то вроде:

Initialized empty Git repository in

/home/user/Desktop/git_exercise/.git/

Это значит, что репозиторий был успешно создан, но пока что пуст. Теперь создайте текстовый файл под названием hello.txt и сохраните его в директории git_exercise.

– Определение состояния.

status – это еще одна важнейшая команда, которая показывает информацию о текущем состоянии репозитория: актуальна ли информация на нём, нет ли чего-то нового, что поменялось, и так далее. Запуск git status на нашем свежесозданном репозитории долженвыдать:

$ git status

On branch masterInitial commit Untracked files:

(use «git add ...» to include in what will be committed)hello.txt

Сообщение говорит о том, что файл hello.txt неотслеживаемый. Это значит, что файл новый и система еще не знает, нужно ли следить за изменениями в файле или его можно просто игнорировать. Для того, чтобы начать отслеживать новый файл, нужно его специальным образом объявить.

– Подготовка файлов.

В git есть концепция области подготовленных файлов. Можно представить ее как холст, на который наносят изменения, которые нужны в коммите. Сперва он пустой, но затем мы добавляем на негофайлы (или части файлов, или даже одиночные строчки) командой add и, наконец, коммитим все нужное в репозиторий (создаем слепок нужного нам состояния) командой commit.

В нашем случае у нас только один файл, так что добавим его: $ git add hello.txt

Если нам нужно добавить все, что находится в директории, мы можем использовать $ git add -A

Проверим статус снова, на этот раз мы должны получить другой ответ: $ git status

On branch masterInitial commit

Changes to be committed: (use «git rm --cached ...» to unstage) new file: hello.txt

Файл готов к коммиту. Сообщение о состоянии также говорит нам о том, какие изменения относительно файла были проведены в области подготовки – в данном случае это новый файл, но файлы могут быть модифицированы или удалены.

– Коммит (фиксация изменений).

Коммит представляет собой состояние репозитория в определенный момент времени. Это похоже на снапшот, к которому мы можем вернуться и увидеть состояние объектов на определенный момент времени.

Чтобы зафиксировать изменения, нужно хотя бы одно изменение в области подготовки (мы только что создали его при помощи git add), после которого можго коммитить: $ git commit -m «Initial commit.»

Эта команда создаст новый коммит со всеми изменениями из области подготовки (добавление файла hello.txt). Ключ -m и сообщение «Initial commit.» – это созданное пользователем описание всех изменений, включенных в коммит. Считается хорошей практикой делать коммиты часто и всегда писать содержательные комментарии.

- Удаленные репозитории.

Сейчас коммит является локальным – существует только в директории .git на нашей файловой системе. Несмотря на то, что сам по себе локальный репозиторий полезен, в большинстве случаев нужно поделиться работой или доставить код на сервер, где он будет выполняться.

1. Подключение к удаленному репозиторию.

Чтобы загрузить что-нибудь в удаленный репозиторий, сначала нужно к нему подключиться. В нашем руководстве мы будем использовать адрес https://github.com/tutorialzine/awesome-project, но вам посоветуем попробовать создать свой репозиторий в GitHub, BitBucket или любом другом сервисе. Регистрация и установка может занять время, но все подобные сервисы предоставляют хорошую документацию.

Чтобы связать локальный репозиторий с репозиторием на GitHub, выполним следующую команду в терминале. Обратите внимание, что нужно обязательно изменить URI репозитория насвой.

This is only an example. Replace the URI with your own reposi- tory address $git remote add originhttps://github.com/tutorialzine/awesome-project.git

Проект может иметь несколько удаленных репозиториев одновременно. Чтобы их различать, дадим им разные имена. Обычно главный репозиторий называется origin.

2. Отправка изменений на сервер.

Сейчас самое время переслать локальный коммит на сервер. Этот процесс происходит каждый раз, когда нужно обновить данные в удаленном репозитории. Команда, предназначенная для этого – push. Она принимает два параметра: имя удаленного репозитория (мы назвали наш origin) и ветку, в которую необходимо внести изменения (master – это ветка по умолчанию для всех репозиториев).

$ git push origin master Counting objects: 3, done.

Writing objects: 100% (3/3), 212 bytes | 0 bytes/s, done. Total 3 (delta 0), reused 0 (delta 0)

To https://github.com/tutorialzine/awesome-project.git

* [new branch] master -> master

В зависимости от сервиса, который вы используете, вам может потребоваться аутентифицироваться, чтобы изменения отправились. Если все сделано правильно, то когда вы посмотрите в удаленный репозиторий при помощи браузера, вы увидите файл hello.txt

3. Клонирование репозитория.

Сейчас другие пользователи GitHub могут просматривать ваш репозиторий. Они могут скачать из него данные и получить полностью работоспособную копию вашего проекта при помощи команды clone.

$ git clone https://github.com/tutorialzine/awesome-project.git

Новый локальный репозиторий создается автоматически с GitHub в качестве удаленного репозитория.

4. Запрос изменений с сервера.

Если вы сделали изменения в вашем репозитории, другие пользователи могут скачать изменения при помощи команды pull.

$ git pull origin master

From https://github.com/tutorialzine/awesome-project

* branch master -> FETCH_HEADAlready up-to-date.

Так как новых коммитов с тех пор, как мы склонировали себе проект, не было, никаких изменений доступных для скачивания нет.

- Ветвление.

Во время разработки новой функциональности считается хорошей практикой работать с копией оригинального проекта, которую называют веткой. Ветви имеют свою собственную историю и изолированные друг от друга изменения до тех пор, пока вы не решаете слить изменения вместе. Это происходит по набору причин:

- Уже рабочая, стабильная версия кода сохраняется.
- Различные новые функции могут разрабатываться параллельно разными программистами.
- Разработчики могут работать с собственными ветками безриска, что кодовая база поменяется из-за чужих изменений.
- В случае сомнений, различные реализации одной и той же идеи могут быть разработаны в разных ветках и затем сравниваться.

1. Создание новой ветки.

Основная ветка в каждом репозитории называется master. Чтобы создать еще одну ветку, используем команду branch <name>

$ git branch amazing_new_feature

Это создаст новую ветку, пока что точную копию ветки master.

2. Переключение между ветками.

Сейчас, если запустить branch, мы увидим две доступные опции:

$ git branch amazing_new_feature

* master

master – это активная ветка, она помечена звездочкой. Но нужно работать с «новой потрясающей фичей», так что нужно переключиться на другую ветку. Для этого воспользуемся командой checkout, она принимает один параметр – имя ветки, на которую необходимо переключиться.

$ git checkout amazing_new_feature

3. Слияние веток.

Наша «потрясающая новая фича» будет еще одним текстовым файлом под названием feature.txt. Создадим его, добавим и закоммитим: $ git add feature.txt

$ git commit -m «New feature complete.»

Изменения завершены, теперь мы можем переключиться обратно на ветку master. $ git checkout master

Теперь, если откроем наш проект в файловом менеджере, не увидим файла feature.txt, потому что мы переключились обратно на ветку master, в которой такого файла не существует. Чтобы он появился, нужно воспользоваться merge для объединения веток (применения изменений из ветки amazing_new_feature к основнойверсии проекта).

$ git merge amazing_new_feature

Теперь ветка master актуальна. Ветка amazing_new_feature больше не нужна, и ее можно удалить.

$ git branch -d awesome_new_feature

- Дополнительно.

В последней части этого руководства мы расскажем о некоторых дополнительных трюках, которые могут вам помочь.

4. Отслеживание изменений, сделанных в коммитах.

У каждого коммита есть свой уникальный идентификатор в виде строки цифр и букв. Чтобы просмотреть список всех коммитов и их идентификаторов, можно использовать команду log:

Вывод git log.

Как можно заметить, идентификаторы довольно длинные, но для работы с ними не обязательно копировать их целиком – первых нескольких символов будет вполне достаточно. Чтобы посмотреть, что нового появилось в коммите, можно воспользоваться командой show [commit]

Вывод git show.

Чтобы увидеть разницу между двумя коммитами, используетсякоманда diff (с указанием промежутка между коммитами):

Вывод git diff.

Мы сравнили первый коммит с последним, чтобы увидеть все изменения, которые были когда-либо сделаны. Обычно проще использовать git difftool, так как эта команда запускает графический клиент, в котором наглядно сопоставляет все изменения.

1. Возвращение файла к предыдущему состоянию.

Гит позволяет вернуть выбранный файл к состоянию на момент определенного коммита. Это делается уже знакомой нам командой checkout, которую ранее использовали для переключения между ветками. Но она также может быть использована для переключения между коммитами (это довольно распространенная ситуация для Гита – использование одной команды для различных, на первый взгляд, слабо связанных задач).

В следующем примере возьмем файл hello.txt и откатим всеизменения, совершенные над ним к первому коммиту. Чтобы сделать это, мы подставим в команду идентификатор нужного коммита, а также путь до файла: $ git checkout 09bd8cc1 hello.txt

2. Исправление коммита.

Если вы опечатались в комментарии или забыли добавить файл и

заметили это сразу после того, как закоммитили изменения, вы легко можете это поправить при помощи commit –amend. Эта команда добавит все из последнего коммита в область подготовленных файлов и попытается сделать новый коммит. Это дает вам возможность поправить комментарий или добавить недостающие файлы в область подготовленных файлов.

Для более сложных исправлений, например, не в последнем коммите или если вы успели отправить изменения на сервер, нужно использовать revert. Эта команда создаст коммит, отменяющий изменения, совершенные в коммите с заданным идентификатором. Самый последний коммит может быть доступен по алиасу HEAD:

$ git revert HEAD

Для остальных будем использовать идентификаторы:

$ git revert b10cc123

При отмене старых коммитов нужно быть готовым к тому, что возникнут конфликты. Такое случается, если файл был изменен еще одним, более новым коммитом. И теперь git не может найти строчки, состояние которых нужно откатить, так как они больше не существуют.

3. Разрешение конфликтов при слиянии.

Помимо сценария, описанного в предыдущем пункте, конфликты регулярно возникают при слиянии ветвей или при отправке чужого кода. Иногда конфликты исправляются автоматически, но обычно с этим приходится разбираться вручную – решать, какой код остается, а какой нужно удалить.

Давайте посмотрим на примеры, где мы попытаемся слить две ветки под названием john_branch и tim_branch. И Тим, и Джон правят один и тот же файл: функцию, которая отображает элементы массива.

Джон использует цикл:

// Use a for loop to console.log contents.for(var i=0; i<arr.length; i++) {

console.log(arr[i]);

}

Тим предпочитает forEach:

// Use forEach to console.log contents. arr.forEach(function(item) { console.log(item);

});

Они оба коммитят свой код в соответствующую ветку. Теперь,если они попытаются слить две ветки, они получат сообщение об ошибке:

$ git merge tim_branch Auto-merging print_array.js

CONFLICT (content): Merge conflict in print_array.js

Automatic merge failed; fix conflicts and then commit the result.

Система не смогла разрешить конфликт автоматически, значит, это придется сделать разработчикам. Приложение отметило строки, содержащие конфликт:

Вывод.

Над разделителем ======= мы видим последний (HEAD) коммит, а под ним – конфликтующий. Таким образом, мы можем увидеть, чем они

отличаются и решать, какая версия лучше. Или вовсе написать новую. В этой ситуации мы так и поступим, перепишем все, удалив разделители, и дадим git понять, что закончили.

// Not using for loop or forEach.

// Use Array.toString() to console.log contents.console.log(arr.toString());

Когда все готово, нужно закоммитить изменения, чтобы закончить процесс: $ git add -A

$ git commit -m «Array printing conflict resolved.»

Как вы можете заметить, процесс довольно утомительный и может быть очень сложным в больших проектах. Многие разработчики предпочитают использовать для разрешения конфликтов клиенты с графическим интерфейсом. (Для запуска нужно набрать git mergetool).

5. Настройка .gitignore.

В большинстве проектов есть файлы или целые директории, в которые мы не хотим (и, скорее всего, не захотим) коммитить. Мы можем удостовериться, что они случайно не попадут в git add -A при помощи файла .gitignore

1. Создайте вручную файл под названием .gitignore и сохраните его в директорию проекта.
2. Внутри файла перечислите названия файлов/папок, которые нужно игнорировать, каждый с новой строки.
3. Файл .gitignore должен быть добавлен, закоммичен и отправлен на сервер, как любой другой файл в проекте.

Вот хорошие примеры файлов, которые нужно игнорировать:

- Логи
- Артефакты систем сборки
- Папки node_modules в проектах node.js
- Папки, созданные IDE, например, Netbeans или IntelliJ
- Разнообразные заметки разработчика.

Файл .gitignore, исключающий все перечисленное выше, будет выглядеть так:

*.log build/

node_modules/

.idea/ my_notes.txt

Символ слэша в конце некоторых линий означает директорию (и тот факт, что мы рекурсивно игнорируем все ее содержимое). Звездочка, как обычно, означает шаблон.

Контрольные вопросы

1. Приведите основные команды git.
2. Как создать новую ветку в git?
3. Как переключиться в существующую ветку?
4. Как отправить изменения на сервер?

Практическая работа №5 «Построение диаграммы Вариантов использования и диаграммы Последовательности»

Цель работы:

Целью работы является изучение основ создания диаграмм прецедентов (вариантов использования) на языке UML.

Задачи:

Основными задачами практической работы являются:

– ознакомиться с теоретическими вопросами построения диаграмм прецедентов с помощью MS Visio;

– получить навыки создания диаграмм прецедентов.

Краткие теоретические сведения

Общие сведения о языке UML

Язык UML представляет собой общецелевой язык визуального моделирования, который разработан для спецификации, визуализации, проектирования и документирования компонентов программного обеспечения,бизнес-процессов и других систем.

Язык UML одновременно является простым и мощным средством моделирования, который может быть эффективно использован для построения концептуальных, логических и графических моделей сложных систем самого различного целевого назначения.

В языке UML используется ***четыре основных вида графических конструкций***:

– **Значки или пиктограммы**. Значок представляет собой графическую фигуру фиксированного размера и формы. Примерами значков могут служить окончания связей элементов диаграмм или некоторые другие дополнительные обозначения.

– **Графические символы на плоскости**. Такие двумерные символы изображаются с помощью некоторых геометрических фигур и могут иметь различную высоту и ширину с целью размещения внутри этих фигур других конструкций языка UML.

Наиболее часто внутри таких символов помещаются строки текста, которые уточняют семантику или фиксируют отдельные свойства соответствующих элементов языка UML. Информация, содержащаяся внутри фигур, имеет важное значение для конкретной модели проектируемой системы, поскольку регламентирует реализацию соответствующих элементов в программном коде.

– **Пути,** которые представляют собой последовательности из отрезков линий, соединяющих отдельные графические символы. При этом концевые точки отрезков линий должны обязательно соприкасаться с геометрическими фигурами, служащими для обозначения вершин диаграмм, как принято в теории графов. С концептуальной точки зрения путям в языке UML придается особое значение, поскольку они являются простыми топологическими сущностями.

– **Строки текста.** Служат для представления различных видов

информации в некоторой грамматической форме. Предполагается, что каждое использование строки текста должно соответствовать синтаксису в нотации языка UML, посредством которого может быть реализован грамматический разбор этой строки.

При графическом изображении диаграмм следует придерживаться следующих ***основных рекомендаций***:

1. Каждая диаграмма должна служить законченным представлением соответствующего фрагмента моделируемой предметной области.
2. Все сущности на диаграмме модели должны быть одного концептуального уровня. Здесь имеется в виду согласованность не только имен одинаковых элементов, но и возможность вложения отдельных диаграмм друг в друга для достижения полноты представлений.
3. Вся информация о сущностях должна быть явно представлена на диаграммах. Речь идет о том, что, хотя в языке UML при отсутствии некоторых символов на диаграмме могут быть использованы их значения по умолчанию (например, в случае неявного указания видимости атрибутов и операций классов), необходимо стремиться к явному указанию свойств всех элементов диаграмм.
4. Диаграммы не должны содержать противоречивой информации. Противоречивость модели может служить причиной серьезнейших проблем при ее реализации и последующем использовании на практике. Например, наличие замкнутых путей при изображении отношений агрегирования или композиции приводит к ошибкам в программном коде, который будет реализовывать соответствующие классы. Наличие элементов с одинаковыми именами и различными атрибутами свойств в одном пространстве имен также приводит к неоднозначной интерпретации и может служить источником проблем.
5. Диаграммы не следует перегружать текстовой информацией. Принято считать, что визуализация модели является наиболее эффективной, если она содержит минимум пояснительного текста.
6. Каждая диаграмма должна быть самодостаточной для правильной интерпретации всех ее элементов и понимания семантики всех используемых графических символов.
7. Количество типов диаграмм для конкретной модели приложения не является строго фиксированным.

3.1 *Диаграмма вариантов использования (usecase diagram)*

Разработка диаграммы вариантов использования преследует цели:

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

Суть данной диаграммы состоит в следующем: проектируемая система представляется в виде множества сущностей или актеров, взаимодействующих с системой.

При этом **актером (actor)** или действующим лицом называется любая сущность, взаимодействующая с системой извне. Это может быть человек, техническое устройство, программа или любая другая система, которая может служить источником воздействия на моделируемую систему так, как определит сам разработчик.

В свою очередь, **вариант использования (usecase)** служит для описания сервисов, которые система предоставляет актеру. Другими словами, каждый вариант использования определяет некоторый набор действий, совершаемый системой при диалоге с актером. При этом ничего не говорится о том, каким образом будет реализовано взаимодействие актеров с системой.

В самом общем случае, диаграмма вариантов использования представляет собой граф специального вида, который является графической нотацией для представления конкретных вариантов использования, актеров, возможно некоторых интерфейсов, и отношений между этими элементами.

Вариант использования (use case) **–** конструкция или стандартный элемент языка UML, который применяется для спецификации общих особенностей поведения системы или любой другой сущности предметной области без рассмотрения внутренней структуры этой сущности. Каждый вариант использования определяет последовательность действий, которые должны быть выполнены проектируемой системой при взаимодействии ее с соответствующим актером.

Диаграмма вариантов может дополняться пояснительным текстом, который раскрывает смысл или семантику составляющих ее компонентов.

Такой пояснительный текст получил название ***примечания или сценария***.

Отдельный вариант использования обозначается на диаграмме эллипсом (рис. 2), внутри которого содержится его краткое название или имя в форме глагола с пояснительными словами.

Рисунок 2 – Графическое обозначение варианта использования

Актер (actor) представляет собой любую внешнюю по отношению к моделируемой системе сущность, которая взаимодействует с системой и использует ее функциональные возможности для достижения определенных целей или решения частных задач. При этом актеры служат для обозначения согласованного множества ролей, которые могут играть пользователи в процессе взаимодействия с проектируемой системой. Каждый актер может рассматриваться как некая отдельная роль относительно конкретного

варианта использования. Стандартным графическим обозначением актера на диаграммах является фигурка «человечка» (рис. 3), под которой записывается конкретное имя актера.

Рисунок 3 – Графическое обозначение актера

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

На диаграмме вариантов использования интерфейс изображается в виде маленького круга, рядом с которым записывается его имя (рис. 4).

Рисунок 4 – Графическое обозначение интерфейса

В качестве имени может быть существительное, которое характеризует соответствующую информацию или сервис (например, «датчик», «сирена», «видеокамера»), но чаще строка текста (например, «запрос к базе данных», «форма ввода», «устройство подачи звукового сигнала»).

Если имя записывается на английском, то оно должно начинаться с заглавной буквы I, например, ISecurelnformation, ISensor.

Графический символ отдельного интерфейса может соединяться на диаграмме сплошной линией с тем вариантом использования, который его поддерживает. Сплошная линия в этом случае указывает на тот факт, что связанный с интерфейсом вариант использования должен реализовывать все операции, необходимые для данного интерфейса, а возможно и больше (рис. 5а). Кроме этого, интерфейсы могут соединяться с вариантами использования пунктирной линией со стрелкой (рис. 5б), означающей, что вариант использования предназначен для спецификации только того сервиса, который необходим для реализации данного интерфейса.

Рисунок 5 – Графическое обозначение взаимосвязей интерфейсов и вариантов использования

Примечания (notes) в языке UML предназначены для включения в модель произвольной текстовой информации, имеющей непосредственное отношение к контексту разрабатываемого проекта. В качестве такой информации могут быть комментарии разработчика (например, дата и версия разработки диаграммы или ее отдельных компонентов), ограничения (например, на значения отдельных связей или экземпляры сущностей) и помеченные значения.

Применительно к диаграммам вариантов использования примечание может носить самую общую информацию, относящуюся к общему контексту системы.

Графически примечания обозначаются прямоугольником с «загнутым» верхним правым уголком (рис. 6). Внутри прямоугольника содержится текст примечания. Примечание может относиться к любому элементу диаграммы, в этом случае их соединяет пунктирная линия. Если примечание относится к нескольким элементам, то от него проводятся, соответственно, несколько линий. Разумеется, примечания могут присутствовать не только на диаграмме вариантов использования, но и на других канонических диаграммах.

Рисунок 6 – Пример обозначения примечания

Отношения на диаграмме вариантов использования.

Между компонентами диаграммы вариантов использования могут существовать различные отношения, которые описывают взаимодействие экземпляров одних актеров и вариантов использования с экземплярами других актеров и вариантов.

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

В языке UML имеется несколько стандартных ***видов отношений между актерами и вариантами использования:***

1. Отношение ассоциации (association relationship)

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

Применительно к диаграммам вариантов использования оно служит для обозначения специфической роли актера в отдельном варианте использования. Другими словами, ассоциация специфицирует семантические особенности взаимодействия актеров и вариантов использования в графической модели системы. Таким образом, это отношение устанавливает, какую конкретную роль играет актер при взаимодействии с экземпляром варианта использования. На диаграмме вариантов использования, так же, как и на других диаграммах, отношение ассоциации обозначается сплошной линией между актером и вариантом использования. Эта линия может иметь дополнительные условные обозначения, такие, например, как имя и кратность (рис. 7).

Рисунок 7 – Графическое обозначение отношения ассоциации

2. Отношение расширения (extend relationship)

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

Так, если имеет место отношение расширения от варианта использования A к варианту использования B, то это означает, что свойства экземпляра варианта использования B могут быть дополнены благодаря наличию свойств у расширенного варианта использования A.

Отношение расширения между вариантами использования обозначается пунктирной линией со стрелкой (вариант отношения зависимости), направленной от того варианта использования, который является расширениемдля исходного варианта использования. Данная линия со стрелкой помечается ключевым словом «extend» («расширяет»), как показано на рис. 8.

Рисунок 8 – Графическое обозначение отношения расширения

Отношение расширения отмечает тот факт, что один из вариантов использования может присоединять к своему поведению некоторое дополнительное поведение, определенное для другого варианта

использования.

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

3. Отношение обобщения (generalization relationship)

Отношение обобщения служит для указания того факта, что некоторый вариант использования А может быть обобщен до варианта использования В.

В этом случае вариант А будет являться специализацией варианта В. При этом В называется предком или родителем по отношению А, а вариант А – потомком по отношению к варианту использования В. Следует подчеркнуть, что потомок наследует все свойства и поведение своего родителя, а также может быть дополнен новыми свойствами и особенностями поведения.

Графически данное отношение обозначается сплошной линией со стрелкой в форме незакрашенного треугольника, которая указывает на родительский вариант использования (рис. 9). Эта линия со стрелкой имеет специальное название – стрелка «обобщение».

Рисунок 9 – Графическое обозначение отношения обобщения

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

Между отдельными актерами также может существовать отношение обобщения. Данное отношение является направленным и указывает на факт специализации одних актеров относительно других. Например, отношение обобщения от актера А к актеру В отмечает тот факт, что каждый экземпляр актера А является одновременно экземпляром актера В и обладает всеми его свойствами. В этом случае актер В является родителем по отношению к актеру А, а актер А, соответственно, потомком актера В. При этом актер А обладает способностью играть такое же множество ролей, что и актер В.

Графически данное отношение также обозначается стрелкой обобщения, т. е. сплошной линией со стрелкой в форме незакрашенного треугольника, которая указывает на родительского актера (рис. 10).

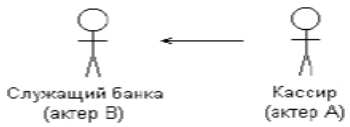


Рисунок 10 – Графическое обозначение отношения обобщения между актерами

4. relationship)

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

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

Рисунок 11 – Графическое обозначение отношения включения

5. Методика выполнения

В качестве примера рассматривается моделирование системы продажи товаров по каталогу.

1. Запустите MS Visio.

2. На экране выбора шаблона выберите категорию *Программы и БД* и в ней элемент *Схема модели UML*. Нажмите кнопку *Создать* в правой части экрана.

3. Окно программы примет вид, подобный рис. 12.

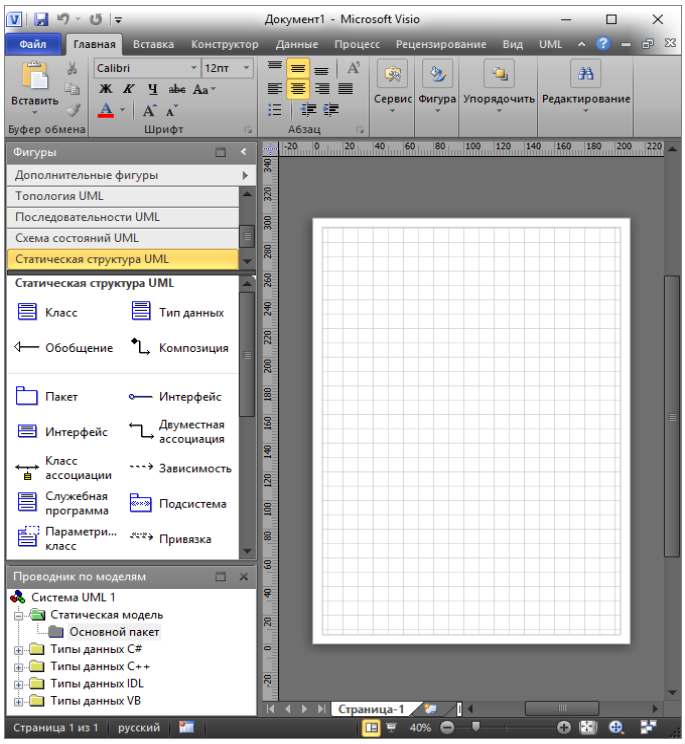


Рисунок 12 – Схема модели UML в MS Visio

4. Далее необходимо открыть все фигуры, необходимые для построения UML-диаграмм. Для этого в левой части экрана необходимо нажать кнопку *Дополнительные фигуры*. В открывшемся вспомогательном меню выбрать *Программы и БД -> Программное обеспечение* и выбрать все доступные фигуры для построения UML (рис. 13).

