МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное бюджетное образовательное учреждение высшего образования
«Кузбасский государственный технический университет имени Т. Ф. Горбачева»

Кафедра информационных и автоматизированных производственных систем

Составитель
И. С. Сыркин

ТЕХНОЛОГИЯ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

Методические материалы

Рекомендовано цикловой методической комиссией специальности
СПО 09.02.07 Информационные системы и программирование
в качестве электронного издания
для использования в образовательном процессе

Кемерово 2018

Рецензенты

Ванеев О. Н. – кандидат технических наук, доцент кафедры информационных и автоматизированных производственных систем ФГБОУ ВО «Кузбасский государственный технический университет имени Т. Ф. Горбачева»

Чичерин И. В. – кандидат технических наук, доцент, зав. кафедрой информационных и автоматизированных производственных систем ФГБОУ ВО «Кузбасский государственный технический университет имени Т. Ф. Горбачева»

Сыркин Илья Сергеевич

Технология разработки программного обеспечения: методические материалы [Электронный ресурс] для студентов специальности СПО 09.02.07 Информационные системы и программирование очной формы обучения / сост. И. С. Сыркин; КузГТУ. – Электрон. издан. – Кемерово, 2018.

Методические материалы для дисциплины «Технология разработки программного обеспечения» содержат перечень выполняемых практических и лабораторных работ, контрольные вопросы к ним.

Предисловие

Целью освоения дисциплины «Технологии разработки программного обеспечения» является приобретение обучающимися знаний в области формирования требований к ИС и тестирования ПО.

Основными задачами изучения дисциплины «Технологии разработки программного обеспечения», являются:

1. Изучение основных принципов процесса разработки программного обеспечения.
2. Изучение встроенных и основных специализированные инструментов анализа качества программных продуктов.
3. Использование специализированных графических средств построения и анализа архитектуры программных продуктов.
4. Умение выполнять ручное и автоматизированное тестирование программного модуля.

Содержание дисциплины в соответствии с учебным планом

В соответствии с учебным планом изучение дисциплины «Технологии разработки программного обеспечения» предусматривает проведение лекционных, практических занятий и самостоятельной работы обучающимися очной формы обучения.

Общая трудоемкость дисциплины составляет 118 часов.

Промежуточный контроль – экзамен (4 семестр).

Содержание практических занятий

При подготовке к практическим занятиям обучающиеся самостоятельно изучают основную и дополнительную литературу, готовят конспекты по темам, предложенным преподавателем.

На практических занятиях преподаватель осуществляет контроль подготовки качества знаний обучающегося, используя: опрос, обсуждение вопросов по темам изучаемой дисциплины, письменный опрос при текущем контроле и предоставление отчетов по практическим занятиям.

1. Практическое занятие №1. Анализ предметной области

Целью работы является изучение методов анализа предметной области Результатом практической работы является отчет, в котором должны быть приведены анализ предметной области и требований к системе.

Для выполнения практической работы № 1 студент должен изучить приведенный ниже теоретический материал. Отчет сдается в распечатанном и электронном (файл Word) видах.

Анализ предметной области

Требования к ПО определяют, какие свойства и характеристики оно должно иметь для удовлетворения потребностей пользователей и других заинтересованных лиц. Однако сформулировать требования к сложной системе не так легко. В большинстве случаев будущие пользователи могут перечислить набор свойств, который они хотели бы видеть, но никто не даст гарантий, что это – исчерпывающий список. Кроме того, часто сама формулировка этих свойств будет непонятна большинству программистов: могут прозвучать фразы типа «должно использоваться и частотное, и временное уплотнение каналов», «передача клиента должна быть мягкой», «для обычных швов отмечайте бригаду, а для доверительных – конкретных сварщиков», и это еще не самые тяжелые для понимания примеры.

Чтобы ПО было действительно полезным, важно, чтобы оно удовлетворяло реальные потребности людей и организаций, которые часто отличаются от непосредственно выражаемых пользователями желаний. Для выявления этих потребностей, а также для выяснения смысла высказанных требований приходится проводить достаточно большую дополнительную работу, которая называется анализом предметной области или бизнес-моделированием, если речь идет о потребностях коммерческой организации. В результате этой деятельности разработчики должны научиться понимать язык, на котором говорят пользователи и заказчики, выявить цели их деятельности, определить набор задач, решаемых ими. В дополнение стоит выяснить, какие вообще задачи нужно уметь решать для достижения этих целей, выяснить свойства результатов, которые хоте-

лось бы получить, а также определить набор сущностей, с которыми приходится иметь дело при решении этих задач. Кроме того, анализ предметной области позволяет выявить места возможных улучшений и оценить последствия принимаемых решений о реализации тех или иных функций.

После этого можно определять область ответственности будущей программной системы – какие именно из выявленных задач будут ею решаться, при решении каких задач она может оказать существенную помощь и чем именно. Определив эти задачи в рамках общей системы задач и деятельностей пользователей, можно уже более точно сформулировать требования к ПО.

Анализом предметной области занимаются системные аналитики или бизнес-аналитики, которые передают полученные ими знания другим членам проектной команды, сформулировав их на более понятном разработчикам языке. Для передачи этих знаний обычно служит некоторый набор моделей, в виде графических схем и текстовых документов.

Анализ деятельности крупной организации, такой как банк с сетью региональных отделений, нефтеперерабатывающий завод или компания, производящая автомобили, дает огромные объемы информации. Из этой информации надо уметь отбирать существенную, а также уметь находить в ней пробелы – области деятельности, информации по которым недостаточно для четкого представления о решаемых задачах. Значит, всю получаемую информацию надо каким-то образом систематизировать.

Выделение и анализ требований

После получения общего представления о деятельности и целях организаций, в которых будет работать будущая программная система, и о ее предметной области, можно определить более четко, какие именно задачи система будет решать. Кроме того, важно понимать, какие из задач стоят наиболее остро и обязательно должны быть поддержаны уже в первой версии, а какие могут быть отложены до следующих версий или вообще вынесены за рамки области ответственности системы. Эта информация выявляется при анализе потребностей возможных пользователей и заказчиков.

Потребности определяются на основе наиболее актуальных проблем и задач, которые пользователи и заказчики видят перед собой. При этом требуется аккуратное выявление значимых проблем,

определение того, насколько хорошо они решаются при текущем положении дел, и расстановка приоритетов при рассмотрении недостаточно хорошо решаемых, поскольку чаще всего решить сразу все проблемы невозможно.

Формулировка потребностей может быть разбита на следующие этапы.

Выделить одну-две-три основных проблемы.

Определить причины возникновения проблем, оценить степень их влияния и выделить наиболее существенные из проблем, влекущие появление остальных.

Определить ограничения на возможные решения.

Формулировка потребностей не должна накладывать лишних ограничений на возможные решения, удовлетворяющие им. Нужно попытаться сформулировать, что именно является проблемой, а не предлагать сразу возможные решения.

Например, формулировки «система должна использовать СУБД Oracle для хранения данных», «нужно, чтобы при вводе неверных данных раздавался звуковой сигнал» не очень хорошо описывают потребности. Исключением в первом случае может быть особая ситуация, например, если СУБД Oracle уже используется для хранения других данных, которые должны быть интегрированы с рассматриваемыми: при этом ее использование становится внешним ограничением. Соответствующие потребности лучше описать так: «нужно организовать надежное и удобное для интеграции с другими системами хранение данных», «необходимо предотвращать попадание некорректных данных в хранилище».

При выявлении потребностей пользователей анализируются модели деятельности пользователей и организаций, в которых они работают, для выявления проблемных мест. Также используются такие приемы, как анкетирование, демонстрация возможных сеансов работы будущей системы, интерактивные опросы, где пользователям предоставляется возможность самим предложить варианты внешнего вида системы и ее работы или поменять предложенные кем-то другим, демонстрация прототипа системы и др.

После выделения основных потребностей нужно решить вопрос о разграничении области ответственности будущей системы, т.е. определить, какие из потребностей надо пытаться удовлетворить в ее рамках, а какие – нет.

При этом все заинтересованные лица делятся на пользователей, которые будут непосредственно использовать создаваемую систему для решения своих задач, и вторичных заинтересованных лиц, которые не решают своих задач с ее помощью, но чьи интересы так или иначе затрагиваются ею. Потребности пользователей нужно удовлетворить в первую очередь и на это нужно выделить больше усилий, а интересы вторичных заинтересованных лиц должны быть только адекватно учтены в итоговой системе.

На основе выделенных потребностей пользователей, отнесенных к области ответственности системы, формулируются возможные функции будущей системы, которые представляют собой услуги, предоставляемые системой и удовлетворяющие потребности одной или нескольких групп пользователей (или других заинтересованных лиц). Идеи для определения таких функций можно брать из имеющегося опыта разработчиков (наиболее часто используемый источник) или из результатов мозговых штурмов и других форм выработки идей.

Формулировка функций должна быть достаточно короткой, ясной для пользователей, без лишних деталей. Например:

Все данные о сделках и клиентах будут сохраняться в базе данных.

Статус выполнения заказа клиент сможет узнать через Интернет.

Система будет поддерживать до 10000 одновременно работающих пользователей.

Расписание проведения ремонтных работ будет строиться автоматически.

Предлагая те или иные функции, нужно уметь аккуратно оценивать их влияние на структуру и деятельность организаций, в рамках которых будет использоваться ПО. Это можно сделать, имея полученные при анализе предметной области модели их текущей деятельности.

Имея набор функций, достаточно хорошо поддерживающий решение наиболее существенных задач, с которыми придется работать разрабатываемой системе, можно составлять требования к ней, представляющие собой детализацию работы этих функций. Соотношение между проблемами, потребностями, функциями и требованиями показано на рис. 1.

Рис. 1. Соотношение между проблемами, потребностями, функциями и требованиями

При этом часто нужно учитывать, что ПО является частью программно-аппаратной системы, требования к которой надо преобразовать в требования к программной и аппаратной ее составляющим. В последнее время, в связи со значительным падением цен на мощное аппаратное обеспечение общего назначения, фокус внимания переместился, в основном, на программное обеспечение. Во многих проектах аппаратная платформа определяется из общих соображений, а поддержку большинства нужных функций осуществляет ПО.

Каждое требование раскрывает детали поведения системы при выполнении ею некоторой функции в некоторых обстоятельствах. При этом часть требований исходит из потребностей и пожеланий заинтересованных лиц и решений, удовлетворяющих эти потребности и пожелания, а часть – из внешних ограничений, накладываемых на систему, например, основными законами той предметной области, в рамках которой системе придется работать, государственным законодательством, корпоративной политикой и пр.

Контрольные вопросы

1. Что такое «требование к информационной системе»?
2. Кто занимается выявлением требований к ИС?
3. Перечислите этапы формулировки потребностей

2. Практическое занятие №2. Разработка и оформление технического задания

Целью работы является изучение порядка разработки и оформления технического задания. Результатом практической работы является отчет, в котором должны быть приведено разработанное ТЗ.

Для выполнения практической работы № 2 студент должен изучить приведенный ниже теоретический материал. Отчет сдается в распечатанном и электронном (файл Word) видах.

Разработка и оформление технического задания

Техническое задание – это документ, определяющий цели, требования и основные исходные данные, необходимые для разработки автоматизированной системы управления.

При разработке технического задания необходимо решить следующие задачи:

- установить общую цель создания ИС, определить состав подсистем и функциональных задач;
- разработать и обосновать требования, предъявляемые к подсистемам;
- разработать и обосновать требования, предъявляемые к информационной базе, математическому и программному обеспечению, комплексу технических средств (включая средства связи и передачи данных);
- установить общие требования к проектируемой системе;
- определить перечень задач создания системы и исполнителей;
- определить этапы создания системы и сроки их выполнения;
- провести предварительный расчет затрат на создание системы и определить уровень экономической эффективности ее внедрения.

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

Таблица

Состав и содержание технического задания
(ГОСТ 34.602-89)

№ п\п	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,

№ п\п	Раздел	Содержание
		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	• перечень стадий и этапов работ

№ п\п	Раздел	Содержание
	работ по созданию системы	• 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

Контрольные вопросы

1. Что такое техническое задание?
2. Какой ГОСТ регламентирует содержание технического задания?
3. Какие пункты должно содержать техническое задание?

3. Практическое занятие №3. Построение архитектуры программного средства

Целью работы является изучение порядка проектирования архитектуры ПО. Результатом практической работы является отчет, в котором должны быть приведена разработанная архитектура ПО.

Для выполнения практической работы № 3 студент должен изучить приведенный ниже теоретический материал. Отчет сдается в распечатанном и электронном (файл Word) видах.

Разработка архитектуры ПО

1. Влияние стандартов на процессы архитектурного проектирования

Осмысленная разработка программных продуктов – комплексное понятие, состоящее из множества высокоинтеллектуальных и взаимосвязанных процессов, таких как:

- анализ требований;
- проектирование;
- кодирование;
- тестирование;
- и т. д.

Каждая из обозначенных активностей должна основываться на базе соответствующих стандартов и лучших практик. Дело в том, что подобный тип документов создается на основе наиболее успешного опыта применения специфического вида деятельности, рабочей технологии или артефактов, эталонное описание которых он содержит. Стандарт концентрирует все лучшее и наиболее эффективное, с точки зрения практического достижения конечного результата, подтвержденного массивом статистических данных.

Перед тем как внедрять стандарты в процессы конкретной организации, следует соответствующим образом адаптировать их под реалии конкретной организации.

Должны быть учтены такие аспекты, как:

- ресурсная база организации;
- количество выделенных для деятельности ресурсов, их доступность, квалификация (если речь идет о специалистах), надежность и т. д.;
- сегмент/домен/направление деятельности компании;

- сфера деятельности, с учетом специфических требований к ним со стороны отраслевых/государственных/международных регуляторных органов;
- степень влияния информационных технологий на поддержку и развитие бизнеса;
- инновационность компании и степень участия в инновациях информационных технологий;
- и т. д.

Если тема применения и использования стандартов в целях повышения эффективности деятельности Вас заинтересовала, то мы рекомендуем обратиться к моделям зрелости CMMI, методологии SPICE и другим подобным документам. В настоящей лекции мы больше к данному аспекту, влияющему на качество архитектуры и архитектурного проектирования, возвращаться не будем.

Перечень стандартов и их содержание на конкретном предприятии должно определяться на стадии планирования.

В момент планирования перечня стандартов, который действительно необходим, целесообразно руководствоваться принципом «золотой середины». Чрезмерное количество внедряемых стандартов требует достаточно большого количества ресурсов. Следует осознавать – внедрение стандарта делает процесс более унифицированным и качественным за счет его регламентации, но при этом, в отдельных аспектах, понижает степень его гибкости. Если компания пришла к пониманию необходимости внедрения определенной практики или стандарта эволюционным путем, то это облегчает и упрощает процесс внедрения и использования соответствующего регламента. Если изменения носят революционный характер, то попытка привнести в организацию избыточное количество стандартов может носить катастрофический характер и остаться «на бумаге», что не является целью их внедрения.

Стандартизация должна обеспечить постепенное повышение качества конкретных процессов (анализ, документирование, кодирование и т. д.), обеспечить прозрачность, однозначность и не противоречивость процессов, минимизацию или устранение появления возможных ошибок при разработке архитектур и программных продуктов.

В качестве отступления мы приведем следующий пример.

Требования к процедуре сертификации авиационных продуктов предусматривают, что в процессе разработки специфического (для авиационных нужд) программного обеспечения, должны использоваться как минимум три стандарта:

- Стандарт на работу с требованиями;

Должен описывать методы, правила и инструменты, применяемые для сбора, разработки и управления требованиями, их возможные форматы и нотации.

- Стандарт на разработку архитектуры программного продукта;

Содержит правила, формирующие архитектуру программного продукта, приемлемые и неприемлемые методы её разработки, описывает возможные функциональные и не функциональные ограничения (запрет на использование рекурсии или максимальная вложенность вызовов).

- Стандарт процесса кодирования;

Регламентирует исходный код программы, ссылки на описания используемого языка программирования, его синтаксис, ограничение, желательные и нежелательные конструкции языка и прочие правила, касающиеся разработки кода программы;

Стандарты определяют рабочие принципы, которые должны действовать на всем протяжении жизненного цикла программного продукта. Например, если в стандарте зафиксировано, что нельзя вставлять комментарии в программный код (комментарии следует располагать в строго определенном месте) то, все комментарии, расположенные не в соответствии с этим правилом, должны быть перенесены или удалены. Несоответствие принятым стандартам грозит возникновением потенциальных ошибок. Если стандарт по каким-то причинам не соблюдается, то все «нарушения» должны устраняться. Если стандарт требует доработок в связи с изменившимися первоначальными предпосылками его создания, то его необходимо либо дорабатывать, либо менять. Слепое следование стандартам губительно, так же, как и постоянные попытки обойти их по необоснованным причинам.

Проектирование архитектуры программных продуктов – это одна из активностей, для которых должны быть приняты определенные стандарты деятельности. При этом, современные требова-

ния к архитектурам предусматривают создание достаточно гибких и адаптивных архитектур.

Первое, «верхнеуровневое» проектирование, должно удовлетворять цели достаточно свободной и абстрактной реализации архитектуры, но при этом учитывать жесткие требования стандартов. Поэтому требования к программным продуктам принято делить на 2 типа критичности – высокоуровневые (функциональные) и низкоуровневые (нефункциональные) требования. Гибкость архитектур реализуется за счет разной степени принципиальности учета требований и их последующей реализации в программном коде будущих информационных систем. Требования к нефункциональным характеристикам должны быть более четкими и однозначными.

Типы требований мы уже немного обсуждали ранее и будем продолжать это делать по ходу нашего курса, изучая различные аспекты архитектур и архитектурного проектирования. Сейчас отметим, что подход к работе с ним должен быть также соответствующим образом стандартизован. За счет стандартов важно достичь согласованности между типами требований в момент их согласования друг с другом в виде отдельных компонентов архитектуры или программного продукта. Это позволит добиться непротиворечивого результата, в виде оптимальной архитектуры и эффективного программного обеспечения.

Большинство современных стандартов, описывающих работу с требованиями, выдвигают следующие характеристики, которым должен отвечать каждый из них:

- конкретность/измеримость;
- корректность/обоснованность;
- соответствие функциональности программного продукта;
- возможность последующей верификации;
- уникальность;
- последовательность;
- непротиворечивость;
- возможность декомпозиции;
- изменяемость с минимальными затратами;
- трассируемость.

На сегодняшний день существует множество программных продуктов, которые позволяют автоматизировать и унифицировать процесс разработки и последующего отслеживания требований.

Применение специализированного инструментария, поддерживающего принятые стандарты, в большинстве случаев, положительно сказывается на качестве процессов архитектурного проектирования, разрабатываемых архитектурах и программных продуктов в целом, за счет того, что их использование позволяет выявить те программные компоненты и детали, которые подвержены изменениям меньше остальных. Именно они и составят основу разрабатываемых архитектур, которая будет наименее гибкой, а те требования, которые чаще других изменяются, будут лежать в основе достаточно гибкой функциональности.

Но изменения, иногда вносятся и в «скелет».

Принципиальное следование всем принятым стандартам возможно только тогда, когда речь идет об устоявшихся отлаженных процессах архитектурного проектирования и разработки информационных систем, которые достигли своей наивысшей точки качественного развития. Утопия.

Принципы и темп развития современного мира определяют его черты.

Программные продукты, как часть этих черт должны быть в состоянии поддерживать соответствующую динамику и темпы. Элементы информационных систем и их архитектура предназначены для удовлетворения требований к разрабатываемым продуктам. Стоит понимать, что требования не только могут изменяться, но и качественно трансформироваться, переходя из категории нефункциональных в функциональные и наоборот.

Можно привести следующий пример:

В случае, если речь идет о стадии внедрения ПО, то такая характеристика как сопровождаемость воспринимается пользователями, как некритичная и не значимая с их точки зрения, но как только начинается стадия промышленной эксплуатации программного продукта, затраты на его поддержку и развития сразу дают о себе знать. Хорошо, если компания имеет достаточный для этого ресурс, но если это не так, то поддержка процессов и соответствующего для их выполнения уровня данных «кочует» на плечи бизнес пользователей и сразу переходит в разряд функциональных.

Все вышесказанное явно свидетельствует о значимости и необходимости стандартизированного подхода к созданию новых, инновационных, прорывных информационных продуктов, которые в

своей основе будут иметь надежные, качественные, производительные и масштабируемые архитектуры, которые позволят создавать функционально гибкие продукты.

Без четких, ясных требований к выполняемым функциям создать программное обеспечение невозможно. Требования – фундамент каждой разработки.

Процессы разработки должны начинаться с процесса разработки требований и вестись в соответствии с планом, корректируемым по «ходу».

2 Инструментарий разработки и моделирования требований к процессам и архитектуре программных продуктов

Для перехода от набора требований, порой разрозненных, к стройной и логичной архитектуре программного продукта, нужно использовать специализированный инструментарий. В нашем случае это специальные технологические средства разработки и моделирования процессов, на основе выявленных требований, которые позволят создать единое видение разрабатываемой архитектуры, с учетом определенных правил представления информации, принятой для конкретного проекта по созданию программного продукта или процесса архитектурного проектирования.

Тему правил, методов, методологий, в соответствии с которыми разрабатывается архитектура программных продуктов, мы будем подробно рассматривать в лекции №4.

Как было отмечено ранее, архитектурное проектирование информационных систем – это область деятельности, которая имеет много общего с архитектурным проектированием в строительстве.

«Строительство» – старший собрат информационных технологий. Поэтому было бы оптимально, для процессов создания программных продуктов многое необходимое для развития данной профессиональной области, почерпнуть, поинтересовавшись тем, как в строительстве успешно решались схожие задачи. В гражданском и промышленном строительстве языком описания архитектуры являются архитектурно-строительные чертежи, объемные прототипы и модели, текстовые описания возводимых объектов. Младший собрат поступил правильно и перенял необходимый опыт у старшего. В качестве средств разработки и моделирования характеристик информационных продуктов, в архитектурном

проектировании используются различные языки, которые принято называть нотациями, а инструментами, которые поддерживают работу с нотациями – CASE средствами.

Из современных нотаций можно выделить следующие:

- Блок-схемы (схемы алгоритмов)

Данный тип схем (графических моделей) визуализирует разрабатываемые алгоритмы или процессы. В описании блок-схем принято отдельные стадии процессов изображать в виде блоков (отсюда и соответствующее название данной нотации). Блоки различаются в зависимости от семантики, содержащейся в представляемой ими форме. Отображаемые блоки соединены между собой линиями, имеющими направление и определенную алгоритмами последовательность.

В широком значении термина «блок-схемы» он является метанотацией. Различные спецификации «блок-схем» получили свое определенное назначение и форму представления необходимой информации, в зависимости от специфики каждой конкретной нотации, но при этом общие правила построения моделей наследуются от правил построения «блок-схем».

- DFD-диаграмма (диаграмма потоков данных)

DFD – это нотация структурного анализа.

В данной нотации принято описывать внешние, по отношению к разрабатываемому продукту:

- Источники данных;
- Адресаты данных;
- Логические функции;
- Потоки данных;
- Хранилища данных.

Диаграмма потоков данных – это один из первых и основных инструментов структурного анализа и проектирования верхнеуровневой архитектуры программных продуктов, существовавших до последующего широкого распространения UML.

В основе данной нотации находится методология проектирования и метод построения модели потоков данных проектируемой информационной системы. В соответствии с соответствующей методологией проектирования модель системы идентифицируется как иерархия диаграмм потоков данных, которые представляют собой последовательный процесс преобразования данных, с момента их

поступления в систему, до момента представления информации конечному (или псевдоконечному) пользователю.

