Министерство образования Республики Беларусь
Учреждение образования «Витебский государственный университет имени П.М. Машерова»
Кафедра прикладного и системного программирования

С.А. Ермоченко

ОСНОВЫ БИЗНЕС-АНАЛИЗА

Курс лекций

Витебск
ВГУ имени П.М. Машерова
2022

УДК 004.414.3(075.8)
ББК 32.972я73
Е74

Печатается по решению научно-методического совета учреждения образования «Витебский государственный университет имени П.М. Машерова». Протокол № 1 от 05.10.2022.

Автор: заведующий кафедрой прикладного и системного программирования ВГУ имени П.М. Машерова, кандидат физико-математических наук, доцент **С.А. Ермоченко**

Р е ц е н з е н т :
заведующий кафедрой информационных систем
и технологий УО «ВГТУ», кандидат технических наук,
доцент *В.Е. Казаков*

Ермоченко, С.А.

Е74 Основы бизнес-анализа : курс лекций / С.А. Ермоченко. – Витебск : ВГУ имени П.М. Машерова, 2022. – 36 с.

В курсе лекций излагаются основы методики сбора, анализа и документирования требований к разрабатываемому программному обеспечению.

Предназначается для студентов специальностей «Управление информационными ресурсами» (дисциплина «Основы бизнес-анализа в области разработки программного обеспечения»), «Программное обеспечение информационных технологий» (дисциплина «Разработка и анализ требований»).

УДК 004.414.3(075.8)
ББК 32.972я73

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Бизнес-анализ является первым и самым важным этапом жизненного цикла разработки программного обеспечения. На этом этапе осуществляется сбор, анализ и документирование требований к разрабатываемой системе.

В приведённом конспекте лекций рассматриваются основы разработки требований, роль этого процесса в общем жизненном цикле разработки, влияние этапа анализа требований на каждый из последующих этапов, а также проектные роли, принимающие участие в процессе сбора, анализа и документирования требований.

В практике разработки программного обеспечения подходы к управлению требованиями могут отличаться в зависимости от применяемых методологий управления проектами, специфики доменных областей, состава проектных команд и т.д. Поэтому в данном конспекте лекций рассматриваются базовые подходы, общие для подавляющего большинства вариантов.

Материал данного учебного издания ориентирован на дисциплины «Основы бизнес-анализа в области разработки программного обеспечения» (специальность 1-26 03 01 «Управление информационными ресурсами») и «Разработка и анализ требований» (специальность 1-40 01 01 «Программное обеспечение информационных технологий»).

1. Жизненный цикл разработки программного обеспечения

1.1. Этапы жизненного цикла

Создание программного обеспечения – это достаточно сложный процесс, проходящий несколько этапов. Количество этапов и их назначение могут отличаться, но, как правило, выделяют пять этапов, в том или ином виде присутствующих всегда.

Термин «этап» сложился исторически, когда каждый из них следовал после окончания предыдущего. И хоть в настоящий момент этапы могут следовать не линейно, некоторые могут идти параллельно, некоторые могут многократно повторяться и т. д., их по-прежнему принято называть «этапами»:

1. Анализ требований.
2. Проектирование.
3. Реализация (программирование).
4. Тестирование.
5. Сопровождение.

Первый этап – анализ. На этом этапе ведется работа с заказчиком или будущими пользователями программного обеспечения. Цель данного этапа – выявить требования к продукту (что он должен и чего не должен делать) и формализовать эти требования для разработчиков, чтобы постановка задачи была ясна программистам, не являющимся специалистами в предметной области программного обеспечения.

Второй этап – проектирование. На этом этапе строятся необходимые модели (в первую очередь модель предметной области), обеспечивающие корректную обработку данных. Далее проектируется внутренняя структура программного обеспечения на различных уровнях. Цель данного этапа – создать такой каркас системы, который удовлетворит сформированным требованиям (производительность, масштабируемость, простота модификации и т.п.).

Третий этап – реализация. На этом этапе с помощью выбранных языков программирования, библиотек и технологий создается программный код программного обеспечения, реализующий спроектированную ранее модель предметной области и алгоритмы обработки пользовательских данных.

Четвертый этап – тестирование. На этом этапе оценивается корректность работы системы, выявляются и исправляются дефекты (работа системы, не совпадающая со сформулированными требованиями). Если программное обеспечение использует сложные математические модели для анализа и обработки данных, проверяется адекватность и точность построения этих моделей. Проверяются используемые алгоритмы на устойчивость их работы на самых разных наборах входных данных. Цель этапа – убедиться, что разработанная система имеет приемлемое качество.

Пятый этап – сопровождение. На этом этапе осуществляется ввод в эксплуатацию разработанного программного обеспечения, консультирование пользователей по возникающим вопросам и проблемам функционирования программного обеспечения (техническая поддержка), обучение (при необходимости) пользователей работе с системой, исправление обнаруженных ошибок, пропущенных при тестировании, сбор сведений о необходимых улучшениях в следующих версиях программного обеспечения, поддержка стабильности работы программного обеспечения (периодическое резервное копирование данных, очистка системы от накапливающихся временных данных, утративших актуальность, обновление версий используемого стороннего программного обеспечения и т.д.).

Как уже было сказано, различные этапы жизненного цикла не обязательно следуют линейно один за другим. Порядок этапов, их повторяемость, задачи, решаемые при очередном повторении некоторого этапа, определяются выбранной менеджером проекта моделью жизненного цикла разработки программного обеспечения. Таких моделей может существовать большое количество. По большому счету модель жизненного цикла разработки программного обеспечения в каждом проекте уникальна. Тем не мене принято выделять несколько типичных моделей, комбинируя которые можно получить оптимальную для конкретного проекта модель.

1.2. Каскадная модель жизненного цикла

Исторически первой моделью можно назвать ***каскадную*** модель (см. рисунок 1.1). Именно такая модель породила термин «этап». Согласно этой модели (в её первоначальном виде), все этапы должны следовать в строго друг за другом и выполняться только один раз (как вода, проходящая каскадный водопад, проходит каждый порог, но только один раз). На практике такая модель стала практически неосуществима, так как на любом из этапов может выявиться ошибка, допущенная на предыдущем этапе, что парализует дальнейшую разработку проекта.

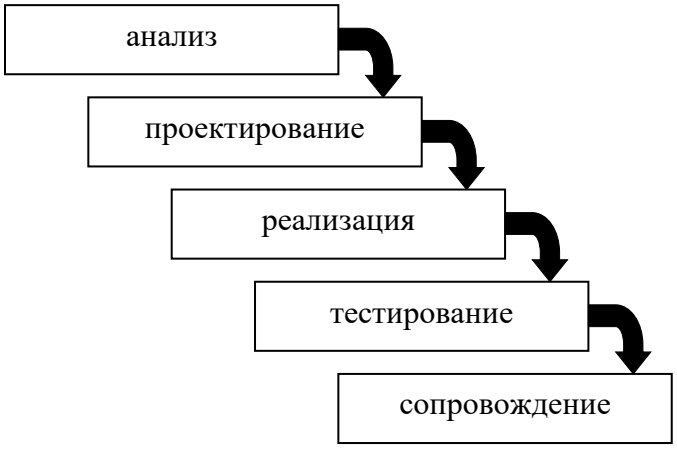

Рисунок 1.1 – Каскадная модель жизненного цикла программного обеспечения

Так как на любом этапе жизненного цикла потенциально возможны ошибки, влияющие на последующие этапы, при выявлении таких ошибок целесообразно вернуться на тот этап, на котором была допущена ошибка, и исправить её. Не всегда, правда, удается определить, на каком из предыдущих этапов была допущена ошибка, поэтому иногда используют возврат непосредственно на предыдущий этап, если там ошибку исправить не удалось, то на этап, предшествующий предыдущему этапу, и так далее (см. рисунок 1.2). Такая модель позволяет учесть самые разные факторы, влияющие на процесс разработки, и в пределе может дать идеальный результат. Но проблема использования такой модели заключается в непрогнозируемых материальных, людских и временных затратах. То есть с применением этой модели можно создать идеальную информационную систему, но остаётся нерешённым вопрос, сколько труда, времени и денег для этого понадобится.