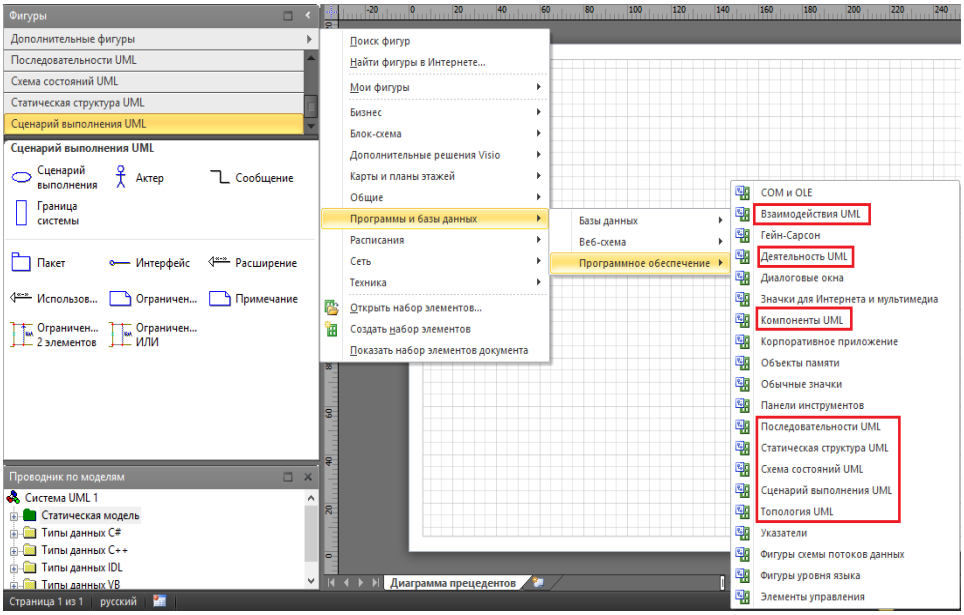


Рисунок 13 – Добавление фигур UML

5. После этого необходимо провести следующие этапы моделирования.

5.1. Выбор актеров.

В качестве актеров данной системы могут выступать два субъекта, один из которых является продавцом, а другой – покупателем. Каждый из

этих актеров взаимодействует с рассматриваемой системой продажи товаров по каталогу и является ее пользователем, т. е. они оба обращаются к соответствующему сервису «Оформить заказ на покупку товара». Как следует из существа выдвигаемых к системе требований, этот сервис выступает в качестве варианта использования разрабатываемой диаграммы, первоначальная структура которой может включать в себя только двух указанных актеров и единственный вариант использования (рис. 14).

– В группе фигур *Сценарий выполнения UML* выбрать блок *Граница системы* и добавить его на лист.

– Внутрь границы системы добавить блок *Сценарий выполнения* и добавить к нему название, дважды щелкнув внутри блока.

– Добавить два блока *Актер* – покупатель и продавец.

– С помощью блока *Сообщение* установите связь актеров и варианта использования. Двойным щелчком правой кнопки мыши по блоку Сообщение откройте окно *Свойств ассоциации UML,* проведите настройки в соответствии с рисунком 15.

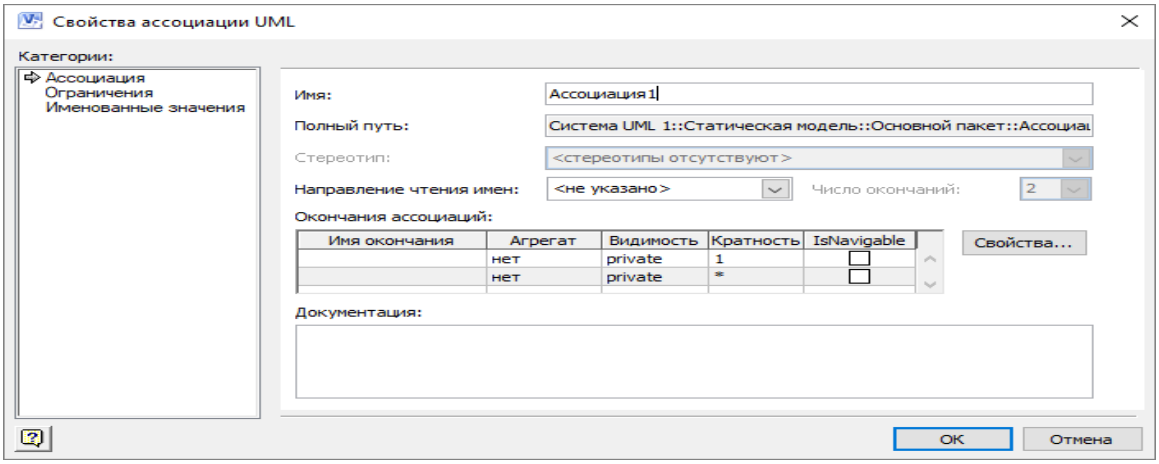


Рисунок 14 – Свойства ассоциации UML

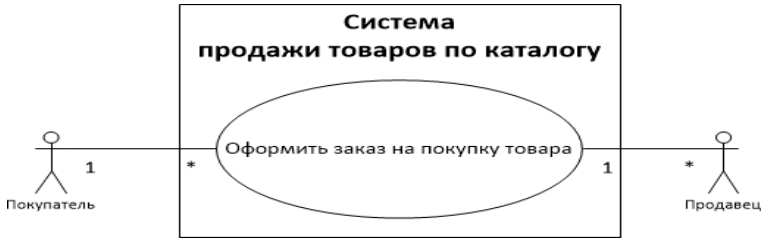


Рисунок 15 – Исходная диаграмма вариантов использования системы по продаже товаров

5.2. Выделение дополнительных вариантов использования.

Детализировать вариант использования «Оформить заказ на продажу товара» можно выделив следующие дополнительные варианты использования:

– обеспечить покупателя информацией – является отношением включения;

– согласовать условия оплаты – является отношением включения;

– заказать товар со склада – является отношением включения;

– запросить каталог товаров – является отношением расширения.

Так как в MS Visio отсутствует отношение включения, его необходимо добавить самостоятельно. Для этого перейти на вкладку *UML* -> в группе *Модель* выбрать пункт *Стереотипы.* В открывшемся окне нажать кнопку *Создать* и настроить стереотип в соответствии с рисунком 16.

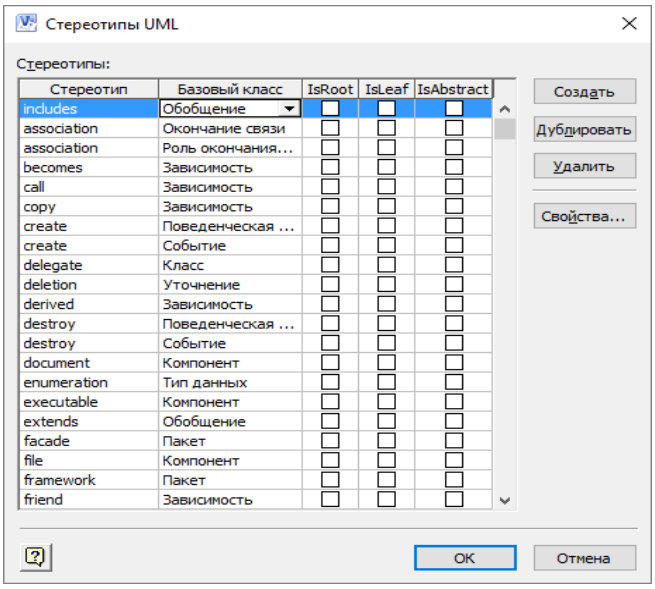


Рисунок 16 – Создание стереотипа

Далее на новом листе необходимо добавить границу системы и все варианты использования. После чего соединить варианты использования с помощью блока *Расширение.*

Для того, чтобы изменить тип отношения дважды щелкните по стрелке и окне свойств задайте необходимыепараметры.

Дополненная диаграмма вариантов использования примет вид, показанный на рисунке 17.

Рисунок 17 – Дополненная диаграмма вариантов использования

5.3. Написание описательной спецификации для каждого варианта использования.

Спецификация для варианта использования «Оформить заказ на покупку компьютера» приведена в таблице 1.

Таблица 1 – Спецификация варианта использования

Раздел	**Описание**
Краткое описание	Покупатель желает оформить заказ на покупку компьютера, который он выбрал в каталоге товаров. При условии, что клиент зарегистрирован и выбранный компьютер есть в наличии оформляется заказ. Если клиент не зарегистрирован, то предлагается ему пройти регистрацию, и после этого заказать выбранный компьютер. Если компьютера нет в наличии, то предлагается заказать товар со склада в течении заданного срока поставки.
Субъекты	Продавец, Покупатель
Предусловия	В каталоге товаров имеются компьютеры, которые можно заказать. У покупателей есть доступ к системе для регистрации. Продавцы умеют пользоваться рассматриваемой системой продажи. У покупателя есть бонусы.
Основной поток	Зарегистрированный покупатель имеет возможность заказать любой компьютер из каталога товаров. В случае наличия выбранного компьютера оформляется заказ с присвоением ему уникального номера. После этого покупателю предлагается выбрать способ оплаты и способ получения компьютера. В случае отсутствия компьютера в наличии предлагается оформить заказ со склада и ожидания его поставки в рамках указанного срока или выбрать другой компьютер.
Альтернативный поток	Покупатель не зарегистрирован. В этом случае, прежде чем оформить заказ на компьютер, ему предлагается пройти регистрацию. Попытка заказать товар, который отсутствует на складеНачисление бонусов
Постусловия	Заказ оформлен и определен срок поставки компьютера и место его получения

На рисунках 18-20 приведены примеры диаграмм вариантов использования для различных систем.

Рисунок 18 – Пример 1

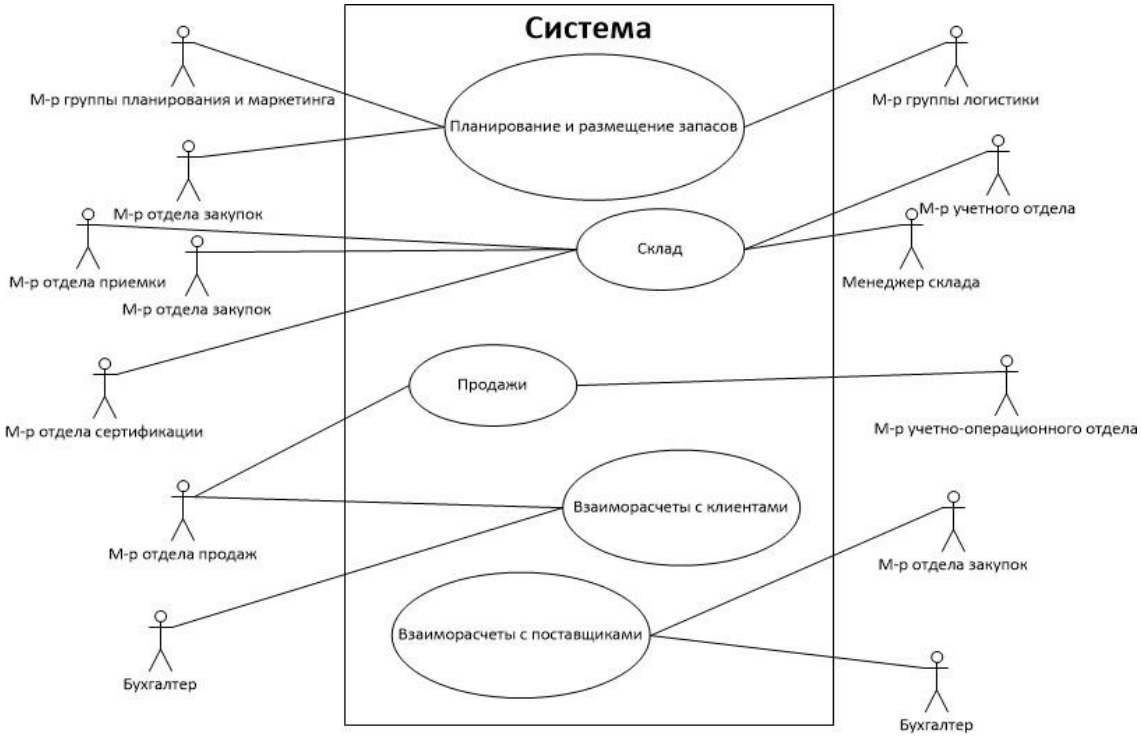


Рисунок 19 – Пример 2

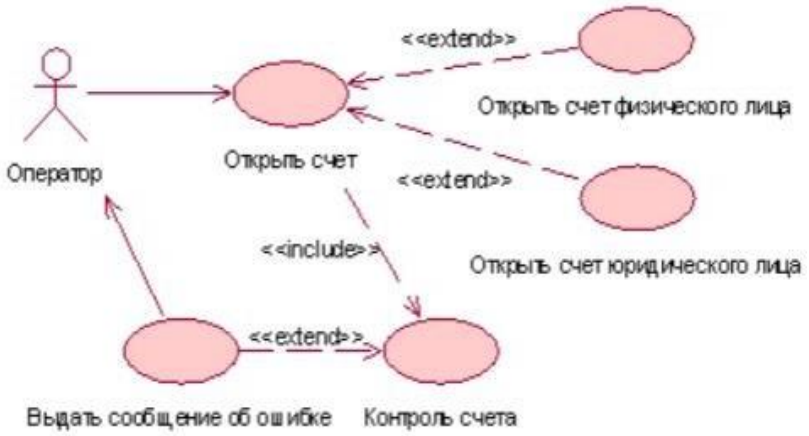


Рисунок 20 – Пример 3

6. Задание

Построить диаграмму прецедентов (вариантов использования) в соответствии с вариантом. Составить спецификацию.

Отчет по практическому занятию выполняется в формате MS Word, который содержит пошаговое описание процесса построения организационной диаграммы, а также скриншоты результатов согласно заданию.

7. Варианты

1. «Отдел кадров»;
2. «Агентство аренды»;
3. «Аптека»;
4. «Ателье»;

5. «Аэропорт»;
6. «Библиотека»;
7. «Кинотеатр»;
8. «Поликлиника»;
9. «Автосалон»;10.«Таксопарк».

8. Контрольные вопросы

1. Для чего используется язык UML?
2. Назначение диаграммы вариантов использования?
3. Что такое «актер»?
4. Что такое «вариант использования»?
5. Что такое «интерфейс»?
6. Что такое «примечание»?
7. Перечислить виды отношений между актерами и вариантами использования, охарактеризовать каждое изних?

Практическая работа №6
«Построение диаграммы Кооперации и диаграммы Развертывания»

Цель работы:

Целью работы является изучение основ создания диаграмм классов на языке UML, получение навыков построения диаграмм классов, применение приобретенных навыков для построения объектно-ориентированных моделей определенной предметной области.

Задачи:

Основными задачами практической работы являются:

– ознакомиться с теоретическими вопросами построения диаграмм классов;

– ознакомиться с теоретическими вопросами построения диаграмм классов с помощью MS Visio;

– получить навыки создания диаграмм классов.

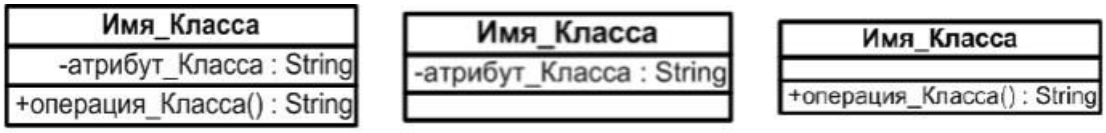
Краткие теоретические сведения.

Диаграмма классов является одной из канонических диаграмм UML, создаваемой для визуализации структурированной статической модели предметной области. Этот вид диаграмм представляет собой графическое изображение объектов – классов с присущими им атрибутами, операциями и различных отношений междуклассами.

Классы.

Класс (class) служит для обозначения множества объектов, обладающих функциональным набором одинаково описывающих параметров (атрибутов), реализуемых операций и однотипными отношениями с объектами других классов.

На диаграмме класс изображается габаритной прямоугольной рамкой, которая дополнительно может быть разделена горизонтальными линиями на секции, каждая из которых предназначена для указания имени, атрибутов (свойств) и реализуемых операций объектов данного класса (рис. 21 а, б, в).



а) б) в)

Рисунок 21 – а) Обозначение класса, б) Обозначение атрибутов, в) Обозначение операций

Имя класса является обязательным элементом в его обозначении и должно быть уникальным (хотя бы в пределах пакета), имеющее непосредственное отношение к контексту моделируемой предметной области.

В соответствии с принятым в языке UML общим соглашением в качестве имени класса используются существительные и прилагательные, каждое из которых начинается с заглавной буквы, записанные без пробелов. Например, в качестве имен классов могут быть использованы профессиональные термины: «Сотрудник», «Компания»,

«Руководитель», «Клиент», «Продавец», «Менеджер», «Офис», «Покупатель», «Датчик_Температуры» и др. Такое имена классов являются простыми.

Иногда возникает необходимость в явном указании пакета, к которому относится данный класс. С этой целью в условном обозначении перед именем класса указывается имя пакета и специальный символ разделитель – двойное двоеточие "::". Такое имя класса является квалифицированным. Текстовая строка имени класса в этом случае записывается в формате <Имя_пакета>::<Имя_класса>. Например, если определен пакет с именем «Банк», то имя класса «Счет» может быть записано так, как показано на рис. 22.

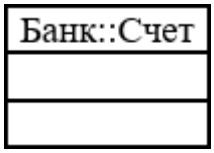


Рисунок 22 – Обозначение квалифицированного имени класса

Атрибуты классов.

Содержательной характеристикой класса является атрибут, содержащий множество значений, которые могут принимать отдельные объекты этого класса. При этом, класс может иметь любое число атрибутов или не иметь ни одного. Так, например, атрибутами класса «Усилитель» являются частотный диапазон, выходная мощность, коэффициент нелинейных искажений, уровень шума и т. д.

Запись атрибута также представляет собой отдельную строку текста, содержащую обязательное имя, в котором обычно каждое слово пишется с заглавной буквы, за исключением первого, например, name (имя) или birth_Date (дата_Рождения).

Например, на рис. 23 указаны атрибуты класса Контейнер, в качестве которых выступают атрибут тип_Контейнера, атрибут регистрационный_Номер_Контейнера, регистрационный_Номер_После_ Перерегистрации, рабочее_Состояние.

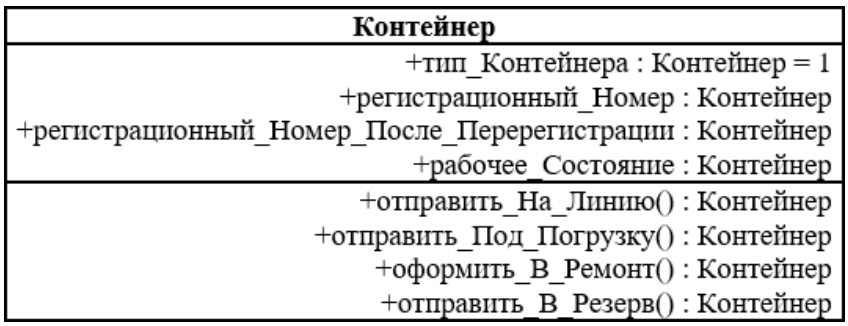


Рисунок 23 – Указание атрибутов класса Контейнер

Качественной характеристикой описания элементов класса является квантор видимости атрибута – потенциальная возможность других объектов

модели оказывать влияние на отдельные аспекты поведения данного класса.

Эта характеристика может принимать одно из трех возможных значений и, соответственно, отображается при помощи специальных символов: символ "+" (public) обозначает атрибут с областью видимости типа общедоступный; атрибут с этой областью видимости доступен или виден из любого другого класса пакета, в котором определена диаграмма; например, для класса Class_1 указан атрибут общедоступного типа (рис. 24а); символ "#" (protected) обозначает атрибут с областью видимости типа защищенный; атрибут с этой областью видимости недоступен или невиден для всех классов, за исключением подклассов данного класса; например, для класса Class_2 указан атрибут защищенного типа (рис. 24б); символ "-" (private) обозначает атрибут с областью видимости типа закрытый; атрибут с этой областью видимости недоступен или невиден для всех классов без исключения; например, для класса Class_3 указан атрибут защищенного типа (рис. 24в);

Квантор видимости при описании атрибутов может быть опущен, что будет означать тот факт, что видимость атрибута не указывается.

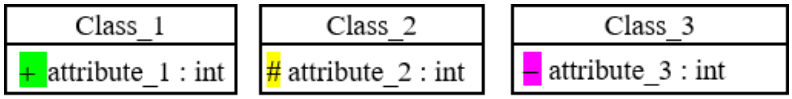


Рисунок 24 – Обозначение видимости атрибутов класса

Операции классов.

Операция (operation) класса – это реализация услуги, которая может быть запрошена у любого объекта данного класса, чтобы вызвать определенное его поведение. Класс может иметь любое число операций либо не иметь ни одной. Так автомобиль может перемещаться по грунту, корабль – перемещаться по воде, компьютер – производить вычисления.

Представление полного синтаксиса записи операций класса также подчиняется определенным синтаксическим правилам: каждой операции класса соответствует отдельная строка, которая состоит из квантора видимости операции, обязательного имени операции, выражения типа возвращаемого операцией значения и, возможно, строки-свойства данной операции:

< квантор видимости >< имя операции > (список параметров) : < выражение типа возвращаемого значения >{строка-свойство}

Квантор видимости, как и в случае атрибутов класса, может принимать одно из трех возможных значений и, соответственно, также отображается при помощи специального символа.

Для именования операции используются короткие глагольные конструкции, описывающие некоторое поведение класса, которому принадлежит операция. Обычно каждое слово в имени операции пишется с заглавной буквы, за исключением первого.

Например, запись +создать() – может обозначать абстрактную

операцию по созданию отдельного объекта класса, которая является общедоступной и не содержит формальных параметров, запись +нарисовать(форма: Многоугольник = прямоугольник, цвет_заливки: Color = (О, О, 255)) – может обозначать операцию по изображению на экране монитора прямоугольной области синего цвета, если не указываются другие значения в качестве аргументов данной операции.

(список–параметров) содержит необязательные аргументы, синтаксис которых совпадает с синтаксисом атрибутов;

< выражение типа возвращаемого значения > является необязательной спецификацией и зависит от конкретного языка программирования; {строка-свойство} показывает значения свойств, которые применяются к данной операции.

Например, запись запросить_Счет_Клиента(номер_счета:Integer) – обозначает операцию по установлению наличия средств на текущем счете клиента банка. При этом аргументом данной операции является номер счета клиента, который записывается в виде целого числа (например, «123456»).

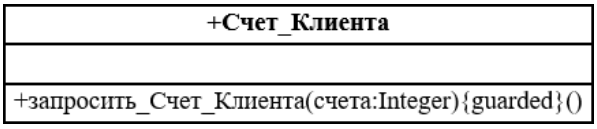


Рисунок 25 – Обозначение операции «запросить_Счет_Клиента»

Квантор видимости для операции может быть опущен. В этом случае его отсутствие означает, что видимость операции не указывается.

Отношения между классами.

Классы на диаграмме связываются различными типами отношений. При этом совокупность типов таких отношений фиксирована в языке UML и предопределена их семантикой.

1.Отношение зависимости.

Отношением зависимости (dependency relationship) называют связь по использованию, когда изменение в спецификации одного класса может повлиять на поведение другого. Отношение зависимости используется в такой ситуации, когда некоторое изменение одного элемента модели может потребовать изменения другого зависимого от него элемента модели. Отношение зависимости графически изображается пунктирной линией между соответствующими элементами со стрелкой на одном из ее концов, направленной к той сущности, от которой зависит данная сущность. Например, (рис. 26).

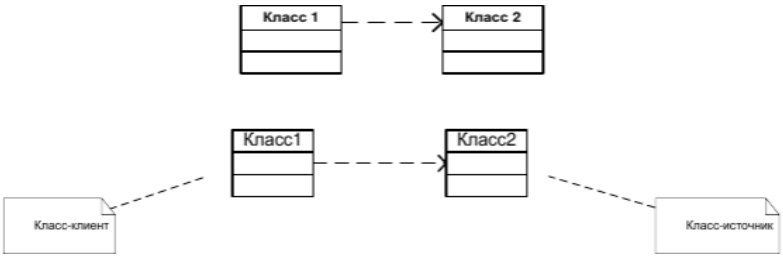


Рисунок 26 – Обозначение отношения зависимости

В качестве класса-клиента и класса-источника зависимости может выступать множество элементов модели. В этом случае одна линия со стрелкой, выходящая от источника зависимости, расщепляется в некоторой точке на несколько отдельных линий, каждая из которых имеет отдельную стрелку для класса-клиента.

Например, если функционирование сущности Класс_С зависит от особенностей реализации сущностей Класс_А и Класс_Б (рис. 27).

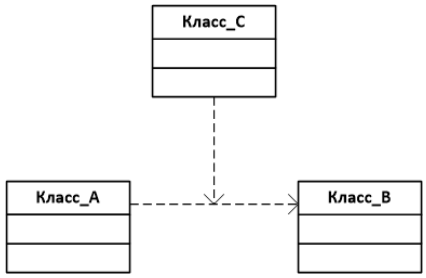


Рисунок 27 – Обозначение зависимости между классом-клиентом (Класс_С) и классами-источниками (Класс_А и Класс_В)

2.Отношение ассоциации.