Диаграммы потоков данных разработаны для определения основных процессов или модулей программного продукта. Далее, в целях комплексного представления разрабатываемой архитектуры, они детализируются диаграммами нижних уровней представления информации (DFD) и процессов (IDEF). Подобный принцип отображения данных продолжается, реализуя многоуровневую иерархию подчиненных и взаимосвязанных диаграмм. В результате будет достигнут нужный уровень декомпозиции, на котором процессы становятся абсолютно понятными и прозрачными.

В современных процессах разработки программного обеспечения подобный подход к проектированию программных продуктов практически не применяется, по причине смещения акцентов создания информационных систем от структурного к объектно-ориентированному подходу.

Данная методология анализа и проектирования на сегодняшний день эффективно используется в бизнес-анализе и соответствующих дисциплинах, по причине ее наглядности и понятности для стэйкхолдеров.

- ER-диаграмма (диаграмма сущность-связь)

ER – это тип нотации, задача которой состоит в создании формальной конструкции, описывающей и визуализирующей верхнеуровневое представление бизнес объектов будущей системы, их атрибуты и связь между ними, в виде основных элементов, которые будут использоваться в качестве фундамента для проектирования таблиц будущей базы данных программного продукта. Таким образом, ER-диаграмма представляется как концептуальная модель создаваемой информационной системы, в терминах конкретной предметной области. Данный тип диаграмм помогает выделить ключевые сущности и определить связи между ними. На сегодняшний день существует достаточно мощный инструментарий, который может позволить выполнить преобразование созданной ER-диаграммы в конкретную схему базы данных на основе выбранной модели данных.

- EPC-диаграмма (диаграмма событийной цепочки процессов)

Событийная цепочка процессов – это тип нотации, задача которой заключается в создании моделей бизнес процессов. Она используется для моделирования, анализа и реорганизации бизнес-процессов. Ее отличие, от ранее рассмотренных, состоит в том, что EPC применяют для создания функциональных моделей, имитирующих конкретные процессы, реализуемые в определенном программном продукте.

История развития данного типа диаграмм связана с разработкой одноименного метода в рамках работ по созданию методологии ARIS (Architecture of Integrated Information Systems – Архитектура интегрированных информационных систем).

Диаграмма EPC оформляется в виде упорядоченной последовательности событий и функций. Каждая функция при этом должна определяться начальными и конечными событиями, могут быть указаны участники, исполнители, материальные и информационные потоки, сопровождающие отдельные функции. Дополнительную ценность диаграммам EPC придает тот факт, что она может использоваться для настройки программных продуктов типа ERP, в части создания или улучшения бизнес-процессов.

- BPMN-диаграммы

Один из самых распространенных типов нотаций на сегодняшний день – это BPMN (Business Process Model and Notation).

EPC, ER, BPMN – это не просто нотации, а методы реализации конкретных моделей процессов. BPMN по своему назначению более уместно сравнивать с диаграммами EPC. Причина этого заключается в том, что они имеют схожую цель – функциональное моделирование бизнес процессов. Рассматриваемая диаграмма самый новый, из приведенных в обзоре, принятый стандарт моделирования бизнес процессов. Главное позиционируемое преимущество диаграммы BPMN – понятная всем стэйкхолдерам разрабатываемой архитектуры и программного продукта визуализация моделируемых процессов.

Нотация имеет достаточно ясный инструментарий, позволяющий моделировать как относительно простые, так и сложные бизнес-процессы. Это достигается за счет применения двух групп элементов:

- Первая группа (simple notation) состоит из основных элементов BPMN, которые удовлетворяют требованиям создания про-

стой графической нотации. Подавляющее большинство бизнес-процессов моделируются с использованием элементов данной группы.

- Вторая группа (powerful notation) содержит полный «пакет» элементов BPMN. В него кроме основных блоков входят специфические комплексные модули, представляющие различные типы и виды элементов, необходимых для моделирования самых сложных процессов.

Подобная преемственность инструментария групп BPMN позволяет удовлетворять требованиям комплексной нотации и управлять более сложными ситуациями моделирования.

Оптимально смоделированные диаграммы BPMN могут применяться для настройки, рефакторинга и реинжиниринга бизнес процессов во многих современных информационных системах класса ECM.

- UML – диаграммы

UML (Unified Modeling Language) – не просто нотация или группа нотаций, а язык графического описания для объектного моделирования.

UML – это язык широкого профиля, который создан для формирования серии нотаций, поддерживающих полный цикл разработки, как архитектуры, так и функциональности программных продуктов, реализуемых по принципам объектно-ориентируемого программирования.

Широкая распространенность и повсеместное применение данного языка привели к тому, что он стал открытым стандартом, использующим графические обозначения для создания моделей систем, которые принято называть UML-моделями. Специфика UML не предполагает ориентацию на описание архитектуры, так как основное его назначение заключается в определении, визуализации, проектировании, документировании, преимущественно программных продуктов, бизнес процессов и структур данных.

Язык UML получил популярность за счет жесткой связки с популярной и распространенной методологией разработки и внедрения программных продуктов RUP, в соответствии с которой организована работа многих консалтинговых и производственных компаний отрасли информационных технологий. Не смотря на множество преимуществ, UML не является панацеей и абсолютным универсу-

мом в процессах проектирования архитектур и программных продуктов.

Применять и разбираться в диаграммах UML могут только узкоспециализированные специалисты области проектирования программных систем. Вовлечь в обсуждение и согласование разрабатываемой архитектуры информационного продукта, задокументированного на UML, стэйкхолдеров, представителей не направления информационных технологий, будет достаточно сложно.

UML, в отличие от ER и BPMN диаграмм, поддерживает генерацию не просто отдельной функциональности, привязанной к специфичным типам информационных систем, а целостного программного кода на основании сформированных UML-моделей.

- EIP-диаграммы

Самая специфичная и наименее распространенная нотация из всех рассматриваемых в данной лекции.

EIP (Enterprise Integration Patterns) диаграммы – это набор из 65 шаблонов диаграмм, которые предназначены для описания специализированных процессов взаимодействия между программными продуктами. Специфика данной нотации состоит в том, что она разработана для описания конкретных процессов, обеспечивающих интеграционное взаимодействие между приложениями и функциональность программных продуктов, ориентированных на обмен сообщениями между информационными системами.

Шаблоны интеграции приложений и необходимая функциональность, описание которых поддерживают EIP диаграммы, встроены в множество современных специализированных открытых программных продуктов и доступны для применения в специализированных интеграционных процессах.

На сегодняшний день, в силу своей специфичности, данный вид нотации не имеет широкого распространения, но, учитывая темпы роста популярности интеграционных процессов, можно спрогнозировать, что в ближайшее время интерес к EIP-диаграммам вырастет.

- Модели компонентов, программных продуктов и функциональных процессов

Модели, как таковые, не являются нотациями, а представляют собой набор необходимых для реализаций процессов/подпроцессов, представленных в виде нескольких взаимодополняющих друг друга

нотаций, с целью моделирования конечного результата. Говоря о модели, со структурной точки зрения, корректно будет сказать, что это «метанотация», включающая в себя несколько основных нотаций, реализация диаграмм которых запланирована для качественной разработки конкретной архитектуры.

Таким образом, модель – это абстракция, которая фокусирует субъект, работающий с ней на основных, критичных для реализации, характеристиках конечного продукта. Из примеров CASE средств, способных реализовать и представить нотации в виде модели можно выделить ARIS, UML.

Так же возможна схема, когда исполнитель разработает необходимые диаграммы в разных CASE средствах, после чего компилирует модель самостоятельно. В таких случаях дальнейшее сопровождение и развитие модели затруднительно (сложность сопровождения, развития, управления моделью) и целиком и полностью лежит на «плечах» ответственных за это специалистов.

- Прототипы

Когда мы говорили о моделях, то отметили, что модель – это следующий, качественный шаг вперед к реализации программного обеспечения, по сравнению с диаграммами, реализуемыми нотациями.

После того, как реализованы несколько моделей требований, можно приступать к реализации прототипов, если это предусмотрено и спланировано в проекте разработки программного продукта.

Прототип представляет собой «черновой» вариант конечной реализации программного продукта или его компонента, разработанного с учетом определенных ограничений для демонстрации заинтересованным сторонам на определенной стадии разработки только архитектуры или программного продукта в целом.

Цель разработки прототипа – подтвердить ожидания стэйкхолдеров от реализации конечного продукта и согласовать её функциональность. Прототипы принято реализовывать в специализированном программном коде, поэтому их следует воспринимать как часть процесса разработки, но при этом не рассчитывать на то, что прототип будет являться частью будущей архитектуры или информационной системы.

Прототипы, как правило, разрабатываются достаточно быстро с помощью специализированного инструментария, освоение кото-

рого требует гораздо меньше времени и сил, чем освоение определенного языка программирования.

Жизненный цикл прототипа заканчивается в тот момент, когда ожидания стэйкхолдеров подтверждены и разработанный облик (не более) будущего продукта запланирован к реализации.

- Интерфейсы:

Работа по созданию интерфейсов – это отдельная область разработки программных продуктов, которая не определяет архитектуры, но в большинстве случаев зависит от неё. Именно поэтому мы решили упомянуть её в конце списка, факультативно.

Эта область требует к себе довольно много внимания и сил. Она определяет многие функциональные характеристики программных продуктов (эргономика, дизайн интерфейсов, воспринимаемость и удобство работы с информацией и т. д.).

Значимость корректных и эффективных интерфейсов все больше и больше растет для разнообразных программных продуктов. Интерфейс представляет собой «верхушку», с помощью которой пользователь будет взаимодействовать с «нутром» программного продукта.

Важно, чтобы уже на ранних стадиях разработки интерфейсов пользователь имел представление о том, с чем и каким образом ему придется работать и постепенно привыкал, при возможности корректируя его или свои ожидания от него.

На текущий момент есть множество инструментов для создания интерфейсов разной степени сложности и детальности. Мы выделим следующие Balsamiq mockup, Axure RP и пр.

Каждый крупный программный продукт или его модуль, который состоит из совокупности компонентов и связей между ними, должен быть детально описан в соответствующий момент процесса архитектурного проектирования. Описание должно быть выполнено по стандартизированным правилам организации или проекта с использованием необходимых нотаций для данного конкретного случая.

По принципам программной инженерии и здравой логики, которая находится в основе процессов разработки информационных систем, любая подсистема или компонент в дальнейшем может выступать в качестве составного элемента более масштабной системы. Поэтому в архитектурном проектировании должно обязательно со-

держаться подробное описание укрупнённых частей системы, выполненных с помощью CASE средств и нотаций, согласованных друг с другом и обеспечивающих последующую преемственность.

Важно, чтобы выбранная нотация или серия нотаций поддерживали существующий процесс архитектурного проектирования, способствовали его развитию и совершенствованию и были «в силах» поддержать соответствующую фиксацию функциональных, нефункциональных и прочих требований к разрабатываемой архитектуре и программному продукту в целом.

Необходимо осознавать, что нет единой нотации или языка проектирования, который мог бы быть решением всех поставленных задач процесса проектирования. Сегодня существует достаточное количество универсальных инструментов и средств, но ни один из них не может обеспечить весь спектр возникающих задач проектирования.

Для того, чтобы создать достаточно комплексную и взаимосвязанную среду проектирования архитектур, моделей и функциональности программных продуктов нужно использовать инструменты и диаграммы, которые смогут поддержать процессы разработки, в зависимости от специфики конкретной задачи, но при этом должен существовать минимальный набор необходимых диаграмм, который должен присутствовать в каждом проекте по разработке программного продукта не смотря на его масштабы и значимость.

3 О функциональных и нефункциональных требованиях к архитектуре и функциональности программного обеспечения

Обсуждение различных типов и видов требований мы начали в предыдущей лекции.

В этой части мы уделим больше внимания рассмотрению функциональных и нефункциональных требований к архитектуре программного обеспечения.

Принято думать, что архитектура формируется под воздействием, прежде всего, нефункциональных требований, которые, в большинстве случаев, отсутствуют в проектируемых функциональных и бизнес моделях систем, но начинают свой жизненный цикл в процессе реализации программных продуктов. Попробуем обосновать, что это не совсем верно.

Многие современные ученые области инженерии требований высказывают мнение о том, что не бывает нефункциональных требований.

Нефункциональные требования – это абсолютно функциональные требования, которые описывают функции системы с точки зрения «каких-то» стэйкхолдеров, о которых забыли в процессе сбора и анализа требований. Вы знаете какое-нибудь программное обеспечение, в процессе разработки которого активно участвовали будущие специалисты групп сопровождения, тестирования, развития его архитектуры и функциональности? Как часто проектируя программные продукты о многих его характеристиках, не значимых на первый взгляд для бизнес стейкхолдеров, но критичных для других пользователей, просто забывают или намеренно игнорируют, считая их не значимыми и второстепенными.

Ранее мы показали, что члены группы поддержки, тестирования, системные администраторы и некоторые другие технические специалисты являются такими же стэйкхолдерами, как и будущие основные бизнес пользователи системы. Игнорирование их требований может привести к тому, что фаза внедрения пройдет достаточно успешно, но вот последующее сопровождение и развитие системы может быть достаточно проблематичным.

Основная задача заключается в том, что для современного бизнес мира «последующее» значит то, о чем можно забыть сегодня и не вспоминать до тех пор, пока забытое не напомнит о себе. В случае с нефункциональными требованиями такой подход к работе может оказаться слишком дорогостоящим. Как только информационный продукт перейдет в стадию промышленной эксплуатации и над ним начнут работать группы специалистов, которые должны будут поддерживать достаточно высокий уровень сервисной поддержки и развития системы, может оказаться, что его реализация не включает в себя множество разнообразных технических аспектов. Ведь именно от таких аспектов зависит качество продукта, обеспечивающее оптимальность бизнес процессов, моделей и данных. Довольно распространенная ситуация, когда на ранних фазах создания программных продуктов о таких характеристиках как надежность, быстродействие, безопасность многие специалисты и стэйкхолдеры, вовлеченные в процесс проектирования архитектуры и функциональности просто не задумываются, отдавая их на

дальнейшую проработку и реализацию разработчикам. Все бы хорошо, но для разработчиков существует множество факторов, в соответствии с которыми, модели программных компонентов, которые не имеют четких требований будут реализованы не так как задумывалось при составлении специализированных документов, а так, как разработчик «сможет». Это «сможет» будет определяться квалификацией, инструментарием и, конечно же тем количеством времени, которое у него/них будет в наличии, а его, как известно, практически всегда не хватает. Таким образом, получается следующий парадокс – качество продукта будет напрямую зависеть не от стэйкхолдеров, а от специалистов, для которых важность продукта, который они разрабатывают, не очевидна. Стэйкхолдерам важно получить законченную функциональность в сроки согласованных работ, а иногда даже намного, намного ранее и получить дополнительное время на реализацию качественных атрибутов, значимость которых определяется только в процессе разработки. Именно поэтому многие характеристики разрабатываемого программного продукта и его архитектуры, скрытые от внимательного взора руководства, необходимо обозначать в активностях, предваряющих стадии проектирования и разработки. Многое на этой стадии зависит от опыта и мастерства системного архитектора и его команды, от профессионализма которых будет зависеть стратегический успех информационной системы, разрабатываемой для бизнес необходимостей конкретной организации.

Этим вступлением мы постарались показать то, что требования не бывают второстепенными или незначимыми. Незначимое сегодня окажется жизненно необходимым завтра. Рамки работ над требованиями должны быть спланированы и ясно понятны всем стэйкхолдерам, участвующим в проекте.

После того, как мы определились с тем, что для создания качественной архитектуры не должно быть разного отношения к функциональным и не функциональным требованиям. Все требования необходимо рассматривать с точки зрения их критичности по отношению к архитектуре программного продукта.

Высокоуровневое назначение программного продукта – приносить выгоду его владельцу за счет автоматизации труда (интеллектуального, физического, монотонного и т. д.) человека, а в идеа-

ле за счет полной замены человеческого ресурса и фактора, если это возможно.

Здесь будет уместно сравнение со строением человека, в котором есть множество органов, каждый из которых отвечает за определенную функцию и каждый развивается не только как самостоятельная единица, но и как часть целого. В тот момент, когда определенный орган дает сбой и не может поддерживать свою целевую функциональность, при условии что это не отражается на деятельности всего организма, то можно считать, что это не критично, но если весь организм выходит из строя на определенный период, то важно понять в чем состоит источник проблемы и устранить его, а если подойти системно, то не допустить подобной возможности. Таким образом, те элементы органов, которые влияют на их полную функциональность и взаимосвязаны с другими, являются наиболее приоритетными для исследования и оптимального содержания и развития. Подобными компонентами в программном продукте являются компоненты, наиболее критичные для рассмотрения с точки зрения архитектуры и сбора требования.

Но при разработке определенной информационной системы, классификация требований необходима для целей дальнейшей их трассировки и управления ими. Традиционно выделяют 2 основные группы описание которых и более подробную литературу по работе с которыми наши коллеги найдут в соответствующей литературе.

Сначала мы рассмотрим функциональные требования.

Функциональные требования описывают «поведение» системы и информацию, с которой система будет работать. Они описывают возможности системы в терминах поведения или выполняемых операций

Цель использования данной группы требований (как следует из названия) заключается в регламентации возможностей и соответствующем им поведении разрабатываемого программного продукта.

Функциональные требования должны отвечать на вопрос – каким образом должны быть алгоритмизированы процессы информационного продукта, чтобы взаимодействие между пользователем и системой удовлетворяло потребности стэйкхолдеров?

Именно с помощью функциональных требований, в большинстве случаев, определяются рамки работ по процессам, сопровож-

дающим цикл создания программных продуктов. Данный тип требований устанавливает:

- Цели разрабатываемого функционала;
- Задачи, которые должны быть выполнены для достижения поставленной цели;
- Сервисы, поддерживающие выполнение задач;
- … (и многое другое, касающееся непосредственно функциональных возможностей и особенностей программного продукта).

На сегодняшний день существует множество подходов к разработке и фиксации функциональных требований. Кратко рассмотрим наиболее популярные из них:

- Классический подход

Суть классического подхода состоит в разработке требований с помощью итеративной работы с верхнеуровневыми требованиями стэйкхолдеров, и постепенной детализации до уровня понятного разработчикам. Это наиболее изученный и популярный подход, который подробно описан в современной литературе. Мы можем порекомендовать для изучения книги К. Вигерса, которые уже упоминались нами ранее.

Дополнительно отметим, что в подобном подходе основная тяжесть создания полноценного документа, охватывающего весь программный продукт, ложится на сотрудника, ответственного за сбор, анализ и синтез информации. При современной динамике изменений бизнес процессов и данных, поступающих от внешних и внутренних аспектов бизнеса, непосредственно влияющих на требования стэйкхолдеров, классический подход становится наименее эффективным. Именно поэтому в последнее время водопадные модели разработки программного обеспечения заменяются «гибкими» подходами (agile), цель которых в быстром и эффективном решении возникающих задач, даже если следующая будет противоречить предыдущей.

- Use cases (Варианты использования);

В этом подходе функциональные требования записываются с помощью системы специализированных правил, которые, к примеру, должны фиксироваться следующим образом: «программный продукт должен обеспечить учетчику возможность формирования акта выдачи бланков строгой отчетности».

Если определить максимальное количество возможных вариантов использования разрабатываемого программного продукта предполагаемыми целевыми пользователями, то мы получим достаточно полные функциональные требования.

Но, при попытке тотального применения к работе над функциональными требованиями этого подхода существует вероятность того, что отдельные, незначимые для определенных стэйкхолдеров, потребности будут не учтены. В этом случае существует вероятность упустить суть конкретных функциональных требований, которые, как правило, скрыты от постороннего взгляда. Чтобы этого не произошло, важно использовать в обработке зафиксированных use cases классический подход, в рамках которого должен быть проведен анализ, систематизация и синтез информации. Это поможет выявить истинное назначение предполагаемого к реализации функционала и не разрешить отдельных деталей.

Нефункциональные требования, в дополнение к функциональным, направленны на обеспечение технической целостности разрабатываемого функционала и поддержку характеристик реализуемого программного обеспечения, которые необходимы для создания оптимальной архитектуры.

Они регламентируют внутренние и внешние условия функционирования программного продукта. Выделяют следующие основные группы нефункциональных требований:

- Атрибуты качества;
- Безопасность;
- Надежность;
- Производительность;
- Скорость и время отклика приложения;
- Пропускная способность workflow;
- Количество необходимой оперативной памяти;
- Ограничения;
- Платформа реализации архитектуры и программного продукта;
- Тип используемого сервера приложений.

Нефункциональные требования описывают условия, которые не относятся к поведению и функциональности системы, но обеспечивают их на уровне компонентов архитектуры программного продукта.

Ранее описанная методология use cases может быть применена для сбора и анализа нефункциональных требований. При взаимодействии со стэйкхолдерами необходимо фиксацию каждого правила подытоживать фиксацией нефункциональных требований, выраженную в виде количественной характеристики определенного параметра: «программный продукт должен обеспечить учетчику возможность формирования акта выдачи бланков строгой отчетности за время не более 0,5 с после того, как поступила соответствующая команда»

В этой части нашего курса мы постарались достаточно подробно рассказать о требованиях, которые оказывают свое влияние на архитектуру и функциональность программных продуктов, в том числе достаточно подробно рассмотрели функциональные и нефункциональные требования. Но предложенный объем является необходимым и достаточным только для того, чтобы у наших коллег сложилось понимание того, какую роль играют требования в процессах проектирования архитектуры и реализации программных продуктов. Более основательное изучение темы инженерии требований изложены в специализированых курсах и литературе, к которым мы и рекомендуем Вам обратиться.

4 Прочие виды требований

Любое предприятие, которое осознанно пришло к необходимости создания или применения программного продукта, обладающего оптимальной архитектурой, для определенных условий и аспектов деятельности, можно считать сложной организационно-технической системой, которая имеет определенные цели и реализует конкретные функциональные задачи. При предпосылках использования программных продуктов в условиях организационного управления, процесс исполнения каждой задачи нуждается в конкретных ресурсах для реализации цели функционирования предприятия.

Сложная организационно-программная система состоит из иерархических уровней и поддерживающих их программных структур (исполнители бизнес процессов и необходимые программные продукты), каждый из которых постоянно расширяется и развивается. Из этого следует, что развитие каждого отдельного элемента (новая версия информационной системы с исправленными ошибками и до-

полнительными функциональными возможностями или развитие конкретного исполнителя) будет приводить к улучшению характеристик программного продукта, используемого на конкретном предприятии. Но целенаправленное развитие можно будет обеспечить только при условии соблюдения требования взаимодействия всех составных компонентов организации.

Запланированное функционирование бизнес процессов, поддерживаемых архитектурой программных продуктов, необходимо обеспечить требованиями к ресурсам, которые могут быть кратко выражены следующими аспектами:

- Время:

Время на обучение пользователей, стабилизацию реализованных и внедренных принципов работы, поддерживающих достижение целей бизнес процессов, время на осознание организацией «хозяйнических» инстинктов по отношению к программному продукту

- Финансы:

Финансовые ресурсы должны обеспечить необходимый уровень поддержки, выраженные в оптимальном количестве квалифицированных исполнителей их профессиональной мотивации, требуемом уровне технической оснащенности и пр.

- Данные и информация:

Для того, чтобы результат деятельности соответствовал ожиданиям стейкхолдеров требуются данные надлежащего качества, на основе которых стало бы возможным достижение поставленных целей и качественной трансформации данных в значимую для бизнеса информацию. В дальнейшем повторное использование подобной информации позволит повысить уровень ценности тактических и стратегических процессов и способствовать повышению уровня экспертности и зрелости компании в целом.