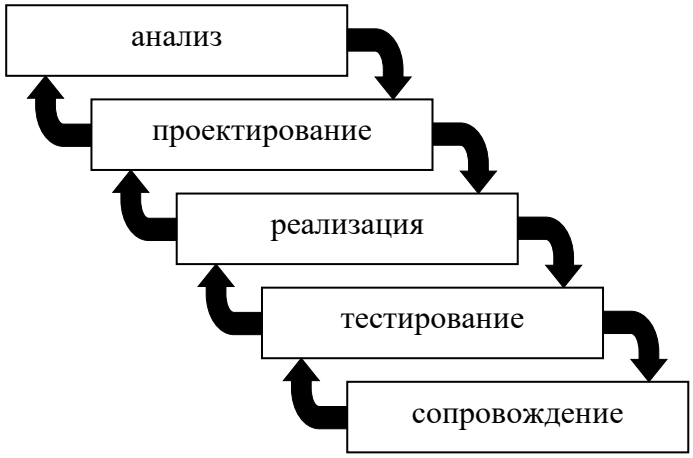

Рисунок 1.2 – Модифицированная каскадная модель жизненного цикла

Одним из способов решения такой проблемы является разбиение программного продукта на ряд подзадач, решение каждой из таких задач представляется как разработка «мини программы», для которой применяется своя модель жизненного цикла.

1.3. Итерационная модель жизненного цикла

Такая модель получила название ***итерационная***. Итерацией в такой модели называется подзадача, отдельно решаемая с применением каждого из этапов. По завершении одной итерации, на которой были пройдены все этапы, начинается следующая итерация, на которой вся последовательность этапов повторяется снова.

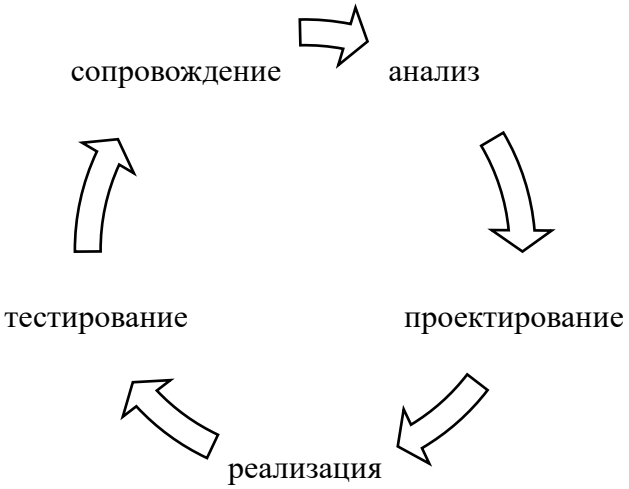

Рисунок 1.3 – Итерационная модель жизненного цикла программного обеспечения

При этом существует большое количество вариаций данной модели. Например, из циклического повторения этапов может исключаться анализ, проводимый только один раз вначале проекта, или выполняющийся параллельно с итерациями, на которых выполняются остальные этапы.

Также из циклического повторения может исключаться этап сопровождения, проводимое в конце проекта как завершающая итерация. Но при этом зачастую в процессе разработки выполняется поэтапное внедрение промежуточных версий разрабатываемого программного обеспечения в некоторое тестовое окружение, доступное специалистам по обеспечению качества (специалистам по тестированию) и / или заказчикам проекта. Такое внедрение может выполняться в конце каждой итерации. При этом такое внедрение в последнее время стараются максимально автоматизировать с помощью практик CI (Continuous Integration – непрерывная интеграция). Возможно также, что в конце определённых итераций после удовлетворительных результатов тестирования промежуточная версия программного обеспечения уже будет пригодна для работы конечных пользователей. В таком случае процесс внедрения этой версии в производственное окружение заказчика также может происходить автоматически с помощью практик CD (Continuous Delivery – непрерывная доставка). В случае применения CI/CD подходов часть этапа сопровождения (внедрение) осуществляется автоматически в конце каждой итерации, а остальные работы, осуществляющиеся на этом этапе (тех. поддержка и обучение пользователей, техническое обслуживание проекта, анализ результатов внедрения), выполняются параллельно с основными итерациями.

Существует вариант итерационной модели, при котором каждая итерация может также «зацикливаться», то есть после завершения основной работы по итерации продолжается улучшение функционала до приемлемо-

го уровня качества (при этом параллельно могут разрабатываться другие итерации).

Также зачастую этап проектирования может проводиться только на нескольких первых итерациях, и в дальнейшем построенные модели только незначительно уточняются, что не требует выделения времени на эту работу, как на отдельный этап. В таком случае проектирование и реализацию рассматривают как один этап, в котором на ранних итерациях проекта преобладают задачи проектирования, а на более поздних – реализация.

Применение итерационной модели при минимально возможной длительности итерации (1-2 недели) при уменьшении количества документации, более плотным сотрудничеством непосредственно с заказчиком, включающем более частое предъявление заказчику промежуточных результатов разработки, стало очень популярным в так называемых гибких методологиях управления проектами.

Гибкие методологии по сути представляют собой итерационную модель, в которой этапы анализа и сопровождения непрерывно следуют параллельно итерациям реализации (совмещённой с проектированием) и тестирования. При этом зачастую в течение одной итерации этап реализации также выполняется параллельно этапу тестирования таким образом, чтобы разработчики выполняли разработку одной версии программного обеспечения (Build-а – сборки), а специалисты по обеспечению качества тестировали другую версию (Build).

1.4. Спиралевидная модель жизненного цикла

Дальнейшим развитием итерационной модели является ***спиралевидная*** модель, которая отличается от итерационной модели ориентацией на управление рисками. В процессе создания программного обеспечения существуют различные риски, такие как:

– дефицит специалистов;
– нереалистичные сроки и бюджет;
– несоответствие спецификации;
– перфекционизм;
– постоянные изменения;
– недостаточная производительность;
– разрыв в квалификации специалистов.

При планировании сроков итерации и объёмов задач, которые должны будут решаться на итерации, менеджер проекта оценивает все эти риски и оперативно реагирует на такие, которые превышают некий допустимый порог. Рассмотрим более подробно эти риски.