Ассоциацией (association relationship) называется структурная связь, показывающая, что объекты одного класса некоторым образом связаны с объектами другого или того же самого класса.

Ассоциация может отображаться графически линией со стрелкой (маркером в виде треугольника), показывающей направление следования классов и кратность – количество объектов, связанных отношением. Отсутствие стрелки рядом с именем ассоциации означает, что порядок следования классов в рассматриваемом отношении не определен (рис. 28).

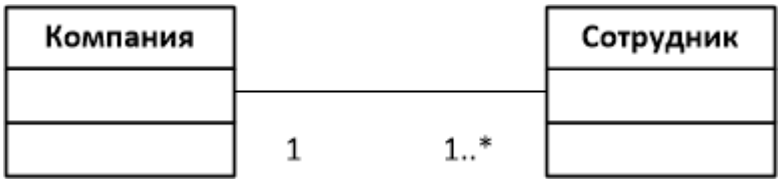


Рисунок 28 – Обозначение отношения ассоциации

Так, в примере на рис. 8 кратность «1» для класса «Компания» означает, что каждый сотрудник может работать только в одной компании. Кратность «1..*» для класса «Сотрудник» означает, что в каждой компании могут работать несколько сотрудников, общее число которых заранее неизвестно и ничем не ограничено.

Специальной формой или частным случаем отношения ассоциации является отношение агрегации, которое, в свою очередь, тоже имеет специальную форму – отношение композиции.

3.Отношение агрегации.

Отношение агрегации (generalization relationship) имеет место между несколькими классами в том случае, если один из классов представляет собой некоторую сущность, включающую в себя в качестве составных частей другие сущности.

Данное отношение имеет фундаментальное значение для описания структуры сложных систем, поскольку применяется для представления системных взаимосвязей типа «часть – целое».

Это отношение по своей сути описывает декомпозицию или разбиение сложной системы на более простые составные части, которые также могут быть подвергнуты декомпозиции, если в этом возникнет необходимость в последующем.

Так, автомобиль состоит из кузова, двигателя, трансмиссии и т.п., а в состав приемопередающего устройства входят передатчик, приемник и антенно-фидерное устройство.

Графически отношение агрегации изображается сплошной линией, один из концов которой представляет собой геометрическую фигуру – ромб. Этот ромб указывает на тот из классов, который представляет собой «целое» (рис. 29).

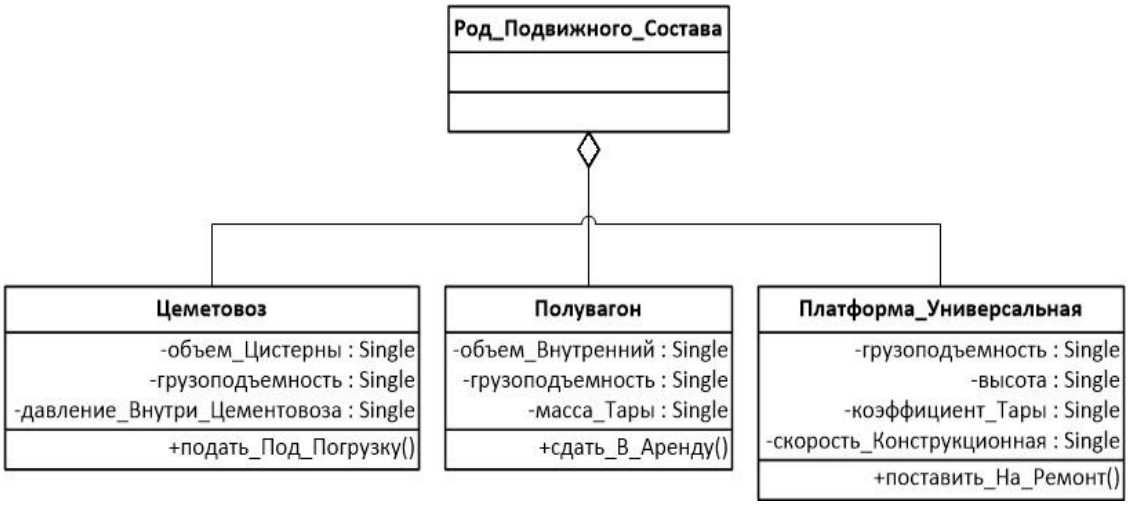


Рисунок 29 – Обозначение отношения агрегации

Примером отношения агрегации может служить деление класса Аналитическая_информация на составные части: Отчет_по_грузу, Отчет_по_контейнерам, Отчет_по_тарифам (рис. 30).

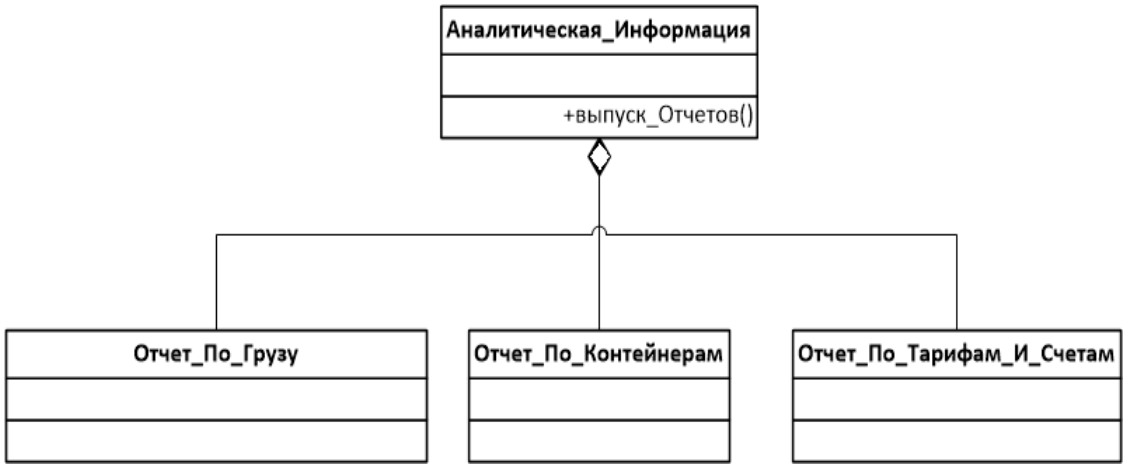


Рисунок 30 – Пример отношения агрегации

Отношение агрегации обладает кратностью. Класс Система_ обеспечения_безопасности_объектов может содержать содержит одну подсистему Сопровождение_грузов, которая в свою очередь может содержать, четыре класса Охрана_вооруженная, каждый из которых может принадлежать лишь одному классу Сопровождение_грузов (рис. 31).

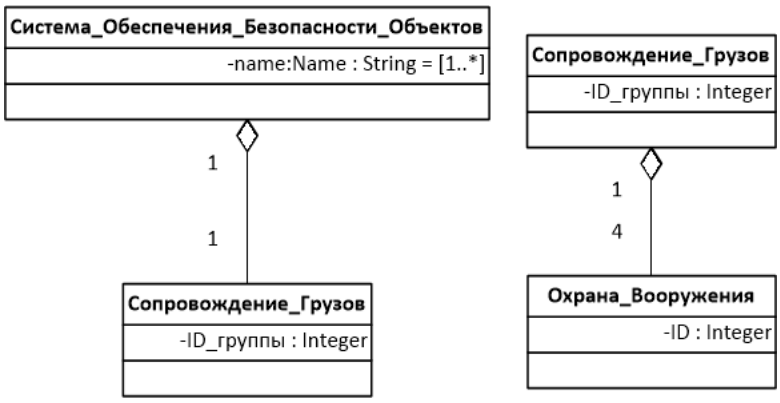


Рисунок 31 – Пример обозначения кратности отношения агрегации

4.Отношение композиции.

Отношение композиции (realization relationship) служит для выделения специальной формы отношения «часть-целое», при которой составляющие части в некотором смысле находятся внутри целого.

Специфика взаимосвязи между ними заключается в том, что части не могут выступать в отрыве от целого, т. е. с уничтожением целого уничтожаются и все его составные части.

Графически отношение композиции изображается сплошной линией, один из концов которой представляет собой закрашенный внутри ромб. Этот ромб указывает на тот из классов, который представляет собой класс-композицию или «целое» (рис. 32).

Рисунок 32 – Обозначение отношения композиции

Применительно к классу Заказ_на_перевозку_грузов отношение композиции может иметь следующий вид (рис. 33).

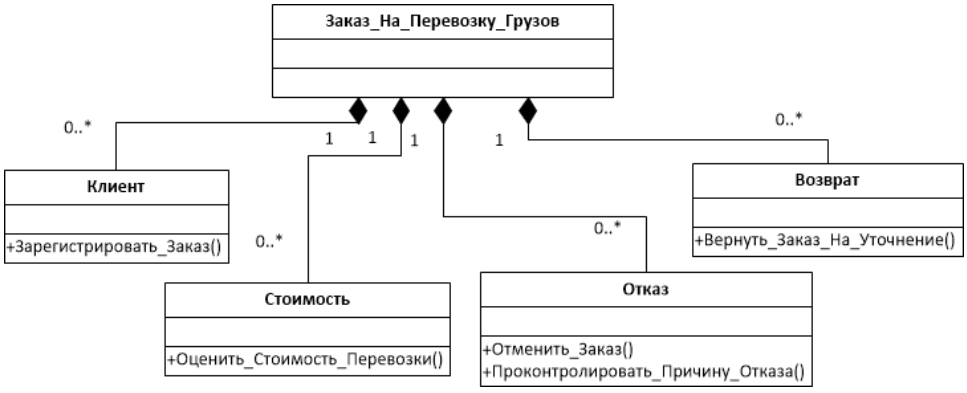


Рисунок 33 – Обозначение на диаграмме отношения композиции

5. Отношение обобщения.

Отношение обобщения (генерализация) является обычным таксономическим отношением между более общим элементом (класс-предок) и более частным или специальным элементом (класс-потомок).

Применительно к диаграмме классов данное отношение описывает иерархическое строение классов и наследование их свойств и поведения. При этом предполагается, что класс-потомок обладает всеми свойствами и поведением класса-предка, а также имеет свои собственные свойства и поведение, которые отсутствуют у класса-предка.

На диаграммах отношение обобщения обозначается сплошной линией с треугольной стрелкой на одном из концов, направленной на более общий класс (класс-предок или суперкласс) от более специального класса (класса-потомка или подкласса) (рис. 34).

Как правило, на диаграмме может указываться несколько линий для одного отношения обобщения, что отражает его таксономический характер. В этом случае более общий класс разбивается на подклассы одним отношением обобщения, например, так, как показано на рис. 14.

В этом случае данные отдельные линии изображаются сходящимися к единственной стрелке, имеющей с ними общую точку пересечения. Родительский Класс Отчет_По_Заказам_На_Перевозку имеет три потомка Отчет_По_Количеству_Заказов, Отчет_По_Клиентам, Отчет_За_Период, которые наследуют структуру и поведение родительского класса.

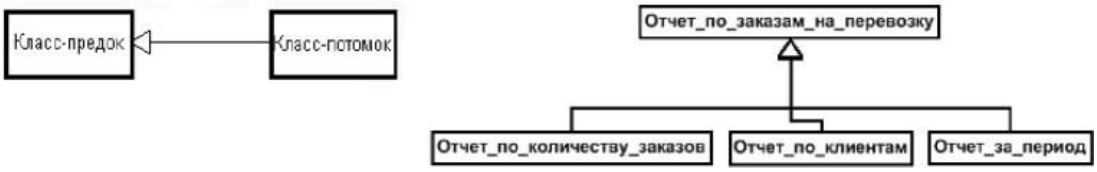


Рисунок 34 – Обозначение на диаграмме отношения обобщения

Для связей обобщения язык UML содержит ограничения. В большинстве случаев ограничение размещается рядом с элементом и заключается в фигурные скобки, например {complete}.

В качестве ограничений могут быть использованы следующие ключевые слова языка UML:

1. {complete} означает, что в данном отношении обобщения специфицированы все классы-потомки, и других классов-потомков у данного класса-предка быть не может.

Например, класс Клиент_банка является предком для двух классов: Физическое_лицо и Компания, и других классов-потомков он не имеет.

На соответствующей диаграмме классов это можно указать явно, записав рядом с линией обобщения данную строку-ограничение (рис. 35).

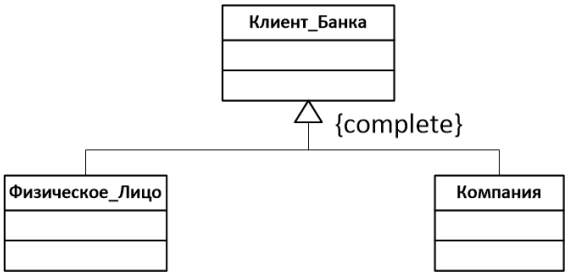


Рисунок 35 – Обозначение ограничения {complete} отношения обобщения

2. {incomplete} означает тот факт, что на диаграмме указаны в обобщении не все классы-потомки. В последующем, возможно, восполнить их перечень, не изменяя уже построенную диаграмму.

3. {disjoint} означает тот факт, что классы-потомки не могут содержать объектов, одновременно являющихся экземплярами двух или более классов.

В приведенном выше примере это условие также выполняется, поскольку предполагается, что никакое конкретное физическое лицо не может являться одновременно и конкретной компанией. В этом случае рядом с линией обобщения можно записать данную строку-ограничение.

4. {overlapping} означает, что отдельные экземпляры классов-потомков могут принадлежать одновременно нескольким классам.

Например, класс Транспорт может быть специализирован путем создания подклассов Наземный_Транспорт и Водный_Транспорт, автомобиль – амфибия относится к обоим классам.

Методика выполнения.

В качестве примера рассматривается моделирование системы продажи товаров по каталогу. 10.Запустите MS Visio.

1. На экране выбора шаблона выберите категорию Программы и БД и в ней элемент Схема модели UML.
2. Нажмите кнопку Создать в правой части экрана.
3. Окно программы примет вид, подобный рис. 36.

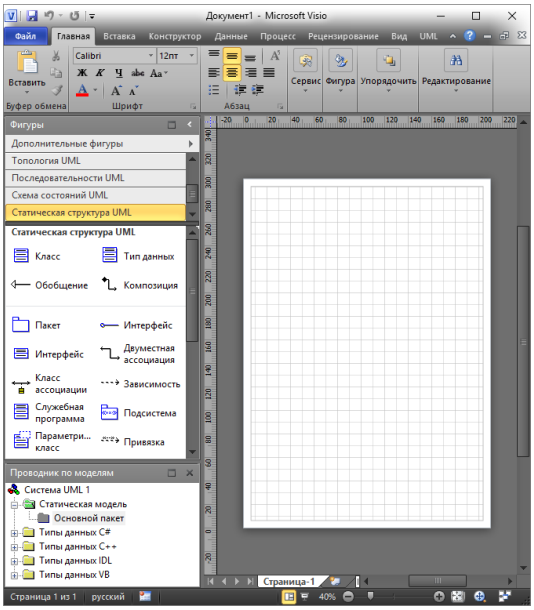


Рисунок 36 – Схема модели UML в MS Visio

4. Ознакомьтесь с элементами графического интерфейса и найдите обязательные панели инструментов Фигуры, содержащие категории Деятельность UML, Взаимодействия UML, Компоненты UML, Топология UML, Последовательности UML, Схема состояний UML, Статическая структура UML, Сценарий выполнения UML, Проводник по моделям, содержащий иерархическую структуру объектов Системы UML1, Рабочую область, ярлык Страница_1, горизонтальную и вертикальную линейки. (рис. 37).

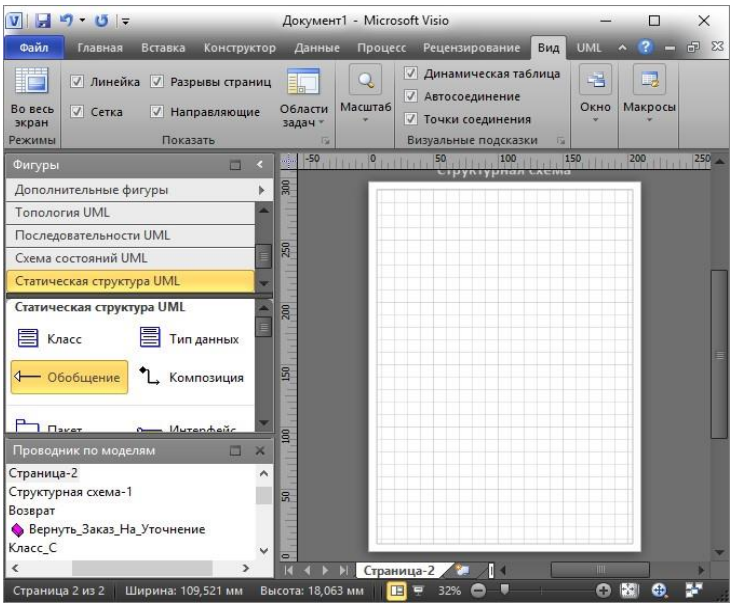


Рисунок 37 – Добавление фигур UML

5. Установите следующие параметры страницы: Ориентация – Альбомная, Автоподбор размера – выключен, Имя страницы – Диаграмма классов для системы продажи товаров по каталогу.

6. Перейдите в категорию Статическая структура UML, ознакомьтесь с содержимым этой категории и найдите элементы: Класс, Пакет, Подсистема, Интерфейс, Метакласс, Двусторонняя ассоциация, Обобщение, Композиция, Примечание, Ограничение и др.

7. Создайте поэтапно статическую структуру классов UML, с помощью которой может быть сформирована некоторая функциональная часть системы, например, *Система продажи товаров по каталогу*. Для чего:

– Выберите структурные элементы (идентифицируйте классы), участвующие в организации продаж, например, *Продавец, Товар, Заказ, Заказ_Оплата, Клиент, Корпоративный_Клиент, Частный_Клиент* и создайте предварительный вариант совокупности классов с указанием имен (один из возможных вариантов представлен нарис. 38).

– Установите для каждого класса атрибуты в соответствии с перечнем и содержательным описанием бизнес-процессов:

например, для класса *Продавец* в качестве атрибутов могут выступать данные: *фамилия, имя, отчество, телефон*. В данном случае все атрибуты видимы, принадлежат основному пакету *Продавец* (рис. 39). для класса *Товар* в качестве атрибутов могут выступать данные: тип, марка, артикул (рис. 40).

Рисунок 38 – Формирование статической структуры объектов диаграммы

Рисунок 39 – Описание атрибутов класса Продавец

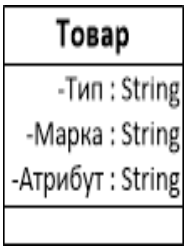


Рисунок 40 – Описание атрибутов класса Товар

Для класса *Заказ* в качестве атрибутов могут выступать данные: количество, цена, статус, а в качестве операций – сформировать заказ.

Рисунок 41 – Описание атрибутов и операций класса Заказ

Для класса *Заказ_Оплата* в качестве атрибутов могут выступать данные: дата получения, проплачен, номер, цена, а в качестве операций – отправить, закрыть.

Рисунок 42 – Описание атрибутов и операций класса Заказ_Оплата

Для класса *Клиент* в качестве атрибутов могут выступать данные: имя, адрес, а в качестве операций –кредитный рейтинг.

Рисунок 43 – Описание атрибутов и операций класса Клиент

Для класса *Корпоративный_Клиент* в качестве атрибутов могут выступать данные: контактное имя, кредитный рейтинг, кредитный лимит, а в качестве операций – сделать, напоминание, счет за месяц.

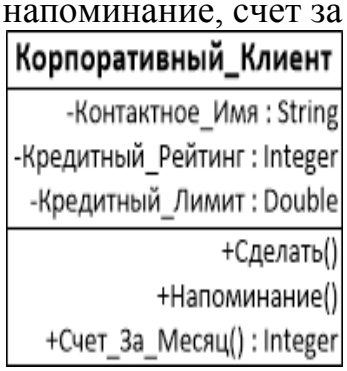


Рисунок 44 – Описание атрибутов и операций класс Корпоративный_Клиент

Для класса *Частный_Клиент* в качестве атрибутов могут выступать данные: номер кредитной карты.

Рисунок 45 – Описание атрибутов класса Частный_Клиент

Для класса *Вариант_Оплаты* в качестве атрибутов могут выступать данные: тип оплаты, а в качестве операций– выбор варианта оплаты.

Рисунок 46 – Описание атрибутов и операций класса Вариант_Оплаты

для класса *Каталог_Товаров* в качестве атрибутов могут выступать данные: тип, марка, артикул, а в качестве операций – проверить наличие.

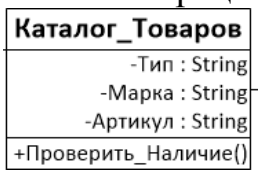


Рисунок 47 – Описание атрибутов и операций класса Каталог_товаров

Для класса *Склад* в качестве атрибутов могут выступать данные: товар, наличие, количество, а в качестве операций – Проверить наличие.

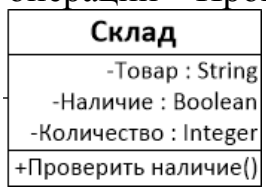


Рисунок 48 – Описание атрибутов и операций класса Склад

– Убедитесь, что все элементы наполнены адекватным содержанием и расположите все структурные элементы диаграммы наиболее оптимально на странице для установления отношений между ними.

В качестве примера на рис. 49 показан набор классов, описывающих реализацию системы продаж товаров по каталогу. Акцент сделан на классе *Клиент*, с которым связан класс *Заказ_Оплата* посредством двусторонней ассоциации «один-ко-многим», *Вариант_Оплаты* – двусторонней ассоциацией «один-к-одному» и классы *Корпоративный_Клиент* и *Частный_Клиент* посредством отношения обобщения. Классы *Заказ_Оплата* и *Товар* связаны с классом *Заказ* посредством двусторонней ассоциации «один-ко-многим». Класс *Товар* связан с классом *Продавец* двусторонней ассоциацией «многие-ко-многим» и классом *Каталог_Товаров* двусторонней ассоциацией «один-ко-многим». Класс *Каталог_Товаров* связан посредством двусторонней ассоциации «многие-ко-многим» с классом *Склад.*

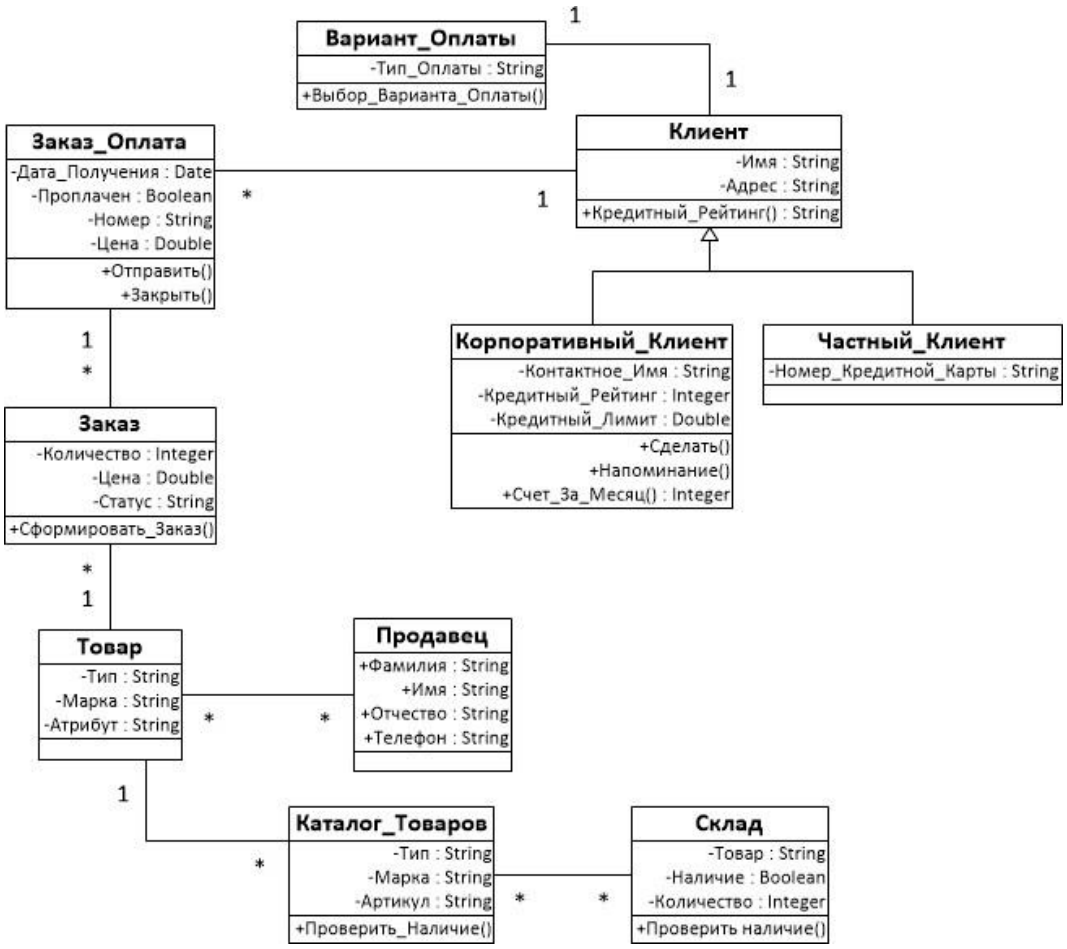


Рисунок 49 – Фрагмент диаграммы классов, описывающей реализацию систем продаж товаров по каталогу

8. Создайте новую страницу с именем Диаграмма классов учета клиентов, и установите следующие опции:

Ориентация – Альбомная, Автоподбор размера – выключен.

9. Идентифицируйте классы учета клиентов, осуществляющих заказы и создайте диаграмму классов с указанием их имен, атрибутов, операций, например, так как показано на рис. 50.