Если перейти от организационных требований к реализации архитектуры программных продуктов, общая специфика большинства современных практик процессов создания архитектур, то можно выявить ряд общих особенностей, которые заключаются в:

1. Преимущественной ориентированности на водопадные модели процессов реализации архитектур и программных продуктов;

2. Подавляющем фокусировании на архитектуре программного продукта и практически полном игнорировании того факта, что система включает в себя не только информационно-программные

компоненты, но и другие аспекты (технические средства, персонал), которые также должны быть рассмотрены и учтены при проектировании решения;

3. Отделение активности технико-экономического обоснования реализации программного продукта от разработки бизнес–процессов и разработки архитектуры системы;

После детального и компетентного изучения специализированных методик (с которыми мы познакомимся чуть позднее) результатом их осмысления можно сформулировать следующие требования к процессам проектирования и реализации архитектуры, в частности, и программных продуктов в целом:

- Проектирование и создание архитектуры, бизнес–анализ, технико-экономическое обоснование создания продукта, моделирование процессов должны быть неразрывно связаны друг с другом и изменение одного из составляющих должно запускать процессы анализа влияния и управления изменениями;
- Сущность процессов проектирования и разработки архитектуры, функциональности программных продуктов должна быть преимущественно итерационной;

В практике создания программных продуктов, каждая реализуемая информационная система имеет собственную специфику, выраженную определенными условиями и факторами, которые имеют различную природу возникновения и силу влияния на архитектуру и функциональность. Часть из них мы рассмотрели в предыдущей лекции, часть из них мы будем рассматривать в следующей.

Активность инженерии требований, которая ставит своей целью работу с требованиями – это отдельная область, которая нуждается в подробном рассмотрении. Если Вас заинтересовало данное направление отрасли информационных технологий, вы сможете самостоятельно приступить к ее изучению, используя общедоступные источники информации.

Мы же далее будем рассматривать только те аспекты, которые оказывают достаточно сильное влияние на предмет нашего курса.

5 Зависимости и связи между различными видами требований, функциональности и архитектуры программного обеспечения

После того, как мы определились с основными видами требований, которые необходимо фиксировать, формализовывать и реализовывать в процессах проектирования и разработки для целей реализации архитектуры и функциональности программных продуктов, стоит задуматься о соблюдении целостности архитектуры, необходимой для:

- создания оптимального функционала;
- реализации значимых аспектов разрабатываемого программного обеспечения;
- разработки кроссбизнес-процессов, поддерживаемых архитектурой и функциональностью программного продукта;
- взаимосвязи между различными стадиями процессов разработки и внедрения программных продуктов (интеграция, миграция данных и пр.);
- и др.;

Каждое требование, зафиксированное в целях создания информационной системы, по ходу стадий проектирования, разработки, тестирования и внедрения программного продукта должно быть трансформировано в определенный программный модуль, тестовую процедуру, пункт инструкции пользователя и т. д. – это один из основных постулатов создания качественного и адекватного программного продукта, который называется трассирование (трассировка).

Трассирование представляет собой процесс или атрибут в рамках реализации информационный системы, который обеспечивает связь между его элементами и функциональными процессами. Трассировка должна способствовать установлению связи между:

- всеми видами требований;
- функциональными и нефункциональными процессами;
- результатами;
- необходимой отчетностью.

Оптимально выстроенный процесс трассирования должен ясно и однозначно позволять понять что (?), откуда (?), каким образом (?) вышло, и куда (?), и в каком виде (?) поступило.

К примеру можно привести стандарты создания программных продуктов для авиационной промышленности. В них зафиксирова-

но, что команда, задействованная в разработке программного продукта должна в любой момент времени жизненного цикла программного продукта уметь отследить цепочку от результатов тестирования, к самим тестам, от тестов к бинарному коду, от бинарного кода к исходному коду, от исходного кода к низкоуровневым требованиям, от низкоуровневых требований к архитектуре, от архитектуры к высокоуровневым требованиям, от высокоуровневых требований к системным требованиям и в обратную сторону. Но, как правило, в стандартах не определяется способ трассирования. Это оставляет системным архитекторам или другим специалистам, ответственным за создание информационных систем, определенную свободу действий, регламентированную принципами здравого смысла и необходимостью получения оптимального результата для конкретных заданных условий.

Когда мы говорим о трассируемости документов, содержащих требования к разрабатываемому продукту, то вполне достаточно обеспечить начальную связность на первых этапах разработки за счет ссылочности создаваемых документов и уникальной идентификации каждого конкретного требования. Если же мы начнем обсуждать непосредственно процессы проектирования, разработки требований и кода программного обеспечения, то необходимо осветить процессы валидации, верификации и тестирования.

Под валидацией понимается процесс, направленный на доказательство того, что верхнеуровневые требования стэйкхолдеров будут полностью удовлетворены и покрыты в разработанной функциональности программного обеспечения.

Верификация является активностью процесса валидации, цель которой проверка и последующее достижение соответствия между требованием и реализованными архитектурой и функциональность программного обеспечения.

Тестирование представляет собой «сугуботехническую» часть активности верификации, направленную на испытание программного продукта.

При выполнении тестирования должны быть решены 2 основные задачи:

1. Демонстрация соответствия требований реализации программного продукта;

2. Выявление ситуаций и аспектов, в которых функциональность и архитектура является несоответствующим зафиксированных в документах требованиям с последующим выполнением п. 1.

Необходимо четко представлять, что ни один из представленных процессов, сам по себе, не обеспечит качественной трассируемости требований с момента их фиксации до момента промышленной эксплуатации программного продукта. Даже наиболее развитый и формализованный процесс тестирования программного обеспечения не позволяет однозначно, точно и полностью выявить все несоответствия и установить адекватную функциональность, необходимую для достижения поставленных результатов. Причины этого заключаются в множестве разнообразных факторов, основной из которых это пресловутый человеческий фактор, к более подробному рассмотрению которого мы периодически возвращаемся в процессе изучения нашего курса.

Только в совокупности, используя свойства, порожденные эффектом эмерджентности, можно добиться достижения системных результатов от процессов трассировки.

При выполнении каждой стадии работы над требованиями и разрабатываемой архитектуры и функциональности информационной системы, в процессы проектирования и реализации должны быть вовлечены все стэйкхолдеры, которые при необходимости смогут прояснить проблемную ситуацию, в рамках зафиксированных требований. Относительно частая связь между разработчиками и стэйкхолдерами будет способствовать:

- повышению прозрачности процессов проектирования и разработки для влиятельных стэйкхолдеров;
- эффективности процессов взаимодействия и оптимизации связей между различными членами команды, заинтересованными сторонами и другими вовлеченными в общие процессы взаимодействия пользователей.

Эти факторы в совокупности приведут к повышению степени доверия между исполнителями и заказчиками и, как следствие, будут позитивно влиять на степень удовлетворенности пользователей и эффективность разрабатываемого программного продукта.

Тема трассирования, как многие рассматриваемые в нашем курсе, является полноценной областью сферы информационных технологий, для успешного изучения которых требуется затратить

определенное время, при этом уже обладая определенным багажом знаний и опыта, полученного во время практической работы над созданием информационных продуктов.

Те аспекты, которые мы осветили, являются основными и позволят нашим коллегам быть готовым к необходимости создания и участия в процессах трассировки.

6 Риски реализации архитектуры и методы управления ими

На текущий момент, когда в процессе изучения нашего курса мы подошли к стадии обсуждения рисков, важно немного более подробно рассказать о том, что именно мы будем понимать под рисками реализации архитектуры.

Риск – это потенциальная возможность наступления вероятного события/явления или их совокупности, которые могут вызывать определенное влияние (негативное или позитивное) на осуществляемую деятельность или реализуемый продукт.

Под рисками реализации архитектуры мы будем понимать совокупность факторов, управленческих решений и других аспектов, которые могут оказать влияние на конечный результат разработки архитектуры и функциональности программных продуктов. Мы не ставим себе целью предложить Вам способ или инструмент, который позволит справиться с рисковой составляющей, но опишем то, как стоит воспринимать конкретный риск и что надо сделать, чтобы быть готовым к работе с ним.

Как правило, риски реализации архитектуры можно ассоциировать со следующими причинами их возникновения:

- Попытка создания оптимальной архитектуры на основе уже существующей, но не удовлетворяющей ожидания заказчиков;

Довольно популярная ситуация, складывается в процессе вынужденного реинжиниринга, когда без обоснованного анализа причинно-следственных связей, разработчик или архитектор спонтанно принимают решение о переделке какого-то компонента. В данном случае риск заключается в том, что идея не перешла на стадию «выдержанного» предложения, а сразу начала претворяться в код. Через какое-то время становится понятно, что какой-то элемент остался недостаточно проработан. В таком случае, достаточно «благостном», приходится отказываться от реализации, но если какие-то части уже внедрены в функционирование и ждут своих доработок,

то приходится в режиме пожарника искать «обходные» варианты и усугублять имеющуюся информационную систему в целом

- В процессе реализации архитектуры программного продукта меняются ключевые участники команды, задействованной в работе над ней;

В том случае, когда новые члены команды, приступающие к работе над архитектурой и функциональностью программного продукта первым делом, не изучив причин реализации и их следствия, начинают недоумевать по поводу выбранного способа реализации. Им кажется, что все сделано не оптимально, не очевидно, слишком сложно. После того, как выполнены определенные доработки, в процессе тестирования начинают появляться дыры в уже разработанном решении и архитектура из целостного монолита становится «костыльной». В итоге систему требуется переделать почти полностью. Чем старше система, тем таких ситуаций больше

Описанные ситуации, как и множество других, особенно выделяются в длительных проектах.

На текущий день, в области создания информационных систем накоплен достаточно большой опыт, на основе которого возможно снижение, локализация или устранение вероятности наступления подобных ситуаций. Среди рисков реализации архитектуры следует выделить следующие, на которых мы сконцентрируемся в дальнейшем:

- риск смены разработчика;
- риск ошибочно принятого архитектурного решения.

Эти риски сильно влияют на предсказуемость процессов перепроектирования и разработки программного продукта. Что характерно, наступление первого усиливает вероятность наступления второго.

Риск смены разработчика связан с потерей ключевой информации об архитектуре и функциональности по причинам изменения ключевого лица. Многие мысли, воплощенные в коде, которые не нашли отражение на бумаге, не всегда понятны тому, кто не является их автором. Подобных рисков можно избежать полным документированием выбранных решений до того момента, как они будут применены. Это позволит относительно безболезненно единовременно выдержать смену любых участников команды. Ключевым

фактором успеха становится поддержка системности процессов документирования.

Причины рисков ошибочно принятых архитектурных решений заключаются в недостаточной трассируемости требований от стадии сбора информации, к стадии проектирования или же в недостаточной компетенции лиц, принимающих важные технические решения.

Для того, чтобы максимально обезопасить программный продукт от подобного рода рисков, необходимо как можно более комплексно подойти к решению некоторых вопросов еще на уровне формирования идеи. Все важные решения, связанные с проектированием, разработкой и последующим изменением архитектуры и функциональности программного продукта необходимо мысленно приостанавливать и стремиться задокументировать в виде концептуально проработанного предложения. Подобный подход помогает не только в принятии обоснованного и согласованного, с уже созданными компонентами информационно системы, решения, но и является своего рода дополнительной тренировкой аналитических способностей.

После того как идея получает свою материализацию на бумаге, в большинстве случаев, она может немного трансформироваться и становится более обдуманной и обоснованной.

Этот метод позволяет подойти к необходимым доработкам с позиции необходимых и достаточных трудозатрат и не тратить ценные ресурсы на «холостые ходы».

Каждое зафиксированное изменение в последующей стадии анализа и проектирования должно быть соотнесено с уже существующей архитектурой с помощью активности анализа влияния. Для этого есть множество разнообразных инструментов от общения с ответственными разработчиками, до построения диаграммы причинно следственных связей, на которых должно быть продемонстрировано то, как предлагаемое изменение будет влиять на программный продукт в целом. Если конструктивных моментов больше и изменение будет влиять положительно, то незамедлительно стоит переходить к его реализации, в противном случае имеет смысл отложить её или поискать более выигрышные варианты изменений.

В процессе работы над рисками важно помнить первоначальные цели, достижение которых является главным результатом нашей деятельности. Иногда бывает, что увлекшись поиском обходных решений и пытаясь минимизировать величину возможного ущерба, проектная команда, занятая поиском наиболее выигрышных ответов, отклоняется в сторону, и архитектура становится расплывчатой и слишком гибкой, что впоследствии отражается на её характеристиках и функциональности программного продукта. Кроме того, необходимо учитывать имеющиеся ресурсы, технологии и возможности бизнеса. Программный продукт в процессе своего функционирования должен поддерживаться и обеспечиваться в соответствии с тем количеством ресурсов, которое предусматривается для конкретной стадии его использования.

В том случае, если по ходу разработки программного продукта первоначальные предпосылки его создания изменились, подобная ситуация является возможностью для появления или увеличения количества и степени проявления всевозможных рисков (не только архитектурных, но и бизнес, операционных и т. д.).

Поэтому важно контролировать:

- Детали реализации архитектуры и функциональности информационной системы;
- Влияние принятых изменений:
- Не только на сам продукт, но и на величину ресурсов, необходимых для разработки и сопровождения (как бизнес, так и технических) принятых изменений;
- Ограничения, которые сопровождают:
- Разработку программного продукта;
- Жизненный цикл программного продукта с момента его выхода на рынок;

Контрольные вопросы

1. Какие технологии для проектирования архитектуры ПО существуют?
2. По каким принципам начинают разрабатывать архитектуру ПО?

4. Практическое занятие №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 практической работы № 4 студент должен изучить приведенный ниже теоретический материал. Отчет сдается в распечатанном и электронном (файл Word) видах.

Изучение работы в системе контроля версий

- Основы

Git – это набор консольных утилит, которые отслеживают и фиксируют изменения в файлах (чаще всего речь идет об исходном коде программ, но вы можете использовать его для любых файлов на ваш вкус). С его помощью вы можете 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, Bitkeeper и 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, вы можете 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

Итак, мы установили git, теперь 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 наше 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 мы отметили ранее, git хранит свои файлы и историю прямо в папке проекта. Чтобы создать новый репозиторий, нам 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 наш 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 должен выдать:

```
$ git status
On branch master
Initial 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 него файлы (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 master
Initial 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, нам нужно хотя бы одно изменение в области подготовки (мы только что создали его при помощи git add), после которого мы может 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 наш 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 мы хотим поделиться нашей 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 наш локальный репозиторий с репозиторием на GitHub, выполним следующую команду в терминале. Обратите внимание, что нужно обязательно изменить URI репозитория на свой.

```
# This is only an example. Replace the URI with your own repository address.
$ git remote add origin https://github.com/tutorialzine/awesome-project.git
```

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

- 2. Отправка изменений на сервер

Сейчас самое время переслать наш локальный коммит на сервер. Этот процесс происходит каждый раз, когда мы хотим 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 увидете 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_HEAD
Already 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 без риска, 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 мы запустим branch, мы увидим две доступные опции:

```
$ git branch
amazing_new_feature
* master
```

master – это активная ветка, она помечена звездочкой. Но мы хотим работать с нашей «новой потрясающей фичей», так что нам понадобится 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. Мы 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 мы откроем наш проект в файловом менеджере, мы 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 основной версии 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.

- 1. 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 вы можете 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, мы можем воспользоваться командой show [commit]

Вывод git show

Чтобы увидеть разницу между двумя коммитами, используется команда 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.

- 2. Возвращение файла к предыдущему состоянию

Гит позволяет вернуть выбранный файл к состоянию на момент определенного коммита. Это делается уже знакомой нам командой checkout, которую мы 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 мы возьмем файл hello.txt и откатим все изменения, 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
```

- 3. 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.

- 4. 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?

5. Лабораторная работа №1. Построение диаграммы Вариантов использования и диаграммы последовательности

Целью работы является изучение порядка построения диаграмм вариантов использования и диаграммы последовательности.

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

Построение диаграммы Вариантов использования и диаграммы последовательности

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

Разработка диаграммы вариантов использования преследует цели:

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

Суть данной диаграммы состоит в следующем: проектируемая система представляется в виде множества сущностей или актеров, взаимодействующих с системой с помощью так называемых вариантов использования. При этом актером (actor) или действующим лицом называется любая сущность, взаимодействующая с системой извне. Это может быть человек, техническое устройство, программа или любая другая система, которая может служить источником воз-

действия на моделируемую систему так, как определит сам разработчик. В свою очередь, вариант использования (use case) служит для описания сервисов, которые система предоставляет актеру. Другими словами, каждый вариант использования определяет некоторый набор действий, совершаемый системой при диалоге с актером. При этом ничего не говорится о том, каким образом будет реализовано взаимодействие актеров с системой.

Примечание

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

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

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

этих понятий мы и приступим к изучению диаграмм вариантов использования.

Вариант использования

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

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

Рис. 1. Графическое обозначение варианта использования

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

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

Варианты использования описывают не только взаимодействия между пользователями и сущностью, но также реакции сущности на получение отдельных сообщений от пользователей и восприятие этих сообщений за пределами сущности. Варианты использования могут включать в себя описание особенностей способов реализации сервиса и различных исключительных ситуаций, таких как корректная обработка ошибок системы. Множество вариантов использования в целом должно определять все возможные стороны ожидаемого поведения системы. Для удобства множество вариантов использования может рассматриваться как отдельный пакет.

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

Примечание

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

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

В метамодели UML вариант использования является подклассом классификатора, который описывает последовательности действий, выполняемых отдельным экземпляром варианта использования. Эти действия включают изменения состояния и взаимодейст-

вия со средой варианта использования. Эти последовательности могут описываться различными способами, включая такие, как графы деятельности и автоматы.

Примерами вариантов использования могут являться следующие действия: проверка состояния текущего счета клиента, оформление заказа на покупку товара, получение дополнительной информации о кредитоспособности клиента, отображение графической формы на экране монитора и другие действия.

Актеры

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

Рис. 2. Графическое обозначение актера

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

Примечание

Имя актера должно быть достаточно информативным с точки зрения семантики. Вполне подходят для этой цели наименования должностей в компании (например, продавец, кассир, менеджер,

президент). Не рекомендуется давать актерам имена собственные (например, «О.Бендер») или моделей конкретных устройств (например, «маршрутизатор Cisco 3640»), даже если это с очевидностью следует из контекста проекта. Дело в том, что одно и то же лицо может выступать в нескольких ролях и, соответственно, обращаться к различным сервисам системы. Например, посетитель банка может являться как потенциальным клиентом, и тогда он востребует один из его сервисов, а может быть и налоговым инспектором или следователем прокуратуры. Сервис для последнего может быть совершенно исключительным по своему характеру.

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

Примечание

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

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

Любая сущность, которая согласуется с подобным неформальным определением актера, представляет собой экземпляр или пример актера. Для моделируемой системы актерами могут быть как субъекты-пользователи, так и другие системы. Поскольку пользова-

тели системы всегда являются внешними по отношению к этой системе, то они всегда представляются в виде актеров.

Так как в общем случае актер всегда находится вне системы, его внутренняя структура никак не определяется. Для актера имеет значение только его внешнее представление, т. е. то, как он воспринимается со стороны системы. Актеры взаимодействуют с системой посредством передачи и приема сообщений от вариантов использования. Сообщение представляет собой запрос актером сервиса от системы и получение этого сервиса. Это взаимодействие может быть выражено посредством ассоциаций между отдельными актерами и вариантами использования или классами. Кроме этого, с актерами могут быть связаны интерфейсы, которые определяют, каким образом другие элементы модели взаимодействуют с этими актерами.

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

Интерфейсы

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

На диаграмме вариантов использования интерфейс изображается в виде маленького круга, рядом с которым записывается его имя (рис. 3, а). В качестве имени может быть существительное, ко-

торое 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 (рис. 3, б).

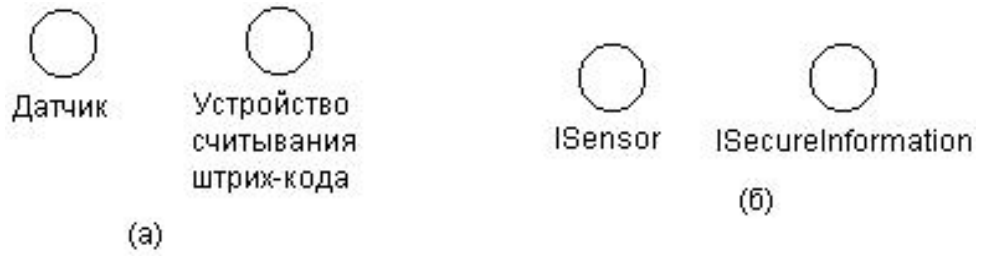


Рис. 3. Графическое изображение интерфейсов на диаграммах вариантов использования

Примечание

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

Графический символ отдельного интерфейса может соединяться на диаграмме сплошной линией с тем вариантом использования, который его поддерживает. Сплошная линия в этом случае указывает на тот факт, что связанный с интерфейсом вариант использования должен реализовывать все операции, необходимые для данного интерфейса, а возможно и больше (рис. 4, а). Кроме этого, интерфейсы могут соединяться с вариантами использования пунктирной линией со стрелкой (рис. 4.4, б), означающей, что вариант использования предназначен для спецификации только того сервиса, который необходим для реализации данного интерфейса.

Рис. 4. Графическое изображение взаимосвязей интерфейсов с вариантами использования

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

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

Примечания

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

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

Рис. 5. Примеры примечаний в языке UML

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

Отношения на диаграмме вариантов использования

Между компонентами диаграммы вариантов использования могут существовать различные отношения, которые описывают взаимодействие экземпляров одних актеров и вариантов использования с экземплярами других актеров и вариантов. Один актер может взаимодействовать с несколькими вариантами использования. В этом случае этот актер обращается к нескольким сервисам данной

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

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

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

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

Отношение ассоциации

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

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

Рис. 6. Пример графического представления отношения ассоциации между актером и вариантом использования

Кратность (multiplicity) ассоциации указывается рядом с обозначением компонента диаграммы, который является участником данной ассоциации. Кратность характеризует общее количество конкретных экземпляров данного компонента, которые могут выступать в качестве элементов данной ассоциации. Применительно к диаграммам вариантов использования кратность имеет специальное обозначение в форме одной или нескольких цифр и, возможно, специального символа «*» (звездочка).

Примечание

Возвращаясь к общей теории множеств, основы которой были рассмотрены в главе 2, следует заметить, что кратность представляет собой мощность множества экземпляров сущности, участвующей в данной ассоциации. Что касается самого понятия ассоциации, то это одна из наиболее общих форм отношений в языке UML.

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

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

Примером этой формы записи кратности ассоциации является указание кратности «1» для актера «Клиент банка» (рис. 6). Эта запись означает, что каждый экземпляр варианта использования «Оформить кредит для клиента банка» может иметь в качестве своего элемента единственный экземпляр актера «Клиент банка». Другими словами, при оформлении кредита в банке необходимо иметь в виду, что каждый конкретный кредит оформляется на единственного клиента этого банка.

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

{первое_число, первое_число+1, первое_число+2, ..., второе_число]. Очевидно, что первое число должно быть строго меньше второго числа в арифметическом смысле, при этом первое число может быть равно 0.

Пример такой формы записи кратности ассоциации – «1..5». Эта запись означает, что количество отдельных экземпляров данного компонента, которые могут выступать в качестве элементов данной ассоциации, равно некоторому заранее неизвестному числу из множества целых чисел {1, 2, 3, 4, 5}. Эта ситуация может иметь место, например, в случае рассмотрения в качестве актера – клиента банка, а в качестве варианта использования – процедуру открытия счета в банке. При этом количество отдельных счетов каждого клиента в данном банке, исходя из некоторых дополнительных соображений, может быть не больше 5. Эти дополнительные соображения как раз и являются внешними требованиями по отношению к проектируемой системе и определяются ее заказчиком на начальных этапах ООАП.

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

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

- Единственный символ «*», который является сокращением записи интервала «0..*». В этом случае количество отдельных экземпляров данного компонента отношения ассоциации может быть

любым целым неотрицательным числом. При этом 0 означает, что для некоторых экземпляров соответствующего компонента данное отношение ассоциации может вовсе не иметь места.