Дефицит специалистов может возникать в тех ситуациях, когда объём задач, решаемых на текущей итерации, требует большого количества

специалистов определённой области, в то время как количество специалистов этой области, задействованное на проекте, меньше требуемого количества. В такой ситуации, менеджеру необходимо перераспределять такие задачи между текущей и последующими итерациями для того, чтобы их можно было решить силами имеющихся специалистов. К этой же области рисков можно отнести и другую проблему планирования задач в рамках проекта, когда на некоторые итерации выставляются достаточно простые задачи, которые вынуждены решать специалисты высокой квалификации. А через несколько итераций накапливается ряд сложных задач, которые распределить равномерно на оставшиеся итерации уже невозможно таким образом, чтобы можно было выполнить их силами имеющихся высококвалифицированных специалистов. В такой ситуации приходится или привлекать дополнительных специалистов, что приводит к превышению бюджета проекта, либо отодвигать сроки окончания проекта. Обе ситуации могут быть неприемлемыми для заказчика. Такая ситуация как раз и создаёт риск нереалистичных сроков и бюджета проекта.

Риск несоответствия спецификации (документированные требования) возрастает с ростом сроков работы над итерациями и времени между началом работы над требованием и представлением промежуточного результата заказчику. Дело в том, что никакая спецификация не может в полной мере отразить ожидания заказчика. Поэтому возникает разрыв между пониманием требований заказчиком и разработчиком. Следствием такого разрыва является выполнение некоторой итерации (или нескольких итераций), результаты которой формально соответствуют спецификации, но на практике не устраивает заказчика. В таком случае часть проделанной работы заказчиком должна оплачиваться, но никакой пользы ему не приносит. Для снижения такого риска рекомендуется как можно чаще демонстрировать промежуточные результаты работы заказчику, что позволит более конкретно понять и детализировать спецификацию и улучшить степень соответствия программного обеспечения этой спецификации.

Также на практике при оценке риска несоответствия спецификации могут прибегать к оценке последствий неточности спецификации в соотнесении с оценкой времени и стоимости разработки детальной спецификации. Например, при описании некоторого требования, может оказаться, что его суть достаточно проста и интуитивно понятна как заказчику, так и разработчику, и специалисту по тестированию. Однако точно выяснить это без детальной проработки и документирования требования невозможно. Но такую интуитивную понятность некоторых требований можно оценить эмпирически. Например, бизнес-аналитик формулирует требование очень кратко и размыто в надежде на то, что разработчик его и так правильно поймёт. Разработчик выполняет требование исходя из своего интуитивного представления о пропущенных деталях этого требования. После этого специалист по тестированию также интуитивно пытается это требование про-

верить. На последнем этапе, в конце итерации заказчику предъявляют полученный результат, и он тоже убеждается, что его требование было понято и реализовано правильно. В таком случае оказывается сэкономлено время аналитика на излишне подробное документирование требования, время разработчика и специалиста по тестированию на изучение детально описанного требования, а также время и средства заказчика для обеспечения этого требования.

Однако следует оценить последствия ситуации, когда, либо на этапе тестирования, либо на этапе предъявления результатов реализации требования заказчику выясняется, что интуитивное понимание требования оказалось неверным. В таком случае наоборот будет затрачено дополнительное время на уточнение требования аналитиком, исправление реализации разработчиком, и повторное тестирование специалистом по качеству, что, естественно, увеличивает время работы над требованием и удорожает процесс разработки. Но стоит отметить, что опытные аналитики, изначально определяя, какие требования стоит описывать детально, а какие – в минимальном объёме, ошибаются не так часто, всего в 10-15% случаев. При этом детального описания в реальности могут требовать всего около 20-25% требований, а около 75-80% требований могут быть описаны в краткой и достаточно размытой форме. Поэтому экономия ресурсов на кратко описанных и интуитивно понимаемых требованиях в опытных командах на практике часто компенсирует потери при иногда возникающих ошибках.

Риск перфекционизма или постоянных изменений возникает в случае, когда многократно повторяются один или два этапа. В первом случае это разработка и тестирование, во втором случае – это анализ требований. При этом некоторая работа на этих этапах проводится, но результат этой работы (прирост «полезности», так называемый Profit, оцениваемый применяемыми в проекте метриками) несоизмерим с затратами. Например, после нескольких итераций по выявлению дефектов на этапе тестирования и их исправлению на этапе разработки, может складываться ситуация, когда новые незначительные дефекты выявляются, программисты тратят значительное время и другие ресурсы на их исправление, а общее качество проекта при этом увеличивается на десятые доли процентов. Или при анализе требований аналитик может тратить много времени на согласование требований из-за того, что заказчик никак не может определиться, как именно должен работать тот или иной функционал. Но при этом из времени, отводимого на выполнение всего проекта, остаётся всё меньше и меньше на реальную работу. В таком случае необходимо директивно прерывать такие повторения, либо отодвигая такую работу на более поздние итерации (в случае риска перфекционизма), либо (в случае постоянных изменений) принимая хотя бы часть требований, которые не в полной мере устраивают заказчика, но дают возможность быстрее техническим специалистам при-

ступить к работе, быстрее создать некоторую промежуточную версию, которую можно продемонстрировать заказчику, что, в том числе, может помочь заказчику лучше понять, что ему действительно необходимо.

Риск недостаточной производительности возникает, когда специалисты долгое время задействованы на решении рутинных однотипных задач, что приводит к снижению эффективности работы. Собственно, итогом такой ситуации и является снижение производительности труда специалистов. В общем случае факторов, приводящих к снижению производительности, может быть и больше. Решением такой проблемы может стать переключение специалистов на новые виды задач в рамках текущего проекта, или даже, если имеется возможность и необходимость, переключение специалиста на другой проект.

Риск разрыва в квалификации специалистов может возникать в случае, когда на одном этапе работают специалисты одной квалификации, а на следующем этапе – специалисты с квалификацией, которая существенно ниже квалификации специалистов предыдущего этапа. В таком случае есть риск, что работа на втором этапе будет выполняться более длительное время и с меньшим качеством, так как специалистам на этом этапе понадобится больше времени на то, чтобы понять и осознать результаты предыдущего этапа.

2. Анализ требований

2.1. Термины и определения

Мы уже употребляли ни один раз слово «требование». Но на самом деле вопрос, что же означает «требование к программному обеспечению» не такой очевидный, как кажется на первый взгляд. Само слово «требование» (в английском языке Requirement) подразумевает обязательность его выполнения, носит приказной характер. Но при разработке программного обеспечения часть требований может вообще оказаться по разным причинам не реализована, что, тем не менее, не мешает успешному функционированию проекта. То есть для заказчика часть требований оказывается лишь желательными, а не обязательными. В процессе управления требованиями вводится понятие приоритетности требований. Все эти термины противоречат исходному смыслу слова «требование».

В данном конспекте лекций мы будем использовать следующее определение понятия «требование», данное Иеном Соммервилем и Питом Сойером в 1997-ом году:

Требования – это указание (спецификация) того, что должно быть реализовано. В них описано **поведение** системы, **свойства** системы или ее **атрибуты**. Они могут служить **ограничениями** в процессе разработки системы.

Однако даже такое определение слишком размытое. Как правило при формулировании требований используют ряд терминов для обозначения различных типов требований. Приведём наиболее часто употребляемые из них:

Бизнес-требование – высокоуровневая бизнес-цель организации или заказчика системы.

Бизнес-правило – политика, предписание, стандарт или правило, определяющее или ограничивающее некоторые стороны бизнес-процессов. По своей сути это не требование к программному обеспечению, но оно служит источником нескольких других типов требований к программному обеспечению.

Ограничение – ограничение на выбор вариантов, доступных разработчику при проектировании и разработке продукта.

Требование к внешнему интерфейсу – описание взаимодействия между программным обеспечением и пользователем, другой программной системой или устройством.

Характеристика – одна или несколько логически связанных возможностей системы, которые представляют ценность для пользователя и описаны рядом функциональных требований.