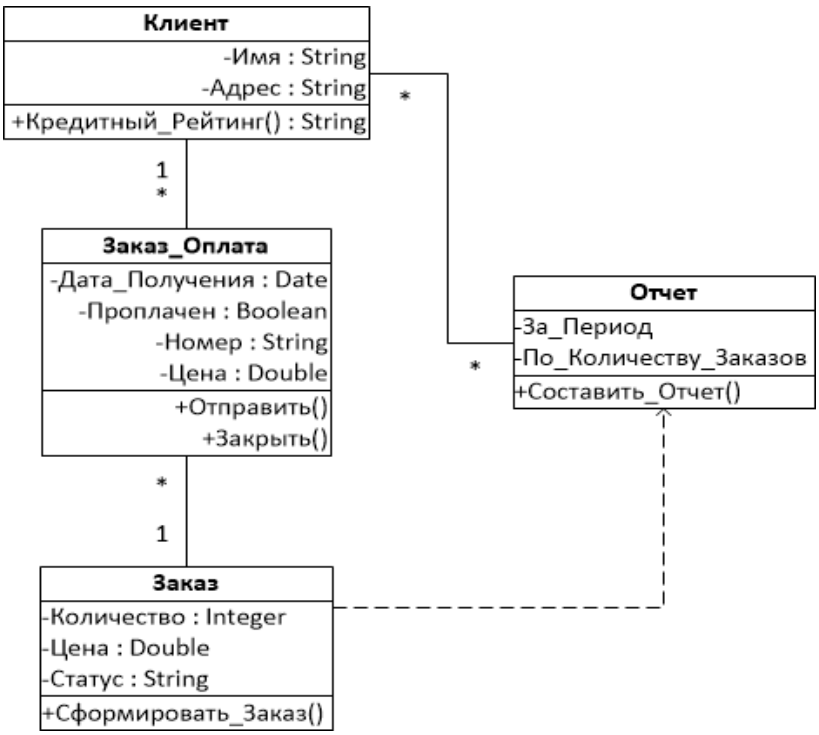


Рисунок 50 – Фрагмент диаграммы классов, описывающей процесс формирования отчетности

Задание.

Построить диаграмму классов в соответствии с вариантом.

Отчет по практическому занятию выполняется в формате MS Word, который содержит пошаговое описание процесса построения диаграммы, а также скриншоты результатов согласно заданию.

Варианты.

1. «Отдел кадров»;
2. «Агентство аренды»;
3. «Аптека»;
4. «Ателье»;
5. «Аэропорт»;
6. «Библиотека»;
7. «Кинотеатр»;
8. «Поликлиника»;
9. «Автосалон»;
10. «Таксопарк».

Контрольные вопросы.

1. Каково назначение диаграммы классов?
2. Назовите основные элементы диаграммы классов.
3. Какие виды связей доступны в диаграмме классов?
4. Для чего используется каждый вид связи?
5. Как создать диаграмму классов в VISIO?

Практическая работа №7 «Построение диаграммы Деятельности, диаграммы Состояний и диаграммы Классов»

Цель работы:

Целью работы является изучение основ создания диаграмм состояний на языке UML, получение навыков построения диаграмм состояний, применение приобретенных навыков для построения объектно-ориентированных моделей определенной предметной области.

Задачи:

Основными задачами практической работы являются:

– ознакомиться с теоретическими вопросами построения диаграмм состояний на языке UML;

– ознакомиться с теоретическими вопросами построения диаграмм состояний с помощью MS Visio.

Краткие теоретические сведения.

Диаграмма состояний описывает процесс изменения состояний только одного класса, а точнее – одного экземпляра определенного класса, т. е. моделирует все возможные изменения в состоянии конкретного объекта. При этом изменение состояния объекта может быть вызвано внешними воздействиями со стороны других объектов или извне. Именно для описания реакции объекта на подобные внешние воздействия и используются диаграммы состояний.

Диаграмма состояний описывает возможные последовательности состояний и переходов, которые в совокупности характеризуют поведение элемента модели в течение его жизненного цикла. Диаграмма состояний представляет динамическое поведение сущностей, на основе спецификации их реакции на восприятие некоторых конкретных событий.

Хотя диаграммы состояний чаще всего используются для описания поведения отдельных экземпляров классов (объектов), но они также могут быть применены для спецификации функциональности других компонентов моделей, таких как варианты использования, актеры, подсистемы, операции и методы.

Диаграмма состояний по существу является графом специального вида, который представляет некоторый автомат. Понятие автомата в контексте UML обладает довольно специфической семантикой, основанной на теории автоматов. Вершинами этого графа являются состояния и некоторые другие типы элементов автомата (псевдосостояния), которые изображаются соответствующими графическими символами. Дуги графа служат для обозначения переходов из состояния в состояние. Диаграммы состояний могут быть вложены друг в друга, образуя вложенные диаграммы более детального представления отдельных элементов модели.

Рисунок 51 – Пример диаграммы состояний для технического устройства типа «Компьютер»

Состояние

Под состоянием понимается абстрактный метакласс, используемый для моделирования отдельной ситуации, в течение которой имеет место выполнение некоторого условия. Состояние может быть задано в виде набора конкретных значений атрибутов класса или объекта, при этом изменение их отдельных значений будет отражать изменение состояния моделируемого класса или объекта.

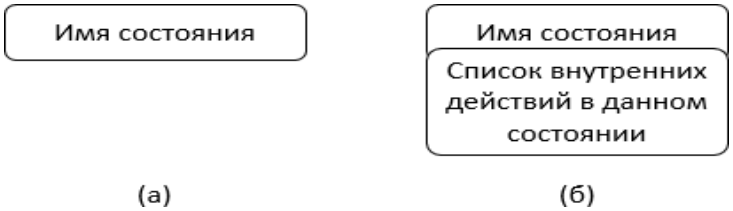


Рисунок 52 – Графическое обозначение состояния

Секция «Список внутренних действий» содержит перечень внутренних действий или деятельностей, которые выполняются в процессе нахождения моделируемого элемента в данном состоянии. Каждое из действий записывается в виде отдельной строки и имеет следующий формат:

<метка-действия '/' выражение-действия>

Метка действия указывает на обстоятельства или условия, при которых будет выполняться деятельность, определенная выражением действия:

– ***entry*** – эта метка указывает на действие, специфицированное следующим за ней выражением действия, которое выполняется в момент входа в данное состояние (входное действие);

– ***exit*** – эта метка указывает на действие, специфицированное следующим за ней выражением действия, которое выполняется в момент выхода из данного состояния (выходное действие);

– ***do*** – эта метка специфицирует выполняющуюся деятельность («doactivity»), которая выполняется в течение всего времени, пока объект находится в данном состоянии, или до тех пор, пока не закончится вычисление, специфицированное следующим за ней выражением действия. В последнем случае при завершении события генерируется соответствующий результат;

– ***include*** – эта метка используется для обращения к подавтомату, при этом следующее за ней выражение действия содержит имя этого подавтомата.

Пример: Аутентификация входа.

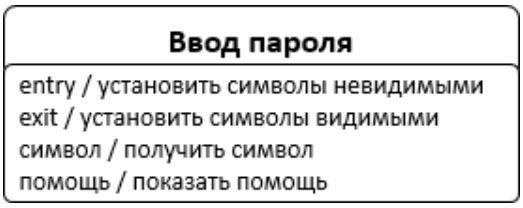


Рисунок 53 – Состояние для ввода пароля

Начальное и конечное состояния.

Начальное состояние представляет собой частный случай состояния, которое не содержит никаких внутренних действий (псевдосостояния). В этом состоянии находится объект по умолчанию в начальный момент времени. Оно служит для указания на диаграмме состояний графической области, от которой начинается процесс изменения состояний.

Конечное (финальное) состояние представляет собой частный случай состояния, которое также не содержит никаких внутренних действий (псевдосостояния). В этом состоянии будет находиться объект по умолчанию послезавершения работы автомата в конечный момент времени.

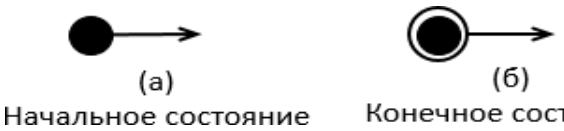


Рисунок 54 – Графическое изображение начального и конечного состояний

Переход.

Простой переход (simpletransition) представляет собой отношение между двумя последовательными состояниями, которое указывает на факт смены одного состояния другим. Пребывание моделируемого объекта в первом состоянии может сопровождаться выполнением некоторых действий, а переход во второе состояние будет возможен после завершения этих действий, а также после удовлетворения некоторых дополнительных условий. В этом случае говорят, что переход срабатывает, Или происходит срабатывание перехода. До срабатывания перехода объект находится в предыдущем от него состоянии, называемым исходным состоянием, или в источнике (не путать с начальным состоянием – это разные понятия), а после его срабатывания объект находится в последующем от него состоянии (целевом состоянии).

Переход осуществляется при наступлении некоторого события: окончания выполнения деятельности (doactivity), получении объектом сообщения или приемом сигнала. На переходе указывается имя события. Кроме того, на переходе могут указываться действия, производимые объектом в ответ на внешние события при переходе из одного состояния в другое. Срабатывание перехода может зависеть не только от наступления некоторого события, но и от выполнения определенного условия, называемого сторожевым условием. Объект перейдет из одного состояния в другое в том случае, если произошло указанное событие и сторожевое

условие приняло значение «истина».

На диаграмме состояний ***переход изображается*** сплошной линией со стрелкой, которая направлена в целевое состояние. Каждый переход может помечен строкой текста, которая имеет следующий общий формат:

<сигнатура события>'['<сторожевое условие>']' <выражение действия>

Событие.

Событие представляет собой спецификацию некоторого факта, имеющего место в пространстве и во времени. Про события говорят, что они «происходят», при этом отдельные события должны быть упорядочены во времени. После наступления некоторого события нельзя уже вернуться к предыдущим событиям, если такая возможность не предусмотрена явно в модели.

События играют роль стимулов, которые инициируют переходы из одних состояний в другие. В качестве событий можно рассматривать сигналы, вызовы, окончание фиксированных промежутков времени или моменты окончания выполнения определенных действий. Имя события идентифицирует каждый отдельный переход на диаграмме состояний и может содержать строку текста, начинающуюся со строчной буквы.

Сторожевое условие.

Сторожевое условие (guardcondition), если оно есть, всегда записывается в прямых скобках после события и представляет собой некоторое булевское выражение.

Если сторожевое условие принимает значение «истина», то соответствующий переход может сработать, в результате чего объект перейдет в целевое состояние. Если же сторожевое условие принимает значение «ложь», то переход не может сработать, и при отсутствии других переходов объект не может перейти в целевое состояние по этому переходу. Однако вычисление истинности сторожевого условия происходит только после возникновения ассоциированного с ним события, инициирующего соответствующий переход.

В общем случае из одного состояния может быть несколько переходов с одним и тем же событием-триггером. При этом никакие два сторожевых условия не должны одновременно принимать значение «истина». Каждое из сторожевых условий необходимо вычислять всякий раз при наступлении соответствующего события.

Пример

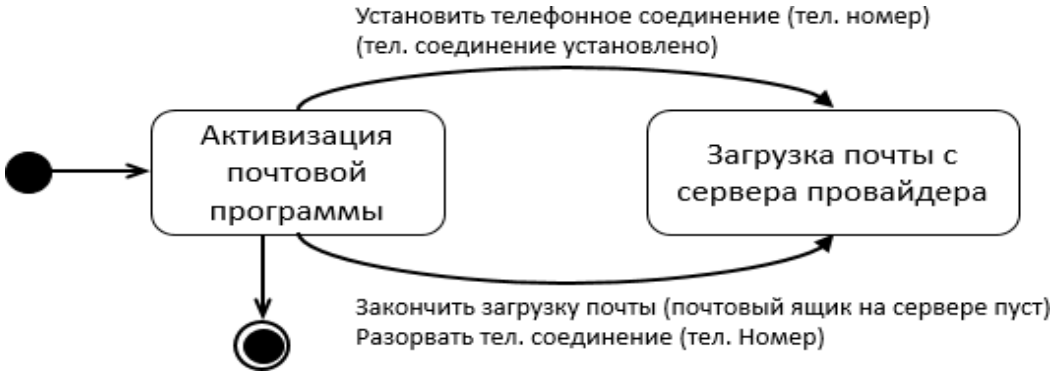


Рисунок 55 – Диаграмма состояний для моделирования почтовой программы клиента

Составное состояние и подсостояние.

Составное состояние (compositestate) – такое сложное состояние, которое состоит из других вложенных в него состояний. Последние будут выступать по отношению к первому как подсостояния (substate).

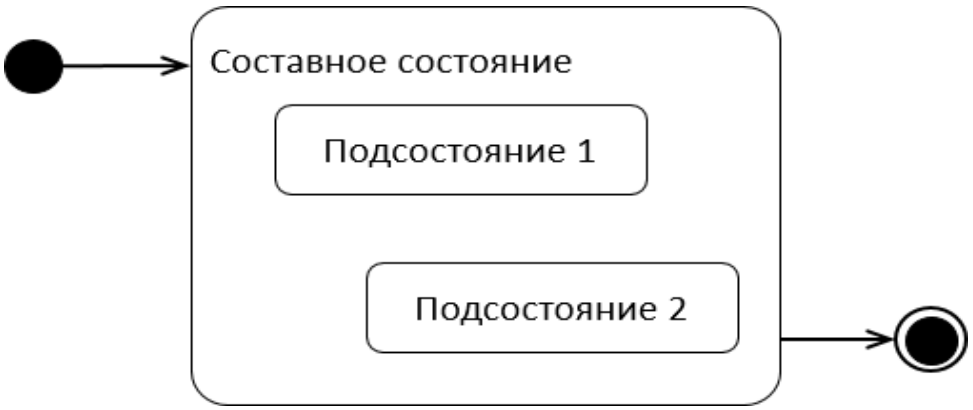


Рисунок 56 – Графическое изображение составного состояния

Последовательные подсостояния (sequential substates) используются для моделирования такого поведения объекта, во время которого в каждый момент времени объект может находиться в одном и только одном подсостояний. Поведение объекта в этом случае представляет собой последовательную смену подсостояний, начиная от начального и заканчивая конечным подсостояниями. Хотя объект продолжает находиться в составном состоянии, введение в рассмотрение последовательных подсостояний позволяет учесть более тонкие логические аспекты его внутреннего поведения.

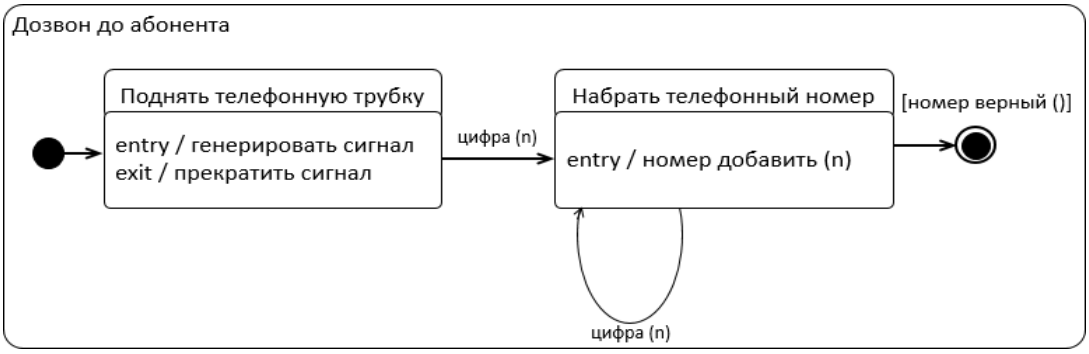


Рисунок 57 – Пример составного состояния с двумя вложенными последовательными подсостояниями

Параллельные подсостояния (concurrentsubstates) позволяют специфицировать два и более подавтомата, которые могут выполняться параллельно внутри составного события. Каждый из подавтоматов занимает некоторую область (регион) внутри составного состояния, которая отделяется от остальных горизонтальной пунктирной линией. Если на диаграмме состояний имеется составное состояние с вложенными параллельными подсостояниями, то объект может одновременно находиться в каждом из этих подсостояний.

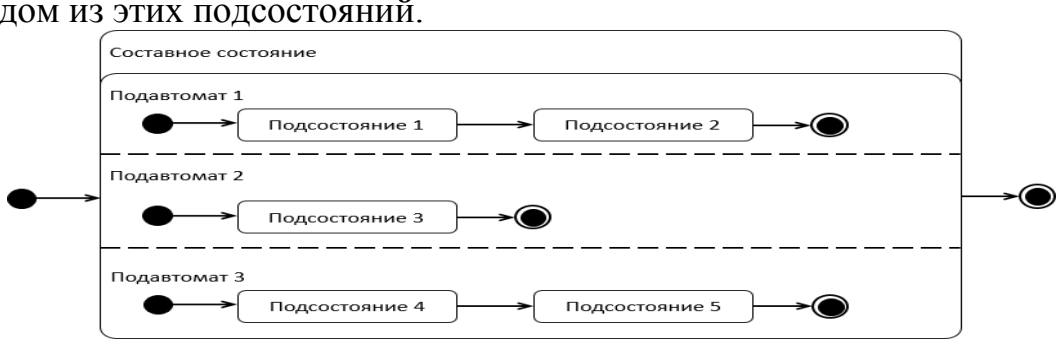


Рисунок 58 – Графическое изображение составного состояния с вложенными параллельными подсостояниями

Синхронизирующие состояния.

Поведение параллельных подавтоматов независимо друг от друга, что позволяет реализовать многозадачность в программных системах. Однако в отдельных случаях может возникнуть необходимость учесть в модели синхронизацию наступления отдельных событий. Для этой цели в языке UML имеется специальное псевдосостояние, которое называется синхронизирующим состоянием.

Синхронизирующее состояние (synch state) обозначается небольшой окружностью, внутри которой помещен символ звездочки "*". Оно используется совместно с переходом-соединением или переходом-ветвлением для того, чтобы явно указать события в других подавтоматах, оказывающие непосредственное влияние на поведение данного подавтомата.

Для иллюстрации использования синхронизирующих состояний рассмотрим упрощенную ситуацию с моделированием процесса постройки дома. Предположим, что постройка дома включает в себя строительные работы (возведение фундамента и стен, возведение крыши и отделочные работы) и работы по электрификации дома (подведение электрической линии, прокладка скрытой электропроводки и установка осветительных ламп). Очевидно, два этих комплекса работ могут выполняться параллельно, однако между ними есть некоторая взаимосвязь.

В частности, прокладка скрытой электропроводки может начаться лишь после того, как будет завершено возведение фундамента и стен. А отделочные работы следует начать лишь после того, как будет закончена прокладка скрытой электропроводки. В противном случае отделочные работы придется проводить повторно. Рассмотренные особенности синхронизации этих параллельных процессов учтены на соответствующей диаграммесостояний с помощью двух синхронизирующих состояний.

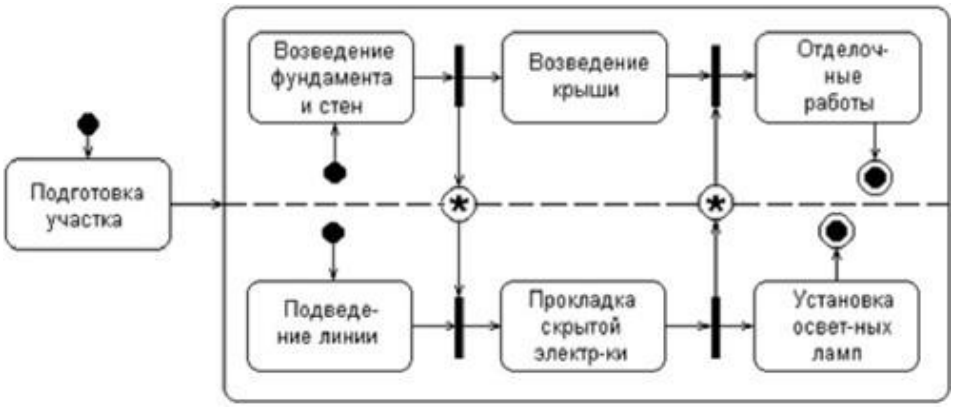


Рисунок 59 – Диаграмма состояний для примера со строительством дома

Примеры диаграмм состояний изображены на рисунках 60-62.

Рисунок 60 – Диаграмма состояний для моделирования запроса данных из БД

Рисунок 61 – Диаграмма состояний для моделирования поведения банкомата

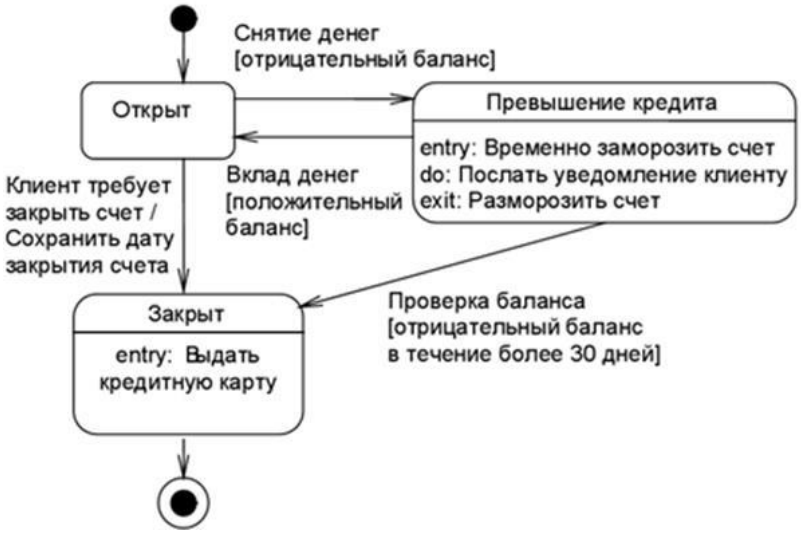


Рисунок 62 – Диаграмма состояний моделирования деятельности сотрудника банка

1. Методика выполнения

В качестве примера рассматривается моделирование системы продажи товаров по каталогу.

Запустите MS Visio.

– На экране выбора шаблона выберите категорию *Программы и БД* и в ней элемент *Схема модели UML*.

Нажмите кнопку *Создать* в правой части экрана.

– Окно программы примет вид, подобный рис. 63.

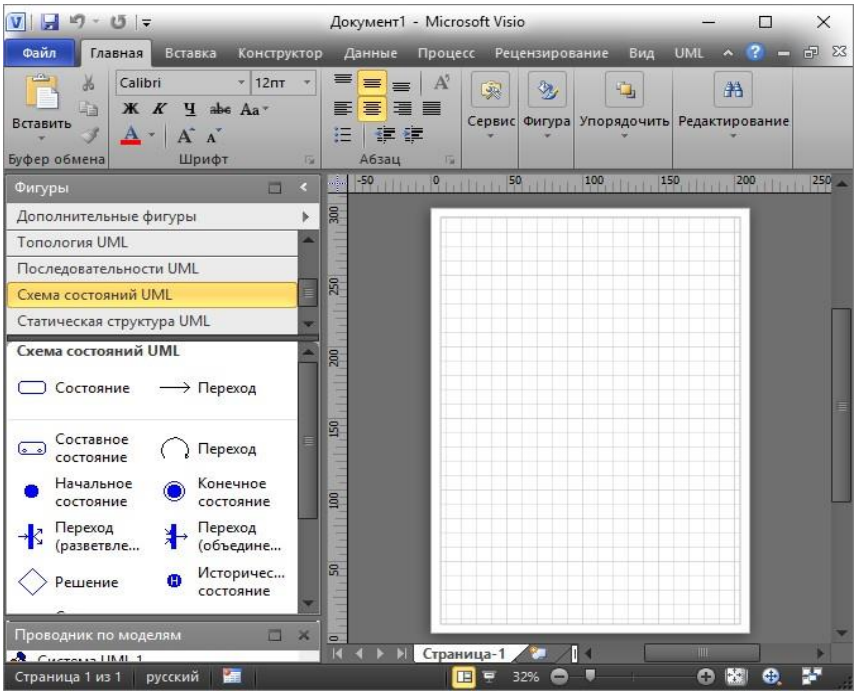


Рисунок 63 – Окно MS Visio для работы с диаграммами состояний

Клиент оформляет заказ. Класс Заказ имеет атрибут Статус. Проследим динамику движения заказов в системе с помощью ***диаграммы состояний, составленной для класса Заказ.***

Данные о сформированном заказе поступают продавцу, который проверяет наличие товаров из заказа, проверяет оплату заказа, комплектует его и делает отметку о готовности. После оплаты заказа он выдается клиенту. Продавец делает отметку о том, что заказ выдан.

Если после проверки кредитного рейтинга клиента, он окажется отрицательным, то заказ будет отклонен.

Построим диаграмму состояний для класса Заказ. Для этого, в файле с диаграммой классов, созданной в практическом занятии 8, необходимо проделать следующие действия:

1. Щелкнуть правой кнопкой мыши по *классу Заказ*.
2. В контекстном меню выбрать пункт *Схемы*.
3. Т.к. в настоящее время уже созданных схем нет, нажать кнопку *Создать* и выбрать *Схема состояний*.
4. Переименовать созданный лист в *Схема состояний-Заказ*.
5. Построить диаграмму состояний для класса Заказ в соответствии с рисунком 64.

После формирования заказа он должен быть оплачен. Обработка заказа подразумевает проверку наличия товара и проверку оплаты. Переход в одно из состояний На комплектации, Укомплектован, Выдан означает смену Статуса заказа.