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

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

Более детальное описание семантических особенностей отношения ассоциации будет дано при рассмотрении других диаграмм в последующих главах книги.

Отношение расширения

Отношение расширения определяет взаимосвязь экземпляров отдельного варианта использования с более общим вариантом, свойства которого определяются на основе способа совместного объединения данных экземпляров. В метамодели отношение расширения является направленным и указывает, что применительно к отдельным примерам некоторого варианта использования должны быть выполнены конкретные условия, определенные для расширения данного варианта использования. Так, если имеет место отношение расширения от варианта использования A к варианту использования B, то это означает, что свойства экземпляра варианта использования B могут быть дополнены благодаря наличию свойств у расширенного варианта использования A.

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

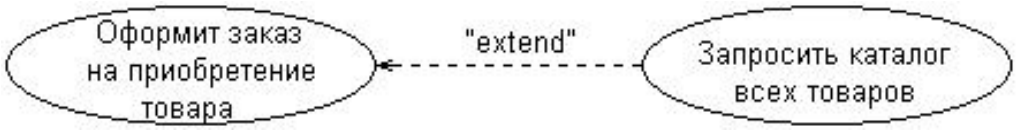


Рис. 7. Пример графического изображения отношения расширения между вариантами использования

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

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

Семантика отношения расширения определяется следующим образом. Если экземпляр варианта использования выполняет некоторую последовательность действий, которая определяет его поведение, и при этом имеется точка расширения на экземпляр другого варианта использования, которая является первой из всех точек расширения у исходного варианта, то проверяется условие данного отношения. Если условие выполняется, исходная последовательность действий расширяется посредством включения действий экземпляра другого варианта использования. Следует заметить, что условие отношения расширения проверяется лишь один раз – при первой ссылке на точку расширения, и если оно выполняется, то все расширяющие варианты использования вставляются в базовый вариант.

В представленном выше примере (рис. 7) при оформлении заказа на приобретение товара только в некоторых случаях может потребоваться предоставление клиенту каталога всех товаров. При этом условием расширения является запрос от клиента на получе-

ние каталога товаров. Очевидно, что после получения каталога клиенту необходимо некоторое время на его изучение, в течение которого оформление заказа приостанавливается. После ознакомления с каталогом клиент решает либо в пользу выбора отдельного товара, либо отказа от покупки вообще. Сервис или вариант использования «Оформить заказ на приобретение товара» может отреагировать на выбор клиента уже после того, как клиент получит для ознакомления каталог товаров.

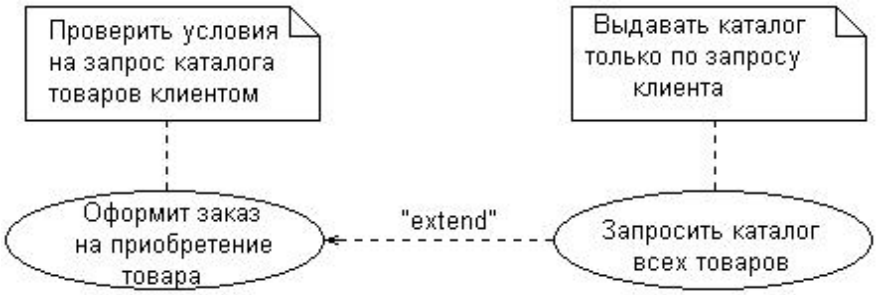


Рис. 8. Графическое изображение отношения расширения с примечаниями условий выполнения вариантов использования

Точка расширения может быть как отдельной точкой в последовательности действий, так и множеством отдельных точек. Важно представлять себе, что если отношение расширения имеет некоторую последовательность точек расширения, только первая из них может определять множество отдельных точек. Все остальные должны определять в точности одну такую точку. Какая из точек должна быть первой точкой расширения, т. е. определяться единственным расширением. Такие ссылки на расположение точек расширения могут быть представлены различными способами, например, с помощью текста примечания на естественном языке (рис. 8), пред- и постусловий, а также с использованием имен состояний в автомате.

Отношение обобщения

Отношение обобщения служит для указания того факта, что некоторый вариант использования А может быть обобщен до варианта использования В. В этом случае вариант А будет являться специализацией варианта В. При этом В называется предком или родителем по отношению А, а вариант А – потомком по отношению к варианту использования В. Следует подчеркнуть, что потомок наследует все свойства и поведение своего родителя, а также может

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

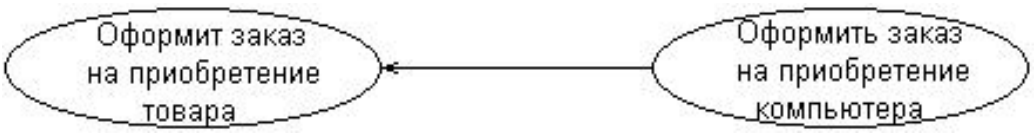


Рис. 9. Пример графического изображения отношения обобщения между вариантами использования

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

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

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

шенного треугольника, которая указывает на родительского актера (рис. 10).

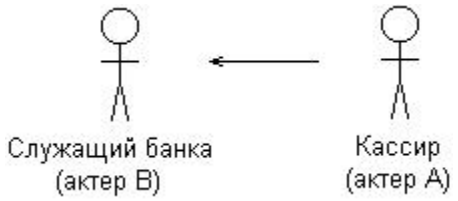


Рис. 10. Пример графического изображения отношения обобщения между актерами

Отношение включения

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

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

Один вариант использования может быть включен в несколько других вариантов, а также включать в себя другие варианты. Включаемый вариант использования может быть независимым от базового варианта в том смысле, что он предоставляет последнему некоторое инкапсулированное поведение, детали реализации которого скрыты от последнего и могут быть легко перераспределены между

несколькими включаемыми вариантами использования. Более того, базовый вариант может зависеть только от результатов выполнения включаемого в него поведения, но не от структуры включаемых в него вариантов.

Отношение включения, направленное от варианта использования 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.

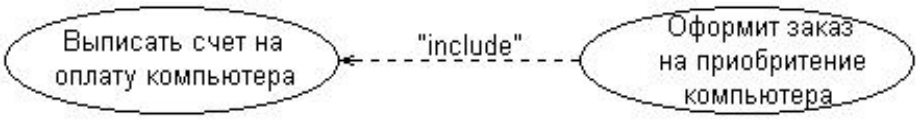


Рис. 11. Пример графического изображения отношения включения между вариантами использования

Примечание

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

пользования подчиняется общим правилам наследования свойств и поведения сущности-предка всеми дочерними сущностями.

Пример построения диаграммы вариантов использования

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

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

Рис. 12. Исходная диаграмма вариантов использования для примера разработки системы продажи товаров по каталогу

Значения указанных на данной диаграмме кратностей отражают общие правила или логику оформления заказов на покупку товаров. Согласно этим правилам, один продавец может участвовать в оформлении нескольких заказов, в то же время каждый заказ может быть оформлен только одним продавцом, который несет ответственность за корректность его оформления и, в связи с этим, будет иметь агентское вознаграждение за его оформление. С другой стороны, каждый покупатель может оформлять на себя несколько заказов, но, в то же время, каждый заказ должен быть оформлен на единственного покупателя, к которому переходят права собственности на товар после его оплаты.

Примечание

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

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

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

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

Рис. 13. Уточненный вариант диаграммы вариантов использования для примера системы продажи товаров по каталогу

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

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

Уточненный таким способом вариант диаграммы вариантов использования содержит одну важную особенность, которую необходимо отметить. А именно, хотя на данной диаграмме (рис. 14) отсутствуют изображения линий отношения ассоциации между акте-

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

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

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

Рис. 15. Фрагмент диаграммы вариантов использования, который в неявном виде присутствует на уточненной диаграмме с отношением ассоциации между отдельными компонентами

Примечание

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

Второе из основных направлений детализации диаграмм вариантов использования связано с последующей структуризацией ее отдельных компонентов в форме элементов других диаграмм. Например, конкретные особенности реализации вариантов использования в терминах взаимодействующих объектов, определенных в виде классов данной сущности, могут быть заданы на диаграмме кооперации. Указанное направление отражает основные особенности ООАП применительно к их реализации в языке UML. Эти особенности являются предметом рассмотрения во всех последующих главах книги.

Построение диаграммы вариантов использования является самым первым этапом процесса объектно-ориентированного анализа и проектирования, цель которого – представить совокупность требований к поведению проектируемой системы. Спецификация требований к проектируемой системе в форме диаграммы вариантов использования представляет собой самостоятельную модель, кото-

рая в языке UML получила название модели вариантов использования и имеет свое специальное стандартное имя или стереотип «useCaseModel».

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

Рекомендации по разработке диаграмм вариантов использования

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

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

Подобное взаимодействие будет продолжаться до тех пор, пока не закончится выполнение требуемой последовательности действий экземпляром варианта использования, и соответствующий эк-

земпляр актера (и никакой другой) не получит требуемый экземпляр сервиса. Окончание взаимодействия означает отсутствие инициализации экземпляров сообщений от экземпляров актеров для соответствующих экземпляров вариантов использования.

Варианты использования могут быть специфицированы в виде текста, а в последующем – с помощью операций и методов вместе с атрибутами, в виде графа деятельности, посредством автомата или любого другого механизма описания поведения, включающего предусловия и постусловия. Взаимодействие между вариантами использования и актерами может уточняться на диаграмме кооперации, когда описываются взаимосвязи между сущностью, содержащей эти варианты использования, и окружением или внешней средой этой сущности.

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

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

Отдельные варианты использования нижнего уровня могут участвовать в нескольких кооперациях, т. е. играть определенную роль при выполнении сервисов нескольких вариантов верхнего

уровня. Для отдельных таких кооперaций могут быть определены соответствующие роли актеров, взаимодействующих с конкретными вариантами использования нижнего уровня. Эти роли будут играть актеры нижнего уровня модели системы. Хотя некоторые из таких актеров могут быть актерами верхнего уровня, это не противоречит принятым в языке UML семантическим правилам построения диаграмм вариантов использования. Более того, интерфейсы вариантов использования верхнего уровня могут полностью совпадать по своей структуре с соответствующими интерфейсами вариантов нижнего уровня.

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

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

Реализация варианта использования зависит от типа элемента модели, в котором он определен. Например, поскольку варианты использования класса определяются посредством операций этого класса, они реализуются соответствующими методами. С другой стороны, варианты использования подсистемы реализуются элементами, из которых состоит данная подсистема. Поскольку подсистема не имеет своего собственного поведения, все предлагаемые подсистемой сервисы должны представлять собой композицию сервисов, предлагаемых отдельными элементами этой подсистемы, т. е., в конечном итоге, классами. Эти элементы могут взаимодействовать друг с другом для совместного обеспечения требуемого поведения

отдельного варианта использования., посвященной построению диаграмм кооперации. Здесь лишь отметим, что кооперации используются как для уточнения спецификаций в виде вариантов использования нижних уровней диаграммы, так и для описания особенностей их последующей реализации.

Если в качестве моделируемой сущности выступает система или подсистема самого верхнего уровня, то отдельные пользователи вариантов использования этой системы моделируются актерами. Такие актеры, являясь внутренними по отношению к моделируемым подсистемам нижних уровней, часто в явном виде не указываются, хотя и присутствуют неявно в модели подсистемы. Вместо этого варианты использования непосредственно обращаются к тем модельным элементам, которые содержат в себе подобные неявные актеры, т. е. экземпляры которых играют роли таких актеров при взаимодействии с вариантами использования. Эти модельные элементы могут содержаться в других пакетах или подсистемах. В последнем случае роли определяются в том пакете, к которому относится соответствующая подсистема.

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

Если же варианты использования применяются для спецификации части системы, то они будут эквивалентны соответствующим вариантам использования в модели подсистемы для части соответствующего пакета. Важно понимать, что все сервисы системы должны быть явно определены на диаграмме вариантов использования, и никаких других сервисов, которые отсутствуют на данной

диаграмме, проектируемая система не может выполнять по определению. Более того, если для моделирования реализации системы используются сразу несколько моделей (например, модель анализа и модель проектирования), то множество вариантов использования всех пакетов системы должно быть эквивалентно множеству вариантов использования модели в целом.

Диаграммы последовательности

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

С помощь диаграммы прецедентов (вариантов использования) выявляются основные пользователи системы и задачи, которые данная система должна решать. С помощью диаграммы деятельности мы описываем последовательность действий для каждого прецедента, необходимая для достижения поставленной цели.

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

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

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

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

Данный вид диаграмм отражает следующие аспекты проектируемой Системы:

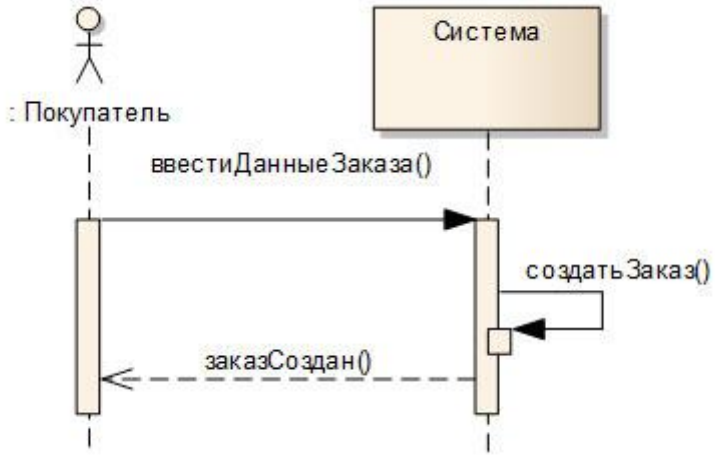
- обмен сообщениями между объектами (в том числе в рамках обмена сообщениями со сторонними Системами)

- ограничения, накладываемые на взаимодействие объектов
- события, инициирующие взаимодействия объектов.

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

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

Одним из основных принципов ООП является способ информационного обмена между элементами Системы, выражающийся в отправке и получении сообщений друг от друга. Таким образом, основные понятия диаграммы последовательности связаны с понятием Объект и Сообщение.

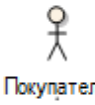


На диаграмме последовательности объекты в основном представляю экземпляры класса или сущности, обладающие поведением. В качестве объектов могут выступать пользователи, инициирующие взаимодействие, классы, обладающие поведением в Системе или программные компоненты, а иногда и Системы в целом.

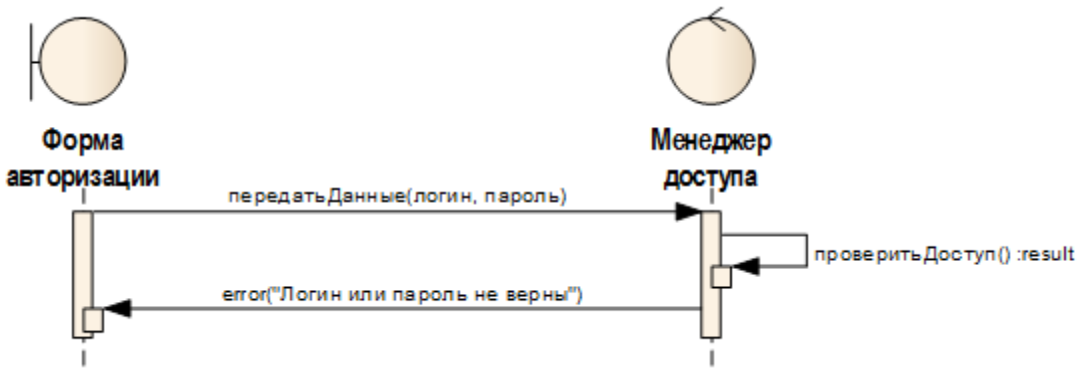
Объекты располагаются с лева на права таким образом, чтобы крайним с лева был тот объект, который инициирует взаимодействие. Неотъемлемой частью объекта на диаграмме последовательности является линия жизни объекта. Линия жизни показывает время,

в течение которого объект существует в Системе. Периоды активности объекта в момент взаимодействия показываются с помощью фокуса управления. Временная шкала на диаграмме направлена сверху вниз.

На диаграммах последовательности допустимо использование стандартных стереотипов класса:

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

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



На диаграмме деятельности выделяются сообщения, инициирующие ту или иную деятельность или являющиеся ее следствием.

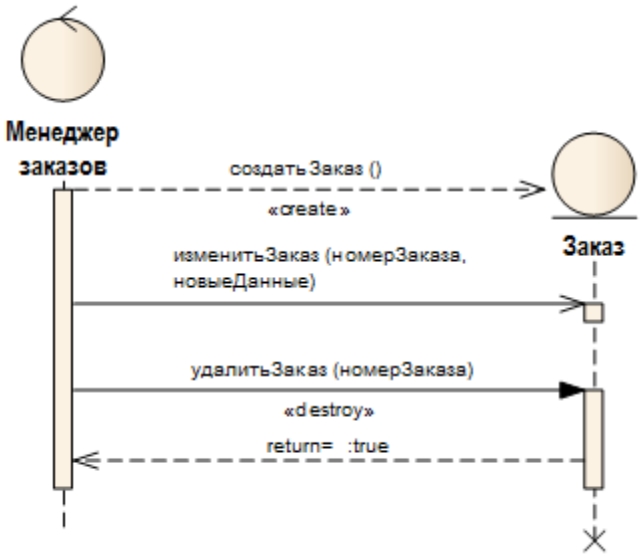
На диаграмме состояний частично показан обмен сообщениями в рамках сообщений инициирующих изменение состояния объекта.

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

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

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

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



В UML различают следующие виды сообщений:

- синхронное сообщение (**synchCall**) – соответствует синхронному вызову операции и подразумевает ожидание ответа от

объекта получателя. Пока ответ не поступит, никаких действий в Системе не производится.

- асинхронное сообщение (**asynchCall**) – которое соответствует асинхронному вызову операции и подразумевает, что объект может продолжать работу, не ожидая ответа.
- ответное сообщение (**reply**) – ответное сообщение от вызванного метода. Данный вид сообщения показывается на диаграмме по мере необходимости или, когда возвращаемые им данные несут смысловую нагрузку.
- потерянное сообщение (**lost**) – сообщение, не имеющее адресата сообщения, т. е. для него существует событие передачи и отсутствует событие приема
- найденное сообщение (**found**) – сообщение, не имеющее инициатора сообщения, т. е. для него существует событие приема и отсутствует событие передачи

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

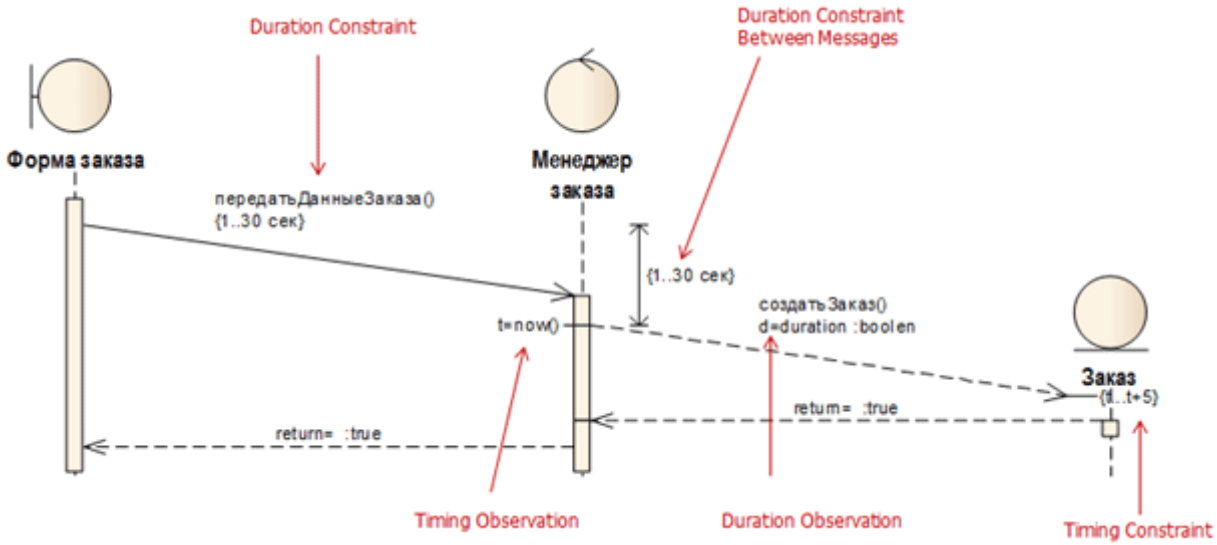
Сообщение со стереотипом **create**, вызывает в классе метод, которые создает экземпляр класса. На диаграмме последовательности не обязательно показывать с самого начала все объекты, участвующие во взаимодействии. При использовании сообщения со стереотипом **create**, создаваемый объект отображается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

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

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

- ограничение продолжительности (Duration Constraint) – минимальное и максимальное значение продолжительности передачи сообщения
- ограничение продолжительности ожидания между передачей и получением сообщения (Duration Constraint Between Messages)

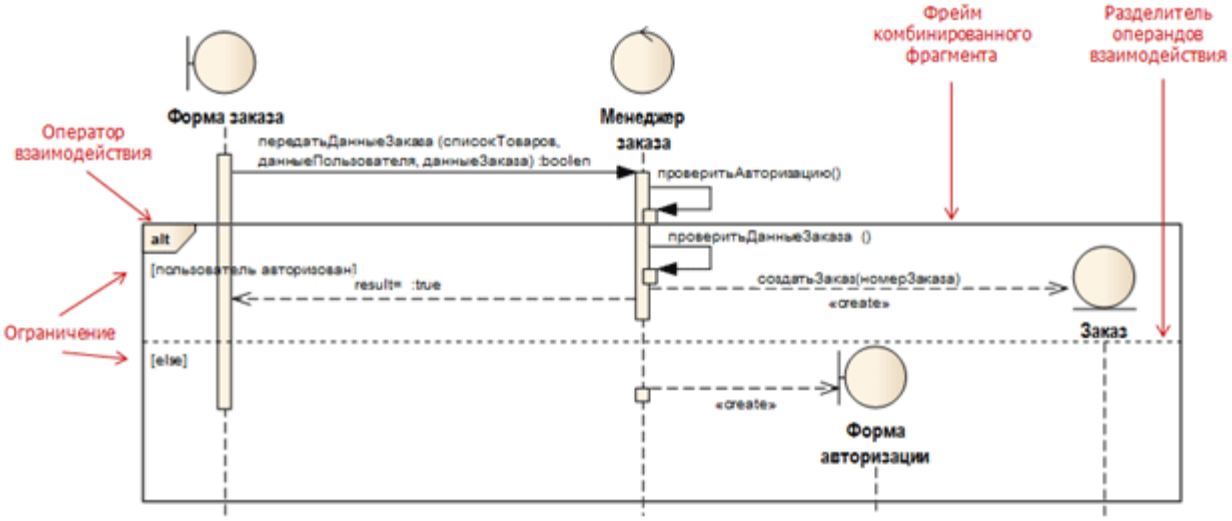
- перехват продолжительности сообщения (Duration Observation)
- временное ограничение (Timing Constraint) – временной интервал, в течение которого сообщение должно прийти к цели (устанавливается на стороне получателя)
- перехват времени, когда сообщение было отправлено (Timing Observation)



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

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

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



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

- **Alt –** Несколько альтернативных фрагментов (alternative); выполняется только тот фрагмент, условие которого истинно
- **Opt –** Необязательный (optional) фрагмент; выполняется, только если условие истинно. Эквивалентно **alt** с одной веткой
- **Par –** Параллельный (parallel); все фрагменты выполняются параллельно
- **loop –** Цикл (loop); фрагмент может выполняться несколько раз, а защита обозначает тело итерации
- **region –** Критическая область (critical region); фрагмент может иметь только один поток, выполняющийся за один прием
- **Neg –** Отрицательный (negative) фрагмент; обозначает неверное взаимодействие
- **ref –** Ссылка (reference); ссылается на взаимодействие, определенное на другой диаграмме. Фрейм рисуется, чтобы охватить

линии жизни, вовлеченные во взаимодействие. Можно определять параметры и возвращать значение

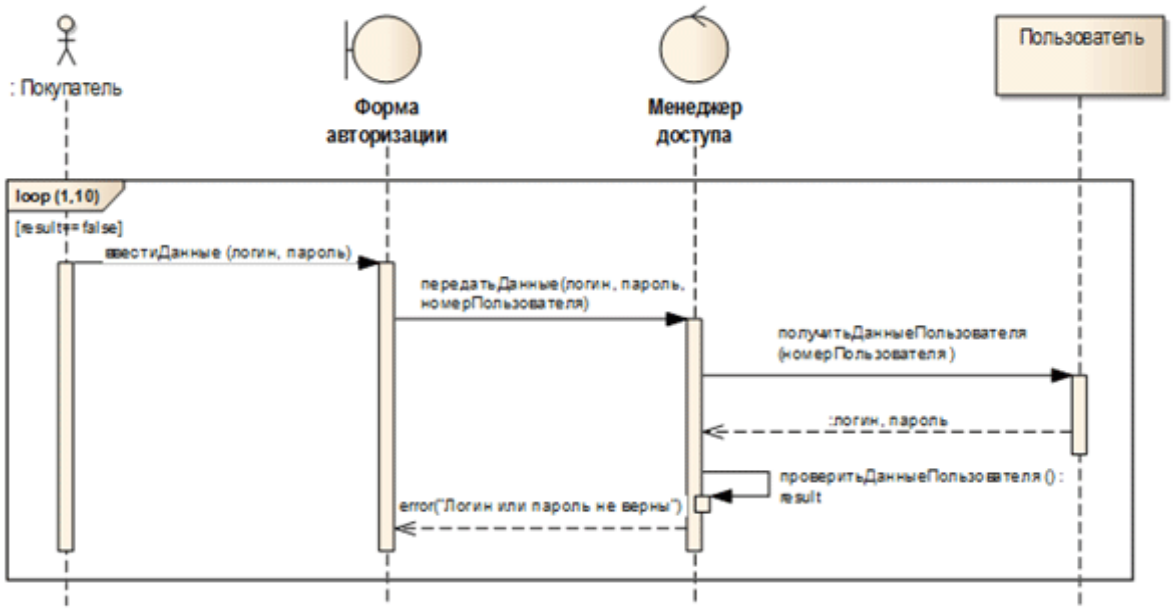
- **Sd –** Диаграмма последовательности (sequence diagram); используется для очерчивания всей диаграммы последовательности, если это необходимо.