Функциональное требование – описание требуемого поведения системы в определенных условиях.

Нефункциональное требование – описание свойства или особенности, которым должна обладать система, или ограничение, которое должна соблюдать система.

Атрибут качества – вид нефункционального требования, описывающего характеристику сервиса или производительности продукта.

Системное требование – требование верхнего уровня к продукту, состоящему из многих подсистем, которые могут представлять собой программное обеспечение или совокупность программного и аппаратного обеспечения.

Пользовательское требование – задача, которую определенные классы пользователей должны иметь возможность выполнять в системе, или требуемый атрибут продукта.

Описанные термины дают общее понимание о существующих типах требований. Однако важно понимать, как эти требования соотносятся между собой, в каких документах как правило эти требования описываются.

При документировании требований могут создаваться различные виды документов, которые могут дополняться визуальными диаграммами, созданными в соответствии с правилами унифицированного языка моделирования UML (Unified Modeling Language). На постсоветском пространстве стандартом на оформление проектной документации является техническое задание (регламентируется одним из документов ГОСТ 19.201-78, ГОСТ 34.602-89, ГОСТ 25.123-82 и др.). В западноевропейской и американской традиции вместо единого документа оформляются различные документы, которые содержат различные уровни требований, к которым относят:

– бизнес-требование;

– пользовательские требования;

– функциональные требования.

Бизнес-требование описывает общую цель создания информационной системы и глобальные задачи, которые она должна решать. Также очерчиваются границы развития системы, за которые заказчик пока не планирует выходить. Фактически это требование описывает, за что готов платить заказчик, и за что он платить не будет. Описываются в документе концепций и границ (Vision and Scope).

Пользовательские требования описывают те действия, которые могут сделать пользователи, играющие различные ролями системе. Пользовательские требования отвечают на вопрос «***что*** *может сделать пользователь?*». Также могут применяться как способ структуризации функциональных требований. Документируются с помощью UML-диаграмм вариантов использования (Use Case Diagram). Также такие диаграммы называются диаграммами прецедентов. Описываются в документе пользовательских требований (User Requirements).

Функциональные требования подробно описывают, каким образом должны функционировать те или иные части программного обеспечения. Именно в функциональных требованиях описываются все тонкости бизнес-логики приложения. Функциональные требования отвечают на вопрос «***как*** *будет работать система?*». Основной вид требования, которым должны руководствоваться разработчики. Описываются в виде текста и

могут сопровождаться различной графикой. В том числе для иллюстрации функциональных требований активно используется прототипирование. Описываются в документе «Спецификация требований к программному обеспечению» (SRS – System Requirements Specification).

Нефункциональные требования описывают различные технические требования к информационной системе. Сложность выявления таких требований заключается в том, что заказчик, как правило, фокусируется лишь на функциональных требованиях, и может не представлять, какие нефункциональные требования критичны для информационной системы. Также описываются в документе «Спецификация требований к программному обеспечению» (SRS – System Requirements Specification), как правило, в отдельном разделе этого документа.

Примерами нефункциональных требований могут быть требования к дизайну и удобству использования, требования к безопасности и надёжности, требования к производительности и др.

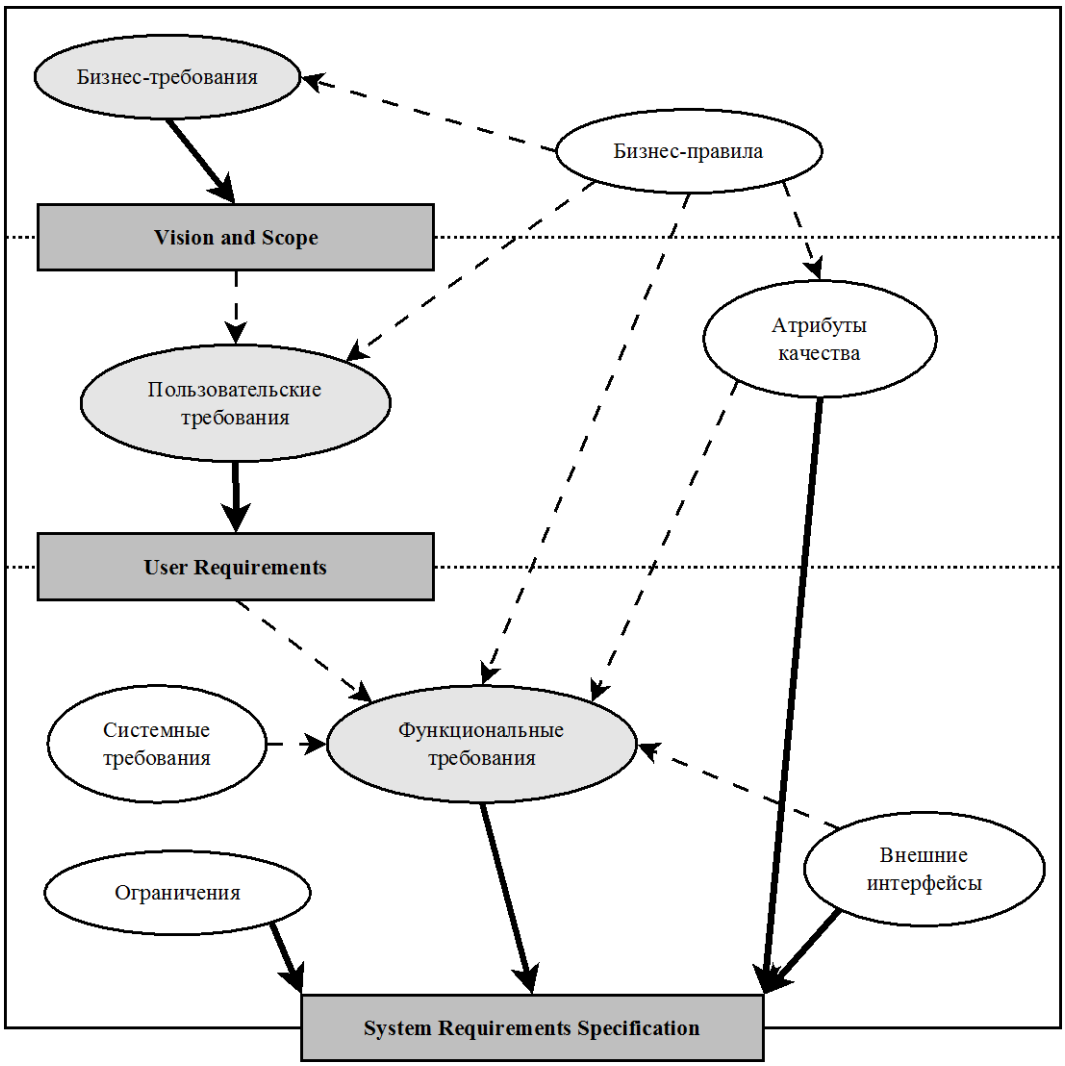

Рисунок 2.1 – Уровни требований и взаимосвязи между требованиями

Сплошные линии показывают, в каких документах содержатся требования
Пунктирные линии показывают, какие требования влияют или являются источниками других требований

2.2. Разработка требований

При работе с требованиями выделяют два основных процесса:

– разработка требований (Requirement Development);

– управление требованиями (Requirement Management).