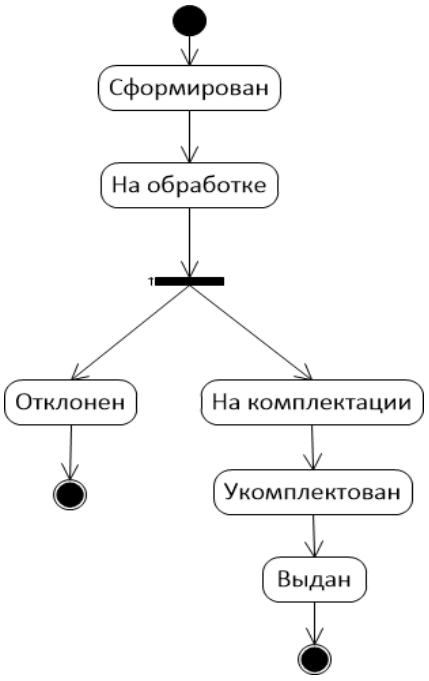


Рисунок 145 – Диаграмма состояний для класса Заказ

Далее опишем с помощью ***диаграммы состояний процесс оплаты заказа клиентом,*** которому соответствуеткласс ЗаказОплата.

Построим диаграмму состояний для проверки оплаты заказа.

Чтобы проверить оплату заказа, необходимо определить, существует ли сам заказ. Результатом проверки оплаты заказа является вывод либо сообщения о произведенной оплате с параметрами (дата оплаты), либо сообщения об ожидании оплаты.

Событием, предшествующим проверке оплаты заказа, является занесение информации о заказе в базу данных заказов.

Чтобы построить диаграмму состояний для класса ЗаказОплаты, необходимо проделать действия, описанные в пунктах 1-4 построения диаграммы состояний для класса Заказ. Полученная диаграмма должна иметь вид, изображенный на рис. 65.

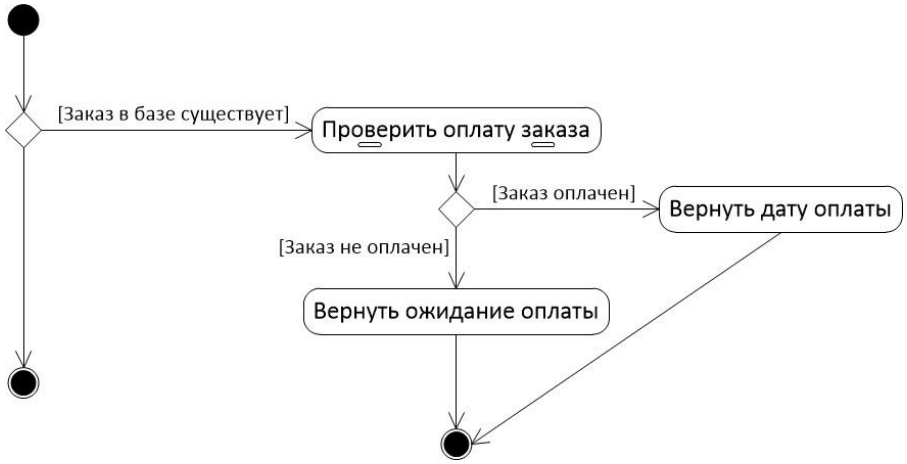


Рисунок 65 – Диаграмма состояний проверки платы заказа

На этой диаграмме есть составное состояние ***«Проверить оплату заказа»***, т.к. оно включает в себя *проверку кредитного рейтинга клиента* и *проверку выбора варианта оплаты клиентом*.

Оплату заказа может произвести только клиент с положительным кредитным рейтингом, поэтому необходимым условием проверки оплаты заказа является проверка кредитоспособности клиента. Если клиент имеет отрицательный кредитный рейтинг, то заказ отклоняется, и на этом дальнейшие события не имеют смысла. Если кредитный рейтинг клиента положительный, то необходимо проверить, выбрал ли клиент вариант оплаты. Событие, которое переводит систему в состояние ожидания выбора варианта оплаты клиентом, является

получение сообщения о кредитоспособности клиента.

Оплата может быть произведена наличными средствами в магазине или с помощью безналичного расчета. В первом случае необходимо договориться с клиентом о дате и времени его прибытия в магазин. Во втором случае необходимо сообщить клиенту о наличии/поступлении товара. Событие, которое переводит систему в состояние ожидания оплаты, является выбор клиентом варианта оплаты.

Соответствующие диаграммы состояний имеют вид (рис. 66 и рис. 67).

Для создания диаграмм состояний, которые входят в состав составного состояния, нужно:

1. Щелкнуть правой кнопкой мыши по Составному состоянию и выбрать пункт Схема.

2. Либо, в Проводнике по моделям выделить название составного состояния и создать новую страницу с помощью кнопки  .

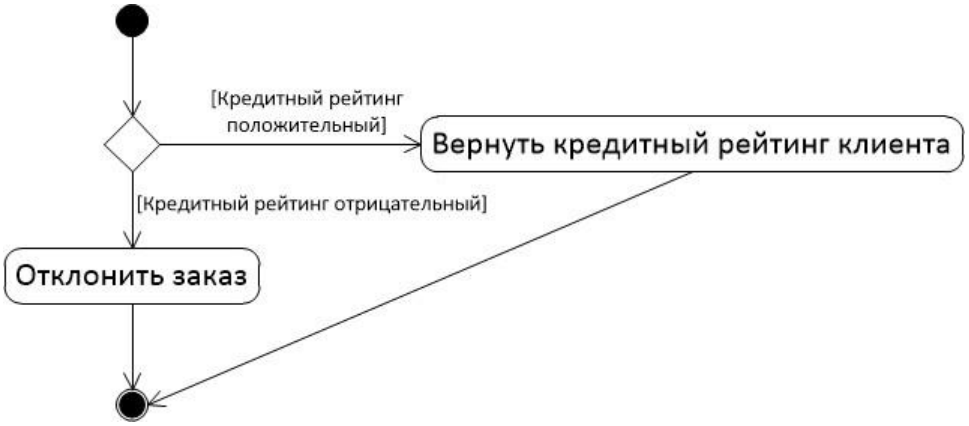


Рисунок 66 – Диаграмма состояний для проверки кредитного рейтинга клиента

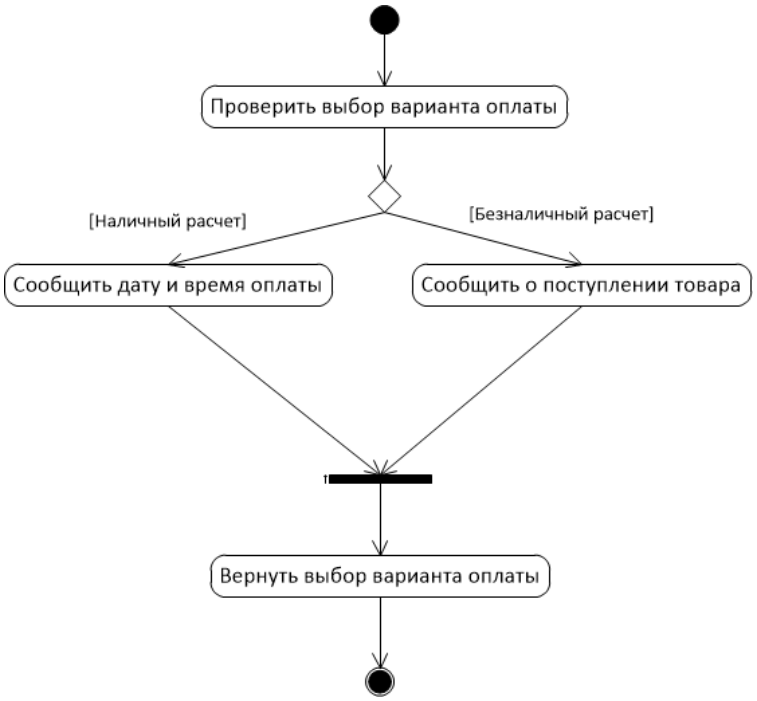


Рисунок 67 – Диаграмма состояний для проверки варианта оплаты

Задание.

Построить диаграмму состояний в соответствии с вариантом.

Отчет по практическому занятию выполняется в формате MS Word, который содержит пошаговое описание процесса построения диаграммы, а также скриншоты результатов согласно заданию.

Создать диаграммы состояния не менее чем для трех классов, описанных в практическом занятии №8.

Варианты.

1. «Отдел кадров»;
2. «Агентство аренды»;
3. «Аптека»;
4. «Ателье»;
5. «Аэропорт»;

6. «Библиотека»;
7. «Кинотеатр»;
8. «Поликлиника»;
9. «Автосалон»;10.«Таксопарк».

Контрольные вопросы.

1. Каково назначение диаграммы состояний?
2. Назовите основные элементы диаграммы состояний.
3. Как создать диаграмму состояний в VISIO?
4. В чем отличие диаграммы классов и состояний?

Практическая работа №8 «Построение диаграммы компонентов»

Цель работы:

Целью работы является изучение основ создания диаграмм деятельности на языке UML, получение навыков построения диаграмм деятельности, применение приобретенных навыков для построения объектно- ориентированных моделей определенной предметной области.

Задачи:

Основными задачами практической работы являются:

– ознакомиться с теоретическими вопросами построения диаграмм деятельности на языке UML;

– ознакомиться с теоретическими вопросами построения диаграмм деятельности с помощью MS Visio.

Краткие теоретические сведения:

При моделировании поведения проектируемой или анализируемой системы возникает необходимость не только представить процесс изменения ее состояний, но и детализировать особенности алгоритмической и логической реализации выполняемых системой операций. Традиционно для этой цели использовались блок-схемы или структурные схемы алгоритмов. Каждая такая схема акцентирует внимание на последовательности выполнения определенных действий или элементарных операций, которые в совокупности приводят к получению желаемого результата.

Алгоритмические и логические операции, требующие выполнения в определенной последовательности, окружают нас постоянно. Например, чтобы позвонить по телефону, нам предварительно нужно снять трубку или включить его. Для приготовления кофе или заваривания чая необходимо вначале вскипятить воду. Чтобы выполнить ремонт двигателя автомобиля, требуется осуществить целый ряд нетривиальных операций, таких как разборка силового агрегата, снятие генератора и некоторых других.

С увеличением сложности системы строгое соблюдение последовательности выполняемых операций приобретает все более важное значение. Если попытаться заварить кофе холодной водой, то мы можем только испортить одну порцию напитка. Нарушение последовательности операций при ремонте двигателя может привести к его поломке или выходу из строя. Еще более катастрофические последствия могут произойти в случае отклонения от установленной последовательности действий при взлете или посадке авиалайнера, запуске ракеты, регламентных работах на АЭС.

Для моделирования процесса выполнения операций в языке UML используются так называемые **диаграммы деятельности**. Применяемая в них графическая нотация во многом похожа на нотацию диаграммы состояний, поскольку на диаграммах деятельности также присутствуют обозначения состояний и переходов. Отличие заключается в семантике состояний, которые используются для представления не деятельностей, а действий, и в отсутствии на переходах сигнатуры событий. Каждое состояние на диаграмме деятельности соответствует выполнению некоторой

элементарной операции, а переход в следующее состояние срабатывает только при завершении этой операции в предыдущем состоянии. Графически диаграмма деятельности представляется в форме графа деятельности, вершинами которого являются состояния действия, а дугами – переходы от одного состояния действия к другому.

Таким образом, диаграммы деятельности можно считать частным случаем диаграмм состояний. Основным направлением использования диаграмм деятельности является визуализация особенностей реализации операций классов, когда необходимо представить алгоритмы их выполнения. При этом каждое состояние может являться выполнением операции некоторого класса либо ее части, позволяя использовать диаграммы деятельности для описания реакций на внутренние события системы.

В контексте языка UML **деятельность (activity)** представляет собой некоторую совокупность отдельных вычислений, выполняемых автоматом. При этом отдельные элементарные вычисления могут приводить к некоторому результату или действию (action). На диаграмме деятельности отображается логика или последовательность перехода от одной деятельности к другой, при этом внимание фиксируется на результате деятельности. Сам же результат может привести к изменению состояния системы или возвращению некоторого значения.

Состояние действия и деятельности.

Состояние деятельности (activity state) – состояние в графе деятельности, которое служит для представления процедурной последовательности действий, требующих определенного времени. Переход из состояния деятельности происходит после выполнения специфицированной в нем ду-деятельности, при этом ключевое слово do в имени деятельности не указывается. Состояние деятельности не может иметь внутренних переходов, поскольку оно является элементарным.

Состояния деятельности могут быть подвергнуты дальнейшей декомпозиции, вследствие чего выполняемую деятельность можно представить с помощью других диаграмм деятельности. Состояния деятельности не являются атомарными, то есть могут быть прерваны. Предполагается, что для их завершения требуется заметное время.

Состояние деятельности можно представлять себе, как составное состояние, поток управления которого включает только другие состояния деятельности и действий.

Состояние действия (action state) является специальным случаем состояния с некоторым входным действием и, по крайней мере, одним выходящим из состояния переходом. Этот переход неявно предполагает, что входное действие уже завершилось. Состояние действия не может иметь внутренних переходов, поскольку оно является элементарным. Обычное использование состояния действия заключается в моделировании одного шага выполнения алгоритма (процедуры) или потока управления.

Графически состояние действия изображается фигурой, напоминающей прямоугольник, боковые стороны которого заменены выпуклыми дугами

(рис. 68). Внутри этой фигуры записывается выражение действия (action-expression), которое должно быть уникальным в пределах одной диаграммы деятельности.

Рисунок 68 – Графическое обозначение состояния действия

Действие может быть записано на естественном языке, некотором псевдокоде или языке программирования. Никаких дополнительных или неявных ограничений при записи действий не накладывается. Рекомендуется в качестве имени простого действия использовать глагол с пояснительными словами (рис. 68а). Если же действие может быть представлено в некотором формальном виде, то целесообразно записать его на том языке программирования, на котором предполагается реализовывать конкретный проект (рис. 68б).

Иногда возникает необходимость представить на диаграмме деятельности некоторое сложное действие, которое, в свою очередь, состоит из нескольких более простых действий. В этом случае можно использовать специальное обозначение так называемого **состояния поддеятельности (subactivity state)**. Такое состояние является графом деятельности и обозначается специальной пиктограммой в правом нижнем углу символа состояния действия (рис. 69). Эта конструкция может применяться к любому элементу языка UML, который поддерживает «вложенность» своей структуры. При этом пиктограмма может быть дополнительно помечена типом вложенной структуры.

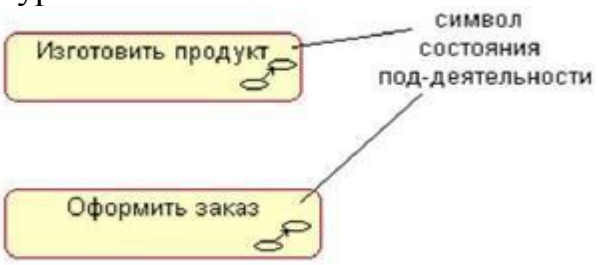


Рисунок 69 – Графическое обозначение состояния поддеятельности

Каждая диаграмма деятельности должна иметь единственное начальное и единственное конечное состояния. Они имеют такие же обозначения, как и на диаграмме состояний (см. практическое занятия №9). При этом каждая деятельность начинается в начальном состоянии и заканчивается в конечном состоянии. Саму диаграмму деятельности принято располагать таким образом, чтобы действия следовали сверху вниз. В этом случае начальное состояние будет изображаться в верхней части диаграммы, а конечное – в ее нижней части.

Переходы

При построении диаграммы деятельности используются только такие переходы, которые срабатывают сразу после завершения деятельности или

выполнения соответствующего действия. Этот переход переводит деятельность в последующее состояние сразу, как только закончится действие в предыдущем состоянии. На диаграмме такой переход изображается сплошной линией со стрелкой.

Если из состояния действия выходит единственный переход, то он может быть никак не помечен. Если же таких переходов несколько, то сработать может только один из них. Именно в этом случае для каждого из таких переходов должно быть явно записано сторожевое условие в прямых скобках. При этом для всех выходящих из некоторого состояния переходов должно выполняться требование истинности только одного из них. Подобный случай встречается тогда, когда последовательно выполняемая деятельность должна разделиться на альтернативные ветви в зависимости от значения некоторого промежуточного результата. Такая ситуация получила название ветвления, а для ее обозначения применяется специальный символ

Графически ветвление на диаграмме деятельности обозначается небольшим ромбом, внутри которого нет никакого текста (рис. 3). В этот ромб может входить только одна стрелка от того состояния действия, после выполнения которого поток управления должен быть продолжен по одной из взаимно исключающих ветвей. Принято входящую стрелку присоединять к верхней или левой вершине символа ветвления. Выходящих стрелок может быть две или более, но для каждой из них явно указывается соответствующее сторожевое условие в формебулевского выражения.

В качестве **примера** рассмотрим фрагмент известного алгоритма нахождения корней квадратного уравнения. В общем случае после приведения уравнения второй степени к каноническому виду: $a * x * x + b * x + c = 0$ необходимо вычислить его дискриминант. Причем, в случае отрицательного дискриминанта уравнение не имеет решения на множестве действительных чисел, и дальнейшие вычисления должны быть прекращены. При неотрицательном дискриминанте уравнение имеет решение, корни которого могут быть получены на основе конкретной расчетной формулы.

Графически фрагмент процедуры вычисления корней квадратного уравнения может быть представлен в виде диаграммы деятельности с тремя состояниями действия и ветвлением (рис. 70). Каждый из переходов, выходящих из состояния «Вычислить дискриминант», имеет сторожевое условие, определяющее единственную ветвь, по которой может быть продолжен процесс вычисления корней в зависимости от знака дискриминанта. Очевидно, что в случае его отрицательности, мы сразу попадаем в конечное состояние, тем самым завершая выполнение алгоритма в целом.

Рисунок 70 – Фрагмент диаграммы деятельности для алгоритма нахождения корней квадратного уравнения

В рассмотренном примере, как видно из рис. 70, выполняемые действия соединяются в конечном состоянии.

Однако это вовсе не является обязательным. Можно изобразить еще один символ ветвления, который будет иметь

несколько входящих переходов и один выходящий.

Один из наиболее значимых недостатков обычных блок-схем или структурных схем алгоритмов связан с проблемой изображения параллельных ветвей отдельных вычислений. Поскольку распараллеливание вычислений существенно повышает общее быстродействие программных систем, необходимы графические примитивы для представления параллельных процессов. В языке UML для этой цели используется специальный символ для разделения и слияния параллельных вычислений или потоков управления. Таким символом является **прямая черточка.**

Такая черточка изображается отрезком горизонтальной линии, толщина которой несколько шире основных сплошных линий диаграммы деятельности. При этом разделение (concurrent fork) имеет один входящий переход и несколько выходящих (рис. 71а). Слияние (concurrent join), наоборот, имеет несколько входящих переходов и один выходящий (рис. 71б).

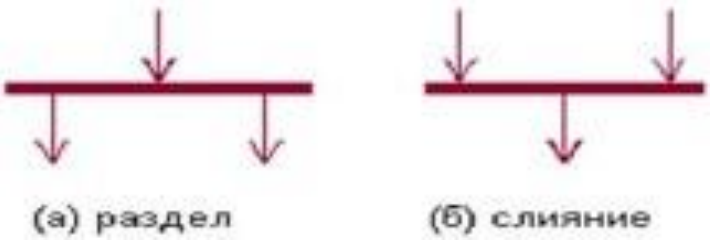


Рисунок 71 – Графическое обозначение разделения и слияния параллельных потоков управления

Для иллюстрации особенностей параллельных процессов выполнения действий рассмотрим **пример** с приготовлением напитка. Достоинство этого примера состоит в том, что он практически не требует никаких дополнительных пояснений в силу своей очевидности (рис. 72).

Хотя диаграмма деятельности предназначена для моделирования поведения систем, время в явном виде отсутствует на этой диаграмме. Ситуация здесь во многом аналогична диаграмме состояний.

Таким образом, диаграмма деятельности есть не что иное, как специальный случай диаграммы состояний, в которой все или большинство состояний являются действиями или состояниями поддеятельности. А все или большинство переходов являются переходами, которые срабатывают по завершении действий или под- деятельностей в состояниях источниках.

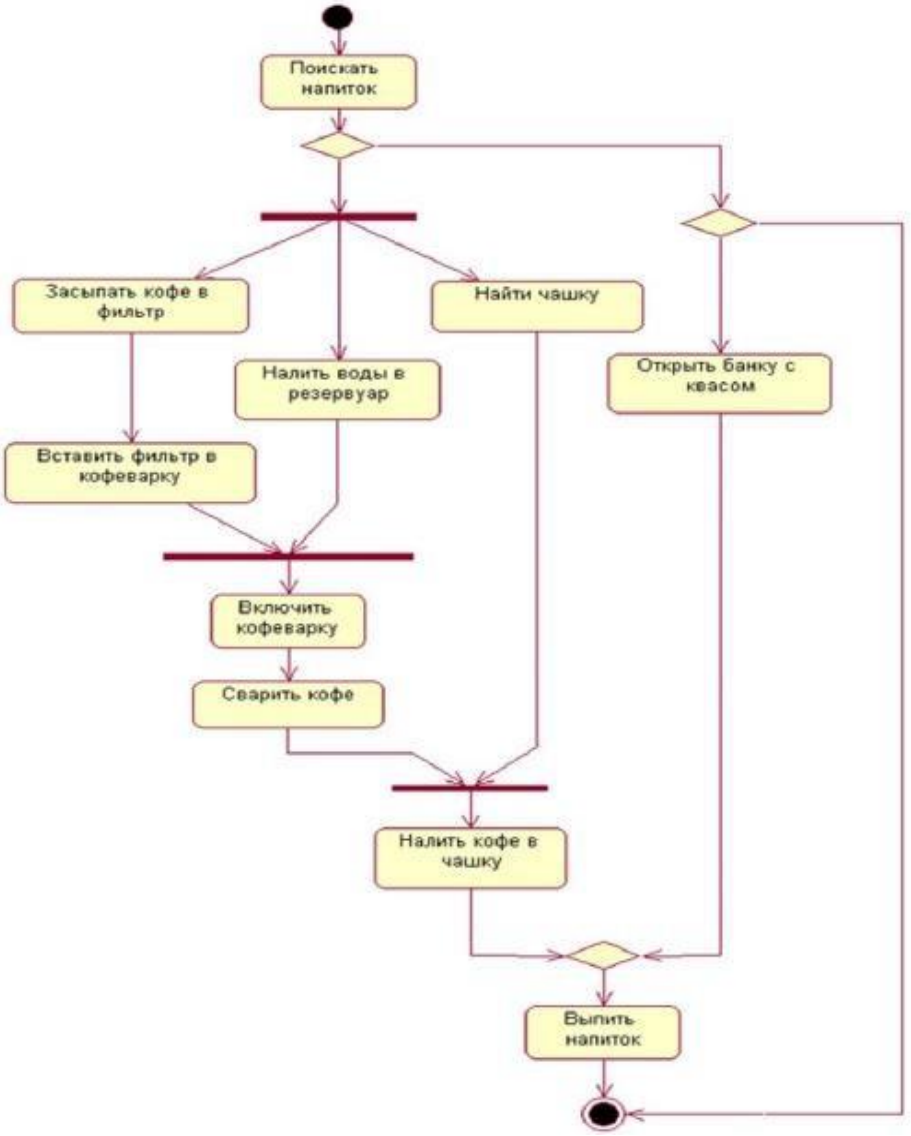


Рисунок 72 – Диаграмма деятельности для примера с приготовлением напитка

Дорожки

Диаграммы деятельности могут быть использованы не только для спецификации алгоритмов вычислений или потоков управления в программных системах. Не менее важная область их применения связана с

моделированием бизнес-процессов. Действительно, деятельность любой компании (фирмы) также представляет собой не что иное, как совокупность отдельных действий, направленных на достижение требуемого результата. Однако применительно к бизнес-процессам желательно выполнение каждого действия ассоциировать с конкретным подразделением компании. В этом случае подразделение несет ответственность за реализацию отдельных действий, а сам бизнес-процесс представляется в виде переходов действий из одного подразделения к другому.

Для моделирования этих особенностей в языке UML используется специальная конструкция, получившее название ***дорожки (swimlanes)***. Имеется в виду визуальная аналогия с плавательными дорожками в бассейне, если смотреть на соответствующую диаграмму. При этом все состояния действия на диаграмме деятельности делятся на отдельные группы, которые отделяются друг от друга вертикальными линиями. Две соседние линии и образуют дорожку, а группа состояний между этими линиями выполняется отдельным подразделением компании (рис. 73).

Рисунок 73 – Вариант диаграммы деятельности с дорожками

Названия подразделений явно указываются в верхней части дорожки. Пересекать линию дорожки могут только переходы, которые в этом случае обозначают выход или вход потока управления в соответствующее подразделение компании. Порядок следования дорожек не несет какой-либо семантической информации и определяется соображениями удобства.

В качестве примера рассмотрим фрагмент диаграммы деятельности торговой компании, обслуживающей клиентов по телефону. Подразделениями компании являются отдел приема и оформления заказов, отдел продаж исклад.

Этим подразделениям будут соответствовать три дорожки на диаграмме деятельности, каждая из которых специфицирует зону ответственности подразделения. В данном случае диаграмма деятельности заключает в себе не только информацию о последовательности выполнения рабочих действий, но и о том, какое из подразделений торговой компании должно выполнять то или иное действие (рис. 74).

Из указанной диаграммы деятельности сразу видно, что после принятия заказа от клиента отделом приема и оформления заказов

осуществляется распараллеливание деятельности на два потока (переход-разделение). Первый из них остается в этом же отделе и связан с получением оплаты от клиента за заказанный товар. Второй инициирует выполнение действия по подбору товара в отделе продаж (модель товара, размеры, цвет, год выпуска и пр.). По окончании этой работы инициируется действие по отпуску товара со склада. Однако подготовка товара к отправке в торговом отделе начинается только после того, как будет получена оплата за товар от клиента и товар будет отпущен со склада (переход-соединение). Только после этого товар отправляется клиенту, переходя в его собственность.

Объекты

В общем случае действия на диаграмме деятельности выполняются над теми или иными объектами. Эти объекты либо инициируют выполнение действий, либо определяют некоторый результат этих действий. При этом действия специфицируют вызовы, которые передаются от одного объекта графа деятельности к другому. Поскольку в таком ракурсе объекты играют определенную роль в понимании процесса деятельности, иногда возникает необходимость явно указать их на диаграмме деятельности.