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

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

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

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



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

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

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

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

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

Контрольные вопросы

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

6. Лабораторная работа №2. Построение диаграммы Деятельности, диаграммы состояний и диаграммы классов

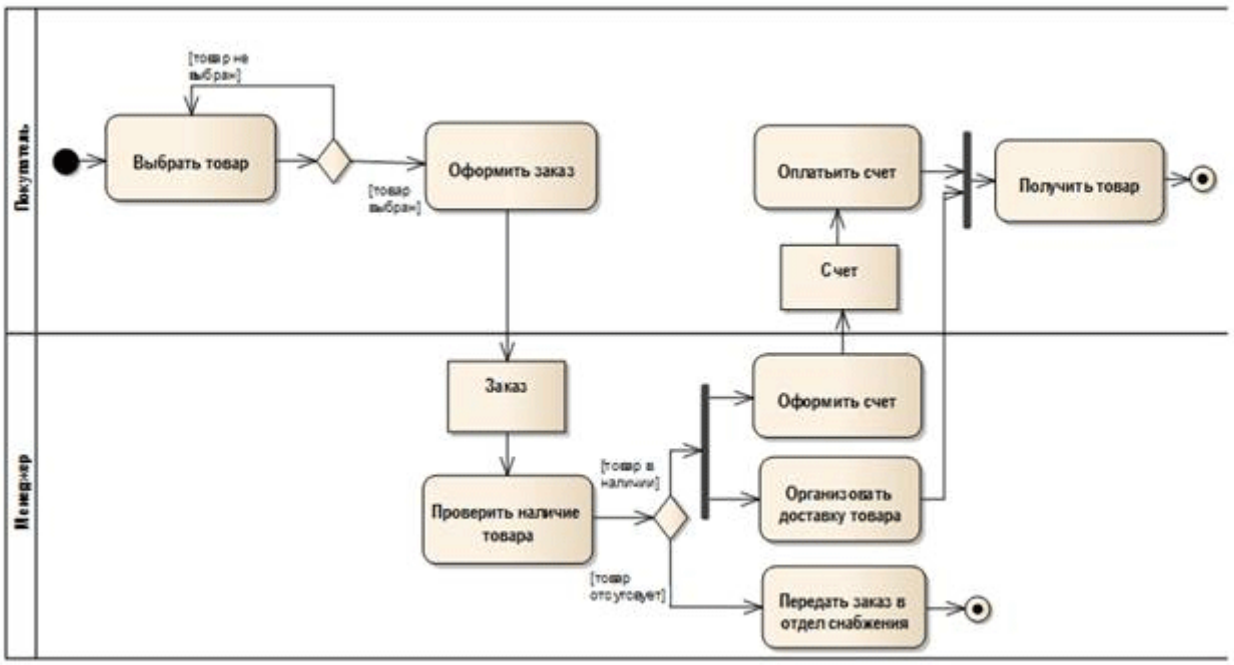
Целью работы является изучение порядка построения различных диаграмм.

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

6.1. Диаграммы деятельности

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

В качестве графического представления для выделения основных функций Системы мы применяем диаграмму вариантов использования (use case). Диаграмма вариантов использования дает нам представление ЧТО должна делать Система. На вопрос КАК мы можем ответить, используя диаграмму активности.



То есть если варианты использования ставят перед Системой цель, то диаграмма деятельности показывает последовательность дей-

ствий, необходимых для ее достижения. Действия (action) это элементарные шаги, которые не предполагают дальнейшую декомпозицию.

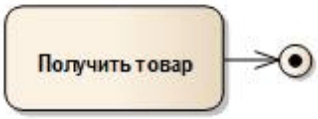
Деятельность может содержать входящие и/или исходящие **дуги деятельности**, показывающие потоки управления и потоки данных. Если поток соединяет две деятельности, он является потоком управления. Если поток заканчивается объектом, он является потоком данных.

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

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

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

Для создания диаграммы деятельности используются следующие узлы:

	Узел управления (control node) – это абстрактный узел действия, которое координирует потоки действий
	**Начальный узел деятельности (или начальное состояние деятельности) (activity initial node)** является узлом управления, в котором начинается поток (или потоки) при вызове данной деятельности извне
	**Конечный узел деятельности (или конечное состояние деятельности) (activity final node)** является узлом управления, который останавливает (stop) все потоки данной диаграммы деятельности. На диаграмме может быть более одного конечного узла
	**Конечный узел потока (или конечное состояние потока) (flow final node)** является узлом управления, который завершает данный поток. На другие потоки и деятельность данной диаграммы это не влияет

	**Объект**, над которым выполняются действия. Это не обязательный элемент диаграммы, но в некоторых случаях необходимо показать объект инициирующий выполнение действий, или являющийся результатом его

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

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

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

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

Для отображения условий соответствующих логическому оператору «и» на диаграмме используются синхронизационная черта.

Точка разделения обеспечивает разделение одного потока на несколько параллельных потоков:

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

Точка слияния обеспечивает синхронизацию нескольких параллельных потоков.

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

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

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

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

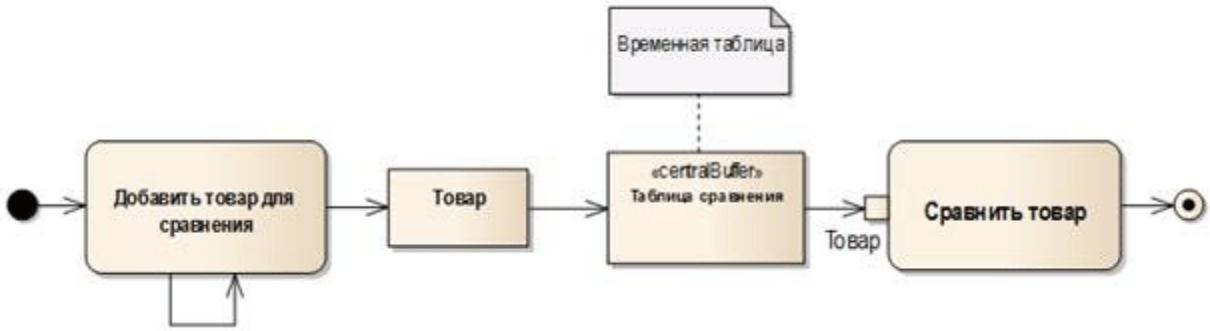
Прием события (receive event action) – действие, которое ожидает некоторого события, принимает и обрабатывает полученное сообщение.

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

- Результатом действия по приему банковской выписки и разнесению оплаты является входящий сигнал для ИС «Веб-магазин» сообщающей об оплате товара.
- Соответственно при получении входящего сигнала в ИС «Веб-магазин» фиксируется факт оплаты, который инициализирует действие «Разрешить отгрузку».

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

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



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

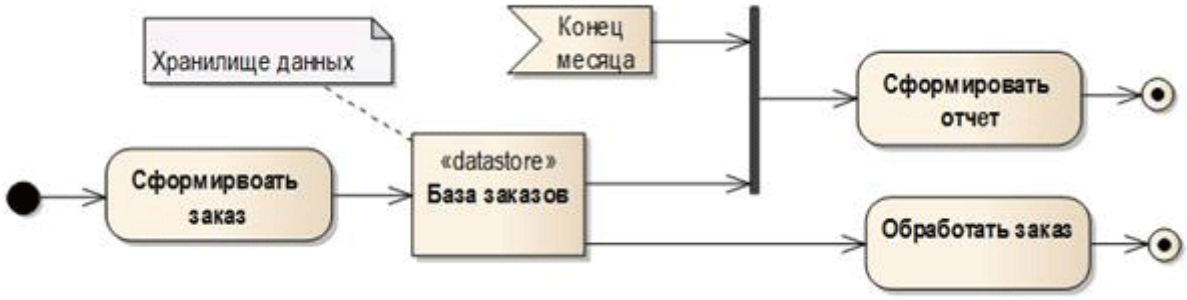
Данный объект может применяться на уровне описания реализации функций Системы для визуализации временных таблиц.

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

- В центральный буфер поступает информация о товаре, выбранном для сравнения пользователем из каталога товаров.
- Данные товара хранятся в центральном буфере какой-то промежуток времени.
- Далее пользователь вызывает для просмотра список сравнения товаров, просматривает его и сохраняет наиболее подходящий товар в корзину.
- Товар, выбранный для покупки, сохраняется в таблице БД, хранящей заказы пользователя, в то время как другие товары из временной таблицы сравнения удаляются.

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

Частный случай Центрального буфера – Хранилище данных.



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

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

Далее следует подробно рассмотреть разбиение деятельности на разделы.

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

Дорожки, используемые при данной структуре диаграммы деятельности, зачастую символизируют роль пользователя или организационное подразделение, осуществляющее определенные действия в рамках данной деятельности.

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

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

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

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

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

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

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

При необходимости описание последовательности действий для под-деятельности может быть размещено непосредственно на

основной диаграмме. Для этого описание под-деятельности размещается в отдельный фрейм.

При таком способе декомпозиции мы можем указать:

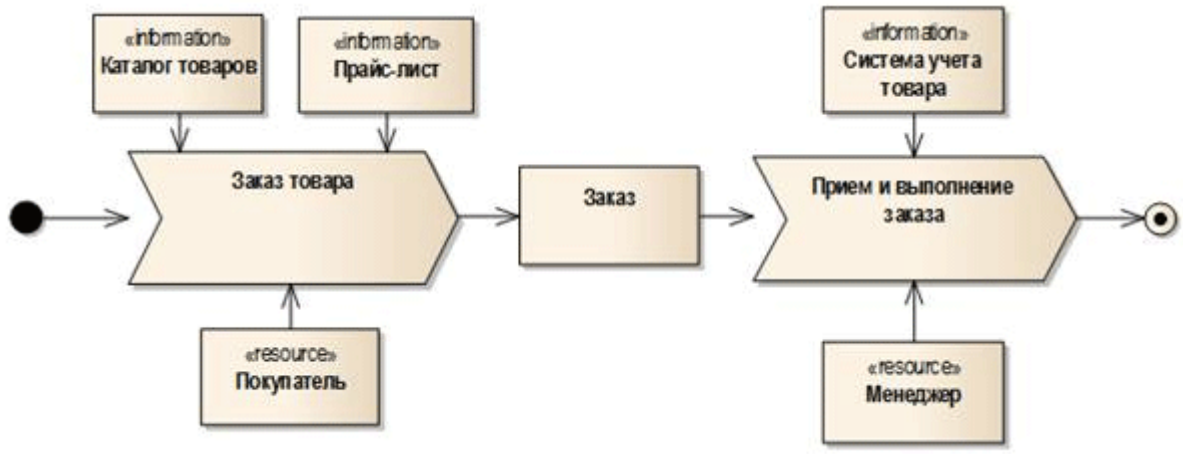
- предусловия и постусловия;
- входные и выходные параметры (объекты);
- внутреннее устройство деятельности.

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

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

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

Цель концептуального описания – показать целостную картину бизнес-процессов предметной области.



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

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

На слайде показаны стандартные UML-объекты «действие» и «объект», но со специальными стереотипами:

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

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

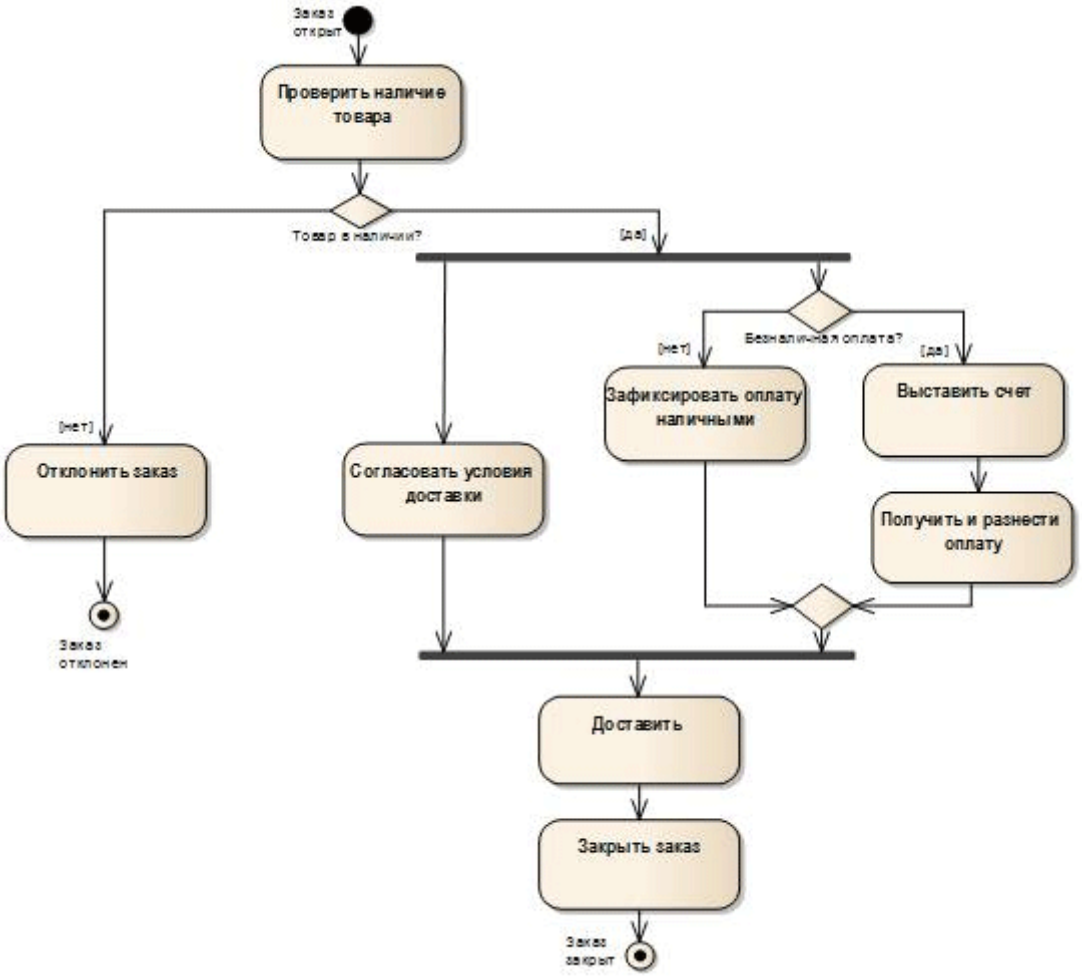


Диаграмма деятельности данного вида хорошо отражает:

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

Для того чтобы отобразить соответствие деятельности определенному пользователю или Системе к данной диаграмме можно

применить «дорожки». Так как в нашем случае все действия выполняются менеджером, применение разделителей не целесообразно.

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

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

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

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

6.2. Диаграмма состояний

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

На сегодняшний день при проектировании сложной Системы принято делить ее на части, каждую из которых затем рассматривать отдельно. Таким образом, при объектной декомпозиции Система разбивается на объекты или компоненты, которые взаимодействуют друг с другом, обмениваясь сообщениями. Сообщения описывают или представляют собой некоторые события. Получение объектом сообщения активизирует его и побуждает выполнять предписанные его программным кодом действия.

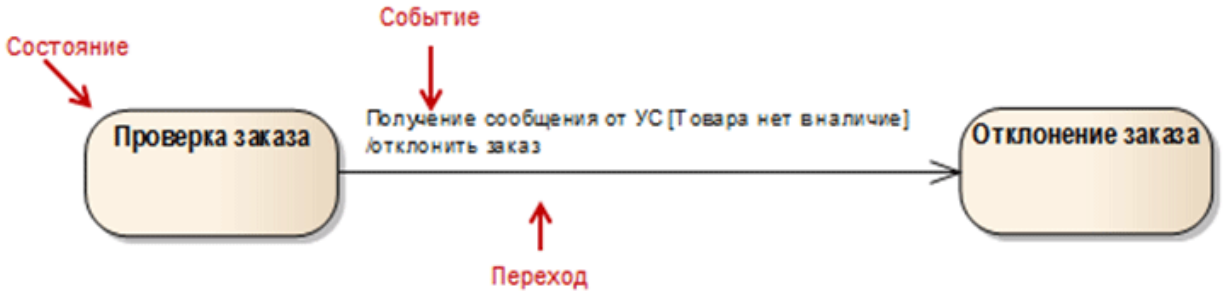
При данном подходе Система становится событийно управляемой, поэтому разработчикам зачастую важно знать, как должен реагировать тот или иной объект на определенные события. Инициаторами событий могут быть как объекты самой Системы, так и её внешнее окружение.

Описать поведение отдельно взятого объекта помогает диаграмма состояний.

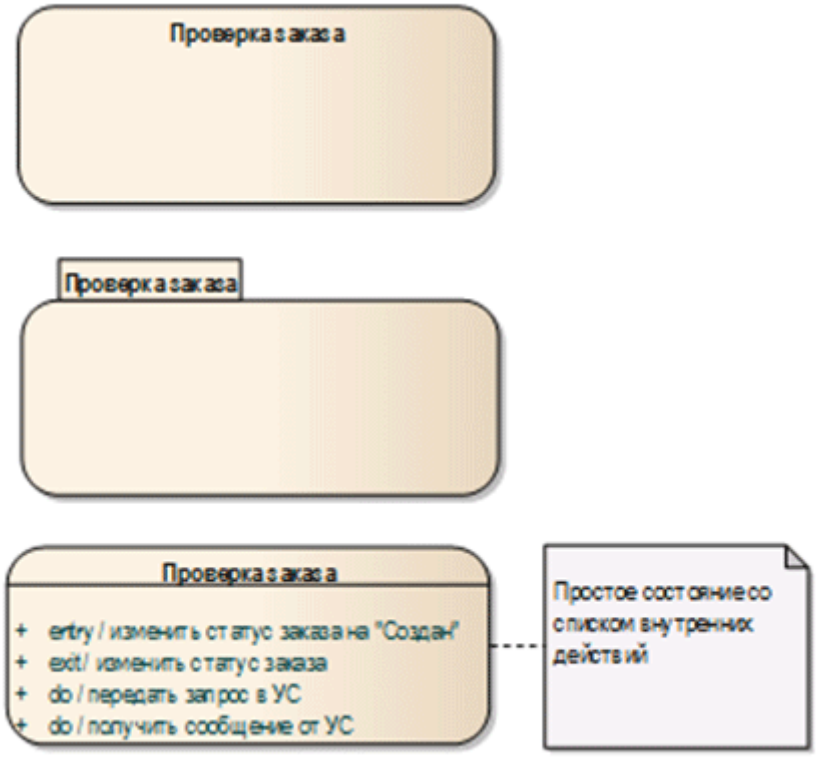
Также зачастую диаграмма состояний используется аналитиками для описания последовательности переходов объекта из одного состояния в другое.

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

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

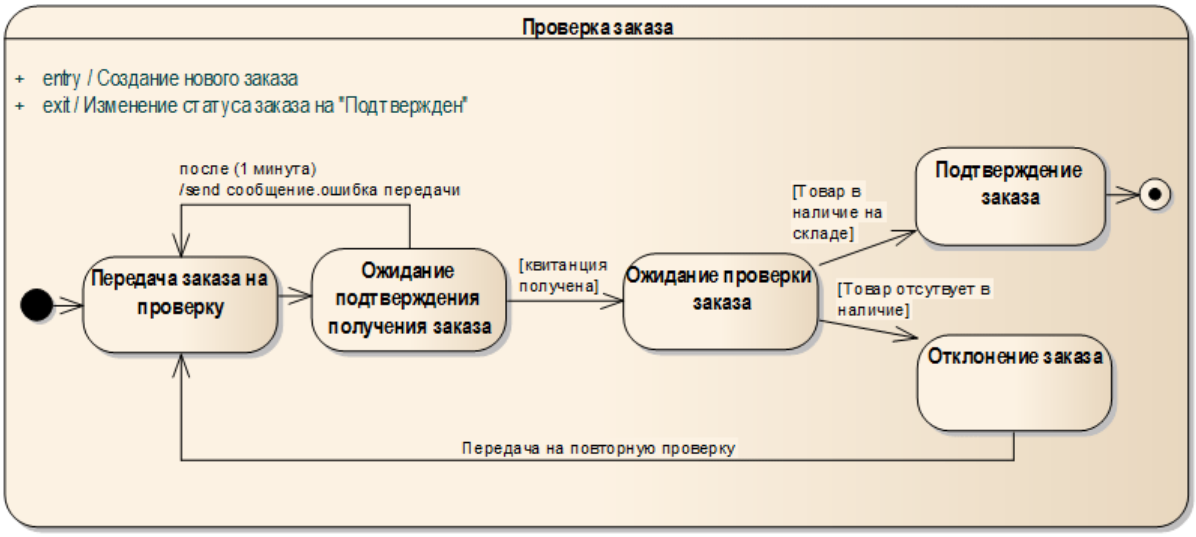


Переход может быть инициирован событием, которое также отражается на диаграмме состояний.



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

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



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

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

Обязательным параметром является только имя события.

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

Различаются следующие виды событий:

- Событие вызова (call event) – событие, возникающее при вызове метода класса. При срабатывании данного вида события объектом ассинхронно создается Сигнал, который принимается другим объектом. Для указания на то, что некоторая операция посылает сигнал, можно воспользоваться зависимостью со стереотипом send.