При этом фактически этап анализа – это и есть этап разработки требований. Управление же требованиями осуществляется после их разработки и осуществляется на протяжении всего жизненного цикла проекта. Как правило за разработку требований отвечают бизнес-аналитики (Business Analyst), а за управление требованиями, в зависимости от уровня этого требования, отвечает менеджер проекта (Project Manager) и главы команд разработчиков и специалистов по тестированию (Team Leaders).

На этапе анализа требований решаются следующие основные задачи:

– сбор (выявление) требований;

– анализ требований;

– документирование требований;

– проверка (валидация) требований.

За решение этих задач отвечают следующие специалисты:

– бизнес-аналитик (Business Analyst – BA) – отвечает за сбор и анализ требований, является экспертом в некоторой предметной области, может не иметь специальных знаний в области информационных технологий и программирования, но имеет опыт формулировки требований понятным для технических специалистов языком;

– дизайнер пользовательских интерфейсов (UI Designer) – занимается проектированием пользовательского интерфейса, является экспертом в области создания эффективных и удобных интерфейсов, может подключаться к работе и на этапе проектирования, или даже на этапе разработки, так как фактически его работа относится к задачам этапа проектирования, но чем раньше начнётся проектирование интерфейса, тем лучше будут согласованы требования, так как визуальный интерфейс легче воспринимается как заказчиком, так и разработчиками.

Примечание: создание интерфейса приложения, позволяющего заказчику протестировать работу с ним и оценить удобство использования, называется ***прототипированием****. Например, прототипом может являться набор свёрстанных HTML-страниц для web-приложения, или настольное или мобильное приложение, содержащее визуальные формы, но не реализующее никакой логики. Также к понятию прототипа можно отнести обычные рисунки в одном из графических форматов, изображающие внешний вид приложения. Прототипами могут служить и схематические рисунки, демонстрирующие лишь концепцию пользовательского интерфейса (например, расположение элементов интерфейса и их состав). В любом случае, способ и объём прототипирования определяются исходя из специфики проекта. Например, для рекламных сайтов определя-*

ющим является именно внешний вид, а не его функциональные возможности. Соответственно, и прототип будет ориентирован, прежде всего, на эффектный внешний вид. А для бизнес-приложения в области бухгалтерского учёта важнее функционал, поэтому на первых этапах пользовательский интерфейс достаточно спроектировать схематически. В настоящее время для создания прототипов сайтов, веб-приложений, мобильных или настольных приложений используется специализированное программное обеспечение, облегчающее и ускоряющее этот процесс.

– технический писатель (Technical Writer) – занимается непосредственным оформлением документов, описывающих требования, следит за соблюдением формальных требований к таким документам.

Основной задачей технического писателя является оформление технической документации на уже разработанное программное обеспечение, изложение её простым, понятным и лаконичным языком. Самым популярным видом работ для технического писателя является создание руководства пользователя. То есть чаще всего технический писатель переводит сложные технические сведения на понятный для конечного пользователя программного продукта язык. Или излагает технические подробности для других разработчиков, которые могут подключиться к проекту позже (так называемый процесс Onboarding), или для программистов, которые будут заниматься сопровождением продукта. Но навыки описания одних и тех же технических сведений на разных языках (один – понятный пользователю, второй – техническому специалисту) позволяет таким специалистам помогать бизнес-аналитикам формулировать требования пользователей для команды технических специалистов. Но последний вид работ технического писателя не столь востребован на проектах в настоящее время (как, впрочем, и вообще в последнее время мало востребована сама отдельная роль технического писателя, которую всё чаще совмещают другие участники проекта).

2.3. Сбор требований

При сборе требований основными путями выявления требований являются следующие.

– Нормативные документы. В случае, если функционал программного обеспечения привязан к бизнес-процессам, которые регламентируются некими законодательными актами или локальными положениями, приказами и регламентами организации-заказчика, в таком случае это основной способ выявления требований, обладающий высокой степенью формализации и документированности.

– Возможности аналогов. Когда разрабатываемое программное обеспечение имеет уже некий реализованный и хорошо известный аналог,

то большую часть функционала можно не описывать в требованиях, а ограничиться лишь требуемыми отличиями и нюансами.

– Ожидания пользователей. С одной стороны – самый естественный способ сбора требований, но и самый трудоёмкий и неоднозначный. Как правило, современное программное обеспечение – это многопользовательская сложная система с распределением различных ролей. В таком случае практически невозможно опросить всех пользователей, которые будут работать с программным обеспечением, и уж тем более учесть все их пожелания. Поэтому такой способ выявления и сбора требований сопряжён с постоянным поиском компромиссов и риском постоянных изменений требований к программному продукту. Однако вообще не учитывать мнение пользователей тоже неправильный подход. Зачастую, бизнес-аналитик должен найти ту золотую середину, которая позволит ему собрать все требования, но при этом не отвлекать потенциальных пользователей на постоянные обсуждения, так как это, всё-таки, не основной их вид деятельности.

При анализе ожиданий пользователей используют различные формы взаимодействия.

Так, например, ***интервью*** может использоваться для беседы с пользователем один на один. Часто применяется для обсуждения узкопрофильных требований, за работу с которыми отвечает конкретный уполномоченный представитель заказчика.

Опросы применяются для того, чтобы выяснить некоторые нюансы требований (обсуждение различных вариантов) среди небольших групп пользователей. Например, для выявления требования к информационной системе, отслеживающей успеваемость студентов некого учреждения высшего образования, нюансы требований для модулей, с которыми будут работать заведующие кафедрой, можно обсудить в форме опроса фокус-группы из нескольких заведующих, выяснив, какие нюансы есть в работе каждого из них. Иногда при проведении такого опроса собирается некоторая встреча с заинтересованными лицами, куда есть возможность пригласить всех пользователей, чьё мнение необходимо получить. Но зачастую собирается некоторая часть таких пользователей. Как правило отбор пользователей происходит на основе опыта этих пользователей в использовании разрабатываемой системы или каких-то её аналогов. Основным критерием включения пользователя в фокус-группу является его возможность (компетентность) и желание предоставить максимум информации по собираемым требованиям, готовность высказывать конструктивную критику и предлагать свои идеи для развития проекта. Как правило, в фокус-группы включают наиболее активных пользователей программного продукта.

Продолжая пример сбора требований для информационной системы сбора и анализа данных об успеваемости учреждения высшего образования, можно отметить, что нюансы требований для модуля, с которым

будут работать непосредственно студенты, можно выяснить с помощью ***анкетирования*** большей части студентов.

Для решения более сложных вопросов, касающихся требований, можно проводить семинары (для больших групп пользователей) или мозговые штурмы (для небольших групп пользователей) совместно с бизнес-аналитиком, дизайнером или другими техническими специалистами со стороны разработчика. Такие формы работы могут характеризоваться отсутствием чёткого понимания, какими должны быть требования для отдельных частей системы, у каждой конкретной группы участников семинара или мозгового штурма (у пользователей, у представителей заказчика, у представителей разработчика). Подобные встречи как раз и проводятся для генерации идей по развитию проекта и выработке требований для программного обеспечения за счёт активного обмена мнениями и идеями, конструктивной критики этих мнений и идей различными участниками мероприятия, а также готовностью участников активно развивать идеи, предложенные остальными участниками.

Также к работе с требованиями могут активно привлекаться технические специалисты со стороны заказчика (если таковые имеются).