Для **графического представления объектов** используются прямоугольник класса, с тем отличием, что имя объекта подчеркивается. Далее после имени может указываться характеристика состояния объекта в прямых скобках. Такие прямоугольники объектов присоединяются к состояниям действия отношением зависимости пунктирной линией со стрелкой. Соответствующая зависимость определяет состояние конкретного объекта послевыполнения предшествующего действия.

На диаграмме деятельности с дорожками расположение объекта может иметь некоторый дополнительный смысл. А именно, если объект расположен на границе двух дорожек, то это может означать, что переход к следующему состоянию действия в соседней дорожке ассоциирован с готовностью некоторого документа (объект в некотором состоянии). Если же объект целиком расположен внутри дорожки, то и состояние этого объекта целиком определяется действиями данной дорожки.

Возвращаясь к предыдущему примеру с торговой компанией, можно заметить, что центральным объектом процесса продажи является заказ или вернее состояние его выполнения. Вначале до звонка от клиента заказ как объект отсутствует и возникает лишь после такого звонка. Однако этот заказ еще не заполнен до конца, поскольку требуется еще подобрать конкретный товар в отделе продаж. После его подготовки он передается на склад, где вместе с отпуском товара заказ окончательно дооформляется. Наконец, после получения подтверждения об оплате товара эта информация заносится в заказ, и он считается выполненным и закрытым. Данная информация может быть представлена графически в виде модифицированного варианта диаграммы деятельности этой же торговой компании (рис. 75).

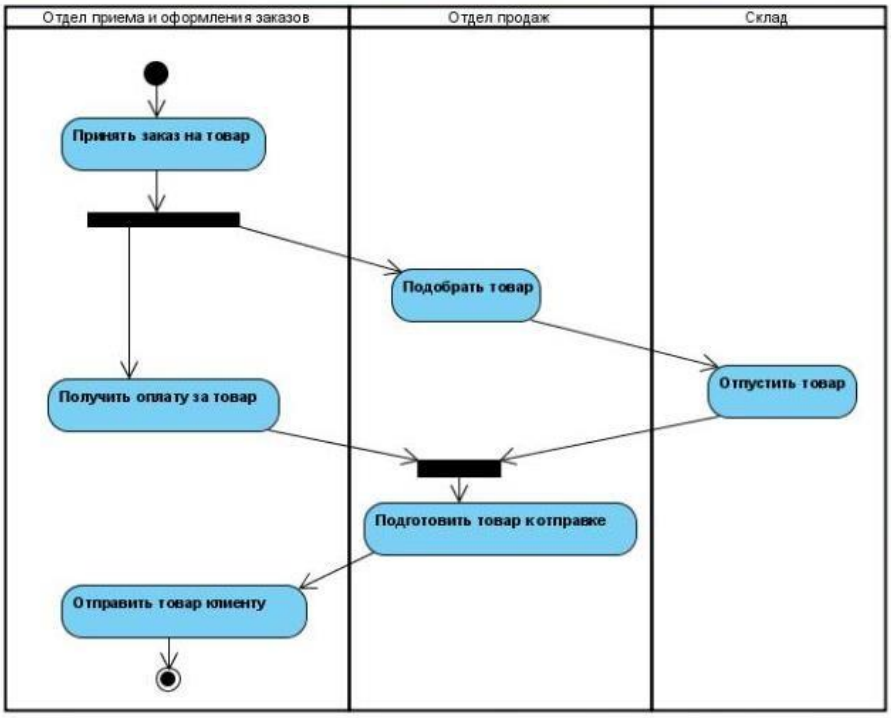


Рисунок 74 – Фрагмент диаграммы деятельности для торговой компании

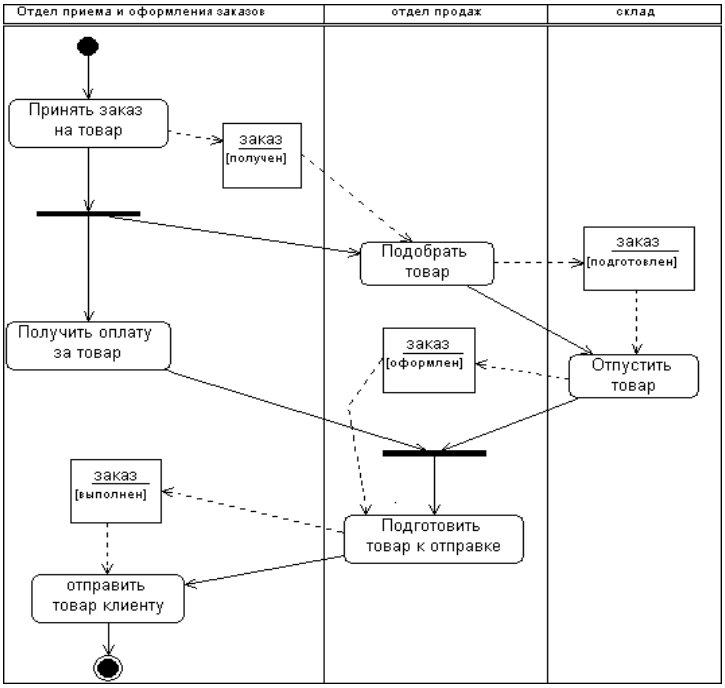


Рисунок 75 – Фрагмент диаграммы деятельности торговой компании с объектом-заказом

Методика выполнения.

В качестве примера рассматривается моделирование системы продажи товаров по каталогу.

1. Запустите MS Visio.
2. Откройте файл, созданный в предыдущих практических занятиях и содержащий диаграмму классов.
3. В проводнике по моделям должны отображаться все классы и диаграммы, созданные ранее (рис. 76).

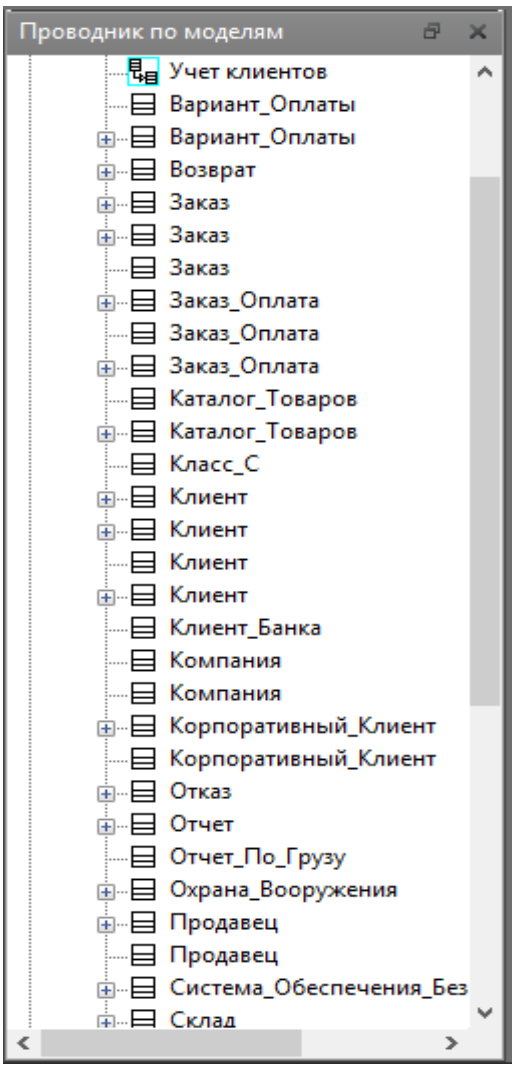


Рисунок 76 – Проводник по моделям

Опишем с помощью диаграммы деятельности процесс формирования заказа и выдачу товара. В бизнес- процессе участвуют 3 действующих лица: клиент, продавец и система оплаты. Следовательно, необходимо добавить 3 дорожки для распределения ответственности между этими лицами.

Для этого, в файле с диаграммой классов, необходимо проделать следующие действия:

1. Щелкнуть правой кнопкой мыши по *классу Заказ*.
2. В контекстном меню выбрать пункт *Схемы*.
3. Нажать кнопку *Создать* и выбрать *Деятельность*.
4. Переименовать созданный лист в *Деятельность-Заказ*.
5. Построить диаграмму деятельности для класса Заказ. Для это выполните действия, описанные ниже.

a. Добавить 3 элемента *Дорожка* и изменить их названия на *Клиент*, *Продавец* и *Система оплаты*

соответственно.

b. Добавить элементы *Начальное состояние*, *Конечное состояние, Состояние действия, Решение, Переход(объединение),* изменить их названия и задать расположение в соответствии с рисунком 77.

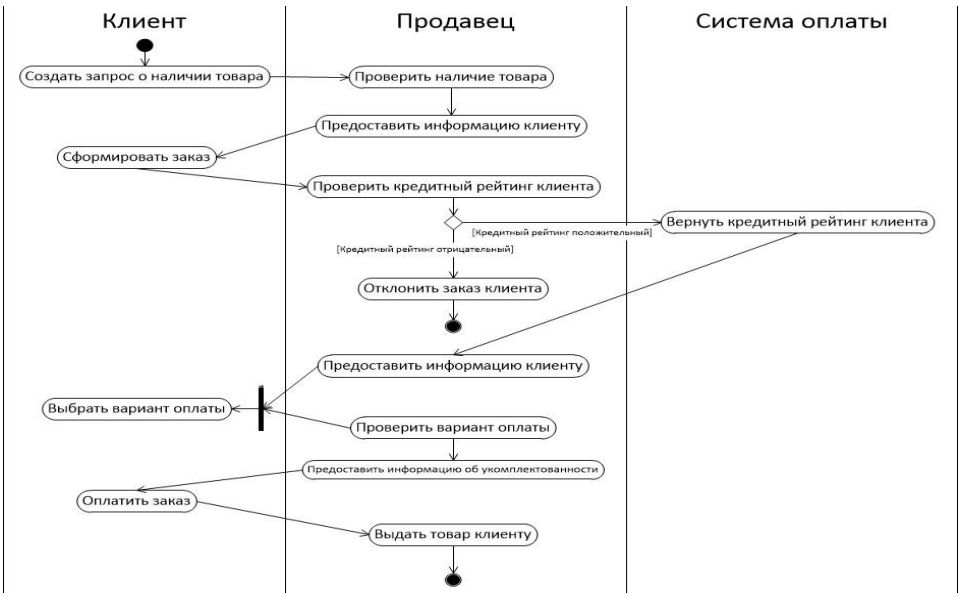


Рисунок 77 – Диаграмма деятельности

Задание.

Построить диаграмму деятельности в соответствии с вариантом.

Отчет по практическому занятию выполняется в формате MS Word, который содержит пошаговое описание процесса построения диаграммы, а также скриншоты результатов согласно заданию.

Создать диаграммы деятельности не менее чем для трех классов, описанных в практическом занятии №8.

Варианты.

1. «Отдел кадров»;
2. «Агентство аренды»;
3. «Аптека»;
4. «Ателье»;
5. «Аэропорт»;
6. «Библиотека»;
7. «Кинотеатр»;
8. «Поликлиника»;
9. «Автосалон»; 10. «Таксопарк».

Контрольные вопросы.

1. Дайте определение понятию «диаграмма деятельности».
2. Опишите назначение диаграммы деятельности.
3. Дайте определение понятиям «состояние деятельности» и «состояние действия». Графическое изображение состояния.
4. Приведите пример ветвления и параллельных потоков управления процессами на диаграмме деятельности.
5. Какие переходы используются на диаграмме деятельности?
6. Что представляет собой дорожка на диаграмме деятельности?
7. Как графически изображаются объекты на диаграмме деятельности?

Практическая работа №9 «Построение диаграмм потоков данных»

Цель работы:

Целью работы является изучение основ создания диаграмм последовательностей на языке UML, получение навыков построения диаграмм последовательностей, применение приобретенных навыков для построения объектно-ориентированных моделей определенной предметной области.

Задачи:

Основными задачами практической работы являются:

– ознакомиться с теоретическими вопросами построения диаграмм последовательностей на языке UML;

– ознакомиться с теоретическими вопросами построения диаграмм последовательностей с помощью MS Visio.

Краткие теоретические сведения.

Диаграммы последовательностей описывают взаимодействия множества объектов, включая сообщения, которыми они обмениваются.

В отличие от диаграммы классов, на которой изображаются абстрактные элементы в виде классов, на диаграмме последовательностей используются конкретные экземпляры классов – ***объекты***. Объекты отображаются прямоугольником без полей. Для того чтобы подчеркнуть, что это экземпляр абстрактной сущности, название объекта подчеркивается. При необходимости через двоеточие после названия можно указать сущность (класс) экземпляром которой является этот объект. Отметим, что объект может быть экземпляром не только класса, но и других абстракций, например, актера. Обратите внимание, что при указании в качестве классификатора актера изменится графическое обозначение объекта (рис. 78).

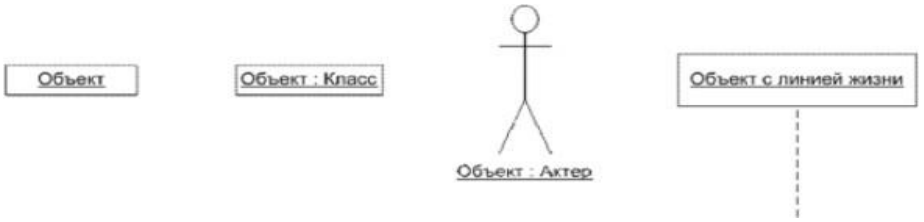


Рисунок 78 – Графическое обозначение объекта UML

На диаграмме последовательностей у объекта может присутствовать ***линия жизни***, на которой отмечаются происходящие с объектом события. Линия жизни отображается пунктирной линией (рис. 79). Между собой объекты могут быть связаны связями. ***Связь*** – это экземпляр отношения ассоциация, и имеет такое же графическое обозначение, что и ассоциация.

На диаграмме последовательностей объекты обмениваются сообщениями. ***Сообщение*** – это спецификация передачи данных от одного объекта другому, который предполагает какое-то ответное действие. Графически сообщение обозначается сплошной линией со стрелкой.

Часто операция вызывает какую-либо операцию в объекте. Очевидно, что класс, экземпляром которого является объект, должен иметь такую

операцию. Привязка сообщения к операции класса объекта выполняется в свойствах сообщения (рис. 79).

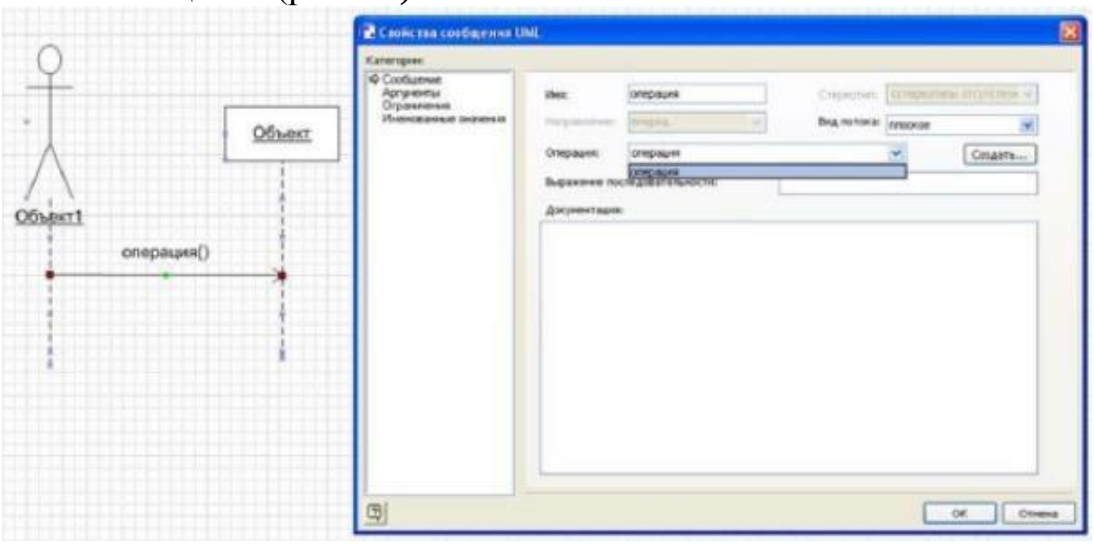


Рисунок 79 – Сообщение UML и его свойства

При построении диаграммы классов обычно определяются только основные свойства сущностей, а такие детали, как операции, удобно создавать при построении диаграммы последовательности, для чего в свойствах сообщения UML есть кнопка создания операции.

Диаграммы последовательностей, как и другие диаграммы для отображения динамических свойств системы, могут быть выполнены в контексте многих сущностей UML. Они могут описывать поведение системы в целом, подсистемы, класса или операции класса и др. К сожалению, Visio недостаточно гибка в плане поддержки раскрытия содержания отдельных элементов с помощью других диаграмм. Например, кликнув правой кнопкой мыши по классу можно обнаружить, что для его описания можно создать лишь диаграммы классов, состояний и деятельности. Поэтому возможность привязать диаграмму последовательностей к элементу, который она реализует, средствами Visio невозможно, эту связь нужно подразумевать.

Диаграммы последовательностей будем делать в контексте прецедентов с диаграммы прецедентов, реализуя те функции, которые должна выполнять наша система.

При построении динамических диаграмм используется уже разработанная структура информационной системы. Для диаграммы последовательностей не нужно придумывать объекты, а достаточно определить, экземпляры каких классов участвуют в этом действии.

Определив необходимые объекты (как экземпляры классов, так и экземпляры актеров), вторым этапом построения диаграмм последовательностей определяются сообщения, пересылаемые между актерами. Фактически определяется последовательность шагов, для выполнения нужного действия.

Методика выполнения.

В качестве примера рассматривается моделирование системы продажи товаров по каталогу.

1. Запустите MS Visio.
2. Откройте файл, созданный в практических занятиях №8-10 и содержащий диаграмму классов.
3. В проводнике по моделям должны отображаться все классы и диаграммы, созданные ранее.

Построим диаграмму последовательности для варианта использования «Обеспечить покупателя информацией» (рис. 4). Для этого добавим на диаграмму последовательности линии жизни и соотнесем объекты сактерами, инициирующими вариант использования «Обеспечить покупателя информацией», и с необходимыми классами.

Для **добавления диаграммы последовательности в проект MS Visio** выполните следующие действия:

1. В проводнике по моделям найдите ветку «Основной пакет».
2. Нажмите по ней правой кнопкой мыши > Создать …
3. В контекстном меню выберите пункт «Схема последовательностей».

Добавим сообщения, которыми обмениваются объекты для исполнения варианта использования. Если объект имеет операцию .

1. Из группы фигур «Последовательности UML» добавить три фигуры типа «Линия жизни объекта». Для изменения названия необходимо дважды щелкнуть левой кнопкой мыши по фигуре. Откроется окно свойств объекта и, если в данном файле нет ранее созданных классов, окно создания нового класса. В данном примере необходимо создать три класса «Товар», «Каталог товаров» и «Заказ» и соответственно три объекта с такими женазваниями.
2. С помощью поиска фигур найти фигуру «Актер» и добавить ее в рабочую область. Двойным щелчком левой кнопки мыши задать имя «Клиент».
3. Добавить фигуру «Линия жизни» и соедините ее начало с фигурой «Клиент».
4. Протянуть все линии жизни вниз листа.
5. Добавить фигуры «Сообщение» и соединить, руководствуясь следующими принципами:

5.1. Соединить фигурой «Сообщение» линию жизни клиента с линией жизни объекта товар. Двойным щелчком по сообщению открыть окно свойств и выбрать операцию запросить товар.

5.2. Соединить фигурой «Сообщение» линию жизни клиента с линией жизни объекта заказ. Двойным щелчком по сообщению открыть окно свойств и выбрать операцию сформировать заказ.

5.3. Соединить фигурой «Сообщение» линию жизни объекта товар с линией жизни объекта каталог товаров. Двойным щелчком по сообщению открыть окно свойств и выбрать операцию проверить наличие.

5.4. Соединить фигурой «Сообщение (возврат)» линию жизни объекта товар и линию жизни клиента. Двойным щелчком по сообщению открыть окно свойств и задать текст сообщения «Предоставить информацию».

6. Добавить фигуры «Активация» и расположить их на диаграмме в соответствии с рисунком 80.

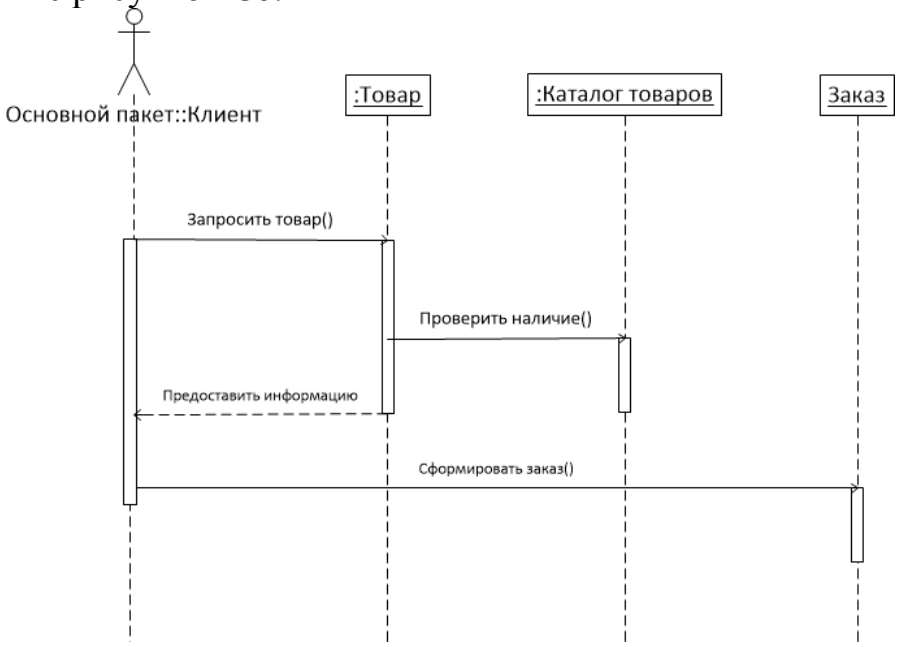


Рисунок 80 – Диаграмма последовательности для варианта использования «Обеспечить покупателя информацией»

При построении диаграмм последовательностей можно вносить коррективы в диаграмму классов. Если объект класса получает новую операцию, то она добавляется в соответствующий класс на диаграмме классов как метод.

Построим диаграмму последовательности для варианта использования «Согласовать условия оплаты» (рис. 81). Действия по построению диаграммы аналогичны построению диаграммы последовательности для варианта использования «Обеспечить покупателя информацией».

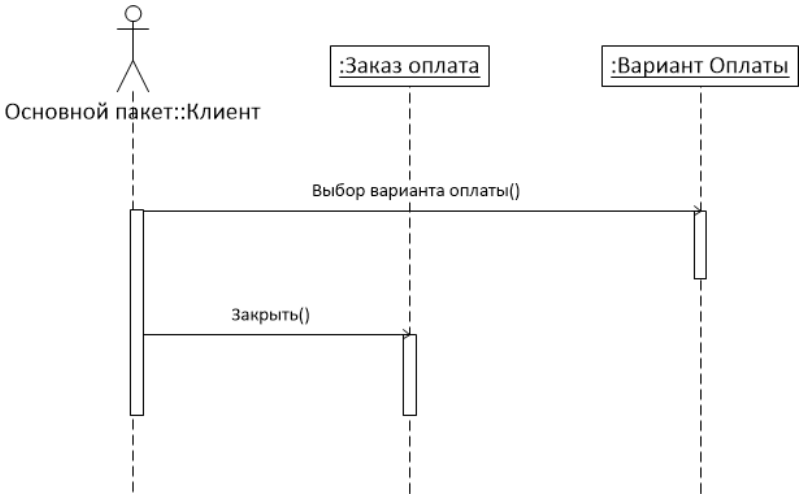


Рисунок 81 – Диаграмма последовательности для варианта использования «Согласовать условия оплаты»

Построим диаграмму последовательности для варианта использования «Заказать товар со склада» (рис. 82).

Действия по построению диаграммы аналогичны построению предыдущих диаграмм последовательностей.

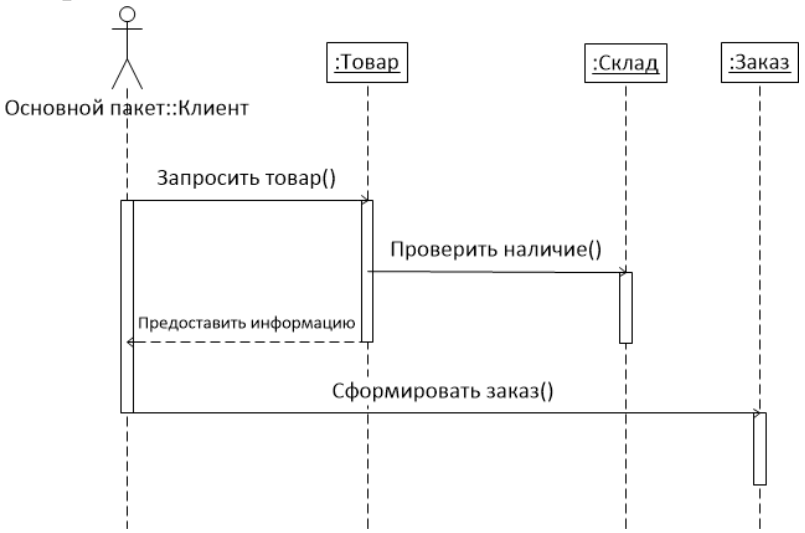


Рисунок 82 – Диаграмма последовательности для варианта использования «Заказать товар со склада»

Построим диаграмму последовательности для системы продажи товаров по каталогу (рис. 83).