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

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

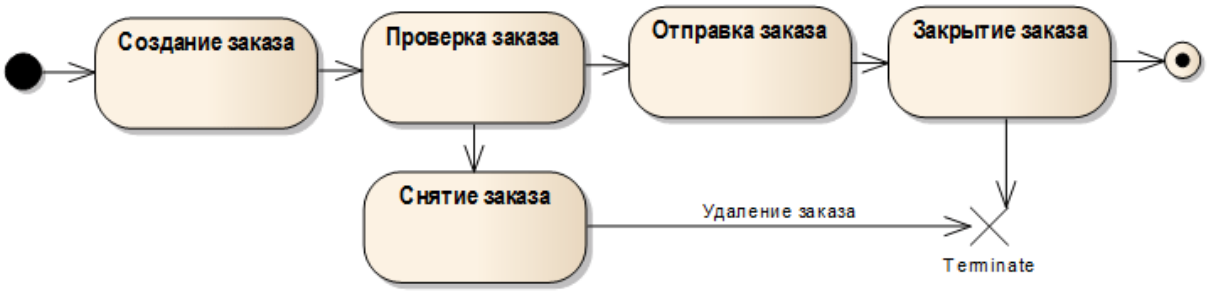
Также имя перехода может содержать **выражение действия (action expression)**. В данном случае указанное действие выполняется сразу при срабатывании перехода и до начала каких бы то ни было действий в целевом состоянии. В общем случае выражение действия может содержать целый список отдельных действий,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

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

Помимо основных узлов, на диаграмме состояний могут использоваться, так называемые, псевдосостояния – вершины которые

не обладают поведением, и объект не находится в ней, а «мгновенно» ее проходит.

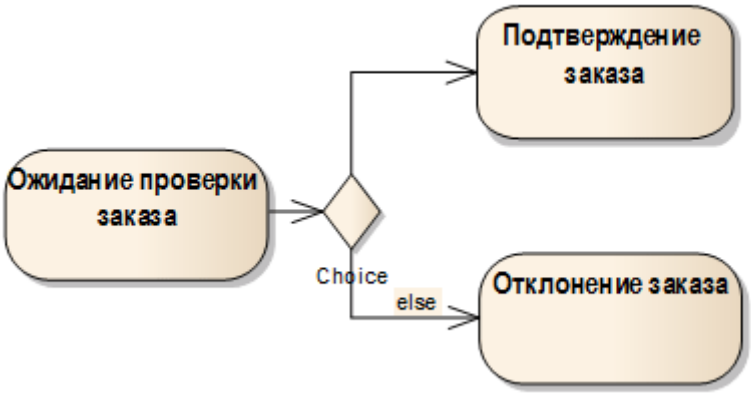
Под псевдосостояниями на диаграмме состояний понимаются, знакомые уже нам начальное и конечное состояние. **Начальное состояние** обычно не содержит никаких внутренних действий и определяет точку, в которой находится объект по умолчанию в начальный момент времени. **Конечное состояние** также не содержит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



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

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

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

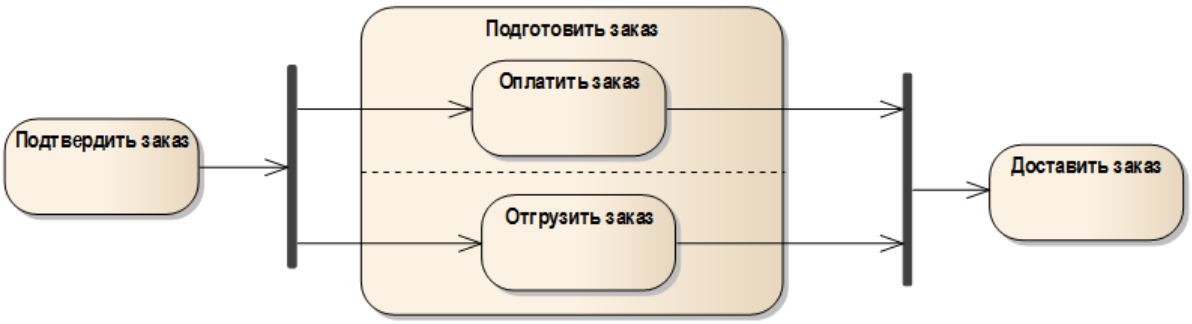


В отличие от диаграммы деятельности, при отображении возможных вариантов перехода на диаграмме состояний узел выбора использовать не обязательно. Диаграмма состояний должна показывать возможное изменение состояния объекта, и не имеет своей целью выстраивать четкую последовательность переходов. Таким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

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

На диаграмме могут быть представлены как простые состояния, так и сложные состояния. Сложные или составные состояния (composite state) включают в себя вложенные подсостояния (слайд 10). Декомпозиция сложного состояния может осуществляться как на основной диаграмме, так и отдельно, при этом на основной диаграмме следует использовать элемент с пиктограммой декомпозиции.

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



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

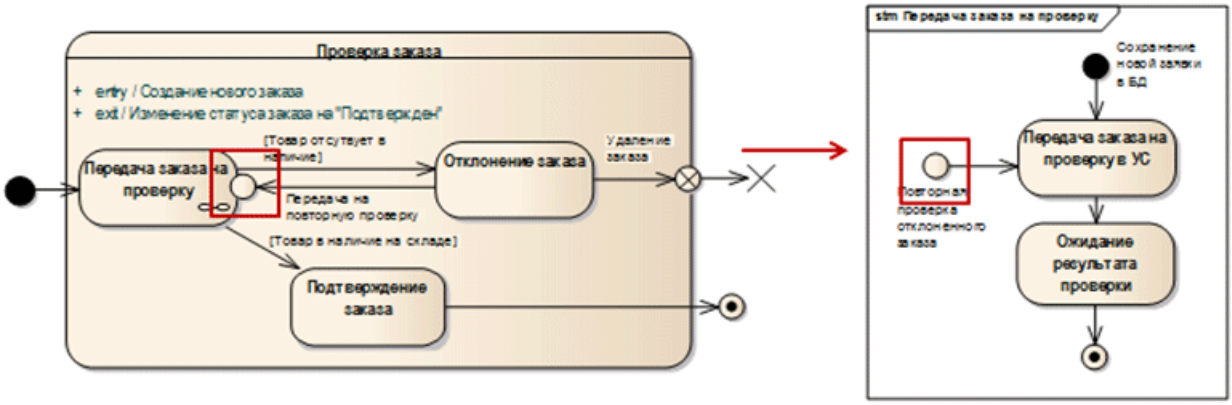
Параллельные подсостояния (concurrent substates) позволяют специфицировать два и более подавтомата, которые могут выполняться параллельно внутри составного события. Каждый из подавтоматов занимает некоторую область (регион) внутри составного состояния.

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

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

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

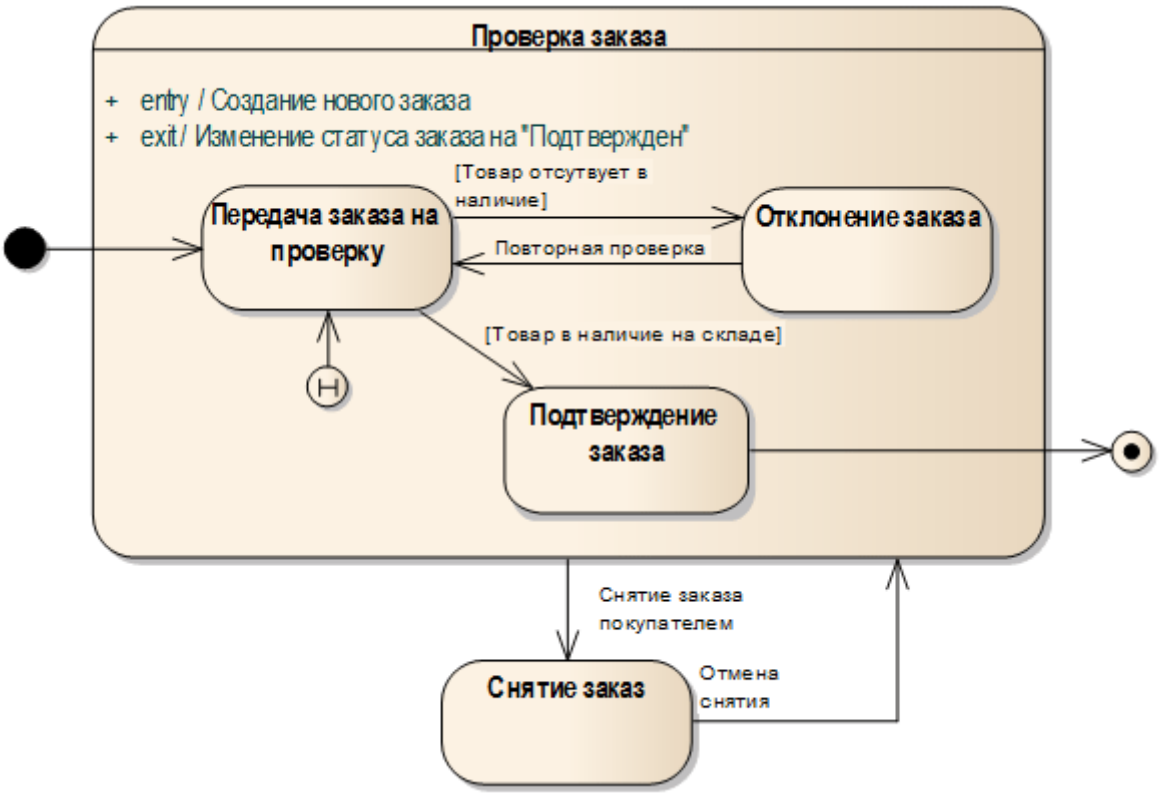
UML 1.0 для подобных случаем использовались элементы «заглушка» и «ссылочное состояние». В UML 2.0 данные элементы были замены «точкой входа» и «точкой выхода».



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

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

Также с понятием композитного состояния тесно связано понятие исторического состояния.



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

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

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

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

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

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

При использовании диаграммы состояний важно следовать следующим правилам:

- Диаграмма состояний должна создаваться только для объектов, обладающих реактивным поведением. Не следует делать диаграмму автоматов для всех классов или объектов, достаточно выбрать только основные классы или объекты, обладающие сложным поведением.
- Диаграмма состояний должна быть сосредоточена на описании только одного аспекта поведения объекта. Следует создавать диаграмму автомата, моделирующую поведение только одного объекта. Если необходимо показать поведение нескольких, взаимосвязанных объектов, допустимо создавать для них диаграмму состояний в рамках определенного варианта использования (диаграмма состояний для варианта использования).

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

6.3. Диаграммы классов

Диаграммы классов UML. Логическое моделирование

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

Представление классов

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

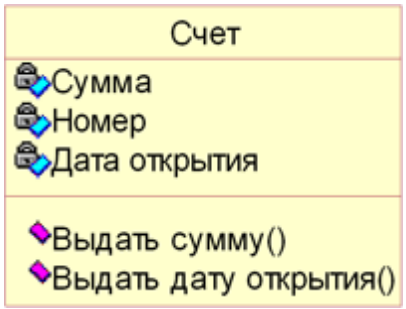


Рис. 1. Изображение класса в нотации UML

Атрибуты класса определяют состав и структуру данных, хранимых в объектах этого класса. Каждый атрибут имеет имя и тип, определяющий, какие данные он представляет. При реализации объекта в программном коде для атрибутов будет выделена память,

необходимая для хранения всех атрибутов, и каждый атрибут будет иметь конкретное значение в любой момент времени работы программы. Объектов одного класса в программе может быть сколь угодно много, все они имеют одинаковый набор атрибутов, описанный в классе, но значения атрибутов у каждого объекта свои и могут изменяться в ходе выполнения программы.

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

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

Последнее значение позволяет реализовать свойство инкапсуляции данных. Например, объявив все атрибуты класса закрытыми, можно полностью скрыть от внешнего мира его данные, гарантируя отсутствие несанкционированного доступа к ним. Это позволяет сократить число ошибок в программе. При этом любые изменения в составе атрибутов класса никак не скажутся на остальной части ПС.

Класс содержит объявления **операций**, представляющих собой определения запросов, которые должны выполнять объекты данного класса. Каждая операция имеет **сигнатуру**, содержащую имя операции, тип возвращаемого значения и список параметров, который может быть пустым. Реализация операции в виде процедуры – это метод, принадлежащий классу. Для операций, как и для атрибутов класса, определено понятие «видимость». Закрытые операции являются внутренними для объектов класса и недоступны из других объектов. Остальные образуют интерфейсную часть класса и являются средством интеграции класса в ПС.

Отношения

На диаграммах классов обычно показываются ассоциации и обобщения (см. предыдущую статью).

Каждая **ассоциация** несет информацию о связях между объектами внутри ПС. Наиболее часто используются бинарные ассоциа-

ции, связывающие два класса. Ассоциация может иметь название, которое должно выражать суть отображаемой связи (рис. 2). Помимо названия, ассоциация может иметь такую характеристику, как **множественность**. Она показывает, сколько объектов каждого класса может участвовать в ассоциации. Множественность указывается у каждого конца ассоциации (полюса) и задается конкретным числом или диапазоном чисел. Множественность, указанная в виде звездочки, предполагает любое количество (в том числе, и ноль). Например, на рис. 2 ассоциация связывает один объект класса «Набор товаров» с одним или более объектами класса «товар». Связаны между собой могут быть и объекты одного класса, поэтому ассоциация может связывать класс с самим собой. Например, для класса «Житель города» можно ввести ассоциацию «Соседство», которая позволит находить всех соседей конкретного жителя.

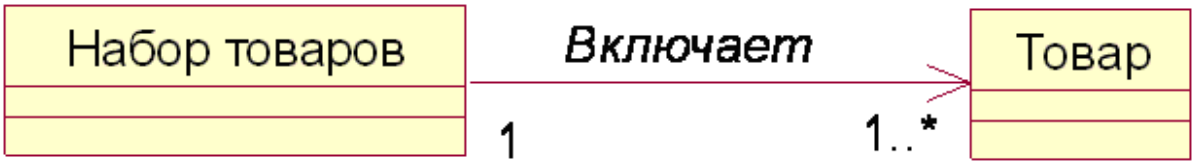


Рис. 2. Применение ассоциаций

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

Ассоциация сама может обладать свойствами класса, то есть, иметь атрибуты и операции. В этом случае она называется класс-ассоциацией и может рассматриваться как класс, у которого помимо явно указанных атрибутов и операций есть ссылки на оба связываемых ею класса. В примере на рис. 2 ассоциация «включает» по существу есть класс-ассоциация, у которой есть атрибут «Количество», показывающий, сколько единиц каждого товара входит в набор (см. рис. 4).

Обобщение на диаграммах классов используется, чтобы показать связь между классом-родителем и классом-потомком. Оно вводится на диаграмму, когда возникает разновидность какого-либо класса (например, при развитии ПС – см. рис. 4), а также в тех слу-

чаях, когда в системе обнаруживаются несколько классов, обладающих сходным поведением (в этом случае общие элементы поведения выносятся на более высокий уровень, образуя класс-родитель – рис. 3).

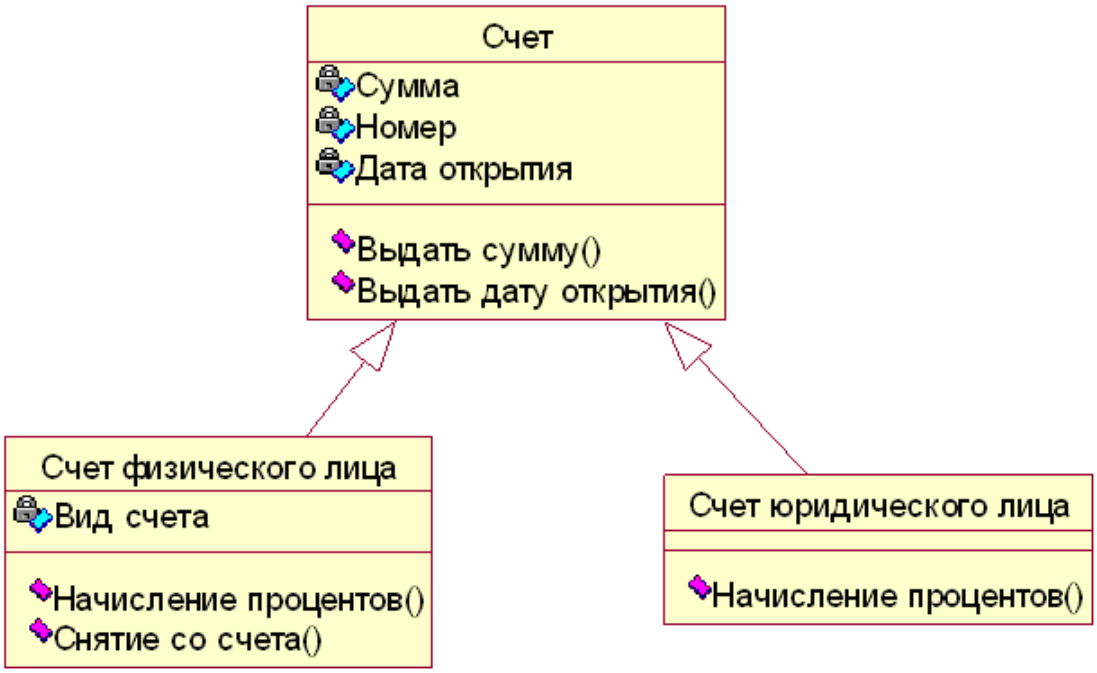


Рис. 3. Наследуются атрибуты и операции

Как уже говорилось ранее, UML позволяет строить модели с различным уровнем детализации. На рис. 4 показана детализация модели, представленной на рис. 2.

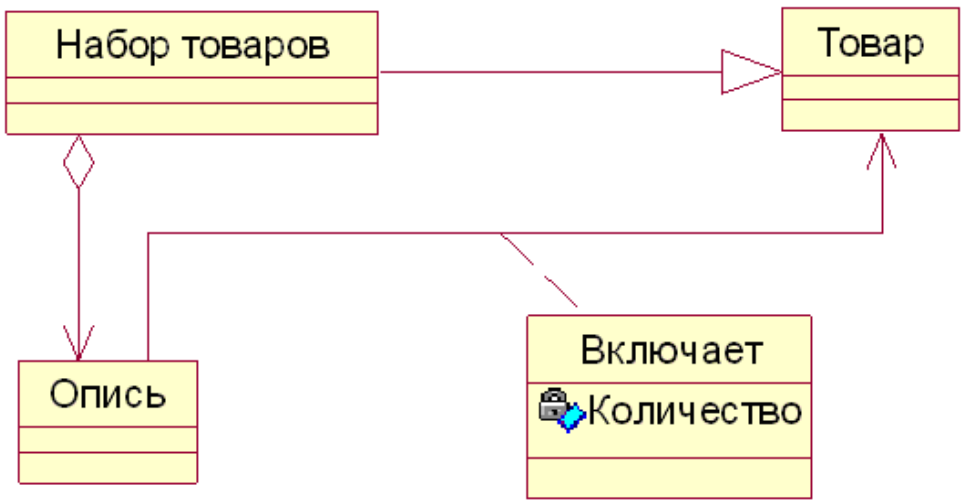


Рис. 4. Детализация модели набора товаров

Обобщение показывает, что набор товаров – это тоже товар, который может быть предметом заказа, продажи, поставки и т. д. Набор включает опись, в которой указывается, какие товары входят в набор, а класс-ассоциация «включает» определяет количество каждого вида товаров в наборе.

Стереотипы классов

При создании диаграмм классов часто пользуются понятием «стереотип». В дальнейшем речь пойдет о стереотипах классов. **Стереотип** класса – это элемент расширения словаря UML, который обозначает отличительные особенности в использовании класса. Стереотип имеет название, которое задается в виде текстовой строки. При изображении класса на диаграмме стереотип показывается в верхней части класса в двойных угловых скобках. Есть четыре стандартных стереотипа классов, для которых предусмотрены специальные графические изображения (рис. 5).

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

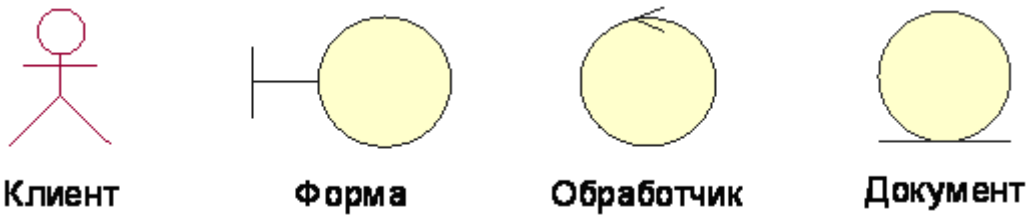


Рис. 5. Стереотипы для классов

Применение диаграмм классов

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

- Для моделирования данных. Анализ предметной области позволяет выявить основные характерные для нее сущности и связи между ними. Это удобно моделируется с помощью диаграмм классов. Эти диаграммы являются основой для построения концептуальной схемы базы данных.

- Для представления архитектуры ПС. Можно выделить архитектурно значимые классы и показать их на диаграммах, описывающих архитектуру ПС.
- Для моделирования навигации экранов. На таких диаграммах показываются пограничные классы и их логическая взаимосвязь. Информационные поля моделируются как атрибуты классов, а управляющие кнопки – как операции и отношения.
- Для моделирования логики программных компонент (будет описано в последующих статьях).
- Для моделирования логики обработки данных.

Контрольные вопросы

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

7. Лабораторная работа №3. Построение диаграммы компонентов

Целью работы является изучение порядка построения диаграммы компонентов

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

7.1. Диаграмма компонентов.

- **Диаграмма компонентов и особенности ее построения**

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

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

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

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

кода, определяемая синтаксисом языка программирования и предполагающая разбиение исходного кода на отдельные модули.

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

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

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

В разработке диаграмм компонентов участвуют как системные аналитики и архитекторы, так и программисты. Диаграмма компонентов обеспечивает согласованный переход от логического представления к конкретной реализации проекта в форме программного кода. Одни компоненты могут существовать только на этапе компиляции программного кода, другие – на этапе его исполнения. Диаграмма компонентов отражает общие зависимости между компо-

нентами, рассматривая последние в качестве отношений между ними.

- **Компоненты**

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

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

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

Компонент служит для общего обозначения элементов физического представления модели и может реализовывать некоторый набор интерфейсов. Для графического представления компонента используется специальный символ – прямоугольник со вставленными слева двумя более мелкими прямоугольниками (рис. 1). Внутри объемлющего прямоугольника записывается имя компонента и, возможно, дополнительная информация. Этот символ является базовым обозначением компонента в языке UML.

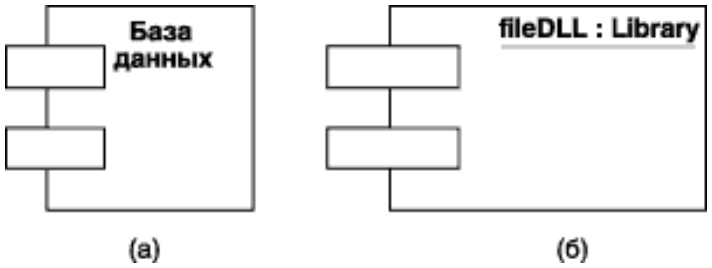


Рис. 1. Графическое изображение компонента

Графическое изображение компонента ведет свое происхождение от обозначения модуля программы, применявшегося некоторое время для отображения особенностей инкапсуляции данных и процедур.

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

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

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

Если компонент представляется на уровне типа, то записывается только имя типа с заглавной буквы в форме: <Имя типа>. Если же компонент представляется на уровне экземпляра, то его имя записывается в форме: <имя компонента ‘:' Имя типа>. При этом вся строка имени подчеркивается. Так, в первом случае (рис. 1, а) для компонента уровня типов указывается имя типа, а во втором (рис. 1, б) для компонента уровня экземпляра – собственное имя компонента и имя типа.