2.4. Анализ требований

При собственно анализе требований главной целью является получение более точного и конкретного понимания всех требований, которые были получены из различных источников. Основными действиями, выполняемыми бизнес-аналитиками при этом, являются следующие:

– анализ и структуризация информации, полученной от пользователей, из нормативных документов и при ознакомлении с аналогами с целью выделение отдельных видов информации, таких как информация для функциональных и нефункциональных требований, бизнес-правил, атрибутов качества, системных требований, ограничений и требований к внешним интерфейсам;

– разбиение высокоуровневых требований на части для формулирования более низкоуровневых требований;

– сбор информации из различных видов требований для синтеза функциональных требований;

– приоритезация атрибутов качества;

– классификация требования по различным компонентам, модулям и иным частям разрабатываемого программного обеспечения, определённым в системной архитектуре (обычно выполняется после того, как начались работы на этапе проектирования, как минимум после начала проектирования графического интерфейса пользователя);

– согласование с заказчиками приоритетов в реализации требований;

– выявление пробелов и нехватки требований на определённом уровне на основе имеющихся требований более высокого уровня;

– уточнение границ проекта, т.е. выявление тех требований, которые за эти границы выходят.

Действия, описанные в последнем пункте, приходится часто выполнять по ходу реализации проекта. Дело в том, что при реализации проекта самые разные группы лиц, в частности, пользователи, представители заказчика, бизнес-аналитики, разработчики, и особенно часто специалисты по тестированию могут формулировать предложения по улучшению разрабатываемого программного обеспечения (Improvement Requests – запросы на улучшение). Но предлагаемые улучшения могут выходить за указанные в документе «Vision and Scope» границы, или находится на этих границах. В таком случае совместно с представителем заказчика, имеющем полномочия принимать решения об увеличении бюджета проекта и его сроков реализации, согласуется необходимость перевести запросы на улучшение в требование.

2.5. Документирование требований

Виды документов и роли участников проекта, составляющих документы, содержащие требования, были уже рассмотрены выше.

Здесь лишь кратко подведём итог того, как и кем уже документированные требования могут использоваться на различных этапах проекта в дальнейшем.

Документ с концепциями и границами (Vision and Scope), как уже было сказано выше, будет всегда использоваться бизнес-аналитиком и менеджером проекта для контроля общей направленности проекта и контроля перевода запросов на улучшение в статус требований.

Также бизнес-аналитик постоянно должен отслеживать необходимость внесения изменений в этот документ, если меняются бизнес-правила у заказчика.

Документ с пользовательскими требованиями анализируется бизнес-аналитиком как вспомогательный для контроля перевода запросов на улучшение в статус требований (если запрос на улучшение будет касаться целей одной конкретной группы пользователей). Также бизнес-аналитик будет отслеживать необходимость внесения изменений в пользовательские требования при изменении бизнес-правил и при внесении изменений в документ «Vision and Scope».

Документ System Requirement Specification будет использоваться:

– менеджером проекта для осуществления общего руководства и управления требованиями;

– бизнес-аналитиком для отслеживания необходимости изменения требований при изменении более высокоуровневых требований, а также при изменении бизнес-правил, атрибутов качества, системных требований, требований к внешним интерфейсам (прежде всего, требований к графическому интерфейсу пользователя), ограничений;

– главами команд разработчиков для управления процессом разработки, планирования объёма работ на каждую итерацию, проектирования внутренней архитектуры проекта, архитектуры источников данных, потоков данных и т.д.

– разработчиками для понимания поставленных перед ними задач;

– специалистами по тестированию для планирования процесса тестирования, для проверки самих требований, а также для понимания ожидаемого поведения системы при выполнении тестов;

– специалистами по внедрению системы для понимания и отслеживания выполнения нефункциональных требований, прежде всего требований к надёжности, доступности, производительности и отказоустойчивости.

2.6. Проверка требований

При проверке (Validation – валидация) сформулированных требований рассматриваются различные характеристики требований:

– завершённость (требование полностью, т. е. в полном объёме описано в одном месте);

– последовательность (непротиворечивость с другими требованиями);

– атомарность (требование не может быть разбито на более мелкие требования без потери завершённости);

– актуальность (требование не устарело с течением времени);

– выполнимость (требование может быть реализовано в пределах проекта, т. е. в пределах отведённого бюджета и временных рамок);

– недвусмысленность (формулировка требования не позволяет двоякого прочтения);

– проверяемость (то есть выполнение требование может быть проверено каким-либо из способов, например, непосредственный осмотр работы готового функционала; демонстрация работы в специально созданных условиях; документально подтверждённые результаты тестирования с указанием процента функционала, покрытого тестированием, процента успешно пройденных критичных, средне-критичных и некритичных тестов; а также анализ внутренних характеристик программного обеспечения).

Для пояснения характеристик требований приведём некоторые примеры соблюдения и несоблюдения характеристик (см. таблицу 2.1).

Таблица 2.1 – Примеры требований

Характеристика	**Пример требования, не обладающего характеристикой**	**Пример требования, обладающего характеристикой**
Завершённость	*Требование #123* Список книг отображается в таблице с заголовками: – заглавие – автор – название издательства – год издания – количество страниц	*Требование #123* Список книг отображается в таблице с заголовками: – заглавие – ***фамилия и инициалы автора*** – название издательства – год издания – количество страниц
Последовательность	*Требование #124* Таблица со списком книг по умолчанию должна быть отсортирована по столбцу «жанр»	*Требование #123* Список книг отображается в таблице с заголовками: – ***название жанра*** – заглавие – фамилия и инициалы автора – название издательства – год издания – количество страниц
Атомарность	*Требование #123* Список книг отображается в таблице с заголовками: – название жанра – заглавие – фамилия и инициалы автора – название издательства – год издания – количество страниц ***Таблица позволяет произвести сортировку списка по любому столбцу***	*Требование #123* Список книг отображается в таблице с заголовками: – название жанра – заглавие – фамилия и инициалы автора – название издательства – год издания – количество страниц *Требование #125* ***Таблица со списком книг из требования #123 позволяет произвести сортировку списка по любому столбцу***
Недвусмысленность	*Требование #234* На кнопке «Сохранить» должна отображаться красивая иконка	*Требование #234* На кнопке «Сохранить» должна отображаться ***иконка из файла "/img/save-icon.png"***
	Требование #235 Таблицы с большим количеством строк разбиваются на несколько страниц	*Требование #235* Таблицы с ***количеством строк более 50*** разбиваются на несколько страниц
	Требование #236 В случае длительного ожидания загрузки данных необходимо отобразить индикатор загрузки	*Требование #236* В случае ожидания загрузки данных ***дольше 50 мс*** необходимо отобразить индикатор загрузки

Рассмотрим теперь некоторые дополнительные характеристики требований, важные для процесса планирования сроков и бюджета проекта.

– Важность (приоритетность). Характеристика показывает, насколько критичным для заказчика является выполнение этого требования. Такая характеристика позволяет выбирать для реализации, прежде всего, самые критичные требования, что помогает концентрировать основные силы разработчиков на первоочередных задачах информационной системы.

– Сложность реализации (в человеко-часах). Характеристика позволяет планировать время работы над реализацией информационной системы. Также в сложности реализации может учитываться требуемая квалификация специалиста, который может реализовать требование. Часто сложность трудно оценить для всего требования, так как в будущем для реализации этого требования различным разработчикам будут ставиться различные задачи в рамках этого требования. В таком случае сложность оценивается для каждой задачи отдельно, с учётом квалификации разработчиков.