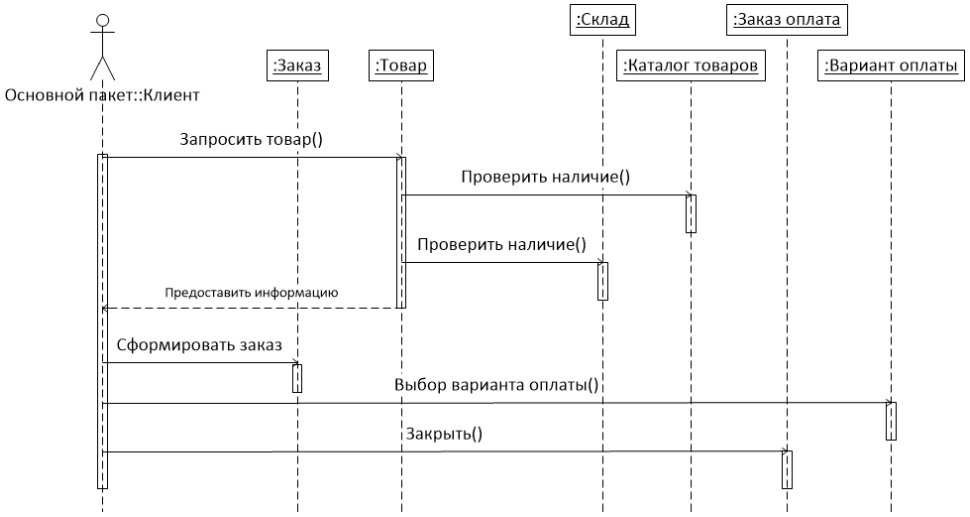


Рисунок 83 – Диаграмма последовательности для системы продажи товаров по каталогу

Задание.

Построить диаграмму последовательности для каждого варианта использования, и для всей системы в целом в соответствии с вариантом.

Отчет по практическому занятию выполняется в формате MS Word, который содержит пошаговое описание процесса построения диаграммы, а также скриншоты результатов согласно заданию.

Варианты.

1. «Отдел кадров»;
2. «Агентство аренды»;

3. «Аптека»;
4. «Ателье»;
5. «Аэропорт»;
6. «Библиотека»;
7. «Кинотеатр»;
8. «Поликлиника»;
9. «Автосалон»;
10. «Таксопарк».

Контрольные вопросы.

1. Для чего предназначена диаграмма последовательности?
2. Назовите и охарактеризуйте элементы диаграммы последовательности.
3. Что такое сообщение?
4. Что такое линия жизни?
5. Назовите виды сообщений

Лабораторная работа №10 «Разработка тестового сценария»

Цели: закрепить знания о типах тестирования; научиться находить и документировать ошибки.

Теоретические сведения

Общепринятая практика состоит в том, что после завершения продукта и до передачи его заказчику независимой группой тестировщиков проводится тестирование ПО.

Уровни тестирования:

- **Модульное тестирование**. Проверяются отдельные компоненты программы, такие как классы или функции.
- **Интеграционное тестирование**. Оценивается корректность взаимодействия компонентов системы, например, передача данных между модулями.
- **Системное тестирование**. Полностью собранная система проверяется на соответствие первоначальным требованиям.

Таблица 2. Виды некоторых ошибок и способы их обнаружения

Виды программных ошибок	Способы их обнаружения
Ошибки выполнения, выявляемые автоматически: а) переполнение, защита памяти; б) несоответствие типов; в) зацикливание	Динамический контроль: аппаратурой процессора; run-time системы программирования; операционной системой – по превышению лимита времени

Тест — это совокупность входных данных и ожидаемых результатов.

Тесты должны удовлетворять следующим критериям:

- **Чувствительность**: высокая вероятность выявления возможных ошибок.
- **Покрытие**: один тест должен охватывать максимальное количество потенциальных проблем.
- **Повторяемость**: возможность воспроизведения ошибки вне зависимости от внешних факторов.

Тестирование помогает обнаружить дефекты в ПО, после чего производится их исправление.

Задание.

Создать приложение Простой калькулятор, в котором реализовать выполнение простых операций с вводимыми двумя операндами. Выполнить тестирование приложения на различных данных, отличающихся по типу и значению.

Программа работы

1. Разработать интерфейс приложения и написать программные коды для событий кнопок.
2. Сохранить проект в отдельной папке, скопировать исполняемый файл на рабочий стол.
3. Составить тесты для проверки работы приложения.
4. Провести тестирование исполняемого файла

Составить отчет по итогам тестирования по устранению выявленных ошибок

Практическая работа №11
«Оценка необходимого количества тестов»

Цель: получить навыки разработки тестовых сценариев.

Теоретические сведения

- Оценка стоимости и причины ошибок в программном обеспечении.
- Виды и методы тестирования.
- Понятие теста.
- Требования к разработке тестовых сценариев.
- Правила разработки тестовых сценариев.

Задание № 1

Написать программу решения квадратного уравнения ax2 + bx + c = 0.

Задание № 2

Найти минимальный набор тестов для программы нахождения вещественных корней квадратного уравнения ax2 + bx + c = 0. Решение представлено в таблице.

Номер теста	a	b	c	Ожидаемый результат	Что проверяется
1	2	-5	2	$x_1=2$, $x_2=0,5$	Случай вещественных корней
2	3	2	5	Сообщение	Случай комплексных корней
3	3	-12	0	$x_1=4$, $x_2=0$	Нулевой корень
4	0	0	10	Сообщение	Неразрешимое уравнение
5	0	0	0	Сообщение	Неразрешимое уравнение
6	0	5	17	Сообщение	Неквадратное уравнение
7	9	0	0	$x_1=x_2=0$	Нулевые корни

Таким образом, для этой программы предлагается минимальный набор функциональных тестов, исходя из 7 классов выходных данных.

Заповеди по отладки программного средства, предложенные Г. Майерсом.

Заповедь 1. Считайте тестирование ключевой задачей разработки ПС, поручайте его самым квалифицированным и одаренным программистам, нежелательно тестировать свою собственную программу.

Заповедь 2. Хорош тот тест, для которого высока вероятность обнаружить ошибку, а не тот, который демонстрирует правильную работу программы.

Заповедь 3. Готовьте тесты как для правильных, так и для неправильных данных.

Заповедь 4. Документируйте пропуск тестов через компьютер, детально изучайте результаты каждого теста, избегайте тестов, пропуск которых нельзя повторить. Заповедь 5. Каждый модуль подключайте к программе только один раз, никогда не изменяйте программу, чтобы облегчить ее тестирование.

Заповедь 6. Пропускайте заново все тесты, связанные с проверкой работы какой-либо программы ПС или ее взаимодействия с другими программами, если в нее были внесены изменения (например, в результате устранения ошибки).

Задание № *3*

Разработайте набор тестовых сценариев (как позитивных, так и негативных) для следующей программы:

Имеется консольное приложение (разработайте самостоятельно). Ему на вход подается 2 строки. На выходе приложение выдает число вхождений второй строки в первую. Например:

Строка 1	Строка 2	Вывод
абвгабвг	аб	2
стстсап	стс	2

Набор тестовых сценариев запишите в виде таблицы, приведенной выше.

Задание № 4

Оформить отчет.

Практическая работа №12 «Разработка тестовых пакетов»

Цель: получить навыки разработки тестовых пакетов.

Теоретические вопросы

– Системные основы разработки требований к сложным комплексам программ.

– Формализация эталонов требований и характеристик комплекса программ.

– Формирование требований компонентов и модулей путем декомпозиции функций комплексов программ.

– Тестирование по принципу «белого ящика».

Задание № 1

В Древней Греции (II в. до н.э.) был известен шифр, называемый "квадрат Полибия".

Шифровальная таблица представляла собой квадрат с пятью столбцами и пятью строками, которые нумеровались цифрами от 1 до 5. В каждую клетку такого квадрата записывалась одна буква. В результате каждой букве соответствовала пара чисел, и шифрование сводилось к замене буквы парой чисел. Для латинского алфавита квадрат Полибия имеет вид:

	1	2	3	4	5
1	A	B	C	D	E
2	F	G	H	I, J	K
3	L	M	N	O	P
4	Q	R	S	T	U
5	V	W	X	Y	Z

Пользуясь изложенным способом создать программу, которая: а) зашифрует введенный текст и сохранит его в файл;

б) считает зашифрованный текст из файла и расшифрует данный текст.

Задание № 2

Спроектировать тесты по принципу «белого ящика» для программы, разработанной в задании № 1. Выбрать несколько алгоритмов для тестирования и обозначить буквами или цифрами ветви этих алгоритмов. Выписать пути алгоритма, которые должны быть проверены тестами для выбранного метода тестирования. Записать тесты, которые позволят пройти по путям алгоритма. Протестировать разработанную вами программу. Результаты оформить в виде таблиц:

Тест	Ожидаемый результат	Фактический результат	Результат тестирования
… …	… …		

Задание № 3

Проверить все виды тестов и сделать выводы об их эффективности

Задание № 4

Оформить отчет.

Практическая работа №13
«Инспекция программного кода на предмет соответствия стандартам кодирования»

Цель: научиться работать с анализатором кода.

Теоретические сведения. Пакет SDK для .NET Compiler Platform предоставляет инструменты для создания пользовательских предупреждений для кода C# или Visual Basic. **Анализатор** содержит код, который распознает нарушения правила. **Исправление кода** содержит код, который исправляет эти нарушения. Правилами, которые вы реализуете, может быть что угодно – от структуры кода до его стиля или соглашений об именовании и многое другое. NET Compiler Platform предоставляет платформу для выполнения анализа во время написания кода, а также все функции пользовательского интерфейса Visual Studio для отладки, включая отображение волнистых линий в редакторе, вывод списка ошибок в Visual Studio и отображение значка лампочки, указывающего на наличие предложений и предлагаемых исправлений.

Анализатор выполняет анализ исходного кода и сообщает о проблеме пользователю. При необходимости анализатор может предложить соответствующее исправление для исходного кода пользователя. В этом руководстве описано, как создать анализатор, ищущий локальные переменные, которые можно объявить с помощью модификатора const, но которые не объявлены. Сопутствующее исправление кода добавляет модификатор const в эти объявления.

Предварительные требования.

Visual Studio 2019 версии 16.7 или более поздней

Необходимо установить пакет SDK для .NET Compiler Platform через Visual Studio Installer:

Инструкции по установке – Visual Studio Installer.

Найти SDK-пакет .NET Compiler Platform в Visual Studio Installer можно двумя способами:

Установка с помощью Visual Studio Installer – представление "Рабочие нагрузки".

SDK-пакет .NET Compiler Platform не выбирается автоматически в рамках рабочей нагрузки разработки расширений Visual Studio. Его необходимо выбрать как дополнительный компонент.

1. Запустите Visual Studio Installer**.**
2. Выберите Изменить**.**
3. Отметьте рабочую нагрузку Разработка расширений Visual Studio**.**
4. Откройте узел Разработка расширений Visual Studio в дереве сводки.
5. Установите флажок SDK-пакет .NET Compiler Platform. Нужный пакет будет представлен последним в списке дополнительных компонентов.

Кроме того, вы можете настроить редактор DGML для отображения диаграмм в средстве визуализации:

1. Откройте узел Отдельные компоненты в дереве сводки.

2. Установите флажок Редактор DGML.

Установка с помощью Visual Studio Installer – вкладка "Отдельные компоненты".

1. Запустите Visual Studio Installer**.**

2. Выберите Изменить**.**

3. Откройте вкладку Отдельные компоненты**.**

4. Установите флажок SDK-пакет .NET Compiler Platform**.** Нужный пакет будет представлен в разделе Компиляторы, средства сборки и среды выполнения в самом начале.

Кроме того, вы можете настроить редактор DGML для отображения диаграмм в средстве визуализации:

1. Установите флажок Редактор DGML. Нужный пакет будет представлен в разделе Средства для работы с кодом**.**

Создать и проверить анализатор можно несколькими способами:

1. Создайте решение.

2. Зарегистрируйте имя и описание анализатора.

3. Создайте отчет анализатора о предупреждениях и рекомендациях.

4. Внедрите исправление кода, чтобы принимать рекомендации.

5. Улучшите анализ с помощью модульных тестов.

Изучение шаблона анализатора.

Анализатор сообщает пользователю о любых объявлениях локальной переменной, которые можно преобразовать в локальные константы. Рассмотрим следующий пример кода:

```
int x = 0;
Console.WriteLine(x);
```

В приведенном выше коде x присваивается значение константы, которое никогда не меняется. Ее можно объявить с помощью модификатора const:

```
const int x = 0;
Console.WriteLine(x);
```

Чтобы определить, можно ли изменить переменную на константу, используется синтаксический анализ, анализ константы из выражения инициализатора, а также анализ потока данных, чтобы убедиться, что переменная не записана. .NET Compiler Platform предоставляет API для упрощения такого анализа. Сначала нужно создать новый проект C# анализатора с исправлением кода**.**

– В Visual Studio последовательно выберите Файл > Создать > Проект , чтобы открыть диалоговое окно "Новый проект".

– В разделе Visual **C# >** Расширяемость выберите Analyzer with code fix (.NET Standard) (Анализатор с исправлением кода (.NET Standard)).

– Присвойте проекту имя MakeConst и нажмите кнопку "ОК".

Анализатор с шаблоном исправления кода создаст три проекта: один содержит анализатор и исправление кода, второй – проект модульного теста и третий – проект VSIX. Запускаемым проектом по умолчанию является

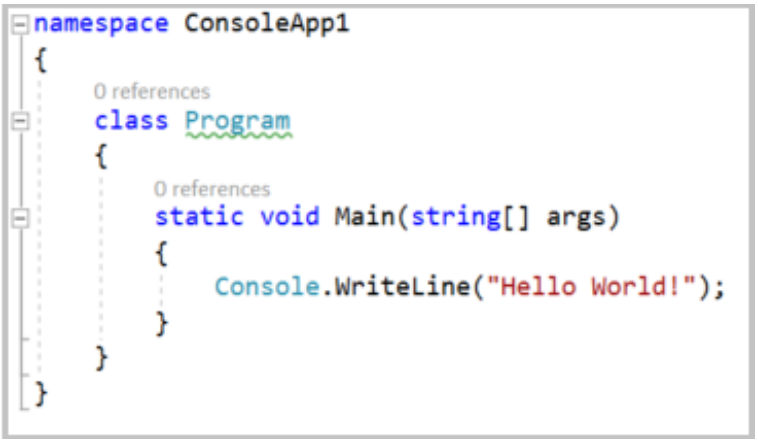
проект VSIX. Нажмите клавишу F5, чтобы запустить проект VSIX. Это запустит второй экземпляр Visual Studio с загруженным в него анализатором.

Совет.

Когда вы запустите анализатор, откроется вторая копия Visual Studio. Эта вторая копия использует другой куст реестра для хранения параметров, что позволяет различить параметры визуальных элементов в обоих копиях Visual Studio. Вы можете выбрать другую тему для экспериментального запуска Visual Studio. Кроме того, не следует перемещать параметры или выполнять вход в учетную запись Visual Studio в экспериментальном экземпляре Visual Studio. Так параметры останутся разными.

Во втором экземпляре Visual Studio, который вы только что создали, создайте проект C# консольного приложения (это может быть как проект .NET Core, так и .NET Framework, так как анализаторы работают на уровне источника). Наведите указатель мыши на токен с волнистым подчеркиванием. Появится текст предупреждения от анализатора.

Шаблон создает анализатор, который выдает предупреждение на каждое объявление типа, где имя типа состоит из букв нижнего регистра, как показано ниже:



Также шаблон содержит исправление кода, которое меняет любые буквы нижнего регистра в имени типа на буквы верхнего регистра. Предлагаемые исправления можно просмотреть, щелкнув значок лампочки рядом с предупреждением. После принятия изменений имя типа и все ссылки на этот тип будут обновлены. Теперь, когда вы увидели работу начального анализатора, закройте второй экземпляр Visual Studio и вернитесь к проекту анализатора.

Для тестирования изменений в анализаторе не требуется каждый раз запускать вторую копию Visual Studio, так как шаблон создает проект модульного теста. В этом проекте содержатся два теста. TestMethod1 показывает обычный формат теста, при котором анализ кода происходит без активации диагностики. TestMethod2 – формат, при котором сначала активируется диагностика, а затем применяется исправление кода. Во время сборки анализатора и исправления кода вы напишете тесты для проверки разных структур. Модульные тесты проводятся гораздо быстрее, чем интерактивное тестирование анализаторов в Visual Studio.

Совет.

Модульные тесты анализатора – отличный инструмент, если вы знаете, какие конструкции кода должны и не должны запускать анализатор. В свою очередь запуск анализатора в другой копии Visual Studio позволяет определить и найти конструкции, о которых вы еще не задумывались.

Регистрация анализатора.

В файле MakeConstAnalyzer.cs с помощью шаблона создается начальный класс DiagnosticAnalyzer. В этом начальном анализаторе отображены два важных свойства каждого анализатора.

- В каждом диагностическом анализаторе должен быть указан атрибут [DiagnosticAnalyzer], который описывает язык, на котором он работает.

- Каждый диагностический анализатор должен наследоваться от класса DiagnosticAnalyzer.

Также в шаблоне отображены базовые функции любого анализатора:

1. Регистрация действий. Действия представляют изменения кода, которые запускают анализатор для проверки нарушений. Когда Visual Studio обнаруживает изменения в коде, которые соответствуют зарегистрированному действию, происходит вызов зарегистрированного метода.

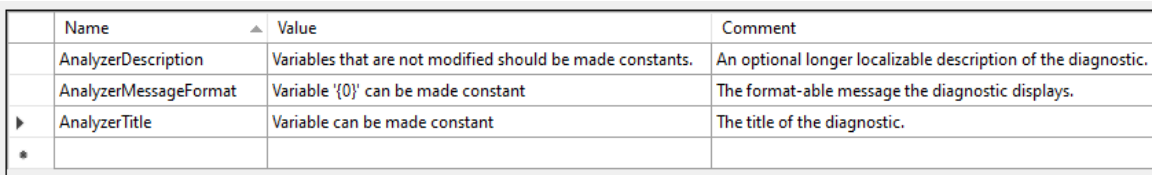
2. Создание диагностики. Обнаружив нарушения, анализатор создает объект диагностики, с помощью которого Visual Studio уведомляет пользователя об этом нарушении.

Действия регистрируются в переопределении метода DiagnosticAnalyzer.Initialize (AnalysisContext). При работе с этим руководстве вы просмотрите синтаксические узлы локальных объявлений и узнаете, какие из них имеют значения констант. Если объявление может быть константой, анализатор создаст диагностику и сформирует отчет.

Сначала обновите константы регистрации и метод Initialize, так как константы определяют ваш анализатор MakeConst. Большинство строковых констант определены в файле строковых ресурсов. Используйте их, чтобы упростить локализацию. Откройте файл Resources.resx для проекта анализатора MakeConst**.** Откроется редактор ресурсов. Измените строковые ресурсы, как показано ниже:

- Компонент AnalyzerTitle измените на Variable can be made constant (Переменная может быть константой).
- Компонент AnalyzerMessageFormat измените на Can be made constant (Может быть константой).
- Компонент AnalyzerDescription измените на Make Constant (Сделать константой).

Также измените раскрывающийся список модификатора доступа на public. Это упростит использование этих констант в модульных тестах. После настройки редактор ресурсов будет иметь следующий вид:



	Name	Value	Comment
	AnalyzerDescription	Variables that are not modified should be made constants.	An optional longer localizable description of the diagnostic.
	AnalyzerMessageFormat	Variable '{0}' can be made constant	The format-able message the diagnostic displays.
▶	AnalyzerTitle	Variable can be made constant	The title of the diagnostic.
*			

Остальные изменения будут в файле анализатора. Откройте файл MakeConstAnalyzer.cs в Visual Studio. Измените зарегистрированное действие на символы на действие на синтаксис. В методе MakeConstAnalyzerAnalyzer.Initialize найдите строку с действием на символы:

context.RegisterSymbolAction(AnalyzeSymbol, SymbolKind.NamedType);

Замените ее приведенным ниже кодом:

context.ConfigureGeneratedCodeAnalysis(GeneratedCodeAnalysisFlags.Analyze | GeneratedCodeAnalysisFlags.None);

context.EnableConcurrentExecution();

context.RegisterSyntaxNodeAction(AnalyzeNode, SyntaxKind.LocalDeclarationStatement);

После этого метод AnalyzeSymbol можно удалить. Этот анализатор проверяет SyntaxKind.LocalDeclarationStatement, а не операторы SymbolKind.NamedType. Обратите внимание на то, что AnalyzeNode подчеркивается красной волнистой линией, так как код, который вы только что вставили, ссылается на метод AnalyzeNode, который еще не объявлен. Объявите этот метод, используя приведенный ниже код:

```
private void AnalyzeNode(SyntaxNodeAnalysisContext context)
{
}
```

В файле MakeConstAnalyzer.cs измените Category на Usage (Использование), как показано ниже:

```
private const string Category = "Usage";
```

Поиск локальных объявлений, которые могут быть константами.

Теперь мы напишем первую версию метода AnalyzeNode. Он должен найти одно локальное объявление, которое может быть const, но таковым не является. Пример приведен ниже:

```
int x = 0;
Console.WriteLine(x);
```

Сначала найдите локальные объявления. Добавьте приведенный ниже код в AnalyzeNode в файле MakeConstAnalyzer.cs :

```
var localDeclaration = (LocalDeclarationStatementSyntax)context.Node;
```

Это приведение всегда завершается успешно, так как ваш анализатор зарегистрирован для отслеживания изменений только локальных объявлений. Другие типы узлов не вызывают метод AnalyzeNode. Затем проверьте объявление на наличие модификаторов const. Если они есть, сразу же выполните возврат. Приведенный ниже код ищет модификаторы const в локальном объявлении:

```
// make sure the declaration isn't already const:
```

```
if (localDeclaration.Modifiers.Any(SyntaxKind.ConstKeyword))
{
   return;
}
```

В конце проверьте, может ли переменная быть const. Это означает, что она не может быть назначена после инициализации.

Выполните семантический анализ с помощью SyntaxNodeAnalysisContext. Используйте аргумент context, чтобы определить, можно ли объявление локальной переменной сделать const. Microsoft.CodeAnalysis.SemanticModel представляет все семантические сведения в одном исходном файле. См. дополнительные сведения о семантических моделях. С помощью Microsoft.CodeAnalysis. SemanticModel выполните анализ потока данных на операторе локального объявления. Затем, используя результаты этого анализа потока данных, убедитесь, что в локальную переменную не записывается новое значение где-либо еще. Вызовите метод расширения GetDeclaredSymbol, чтобы извлечь ILocalSymbol переменной и убедиться, что она не содержится в коллекции DataFlowAnalysis.WrittenOutside из анализа потока данных. Добавьте в конце метода AnalyzeNode приведенный ниже код:

```
// Perform data flow analysis on the local declaration.
var dataFlowAnalysis =
context.SemanticModel.AnalyzeDataFlow(localDeclaration);

// Retrieve the local symbol for each variable in the local declaration
// and ensure that it is not written outside of the data flow analysis region.
var variable = localDeclaration.Declaration.Variables.Single();
var variableSymbol =
context.SemanticModel.GetDeclaredSymbol(variable);
if (dataFlowAnalysis.WrittenOutside.Contains(variableSymbol))
{
   return;
}
```

Этот код гарантирует, что переменная не изменена и может быть const. Теперь мы активируем диагностику. Добавьте приведенный ниже код в последнюю строку метода AnalyzeNode:

```
context.ReportDiagnostic(Diagnostic.Create(Rule,
context.Node.GetLocation()));
```

Чтобы проверить состояние, запустите анализатор, нажав клавишу F5. Загрузите консольное приложение, созданное ранее, и добавьте приведенный ниже код теста:

```
int x = 0;
Console.WriteLine(x);
```

Появится лампочка и анализатор выдаст отчет с диагностическими сведениями. При этом поведение лампочки по-прежнему основано на исправлении кода, сгенерированном шаблоном, указывая на то, что можно

использовать буквы верхнего регистра. В следующем разделе показано, как написать исправление кода.

Написание исправления кода.

Анализатор поддерживает одно или несколько исправлений кода. Исправление кода определяет, какие правки нужно внести для решения обнаруженной проблемы. Исправление кода вашего анализатора предоставляет код с ключевым словом const:

```
const int x = 0;
Console.WriteLine(x);
```

Пользователь выбирает исправление в интерфейсе лампочки в редакторе, а Visual Studio изменяет код.

Откройте файл MakeConstCodeFixProvider.cs , добавленный шаблоном. Это исправление уже привязано к идентификатору диагностики вашего анализатора, но оно пока не реализует преобразование кода. Сначала удалите часть кода шаблона. Измените строку заголовка на Make constant (Сделать константой):

```
private const string title = "Make constant";
```

Затем удалите метод MakeUppercaseAsync. Он больше не применяется.

Все поставщики исправлений кода являются производными от CodeFixProvider. и переопределяют CodeFixProvider.Register CodeFixesAsync(CodeFixContext) на сообщение о доступных исправлениях. В LocalDeclarationStatementSyntax измените тип узла-предка на RegisterCodeFixesAsync в соответствии с диагностикой:

```
var declaration =
root.FindToken(diagnosticSpan.Start).Parent.AncestorsAndSelf().OfType<LocalD
eclarationStatementSyntax>().First();
```

Затем, чтобы зарегистрировать исправление кода, измените последнюю строку. Исправление создаст документ, полученный после добавления модификатора const к существующему объявлению:

```
// Register a code action that will invoke the fix.
context.RegisterCodeFix(
   CodeAction.Create(
      title: title,
      createChangedDocument: c => MakeConstAsync(context.Document,
declaration, c),
      equivalenceKey: title),
   diagnostic);
```

Под символом MakeConstAsync появятся красные волнистые линии. Добавьте объявление в MakeConstAsync, например как указано ниже:

```
private async Task<Document> MakeConstAsync(Document document,
  LocalDeclarationStatementSyntax localDeclaration,
  CancellationToken cancellationToken)
{
}
```

Новый метод MakeConstAsync преобразует Document исходного файла пользователя в новый экземпляр Document, содержащий объявление const.