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

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

В отдельных случаях к простому имени компонента может быть добавлена информация об имени объемлющего пакета и о конкретной версии реализации данного компонента. Необходимо заметить, что в этом случае номер версии записывается как помеченное значение в фигурных скобках. В других случаях символ компонента может быть разделен на секции, чтобы явно указать

имена реализованных в нем классов или интерфейсов. Такое обозначение компонента называется **расширенным .**

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

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

- Во-первых, стереотипы для компонентов развертывания, которые обеспечивают непосредственное выполнение системой своих функций. Такими компонентами могут быть динамически подключаемые библиотеки (рис. 2, а), Web-страницы на языке разметки гипертекста (рис. 2, б) и файлы справки (рис. 2, в).
- Во-вторых, стереотипы для компонентов в форме рабочих продуктов. Как правило – это файлы с исходными текстами программ (рис. 2, г).

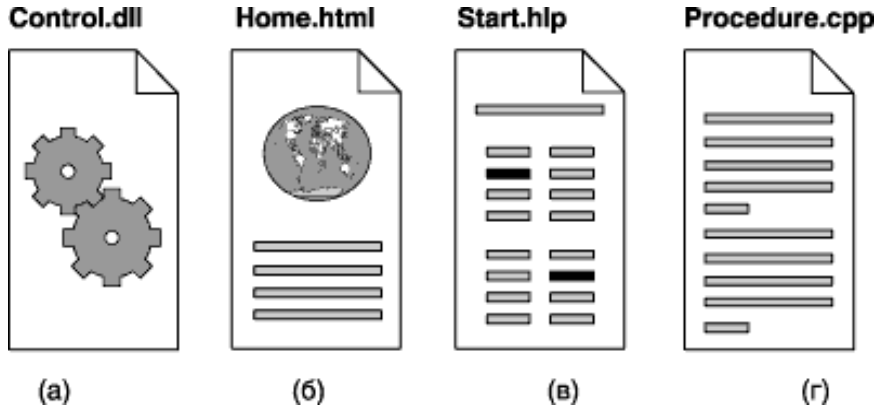


Рис. 2. Варианты графического изображения компонентов на диаграмме компонентов

Эти элементы иногда называют **артефактами**, подчеркивая при этом их законченное информационное содержание, зависящее от конкретной технологии реализации соответствующих компонентов. Более того, разработчики могут для этой цели использовать самостоятельные обозначения, поскольку в язы-

ке UML нет строгой нотации для графического представления артефактов.

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

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

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

- **Интерфейсы**

Следующим графическим элементом диаграммы компонентов являются интерфейсы. В общем случае интерфейс графически изображается окружностью, которая соединяется с компонентом отрезком линии без стрелок (рис. 3, а). При этом имя интерфейса, которое рекомендуется начинать с заглавной буквы «I», записывается рядом с окружностью. Семантически линия означает реализа-

цию интерфейса, а наличие интерфейсов у компонента означает, что данный компонент реализует соответствующий набор интерфейсов.

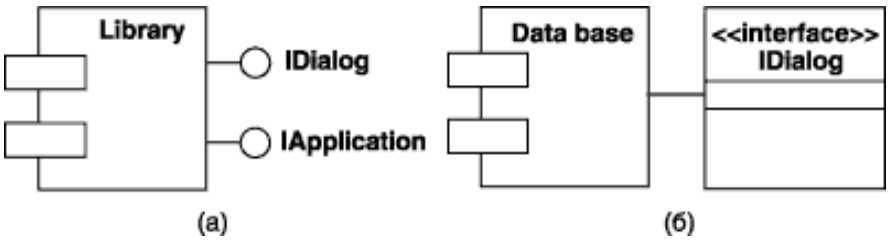


Рис. 3. Графическое изображение интерфейсов на диаграмме компонентов

Кроме того, интерфейс на диаграмме компонентов может быть изображен в виде прямоугольника класса со стереотипом << interface>> и секцией поддерживаемых операций (рис. 3, б). Как правило, этот вариант обозначения используется для представления внутренней структуры интерфейса.

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

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

- **Зависимости между компонентами**

В общем случае отношение зависимости также было рассмотрено ранее. Отношение зависимости служит для представления факта наличия специальной формы связи между двумя элементами модели, когда изменение одного элемента модели оказывает влияние или приводит к изменению другого элемента модели. Отношение зависимости на диаграмме компонентов изображается пунктир-

ной линией со стрелкой, направленной от клиента или зависимого элемента к источнику или независимому элементу модели.

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

В этом случае рисуют стрелку от компонента-клиента к импортируемому интерфейсу (рис. 4). Наличие такой стрелки означает, что компонент не реализует соответствующий интерфейс, а использует его в процессе своего выполнения. При этом на этой же диаграмме может присутствовать и другой компонент, который реализует этот интерфейс. Отношение реализации интерфейса обозначается на диаграмме компонентов обычной линией без стрелки.

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

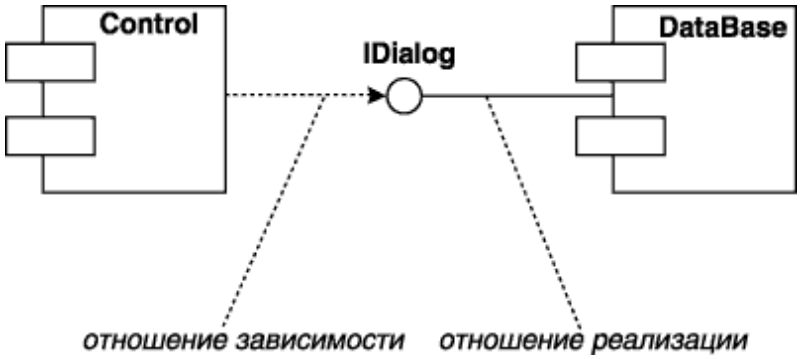


Рис. 4. Фрагмент диаграммы компонентов
с отношениями зависимости и реализации

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

между различными видами компонентов. Для рассмотренного фрагмента диаграммы компонентов (рис. 5) наличие подобной зависимости означает, что исполнимый компонент Control .exe использует или импортирует некоторую функциональность компонента Library .dll, вызывает страницу гипертекста Home .html и файл помощи Search .hlp, а исходный текст этого исполнимого компонента хранится в файле Control .cpp. При этом характер отдельных видов зависимостей может быть отмечен дополнительно с помощью текстовых стереотипов.

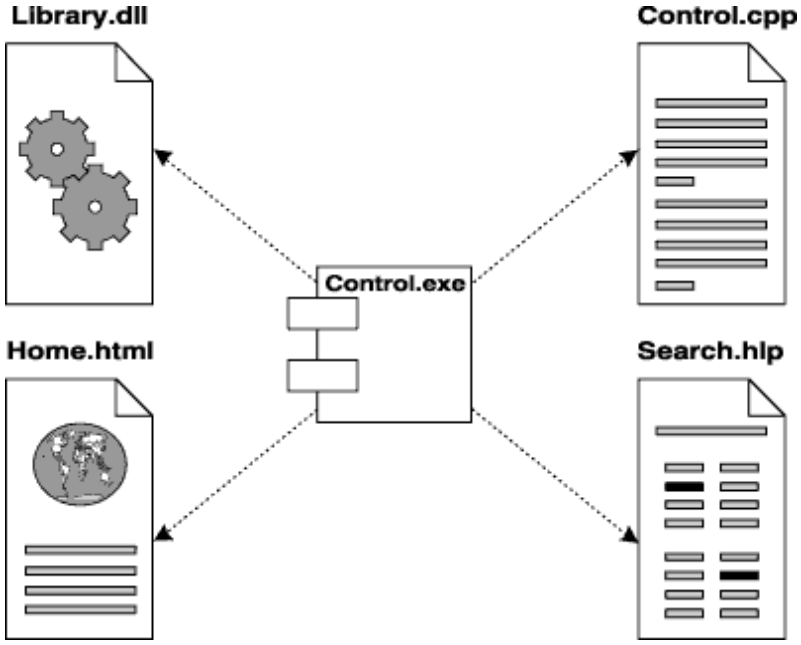


Рис. 5. Графическое изображение отношения зависимости между компонентами

На диаграмме компонентов могут быть также представлены отношения зависимости между компонентами и реализованными в них классами. Эта информация имеет значение для обеспечения согласования логического и физического представлений модели системы. Разумеется, изменения в структуре описаний классов могут привести к изменению этой зависимости. Ниже приводится фрагмент зависимости подобного рода, когда исполнимый компонент Control .exe зависит от соответствующих классов (рис. 6).

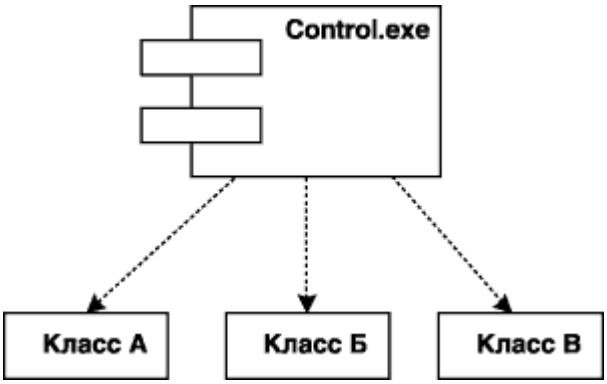


Рис. 6. Графическое изображение зависимости
между компонентом и классами

В этом случае из диаграммы компонентов не следует, что классы реализованы данным компонентом. Если требуется подчеркнуть, что некоторый компонент реализует отдельные классы, то для обозначения компонента используется расширенный символ прямоугольника. При этом прямоугольник компонента делится на две секции горизонтальной линией. Верхняя секция служит для записи имени компонента и, возможно, дополнительной информации, а нижняя секция – для указания реализуемых данным компонентом классов (рис. 7).

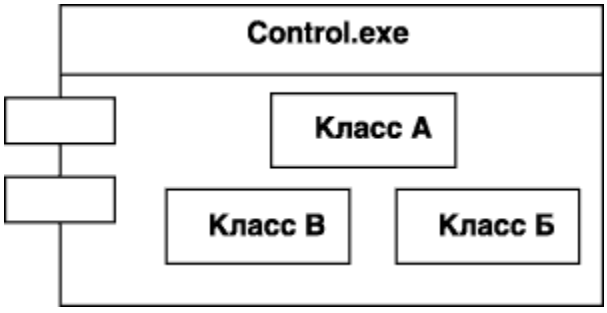


Рис. 7. Графическое изображение компонента
с информацией о реализуемых им классах

В случае если компонент является экземпляром и реализует три отдельных объекта, он изображается в форме компонента уровня экземпляров (рис. 8). Объекты, которые находятся в отдельном компоненте-экземпляре, изображаются вложенными в символ данного компонента. Подобная вложенность означает, что выполнение компонента влечет за собой выполнение операций соответствующих объектов. При этом существование компонента в течение вре-

мени исполнения программы обеспечивает функциональность всех вложенных в него объектов. Что касается доступа к этим объектам, то он может быть дополнительно специфицирован с помощью видимости, подобно видимости пакетов.

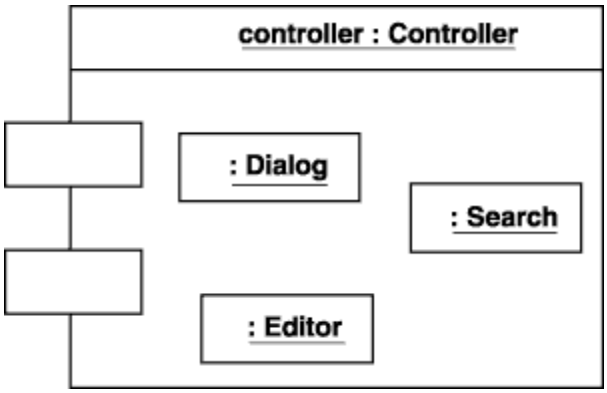


Рис. 8. Графическое изображение компонента-экземпляра, реализующего отдельные объекты

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

- **Рекомендации по построению диаграммы компонентов**

Разработка диаграммы компонентов предполагает использование информации не только о логическом представлении модели системы, но и об особенностях ее физической реализации. В первую очередь, необходимо решить, из каких физических частей или файлов будет состоять программная система. На этом этапе следует обратить внимание на такую реализацию системы, которая обеспечивала бы возможность повторного использования кода за счет рациональной декомпозиции компонентов, а также создание объектов только при их необходимости.

Общая производительность программной системы существенно зависит от рационального использования вычислительных ресурсов. Для этой цели необходимо большую часть описаний классов, их операций и методов вынести в динамические библиотеки,

оставив в исполняемых компонентах только самые необходимые для инициализации программы фрагменты программного кода.

После общей структуризации физического представления системы необходимо дополнить модель интерфейсами и схемами базы данных. При разработке интерфейсов следует обращать внимание на согласование различных частей программной системы. Включение в модель схемы базы данных предполагает спецификацию отдельных таблиц и установление информационных связей между ними.

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

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

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

Контрольные вопросы

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

8. Лабораторная работа №4. Построение диаграммы потоков данных

Целью работы является изучение порядка построения диаграммы потоков данных

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

8.1. Диаграмма потоков данных

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

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

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

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

[Грошев А. Основы работы с базами данных: http://www.intuit.ru/studies/courses/93/93/info]

Состав диаграмм потоков данных

Основными компонентами диаграмм потоков данных являются:

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

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

Внешняя сущность обозначается квадратом (рис. 1), расположенным над диаграммой и бросающим на нее тень для того, чтобы можно было выделить этот символ среди других обозначений.

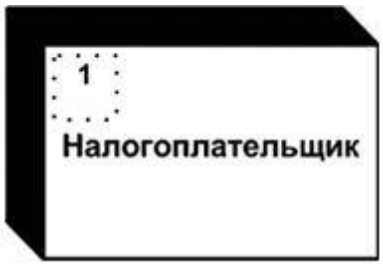


Рис. 1. Графическое изображение внешней сущности

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

Подсистема (или система) на контекстной диаграмме изображается так, как она представлена на рис. 2.

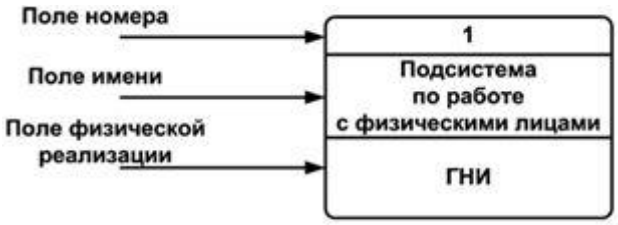


Рис. 2. Подсистема по работе с физическими лицами (ГНИ – Государственная налоговая инспекция)

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

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

Процесс на диаграмме потоков данных изображается, как показано на рис. 3.

Рис. 3. Графическое изображение процесса

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

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

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

Накопитель данных может быть реализован физически в виде микрофиши, ящика в картотеке, таблицы в оперативной памяти, файла на магнитном носителе и т. д. Накопитель данных на диаграмме потоков данных изображается, как показано на рис. 4.

Рис. 4. Графическое изображение накопителя данных

Накопитель данных идентифицируется буквой «D» и произвольным числом. Имя накопителя выбирается из соображения наибольшей информативности для проектировщика.

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

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

Поток данных на диаграмме изображается линией, оканчивающейся стрелкой, которая показывает направление потока (рис. 5). Каждый поток данных имеет имя, отражающее его содержание.

Рис. 5. Поток данных

Построение иерархии диаграмм потоков данных

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

- Размещать на каждой диаграмме от 3 до 6–7 процессов (аналогично SADT). Верхняя граница соответствует человеческим возможностям одновременного восприятия и понимания структуры сложной системы с множеством внутренних связей, нижняя граница выбрана по соображениям здравого смысла: нет необходимости детализировать процесс диаграммой, содержащей всего один или два процесса.
- Не загромождать диаграммы несущественными на данном уровне деталями.
- Декомпозицию потоков данных осуществлять параллельно с декомпозицией процессов. Эти две работы должны выполняться одновременно, а не одна после завершения другой.
- Выбирать ясные, отражающие суть дела имена процессов и потоков, при этом стараться не использовать аббревиатуры.

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

Для проверки контекстной диаграммы можно составить список событий. Список событий должен состоять из описаний действий внешних сущностей (событий) и соответствующих реакций системы на события. Каждое событие должно соответствовать одному или более потокам данных: входные потоки интерпретируются как

воздействия, а выходные потоки – как реакции системы на входные потоки.

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

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

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

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

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

Спецификации представляют собой описания алгоритмов задач, выполняемых процессами. Они содержат номер и/или имя про-

цесса, списки входных и выходных данных и тело (описание) процесса, являющееся спецификацией алгоритма или операции, трансформирующей входные потоки данных в выходные. Языки спецификаций могут варьироваться от структурированного естественного языка или псевдокода до визуальных языков моделирования.

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

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

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

После построения законченной модели системы ее необходимо верифицировать (проверить на полноту и согласованность). В пол-

ной модели все ее объекты (подсистемы, процессы, потоки данных) должны быть подробно описаны и детализированы. Выявленные недетализированные объекты следует детализировать, вернувшись на предыдущие шаги разработки. В согласованной модели для всех потоков данных и накопителей данных должно выполняться правило сохранения информации: все поступающие куда-либо данные должны быть считаны, а все считываемые данные должны быть записаны.

При моделировании бизнес-процессов диаграммы потоков данных (DFD) используются для построения моделей «AS-IS» и «AS-TO-BE», отражая, таким образом, существующую и предлагаемую структуру бизнес-процессов организации и взаимодействие между ними. При этом описание используемых в организации данных на концептуальном уровне, независимом от средств реализации базы данных, выполняется с помощью модели «сущность-связь».

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

1. Описание контекста процессов и построение начальной контекстной диаграммы.

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

2. Спецификация структур данных.

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

3. Построение начального варианта концептуальной модели данных.

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

4. Построение диаграмм потоков данных нулевого и последующих уровней.

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

Процессы разделяются на группы, которые имеют много общего (работают с одинаковыми данными и/или имеют сходные функции). Они изображаются вместе на диаграмме более низкого (первого) уровня, а на диаграмме нулевого уровня объединяются в один процесс. Выделяются накопители данных, используемые процессами из одной группы.

Декомпозируются сложные процессы и проверяется соответствие различных уровней модели процессов.

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

5. Уточнение концептуальной модели данных.

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

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

Контрольные вопросы

1. Что такое поток данных?
2. Как изображается объект на DFD?
3. Как изображается накопитель на DFD?

9. Лабораторная работа №5. Разработка тестового сценария

Целью работы является изучение порядка построения тестовых сценариев.

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

9.1. Разработка тестового сценария

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

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

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

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

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

Характеристики хорошего теста:

- Существует обоснованная вероятность выявления тестом ошибки.

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

В настоящее время наблюдается несколько методологий разработки тест кейсов. Они отличаются и теоретическим подходом, и практической реализацией.

Наиболее часто употребляемая методология разработки тестовых случаев – методология, при которой источниками тестовых случаев выступают случаи использования.

- Методология разработки тестовых случаев на основе сценариев использования.

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

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

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

Комбинация рейтингов частоты использования и критичности, применяемая для того, чтоб упорядочить случаи использования,

обеспечивает получение определенного критерия качества. Например, можно нарисовать эмблему в правом нижнем углу каждого окна. Это повторяется довольно часто, но если это не получится, система все еще способна выполнять наиболее важные функции для пользователя. Аналогично, соединение с сервером локальной базы данных происходит крайне редко, однако неудача этой операции сделает невозможным успешное выполнение множества других функций. Количество тестовых случаев, приходящихся на один случай использования, выбирается в зависимости от положения этого случая использования в рейтинговой таблице (чем чаще встречается данный случай использования и чем критичней его неверное выполнение для системы – там больше тест кейсов должно быть разработано). На этом этапе тестирования поддерживается проведение тестирования приложения в таком режиме, в каком оно будет использовано на практике.

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

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

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

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

Разработка сценария для случая использования предусматривает выполнение четырех действий:

- Идентификация всех значений, которые вводятся действующими субъектами, содержащимися в модели случая использования.

- Выделение классов эквивалентности значений каждого типа входных данных.
- Построение таблиц, в которые помещен список комбинаций значений из различных классов эквивалентности.
- Построение тестовых случаев, в которых сочетаются одна перестановка значений с необходимыми внешними ограничениями.

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

В табл. 1 показаны классы эквивалентности этих трех переменных.

Таблица 1

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

- Спецификация входных значений для системы управления кадрами

В табл. 2 каждой из этих переменных отводится отдельный столбец. В эти столбцы помещены значения из различных классов эквивалентности рассматриваемых переменных. Каждая строка таблицы представляет собой описание конкретного теста.

Количество тестовых случаев, которые необходимо построить, зависит от значения атрибута частоты использования каждого слу-

чая использования. Один из способов оценки соответствующего числа тестовых случаев заключается в том, что вычисляется произведение количества различных входов и количества классов эквивалентности каждого типа ввода с целью получения максимального количества перестановок.

Таблица 2

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

- Перестановки входных данных системы управления персоналом

На практике количество тестовых случаев может быть ограничено, если принимать во внимание важность того или иного случая использования или объем доступных системных ресурсов. Можно предпринять пробную попытку либо воспользоваться подходящими статистическими данными для определения, какой объем ресурсов необходим для выполнения типичного случая использования. Если известно количество случаев использования, то можно получить оценку трудозатрат, необходимых для реализации проекта в полном объеме.

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

целью проверки использования системы в полном объеме. С расширением тестовых случаев, тестовые результаты тоже расширяются.

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

Второй подход заключается в разработке **тестовых случаев большого цикла** (grand tour test cases), в котором каждый тестовый случай генерирует данные, которые служат входом для следующего тестового случая. По условиям такого подхода каждый тест переносит тестовые данные через весь жизненный цикл. Полученное при этом состояние базы данных используется в качестве входного для следующего теста. Этот метод особенно эффективен при тестировании жизненного цикла после того, как тестирование нижнего уровня позволило выявить большую часть дефектов, вызывающих отказы. Если выстроить тестовые случаи в соответствующую последовательность, то после успешного выполнения тестового случая 1 устанавливается такое состояние программы, которое ожидается как входное для тестового случая 2. Очевидная проблема в условиях очерченного подхода заключается в том, что неудачное выполнение тестового случая 1 оставляет программу в состоянии, которого мы не ожидали, в результате чего мы не можем выполнять прогон тестового случая 2 или даже вернуть программу в рабочее состояние.

Итак, при разработке тест кейсов на основе случаев использования процедура в общем-то ясна. Однако возникает другой вопрос – что необходимо подвергнуть тестированию? Какие аспекты функционирования системы? Рекомендуется проводить следующие виды тестирования:

- Тестирование на соответствие функциональным требованиям.

- Проверка качественных системных атрибутов. Добротная организация разработок программного обеспечения предусматривает методы подтверждения всех системных «требований», включая и претензии на придание программному продукту особых качеств. Существуют два вида претензий, с которыми может столкнуться программа при разработке продукта. Первый вид претензий представляет интерес только для организаций, занимающейся разработкой программных продуктов. Например, утверждение, что «программный код допускает многократное использование». Второй тип претензий представляет интерес для пользователей системы. Например, утверждение о том, что система является более полной, нежели другие системы подобного класса, предлагаемые на текущий момент на рынке программных продуктов. Вполне понятно, что не все эти претензии могут подвергаться проверке через тестирование. Однако на это следует обратить внимание.
- Тестирование механизма развертывания системы.
- Тестирование после развертывания системы. Естественное расширение тестирования механизма развертывания системы заключается в добавлении в тестируемый программный продукт функциональных средств самопроверки. Считается, что система «изнашивается» во времени по причине изменений, имеющих место в ее взаимодействии с окружением, примерно так же, как со временем изнашивается механическая система из-за трений между ее компонентами. По мере того, как устанавливаются все более новые версии стандартных драйверов и библиотек, несоответствия возрастают вместе с ростом вероятности возникновения отказов. Каждая новая версия dll-библиотеки привносит возможность появления новых областей нестыковки стандартных интерфейсов или появления состояния гонок между этой библиотекой и приложением. Функциональные средства самотестирования должны обеспечивать выполнение тестов, которые исследуют работу интерфейсов между этими программными продуктами.
- Тестирование взаимодействий окружения.
- Тестирование системы безопасности.