– Устойчивость (вероятность изменения в будущем). Характеристику трудно измерить некоторым конкретным числом. Как правило, используют одно из значений: высокая устойчивость (требование вряд ли изменится в будущем), средняя устойчивость (изменения возможны), низкая устойчивость (изменения очень вероятны).

3. Проектирование

3.1. Язык моделирования UML

Проектирование – это уже следующий этап жизненного цикла разработки программного обеспечения, формально не относящийся к этапу анализа требований. Однако одним из активно применяемых на этом этапе инструментов является графический унифицированный язык моделирования UML (Unified Modeling Language), который также применяется и для документирования требований, в частности – пользовательских требований. Поэтому рассмотрим кратко этот этап.

На этапе проектирования решаются следующие основные задачи.

– Проектирование модели предметной области. При этом разрабатываются структуры данных, которые будут хранить информацию о предметной области, структурируя её понятным для разработчиков способом. Для объектно-ориентированного проектирования это, прежде все, разработка структуры классов, поля которых будут хранить необходимую информацию, и определение взаимосвязей между этими классами. Также при решении этой задачи проектируется структура постоянного хранилища данных. В подавляющем большинстве случаев для этого используется одна из систем управления базами данных. Чаще всего это некая серверная система управления реляционными базами данных, поддерживающая структурированный язык запросов SQL (Oracle, Microsoft SQL Server, PostgreSQL, MySQL и т. д.). В таком случае под проектированием структуры постоянного хранилища понимается разработка ER-диаграммы базы данных и создание схемы базы данных (создание таблиц, связей, ключей и других ограничений).

– Проектирование аппаратного обеспечения. При решении этой задачи осуществляется выбор компонент вычислительной системы, на базе которой будет разворачиваться разработанное программное обеспечение. Прежде всего, это актуально для распределённых высоконагруженных информационных систем, которые способны одновременного обрабатывать большое число клиентских запросов. Для таких систем необходимо подобрать необходимое серверное оборудование, подходящее под планируемую нагрузку. А также спроектировать компьютерную сеть, объединяющую сервера в единую вычислительную систему.

– Проектирование программного обеспечения. При решении этой задачи проектируется набор модулей и способы их взаимодействия между собой. При использовании объектно-ориентированного проектирования разрабатываются абстрактные классы и интерфейсы, обеспечивающие функционирование программного обеспечения на верхнем уровне абстракций. Эта часть проектирования называется проектированием архитектуры программного обеспечения. В последнее время зачастую проектирование внутренней архитектуры системы сводится к выбору и настройке готового

библиотечного программного каркаса (Framework) для выбранного языка программирования и технологии. Например, Spring Framework для Java или ASP.NET MVC для платформы .NET и языка программирования C#. Также на данном этапе может проектироваться и внутреннее устройство компонентов программного обеспечения, но чаще всего это выполняется уже на этапе разработки. Ещё на данном этапе продолжается (или начинается) проектирование пользовательского интерфейса приложения.

За решение перечисленных задач на этапе проектирования отвечают:

– бизнес-архитектор (Business Architect), который отвечает за проектирование так называемых бизнес-процессов, то есть основной бизнес-логики функционирования информационной системы (данный специалист должен быть, в первую очередь, экспертом в области разработки программного обеспечения, но также и иметь опыт работы с данной предметной областью, именно он отвечает за её моделирование);

– системный архитектор (System Architect), который отвечает за проектирование аппаратного и программного обеспечения (у таких специалистов может быть своя узкая специализация в области построения и администрирования компьютерных сетей, проектирования распределённых приложений, в области объектно-ориентированного проектирования и т.д.);

– администратор баз данных (DataBase Administrator – DBA), специалист в области использования баз данных, в первую очередь реляционных баз данных с поддержкой SQL-запросов (отвечает за проектирование схемы базы данных, её оптимизации для специфики разрабатываемой информационной системы и т.п.);

– инженер по обработке данных (Data Engineer), специалист в области сбора и консолидации данных из нескольких источников, который настраивает потоки обработки данных как из внутренних источников данных, спроектированных администратором баз данных, так и внешних источников данных, таких как облачные хранилища, социальные сети и публичные ресурсы;

– аналитик данных (Data Scientist), специалист в области анализа данных с помощью различных математических методов, в том числе с помощью методов математической статистики, методов искусственного интеллекта и т.д., продумывает и проектирует алгоритмы анализа данных, собранных инженерами по обработке данных, строит математические модели, описывающие предметную область;

– ведущий разработчик (Lead Developer), в основном работает на этапе разработки (программирования), но так как на этапе проектирования, как правило, программное обеспечение не проектируется в полном объёме, проектируются лишь общие модули и интерфейсы их взаимодействия, то подробности проектирования и реализации модулей прорабатываются на следующем этапе, где за такие работы отвечает, прежде всего, ведущий разработчик.

Основным инструментом на этапе проектирования программного обеспечения является унифицированный язык моделирования (Unified

Modeling Language – UML). Этот язык состоит из различных графических диаграмм. Графическая форма языка позволяет быстрее создавать проектные решения и быстро и интуитивно их воспринимать. Рассмотрим наиболее популярные виды UML-диаграмм, применяющиеся при проектировании информационных систем.

3.2. Диаграмма вариантов использования

Диаграмма вариантов использования (Use Case Diagram, или диаграмма прецедентов) отражает отношения между ***актёрами*** и ***вариантами использования*** (прецедентами).

Прецедент – это некоторая возможность моделируемой системы, благодаря которой пользователь может получить конкретный, измеримый и нужный ему результат. Используется для спецификации пользовательского требования к приложению. Показывает, что можно сделать, но не как. Каждый вариант использования представляется на диаграмме в виде овала, в который вписывается, что может сделать пользователь. Примеры вариантов использования приведены на рисунке ниже.

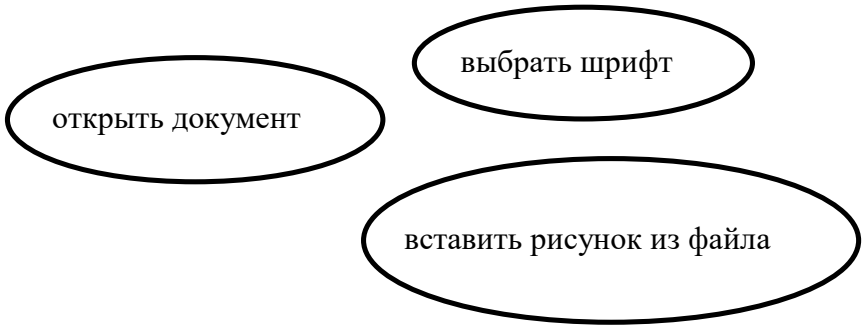

Рисунок 3.1 – Примеры вариантов использования

Актёр – это роль, которую пользователи исполняют во время взаимодействия с вариантами использования. Однако в качестве актёров может выступать не только человек, но также аппаратное устройство, время или другая система. Примеры актёров представлены на рисунке ниже.

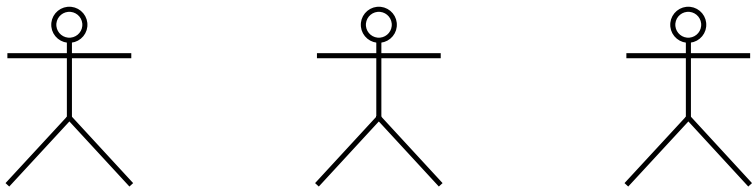

Рисунок 3.2 – Примеры вариантов использования

Кроме самих актёров и вариантов использования на диаграмме используются различные виды связи между объектами диаграммы. Наиболее часто используемая связь, без которой сама диаграмма теряет смысл, это связь между актёром и вариантом использования. Такая связь называется ассоциацией (Association), и она показывает, что актёр может инициировать действие, указанное в варианте использования. Примеры таких ассоциаций показы на рисунке ниже.