Создайте новый токен ключевого слова const и вставьте его перед оператором объявления. Не забудьте сначала удалить все элементы trivia из первого оператора объявления и подключить к токену const. Добавьте следующий код в метод MakeConstAsync:

```
// Remove the leading trivia from the local declaration.
var firstToken = localDeclaration.GetFirstToken();
var leadingTrivia = firstToken.LeadingTrivia;
var trimmedLocal = localDeclaration.ReplaceToken(
    firstToken, firstToken.WithLeadingTrivia(SyntaxTriviaList.Empty));
// Create a const token with the leading trivia.
var constToken = SyntaxFactory.Token(leadingTrivia,
SyntaxKind.ConstKeyword,
SyntaxFactory.TriviaList(SyntaxFactory.ElasticMarker));
```

Затем добавьте токен const к объявлению с помощью приведенного ниже кода:

```
// Insert the const token into the modifiers list, creating a new modifiers list.
var newModifiers = trimmedLocal.Modifiers.Insert(0, constToken);
// Produce the new local declaration.
var newLocal = trimmedLocal
    .WithModifiers(newModifiers)
    .WithDeclaration(localDeclaration.Declaration);
```

Отформатируйте новое объявление в соответствии с правилами форматирования C#. Форматирование изменений в соответствии с существующим кодом упрощает работу. Добавьте приведенный ниже оператор сразу после существующего кода:

```
// Add an annotation to format the new local declaration.
var formattedLocal =
newLocal.WithAdditionalAnnotations(Formatter.Annotation);
```

Для выполнения этого кода требуется новое пространство имен. Добавьте следующую директиву using в начало файла.

```
using Microsoft.CodeAnalysis.Formatting;
```

В конце нужно внести правки. Для этого выполните эти три шага:

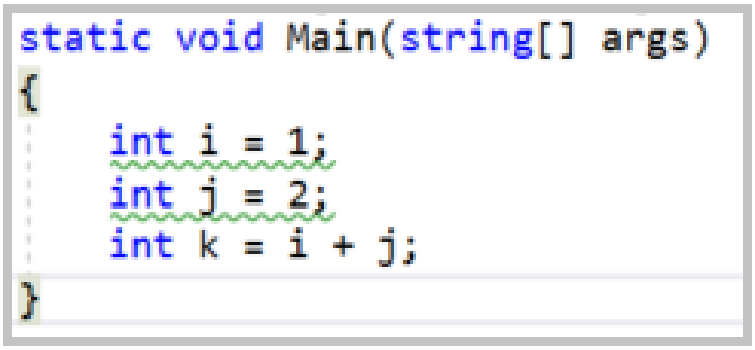
1. Получите дескриптор существующего документа.
2. Создайте документ, заменив существующее объявление на новое.
3. Верните новый документ.

Добавьте в конце метода MakeConstAsync приведенный ниже код:

```
// Replace the old local declaration with the new local declaration.
var oldRoot = await document.GetSyntaxRootAsync(cancellationToken);
var newRoot = oldRoot.ReplaceNode(localDeclaration, formattedLocal);
// Return document with transformed tree.
return document.WithSyntaxRoot(newRoot);
```

Ваше исправление кода готово. Нажмите клавишу F5, чтобы запустить проект анализатора во втором экземпляре Visual Studio. Во втором

экземпляре Visual Studio создайте проект C# консольного приложения и добавьте несколько объявлений локальной переменной, инициализированных со значениями константы в методе main. Они будут отмечены как предупреждения. См. пример ниже.



Мы уже много сделали. Объявления, которые могут быть const, подчеркнуты волнистой линией. Но нам еще нужно с ними поработать. Здесь следует добавить const в объявления i, затем j и k. Но если вы добавите модификатор const в обратном порядке, начиная с k, анализатор выдаст ошибки: k не может быть объявлен как const, пока i и j не являются const. Нужно выполнить дополнительный анализ, чтобы убедиться, что переменные можно объявить и инициализировать различными путями.

Выполнение теста, управляемого данными.

Анализатор и исправление кода работают на простом примере одного объявления, которое может быть константой. Есть множество возможных операторов объявления, где они выдают ошибки. В этих ситуациях можно обратиться к библиотеке модульных тестов, созданной шаблоном. Это гораздо быстрее, чем каждый раз запускать вторую копию Visual Studio.

Откройте файл MakeConstUnitTests.cs в проекте модульного теста. В нем созданы два теста по общим шаблонам для анализатора и для модульного теста исправления кода. Метод TestMethod1 гарантирует, что анализатор не выдаст отчет о диагностике, когда это не нужно. МетодTestMethod2 выдает отчет о диагностике и запускает исправление кода.

Код почти каждого теста вашего анализатора работает по одному из этих шаблонов. Сначала переделайте эти тесты в тесты, управляемые данными. Так будет проще создавать новые тесты, добавляя новые строковые константы для представления различных входных данных.

Для повышения эффективности сначала выполните рефакторинг двух тестов в тесты, управляемые данными. Затем можно определять несколько строковых констант для каждого нового теста. Во время рефакторинга переименуйте оба метода. Замените TestMethod1 тестом, который гарантирует отсутствие диагностики:

```
 [DataTestMethod]
[DataRow("")]
public void WhenTestCodeIsValidNoDiagnosticIsTriggered(string testCode)
{
   VerifyCSharpDiagnostic(testCode);
```

```
}
```

Вы можете создать новую строку данных для теста. Для этого определите фрагмент кода, который не должен вызывать предупреждение диагностики. Эта перегрузка VerifyCSharpDiagnostic передается, если для фрагмента исходного кода диагностика не вызывалась.

Затем замените TestMethod2 этим тестом, который гарантирует вызов диагностики, а также то, что исправление применяется к этому фрагменту исходного кода:

```
 [DataTestMethod]
[DataRow(LocalIntCouldBeConstant, LocalIntCouldBeConstantFixed, 10, 13)]
public void WhenDiagnosticIsRaisedFixUpdatesCode(
   string test,
   string fixTest,
   int line,
   int column)
{
   var expected = new DiagnosticResult
   {
      Id = MakeConstAnalyzer.DiagnosticId,
      Message = new
LocalizableResourceString(nameof(MakeConst.Resources.AnalyzerMessageFormat),
MakeConst.Resources.ResourceManager, typeof(MakeConst.Resources)).ToString(),
      Severity = DiagnosticSeverity.Warning,
      Locations =
         new[] {
               new DiagnosticResultLocation("Test0.cs", line, column)
            }
   };
   VerifyCSharpDiagnostic(test, expected);
   VerifyCSharpFix(test, fixTest);
}
```

Приведенный выше код также вносит изменения в код, который компилирует ожидаемый результат диагностики. Для этого используются открытые константы, зарегистрированные в анализаторе MakeConst. Также используются две строковые константы для введенного и исправленного источника. Добавьте приведенные ниже строковые константы в класс UnitTest:

```
private const string LocalIntCouldBeConstant = @"
using System;

namespace MakeConstTest
{
   class Program
   {
      static void Main(string[] args)
      {
         int i = 0;
         Console.WriteLine(i);
      }
   }
```

```
}";
private const string LocalIntCouldBeConstantFixed = @"
using System;
namespace MakeConstTest
{
   class Program
   {
      static void Main(string[] args)
      {
         const int i = 0;
         Console.WriteLine(i);
      }
   }
}";
```

Чтобы убедиться в том, что они переданы, запустите оба теста. Откройте обозреватель тестов в Visual Studio, последовательно выбрав Тест > Windows > Обозреватель тестов . Затем щелкните ссылку Выполнить все **.**

Создание тестов для допустимых объявлений.

Как правило, анализаторы должны завершать работу как можно быстрее, выполняя минимум операций. Когда пользователь правит код, Visual Studio вызывает зарегистрированные анализаторы. Основным требованием здесь является скорость реагирования. Существует несколько тестовых случаев для кода, который не должен вызывать диагностику. Анализатор уже обрабатывает один из них – тот, при котором переменная присваивается после инициализации. Чтобы воспроизвести этот случай, добавьте приведенную ниже строковую константу в тест:

```
private const string VariableAssigned = @"
using System;

namespace MakeConstTest
{
   class Program
   {
      static void Main(string[] args)
      {
         int i = 0;
         Console.WriteLine(i++);
      }
   }
}";
```

Затем добавьте строку данных, как показано во фрагменте ниже:

```
 [DataTestMethod]
[DataRow(""),
 DataRow(VariableAssigned)]
public void WhenTestCodeIsValidNoDiagnosticIsTriggered(string testCode)
```

Этот тест также проходит. Далее добавьте константы для условий, которые еще не обработаны:

Объявления, которые представляют const, так как они уже являются константами:

```
private const string AlreadyConst = @"
using System;
namespace MakeConstTest
{
   class Program
   {
      static void Main(string[] args)
      {
         const int i = 0;
         Console.WriteLine(i);
      }
   }
}";
```

Объявления, у которых нет инициализатора, так как нет соответствующего значения:

```
private const string NoInitializer = @"
using System;
namespace MakeConstTest
{
   class Program
   {
      static void Main(string[] args)
      {
         int i;
         i = 0;
         Console.WriteLine(i);
      }
   }
}";
```

Объявления, у которых инициализатор не является константой, так как они не могут быть константами времени компиляции:

```
private const string InitializerNotConstant = @"
using System;
namespace MakeConstTest
{
   class Program
   {
      static void Main(string[] args)
      {
         int i = DateTime.Now.DayOfYear;
         Console.WriteLine(i);
      }
   }
}";
```

Трудность заключается еще в том, что в C# допускается несколько объявлений, работающих как один оператор. Рассмотрите приведенную ниже строковую константу тестового случая:

```
private const string MultipleInitializers = @"
using System;
namespace MakeConstTest
{
```

```
    class Program
    {
        static void Main(string[] args)
        {
            int i = 0, j = DateTime.Now.DayOfYear;
            Console.WriteLine(i, j);
        }
    }
}";
```

Переменная i может быть константой, а переменная j – нет. Поэтому этот оператор не может быть объявлением константы. Добавьте объявления DataRow для всех тестов:

```
 [DataTestMethod]
[DataRow(""),
    DataRow(VariableAssigned),
    DataRow(AlreadyConst),
    DataRow(NoInitializer),
    DataRow(InitializerNotConstant),
    DataRow(MultipleInitializers)]
public void WhenTestCodeIsValidNoDiagnosticIsTriggered(string testCode)
```

Снова запустите тесты. Произойдет сбой в новых тестовых случаях.

Обновление анализатора для пропуска верных объявлений.

Чтобы отфильтровать код, который соответствует условиям, необходимо преобразовать метод AnalyzeNode анализатора. Эти условия связаны между собой, и изменения в одном из них будут применены ко всем. Внесите следующие изменения в AnalyzeNode:

- Для объявления одной переменной был выполнен семантический анализ. Этот код должен находиться в цикле foreach, который проверяет все переменные, объявленные в одном и том же операторе.
- Каждая объявленная переменная должна иметь инициализатор.
- Каждый инициализатор переменной должен быть константой времени компиляции.

В методе AnalyzeNode замените исходный семантический анализ:

```
// Perform data flow analysis on the local declaration.
var dataFlowAnalysis = context.SemanticModel.AnalyzeDataFlow(localDeclaration);

// Retrieve the local symbol for each variable in the local declaration
// and ensure that it is not written outside of the data flow analysis region.
var variable = localDeclaration.Declaration.Variables.Single();
var variableSymbol = context.SemanticModel.GetDeclaredSymbol(variable);
if (dataFlowAnalysis.WrittenOutside.Contains(variableSymbol))
{
    return;
}
```

приведенным ниже фрагментом кода:

```
// Ensure that all variables in the local declaration have initializers that
// are assigned with constant values.
foreach (var variable in localDeclaration.Declaration.Variables)
```

```
        {
            var initializer = variable.Initializer;
            if (initializer == null)
            {
                return;
            }

            var constantValue =
context.SemanticModel.GetConstantValue(initializer.Value);
            if (!constantValue.HasValue)
            {
                return;
            }
        }

        // Perform data flow analysis on the local declaration.
        var dataFlowAnalysis =
context.SemanticModel.AnalyzeDataFlow(localDeclaration);

        foreach (var variable in localDeclaration.Declaration.Variables)
        {
            // Retrieve the local symbol for each variable in the local declaration
            // and ensure that it is not written outside of the data flow analysis region.
            var variableSymbol =
context.SemanticModel.GetDeclaredSymbol(variable);
            if (dataFlowAnalysis.WrittenOutside.Contains(variableSymbol))
            {
                return;
            }
        }
```

Первый цикл foreach проверяет каждое объявление переменной с помощью синтаксического анализа. В первой проверке подтверждается наличие инициализатора. Во второй – что этот инициализатор является константой. Во втором цикле запускается исходный семантический анализ. Семантические проверки проходят в отдельных циклах, так как они серьезно влияют на производительность. Снова запустите тест. Все проверки должны быть пройдены.

Финальные штрихи.

Вы почти у цели. Анализатор должен обработать еще несколько условий. Пока пользователь пишет код, Visual Studio вызывает анализаторы. Часто бывает так, что анализатор вызван для кода, который не компилируется. Метод AnalyzeNode диагностического анализатора не проверяет, можно ли преобразовать значение константы в тип переменной. Поэтому в текущей реализации все неверные объявления, такие как int i =

"abc", преобразуются в локальные константы. Добавьте исходную строковую константу в это условие:

```
private const string DeclarationIsInvalid = @"
using System;

namespace MakeConstTest
{
   class Program
   {
      static void Main(string[] args)
      {
         int x = ""abc"";
      }
   }
}";
```

Кроме того, ссылочные типы обрабатываются неправильно. Единственное допустимое значение константы для ссылочного типа – null, кроме случаев с System.String, в которых работают строковые литералы. Другими словами, const string s = "abc" является допустимым, а const object s = "abc" – нет. Этот фрагмент кода проверяет это условие:

```
private const string ReferenceTypeIsntString = @"
using System;
namespace MakeConstTest
{
   class Program
   {
      static void Main(string[] args)
      {
         object s = ""abc"";
      }
   }
}";
```

Чтобы убедиться в том, что можно создать объявление константы для строки, добавьте еще один тест. В приведенным ниже фрагменте кода определен как код, вызывающий диагностику, так и код после исправления:

```
private const string ConstantIsString = @"
using System;
namespace MakeConstTest
{
   class Program
   {
      static void Main(string[] args)
      {
         string s = ""abc"";
      }
   }
}";
private const string ConstantIsStringFixed = @"
using System;
```

```
namespace MakeConstTest
{
   class Program
   {
      static void Main(string[] args)
      {
         const string s = ""abc"";
      }
   }
}";
```

Если переменная объявлена с ключевым словом var, исправление выдает объявление const var, которое не поддерживается в C#. Чтобы исправить эту ошибку, исправление должно заменить ключевое слово var именем выведенного типа:

```
private const string DeclarationUsesVar = @"
using System;
namespace MakeConstTest
{
   class Program
   {
      static void Main(string[] args)
      {
         var item = 4;
      }
   }
}";

private const string DeclarationUsesVarFixedHasType = @"
using System;
namespace MakeConstTest
{
   class Program
   {
      static void Main(string[] args)
      {
         const int item = 4;
      }
   }
}";
private const string StringDeclarationUsesVar = @"
using System;
namespace MakeConstTest
{
   class Program
   {
      static void Main(string[] args)
      {
         var item = ""abc"";
      }
   }
}";
private const string StringDeclarationUsesVarFixedHasType = @"
```

```
using System;
namespace MakeConstTest
{
   class Program
   {
      static void Main(string[] args)
      {
         const string item = ""abc"";
      }
   }
}";
```

Эти изменения обновят объявления строк данных для обоих тестов. В приведенном ниже коде показаны тесты со всеми атрибутами строк данных:

```
//No diagnostics expected to show up
[DataTestMethod]
[DataRow(""),
 DataRow(VariableAssigned),
 DataRow(AlreadyConst),
 DataRow(NoInitializer),
 DataRow(InitializerNotConstant),
 DataRow(MultipleInitializers),
 DataRow(DeclarationIsInvalid),
 DataRow(ReferenceTypeIsntString)]
public void WhenTestCodeIsValidNoDiagnosticIsTriggered(string testCode)
{
   VerifyCSharpDiagnostic(testCode);
}
[DataTestMethod]
[DataRow(LocalIntCouldBeConstant, LocalIntCouldBeConstantFixed, 10, 13),
 DataRow(ConstantIsString, ConstantIsStringFixed, 10, 13),
 DataRow(DeclarationUsesVar, DeclarationUsesVarFixedHasType, 10, 13),
 DataRow(StringDeclarationUsesVar, StringDeclarationUsesVarFixedHasType, 10, 13)]
public void WhenDiagosticIsRaisedFixUpdatesCode(
   string test,
   string fixTest,
   int line,
   int column)
{
   var expected = new DiagnosticResult
   {
      Id = MakeConstAnalyzer.DiagnosticId,
      Message = new
LocalizableResourceString(nameof(MakeConst.Resources.AnalyzerMessageFormat),
MakeConst.Resources.ResourceManager, typeof(MakeConst.Resources)).ToString(),
      Severity = DiagnosticSeverity.Warning,
      Locations =
         new[] {
               new DiagnosticResultLocation("Test0.cs", line, column)
            }
   };
   VerifyCSharpDiagnostic(test, expected);
   VerifyCSharpFix(test, fixTest);
```

```
}
```

К счастью, все вышеперечисленные ошибки можно устранить с помощью методов, которые вы только что изучили.

Чтобы исправить первую ошибку, сначала откройте файл DiagnosticAnalyzer.cs и найдите цикл foreach, в котором проверяются инициализаторы локальных объявлений. Им должны быть назначены значения констант. Сразу же, *до* выполнения цикла foreach, вызовите context.SemanticModel.GetTypeInfo(), чтобы извлечь подробные сведения об объявленном типе из локального объявления:

```
var variableTypeName = localDeclaration.Declaration.Type;
var variableType =
context.SemanticModel.GetTypeInfo(variableTypeName).ConvertedType;
```

Затем проверьте каждый инициализатор внутри цикла foreach, чтобы убедиться в том, что его можно преобразовать в тип переменной. Убедившись в том, что инициализатор является константой, добавьте приведенную ниже проверку:

```
// Ensure that the initializer value can be converted to the type of the
// local declaration without a user-defined conversion.
var conversion = context.SemanticModel.ClassifyConversion(initializer.Value,
variableType);
if (!conversion.Exists || conversion.IsUserDefined)
{
   return;
}
```

Следующее изменение основано на последнем.Перед закрывающей фигурной скобкой первого цикла foreach добавьте приведенный ниже код. Он проверит тип локального объявления, когда константа является строкой или имеет значение null.

```
// Special cases:
//  * If the constant value is a string, the type of the local declaration
//    must be System.String.
//  * If the constant value is null, the type of the local declaration must
//    be a reference type.
if (constantValue.Value is string)
{
   if (variableType.SpecialType != SpecialType.System_String)
   {
      return;
   }
}
else if (variableType.IsReferenceType && constantValue.Value != null)
{
   return;
}
```

Необходимо заменить ключевое слово var правильным именем типа в вашем поставщике исправлений кода. Вернитесь к файлу CodeFixProvider.cs. Код, который вы добавите, выполняет следующие шаги:

- Проверяет, является ли объявление var, если да:
- Создает тип для выводимого типа.

- Проверяет, что объявление типа не является псевдонимом. Если это так, можно объявить const var.
- Проверяет, что var не является именем типа в программе (если это так, const var является допустимым).
- Упрощает полное имя типа.

Кажется, что здесь довольно много кода. Но это не так. Замените строку, которая объявляет и инициализирует newLocal приведенным ниже кодом. Он выполняется сразу после инициализации newModifiers:

```
// If the type of the declaration is 'var', create a new type name
// for the inferred type.
var variableDeclaration = localDeclaration.Declaration;
var variableTypeName = variableDeclaration.Type;
if (variableTypeName.IsVar)
{
   var semanticModel = await document.GetSemanticModelAsync(cancellationToken);
   // Special case: Ensure that 'var' isn't actually an alias to another type
   // (e.g. using var = System.String).
   var aliasInfo = semanticModel.GetAliasInfo(variableTypeName);
   if (aliasInfo == null)
   {
      // Retrieve the type inferred for var.
      var type = semanticModel.GetTypeInfo(variableTypeName).ConvertedType;
      // Special case: Ensure that 'var' isn't actually a type named 'var'.
      if (type.Name != "var")
      {
         // Create a new TypeSyntax for the inferred type. Be careful
         // to keep any leading and trailing trivia from the var keyword.
         var typeName = SyntaxFactory.ParseTypeName(type.ToDisplayString())
            .WithLeadingTrivia(variableTypeName.GetLeadingTrivia())
            .WithTrailingTrivia(variableTypeName.GetTrailingTrivia());
         // Add an annotation to simplify the type name.
         var simplifiedTypeName =
typeName.WithAdditionalAnnotations(Simplifier.Annotation);
         // Replace the type in the variable declaration.
         variableDeclaration = variableDeclaration.WithType(simplifiedTypeName);
      }
   }
}
// Produce the new local declaration.
var newLocal = trimmedLocal.WithModifiers(newModifiers)
                  .WithDeclaration(variableDeclaration);
```

Чтобы использовать тип Simplifier, потребуется добавить одну директиву using:

```
using Microsoft.CodeAnalysis.Simplification;
```

Запустите тесты. Они все должны быть пройдены. Можете поздравить себя, запустив готовый анализатор. Чтобы запустить проект анализатора во втором экземпляре Visual Studio с загруженным расширением Roslyn (предварительная версия), нажмите клавиши CTRL+F5.

– Во втором экземпляре Visual Studio создайте новый проект C# консольного приложения и добавьте int x = "abc"; в метод main. Так как первая ошибка была исправлена, для объявления локальной переменной не должно выдаваться предупреждение (хотя есть ошибка компилятора, как и предполагалось).

– Затем добавьте object s = "abc"; в метод main. Так как вторая ошибка была исправлена, не должно быть никаких предупреждений.

– Наконец, добавьте другую локальную переменную, использующую ключевое слово var. Внизу слева появится предупреждение и предложение исправлений.

– Наведите курсор редактора на волнистую линию и нажмите клавиши CTRL+. Отобразятся предложенные исправления. Обратите внимание, что после выбора исправления ключевое слово var теперь обрабатывается правильно.

В конце добавьте приведенный ниже код:

```
int i = 2;
int j = 32;
int k = i + j;
```

После этого только две переменные будут подчеркнуты красными волнистыми линиями. Добавьте const в i и j. Появится предупреждение, указывающее на то, что k может быть const.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

Электронные издания (электронные ресурсы)

1. Гниденко, И. Г. Технология разработки программного обеспечения : учеб. пособие для СПО / И. Г. Гниденко, Ф. Ф. Павлов, Д. Ю. Федоров. — Москва : Издательство Юрайт, 2019. — 235 с. Текст : электронный // ЭБС Юрайт [сайт]. — Режим доступа: https://biblio-online.ru/bcode/438444

2. Разработка, внедрение и адаптация программного обеспечения отраслевой направленности [Электронный ресурс]: учеб. пособие / Г.Н. Федорова. — М. :КУРС : ИНФРА-М, 2019. — 336 с. (Среднее Профессиональное Образование). - Режим доступа: http://znanium.com/catalog/product/989682

3. Программное обеспечение компьютерных сетей и web-серверов [Электронный ресурс]: учеб. пособие / Г.А. Лисьев, П.Ю. Романов, Ю.И. Аскерко. — М. : ИНФРА-М, 2019. — 145 с. - Режим доступа: http://znanium.com/catalog/product/988332

4. Гагарина, Л. Г. Разработка и эксплуатация автоматизированных информационных систем [Электронный ресурс]: учеб. пособие / Л.Г. Гагарина. — М. : ФОРУМ : ИНФРА-М, 2019. — 384 с.- Режим доступа: http://znanium.com/catalog/product/1003025 (ЭБСZnanium)

5. Технология разработки программного обеспечения [Электронный ресурс]: учеб. пособие / Л.Г. Гагарина, Е.В. Кокорева, Б.Д. Сидорова-Виснадул ; под ред. Л.Г. Гагариной. — М. : ИД «ФОРУМ» : ИНФРА-М, 2019. — 400 с. - Режим доступа: http://znanium.com/catalog/product/1011120 (ЭБСZnanium)

6. Программное обеспечение компьютерных сетей [Электронный ресурс]: учеб. пособие / О.В. Исаченко. — М. : ИНФРА-М, 2019. — 117 с. — (Среднее профессиональное образование). - Режим доступа: http://znanium.com/catalog/product/989894 (ЭБСZnanium)

7. Плохотников, К.Э. Метод и искусство математического моделирования : курс лекций / К.Э. Плохотников. — 2-е изд., стер. — Москва : ФЛИНТА, 2017. — 519 с. - Текст : электронный. - Режим доступа: https://new.znanium.com/catalog/product/1034329

8. Математическое моделирование технических систем [Электронный ресурс]: учебник / В.П. Тарасик. — Минск : Новое знание ; М. : ИНФРА-М, 2019. — 592 с. - Режим доступа: http://znanium.com/catalog/product/1019246

Методические издания

1. Игнатенко, Е.С.. Методические указания по выполнению практических работ по МДК

02.01 Технология разработки программного обеспечения.– Нефтеюганск: НИК(филиал) ФГБОУ ВО «ЮГУ», 2019 [Электронный ресурс] Режим доступа: локальная сеть филиала.

ЧАГАРОВА Зухра Нуржигидовна

ТЕХНОЛОГИЯ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

Практикум для III-IV курса, обучающихся по специальности 09.02.07 «Информационные системы и программирование»

Корректор Чагова О.Х.
Редактор Чагова О.Х.

Сдано в набор 28.05.2025 г.
Формат 60х84/16
Бумага офсетная.
Печать офсетная.
Усл. печ. л. 6,27
Заказ № 5120
Тираж 100 экз.

Оригинал-макет подготовлен
в Библиотечно-издательском центре СКГА
369000, г. Черкесск, ул. Ставропольская, 36