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

Контрольные вопросы

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

10. Лабораторная работа №6. Оценка необходимого количества тестов

Целью работы является изучение порядка оценки необходимого количества тестов

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

10.1. Оценка необходимого количества тестов

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

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

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

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

Существуют следущие подходы к оценке и измерению тестового покрытия:

1. **Покрытие требований (Requirements Coverage)** – оценка покрытия тестами функциональных и нефункциональных требований к продукту путем построения матриц трассировки (traceability matrix).
2. **Покрытие кода (Code Coverage)** – оценка покрытия исполняемого кода тестами, путем отслеживания непроверенных в процессе тестирования частей программного обеспечения.
3. **Тестовое покрытие на базе анализа потока управления** – оценка покрытия основанная на определении путей выполнения ко-

да программного модуля и создания выполняемых тест кейсов для покрытия этих путей.

Различия:

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

Ограничения:

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

- **Покрытие требований (Requirements Coverage)**

Расчет тестового покрытия относительно требований проводится по формуле:

$$Tcov = (Lcov/Ltotal) * 100\%$$

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

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

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

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

- **Покрытие кода (Code Coverage)**

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

$$Tcov = (Ltc/Lcode) * 100\%$$

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

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

- **Тестовое покрытие на базе анализа потока управления**

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

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

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

Для тестирования потоков управления определены разные **уровни тестового покрытия**:

Таблица 1

Уровни тестового покрытия

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

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

Контрольные вопросы

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

11. Лабораторная работа №6. Разработка тестовых пакетов

Целью работы является изучение порядка Разработка тестовых пакетов

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

11.1. Разработка тестовых пакетов

1. Создайте проект для тестирования

1. Запустите Visual Studio.
2. В меню **Файл** выберите пункт **Создать** > **Проект**.

Откроется диалоговое окно **Новый проект**.

3. В области **Установленные шаблоны**выберите шаблон **Visual C#**.
4. В списке типов приложения выберите пункт **Библиотека классов**.
5. В поле **Имя** введите **Bank** и нажмите **OK**.

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

Примечание

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

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

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

В этой статье проводится тестирование на примере метода Debit. Метод Debit вызывается, когда денежные средства снимаются со счета. Так выглядит определение метода:

```
// Method to be tested.
```

```
public void Debit(double amount)
{
 if(amount > m_balance)
 {
  throw new ArgumentOutOfRangeException(«amount»);
 }
 if (amount < 0)
 {
  throw new ArgumentOutOfRangeException(«amount»);
 }
 m_balance += amount;
}
```

2. Создание проекта модульного теста

3. В меню **Файл** выберите **Добавить** > **Создать проект**.

4. В диалоговом окне **Новый проект** разверните узлы **Установленные** и **Visual C#** и выберите **Тест**.

5. В списке шаблонов выберите **Проект модульного теста**.

6. В поле **Имя** введите BankTests, а затем нажмите кнопку **ОК**.

Проект **BankTests** добавляется в решение **Банк**.

7. В проекте **BankTests** добавьте ссылку на проект **Банк**.

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

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

9. Создание тестового класса

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

10. Переименование файла класса

В **обозревателе решений** выберите файл UnitTest1.cs в проекте BankTests. В контекстном меню выберите команду **Переименовать**, а затем переименуйте файл в BankAccountTests.cs. Выберите **Да** в диалоговом окне, предлагающем переименовать все ссылки на элемент кода UnitTest1 в проекте.

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

```
using System;
using Microsoft.VisualStudio.TestTools.UnitTesting;

namespace BankTests
{
 [TestClass]
 public class BankAccountTests
 {
  [TestMethod]
  public void TestMethod1()
  {
  }
 }
}
```

11. Добавление оператора using в тестируемый проект

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

C#Копировать

using BankAccountNS;

12. Требования к тестовому классу

Минимальные требования к тестовому классу следующие:

- Атрибут [TestClass] является обязательным для платформы модульных тестов Microsoft для управляемого кода в любом классе, содержащем методы модульных тестов, которые необходимо выполнить в обозревателе тестов.
- Каждый метод теста, предназначенный для запуска в обозревателе тестов, должен иметь атрибут [TestMethod].

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

13. Создание первого тестового метода

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

Существует по крайней мере три поведения, которые требуется проверить:

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

Совет

Метод по умолчанию TestMethod1 можно удалять, так как он не используется в этом руководстве.

14. Создание метода теста

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

```
[TestMethod]
public void Debit_WithValidAmount_UpdatesBalance()
{
 // Arrange
 double beginningBalance = 11.99;
 double debitAmount = 4.55;
 double expected = 7.44;
 BankAccount account = new BankAccount(«Mr. Bryan Walton», beginning-
Balance);

 // Act
 account.Debit(debitAmount);

 // Assert
 double actual = account.Balance;
 Assert.AreEqual(expected, actual, 0.001, «Account not debited correctly»);
}
```

Метод очень прост: он создает новый объект BankAccount с начальным балансом, а затем снимает допустимое значение. Он ис-

пользует метод AreEqual, чтобы проверить, что конечный баланс соответствует ожидаемому.

15. Требования к методу теста

Метод теста должен удовлетворять следующим требованиям:

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

16. Сборка и запуск теста

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

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

Совет

Если **обозреватель тестов** не откроется после успешной сборки, выберите в меню пункт **Тест**, щелкните **Windows**, а затем – **Обозреватель тестов**.

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

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

17. Исправление кода и повторный запуск тестов

18. Анализ результатов теста

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

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

19. Исправление ошибки

Для исправления ошибки замените строку:

```
m_balance += amount;
```

на:

```
C#Копировать
```

```
m_balance -= amount;
```

20. Повторный запуск теста

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

21. Использование модульных тестов для улучшения кода

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

22. Анализ проблем

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

- больше баланса или
- меньше нуля.

23. Создание методов теста

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

```
 [TestMethod]
[ExpectedException(typeof(ArgumentOutOfRangeException))]
public void Debit_WhenAmountIsLessThanZero_ShouldThrowArgumentOutOfRange()
{
  // Arrange
  double beginningBalance = 11.99;
  double debitAmount = -100.00;
  BankAccount account = new BankAccount(«Mr. Bryan Walton», beginningBalance);

  // Act
  account.Debit(debitAmount);

  // Assert is handled by the ExpectedException attribute on the test method.
}
```

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

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

1. Создать новый метод теста с именем Debit_WhenAmountIsMoreThanBalance_ShouldThrowArgumentOutOfRange.
2. Скопировать тело метода из Debit_WhenAmountIsLessThanZero_ShouldThrowArgumentOutOfRange в новый метод.
3. Присвоить debitAmount значение, превышающее баланс.

24. Запуск тестов

Запуск двух методов теста показывает, что тесты работают правильно.

25. Продолжение анализа

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

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

C#Копировать

```
throw new ArgumentOutOfRangeException(«amount»);
```

Так выглядит конструктор, который можно использовать для сообщения более детальной информации: ArgumentOutOfRangeException(String, Object, String) включает имя аргумента, значения аргумента и определяемое пользователем сообщение. Мы можем выполнить рефакторинг тестируемого метода для использования дан-

ного конструктора. Более того, можно использовать открытые для общего доступа члены типа для указания ошибок.

26. Рефакторинг тестируемого кода

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

```
public const string DebitAmountExceedsBalanceMessage = «Debit amount exceeds balance»;
public const string DebitAmountLessThanZeroMessage = «Debit amount is less than zero»;
```

Затем изменим два условных оператора в методе Debit:

C#Копировать

```
if (amount > m_balance)
{
  throw new ArgumentOutOfRangeException(«amount», amount, DebitAmountExceedsBalanceMessage);
}

if (amount < 0)
{
  throw new ArgumentOutOfRangeException(«amount», amount, DebitAmountLessThanZeroMessage);
}
```

27. Рефакторинг тестовых методов

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

В этом случае метод Debit_WhenAmountIsMoreThanBalance_ShouldThrowArgumentOutOfRange может выглядеть следующим образом:

```
[TestMethod]
public void Debit_WhenAmountIsMoreThanBalance_ShouldThrowArgumentOutOfRange()
{
  // Arrange
  double beginningBalance = 11.99;
  double debitAmount = 20.0;
```

```
BankAccount account = new BankAccount(«Mr. Bryan Walton», beginning-
Balance);

// Act
try
{
  account.Debit(debitAmount);
}
catch (ArgumentOutOfRangeException e)
{
  // Assert
  StringAssert.Contains(e.Message, BankAc-
count.DebitAmountExceedsBalanceMessage);
}
}
```

28. Повторное тестирование, переписывание и анализ

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

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

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

```
[TestMethod]
```

```
        public void De-
bit_WhenAmountIsMoreThanBalance_ShouldThrowArgumentOutOfRange()
        {
         // Arrange
         double beginningBalance = 11.99;
         double debitAmount = 20.0;
         BankAccount account = new BankAccount(«Mr. Bryan Walton», beginning-
Balance);

         // Act
         try
         {
          account.Debit(debitAmount);
         }
         catch (ArgumentOutOfRangeException e)
         {
          // Assert
          StringAssert.Contains(e.Message, BankAc-
count.DebitAmountExceedsBalanceMessage);
          return;
         }

         Assert.Fail(«The expected exception was not thrown.»);
        }
```

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

Контрольные вопросы

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

12. Лабораторная работа №7. Оценка программных средств с помощью метрик

Целью работы является изучение порядка оценки программных средств с помощью метрик

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

Понятие качества

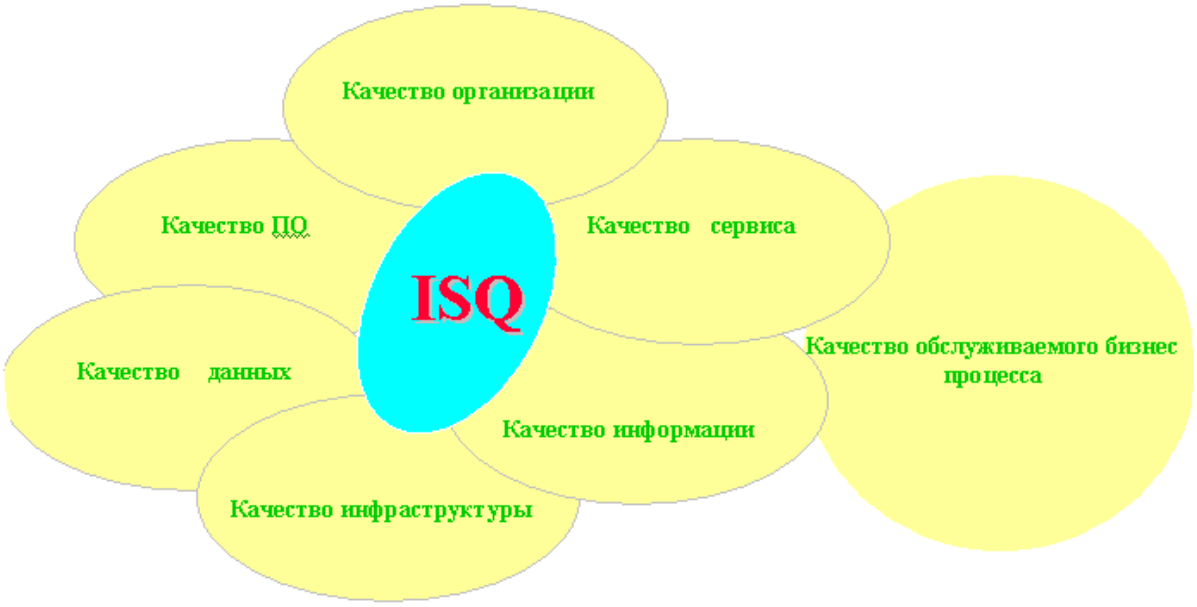
Сейчас существует несколько определений качества, которые в целом совместимы друг с другом. Приведем наиболее распространенные:

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

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

Многомерность качества

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



ISQ – Information System Quality

Составляющие качества информационной системы:

- Качество инфраструктуры (infrastructure quality): качество аппаратного и поддерживающего программного обеспечения (например, качество операционных систем, компьютерных сетей и т. п.).
- Качество программного обеспечения (software quality): качество программного обеспечения информационной системы.
- Качество данных (data quality): качество данных, использующихся информационной системой на входе.
- Качество информации (information quality): качество информации, продуцируемое информационной системой.
- Качество организации (administrative quality) – качество менеджмента, включая качество бюджетирования, планирования и календарного контроля.
- Качество сервиса (service quality) – качество обучения, системной поддержки и т. п.

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

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

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

- **Дерево характеристик качества**

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

Следует отметить, что метрики качества являются изначально неочевидной категорией. Исторически сначала были выделены ряд универсальных и неполных метрик на основе следующих шагов:

- Определение множества характеристик, которые, являясь важными для программного обеспечения, допускают несложное измерение и не перекрываются.

• Выделение кандидатов в метрики, которые измеряют степень удовлетворения указанным характеристикам.

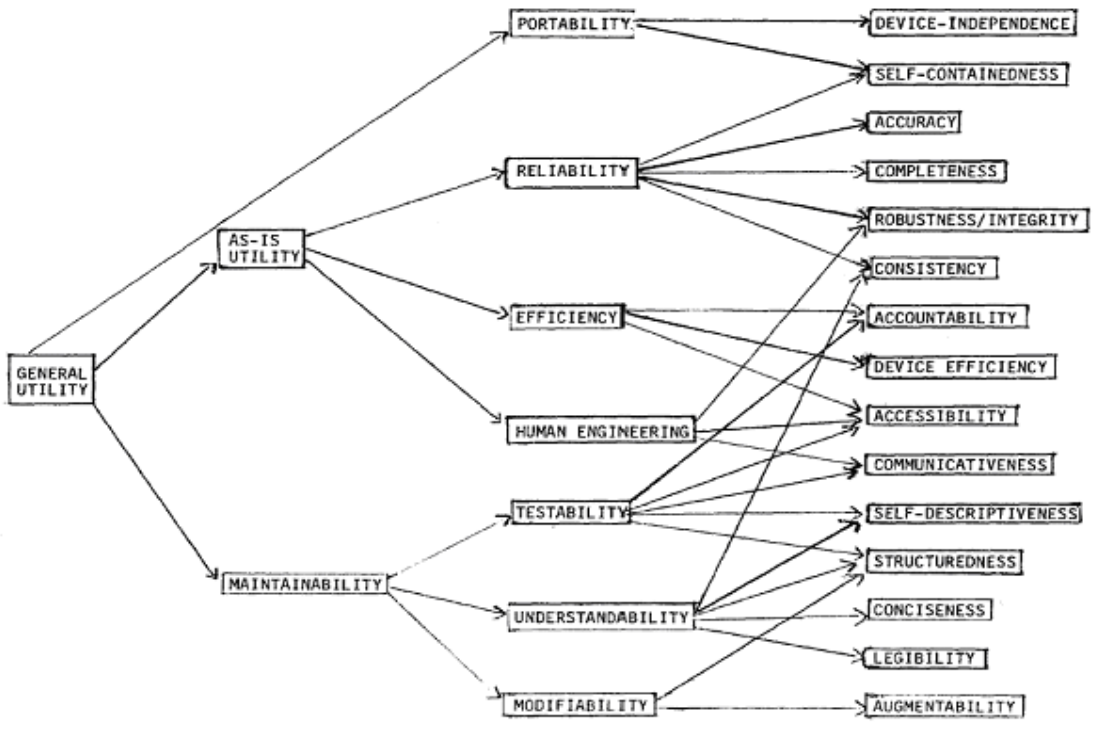
• Исследование характеристик и связанных метрик, для определения корреляции, значимости, степени автоматизируемости.

• Исследование корреляции между метриками, степени перекрытия, зависимости и недостатков.

• Рафинирование множества метрик в целом во множество метрик, которые в совокупности адекватно отражают качество программного обеспечения.

• Корректировка каждой метрики в итоговом множестве в контексте зафиксированных множеств характеристик и метрик.

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



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

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

- **Шкала измерения характеристик (ISO 12207) – введение в метрики**

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

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

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

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

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

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

Потребность в ресурсах памяти и производительности компьютера в процессе решения задач значительно изменяется в зависимости от состава и объема исходных данных. Для корректного определения предельной пропускной способности информационной системы с данным программным средством нужно измерить экстремальные и средние значения длительностей исполнения функциональных групп программ и маршруты, на которых они достигаются. Если предварительно в процессе проектирования производительность компьютера не оценивалась, то, скорее всего, понадобится большая доработка или даже замена компьютера на более быстродействующий.

Оценка практичности программных средств проводится экспертами и включает определение понятности, простоты использования, изучаемости и привлекательности программного средства. В основном это качественная (и субъективная) оценка в баллах, однако некоторые атрибуты можно оценить количественно по трудоемкости и длительности выполнения операций при использовании программного средства, а также по объему документации, необходимой для их изучения.

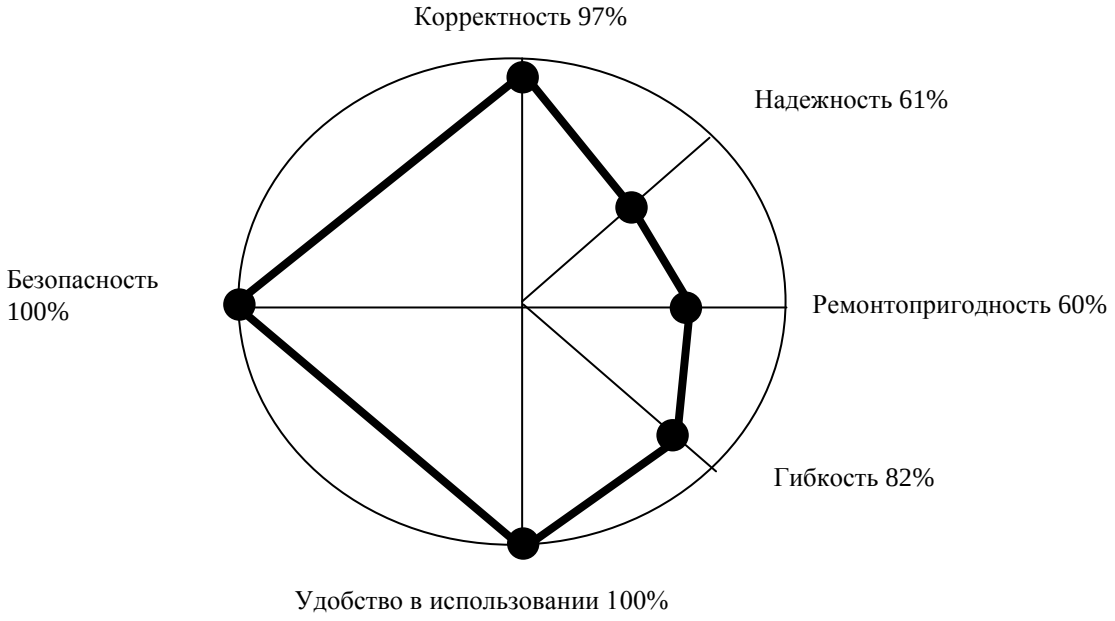
Сопровождаемость можно оценивать полнотой и достоверностью документации о состояниях программного средства и его компонентов, всех предполагаемых и выполненных изменениях, позволяющей установить текущее состояние версий программ в любой момент времени и историю их развития. Она должна определять стратегию,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

создания, изменения и применения документов на программы и данные.

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

- **Пример графического изображения качества**

Для мониторинга метрик качества и подготовки информации для принятия решений собранные метрики должны представляются в наглядном виде, обеспечивающим полноту информации, что особенно важно при отсутствии консолидированных метрик качества «Пример графического изображения качества».



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

Контрольные вопросы

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

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

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

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

Что такое code review

Code review – инженерная практика в терминах гибкой методологии разработки. Это анализ (инспекция) кода с целью выявить ошибки, недочеты, расхождения в стиле написания кода, в соответствии написанного кода и поставленной задачи. К очевидным плюсам этой практики можно отнести:

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

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

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

- Как проводить review.

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

- Из чего состоит review.

- Сначала **design review** – анализ будущего дизайна (архитектуры).Данный этап очень важен, так как без него ревью кода будет менее полезным или вообще бесполезным (если программист написал код, но этот код полностью неверен – не решает поставленную задачу, не удовлетворяет требованиям по памяти, времени). Пример: программисту поставили задачу написать алгоритм сортировки массива. Программист реализовал алгоритм bogo-sort, причем с точки зрения качества кода – не придраться (стиль написания, проверка на ошибки), но этот алгоритм совершенно не подходит по времени работы. Поэтому ревью в данном случае бесполезно (конечно – это утрированный пример, но я думаю, суть ясна), здесь необходимо полностью переписывать алгоритм.
- Собственно, сам **code review** – анализ написанного кода. На данном этапе автору кода отправляются замечания, пожелания по написанному коду.

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

- Как проводить design review.

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

- Как проводить code review.

Можно проводить code review разными способами – дистанционно, когда каждый разработчик сидит за своим рабочим местом, и совместно – сидя перед монитором одного из коллег, либо в специально выделенным для этого месте, например meeting room. В принципе существует много способов (можно даже распечатать исходный код и вносить изменения на бумаге).

- Pre-commit review.

Данный вид review проводится перед внесением изменений в VCS. Этот подход позволяет содержать в репозитории только про-

веренный код. В microsoft используется этот подход: всем участникам review рассылаются патчи с изменениями. После того как собран и обработан фидбэк, процесс повторяется до тех пор пока все ревьюверы не согласятся с изменениями.

- Post-commit review.

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

- Тематические review.

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

- «Давайте поищем ошибки в этом модуле» – не подходит в качестве цели, так как она необозрима.
- «Анализ алгоритма на соответствие спецификации RFC 1149» – уже лучше.

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

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

- Результаты review.

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

- Описание способа решения задачи (design review).
- UML диаграммы (design review).

- Комментарии к стилю кода (code review).
- Более правильный вариант (быстрый, легкочитаемый) реализации (design review, code review).
- Указание на ошибки в коде (забытое условие в switch и т. д.) (code review).
- Юнит тесты (design review, code review).

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

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

Для преодоления этих проблем частично может помочь:

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

Контрольные вопросы

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

СОДЕРЖАНИЕ

Предисловие 2
Содержание дисциплины в соответствии с учебным планом 2
Содержание практических занятий 2
1. Практическое занятие №1. Анализ предметной области 3
2. Практическое занятие №2. Разработка и оформление технического задания 8
3. Практическое занятие №3. Построение архитектуры программного средства 12
4. Практическое занятие №4. Изучение работы в системе контроля версий 41
5. Лабораторная работа №1. Построение диаграммы Вариантов использования и диаграммы последовательности 51
6. Лабораторная работа №2. Построение диаграммы Деятельности, диаграммы состояний и диаграммы классов 89
6.1. Диаграммы деятельности 89
6.2. Диаграмма состояний 99
6.3. Диаграммы классов 110
7. Лабораторная работа №3. Построение диаграммы компонентов 116
7.1. Диаграмма компонентов. 116
8. Лабораторная работа №4. Построение диаграммы потоков данных 128
8.1. Диаграмма потоков данных 128
9. Лабораторная работа №5. Разработка тестового сценария 138
9.1. Разработка тестового сценария 138
10. Лабораторная работа №6. Оценка необходимого количества тестов 146
10.1. Оценка необходимого количества тестов 146
11. Лабораторная работа №6. Разработка тестовых пакетов 150
11.1. Разработка тестовых пакетов 150
12. Лабораторная работа №7. Оценка программных средств с помощью метрик 160
13. Лабораторная работа №8. Инспекция программного кода на предмет соответствия стандартам кодирования 167