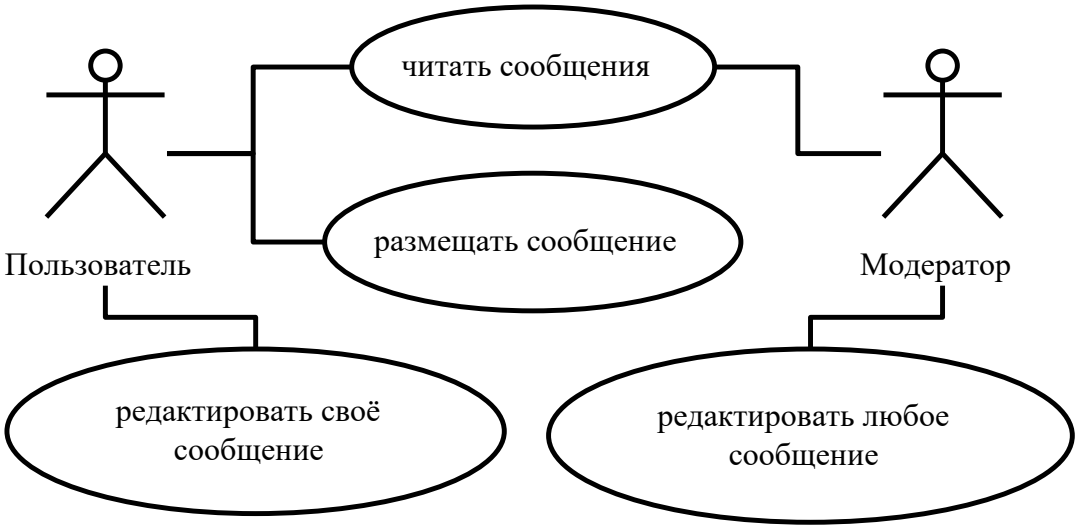

Рисунок 3.3 – Пример ассоциации между актёрами и вариантами использования

Далее рассмотрим связи между актёрами. Такой вид связи называется обобщением (Generalization). Обобщение актёров (наследование) показывает, что одна из ролей является частным случаем другой роли. Фактически это обозначает, что все варианты использования, которые может инициировать более общая роль, доступны и для её наследников (но не наоборот). Пример такой связи приведён на рисунке 3.4 При этом некоторые актёры могут быть абстрактными (выделенные курсивом), что значит, что пользователя конкретно с такой ролью существовать не может.

Между вариантами использования также возможны связи, при этом различных типов:

– обобщение (Generalization);

– зависимость (Dependency), которая подразделяется на включение (Include) и расширение (Extend).

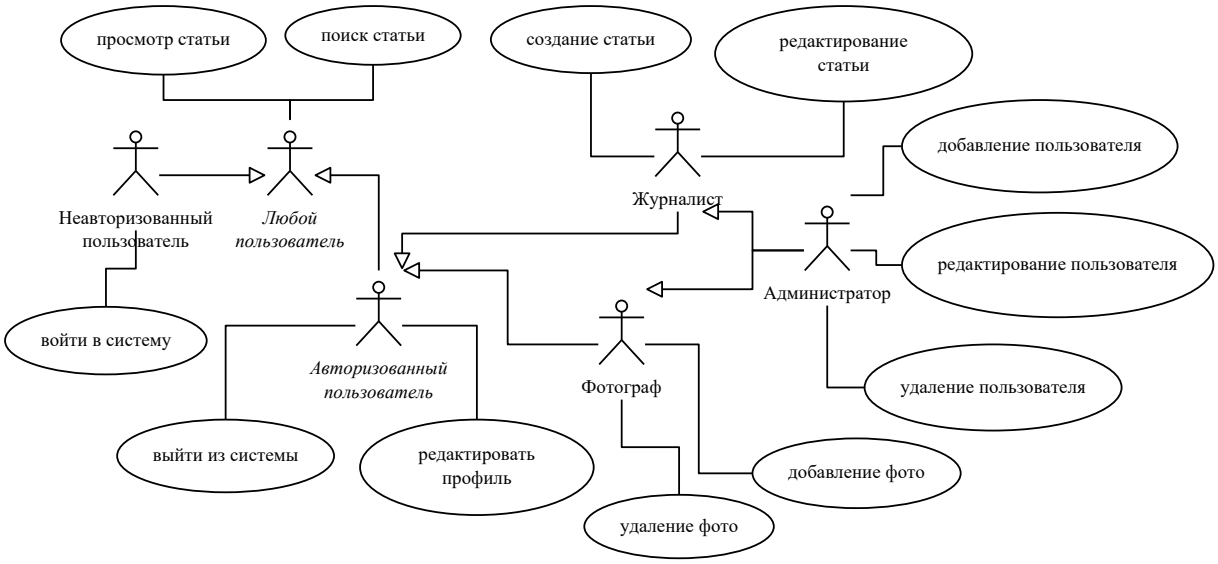

Рисунок 3.4 – Пример обобщения между актёрами

Обобщение (наследование) используется, когда вариант использования, являющийся потомком, наследует поведение варианта использования, являющегося предком, дополняя его или заменяя, сохраняя общий интерфейс взаимодействия.

Обобщение между вариантами используется в том случае, когда наследник варианта использования может использоваться вместо предка. Пример таких вариантов использования смотри на рисунке ниже:

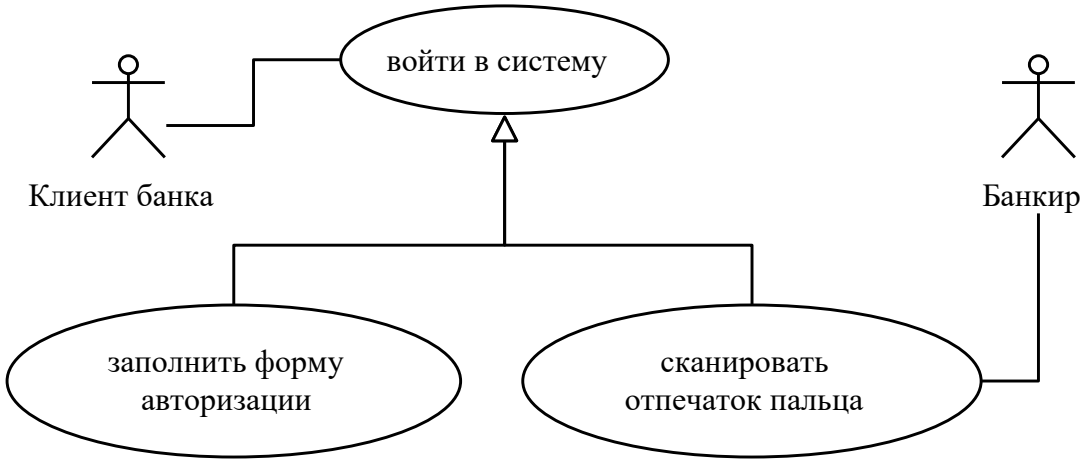

Рисунок 3.5. – Пример обобщения между вариантами использования

Зависимости между вариантами использования, в общем случае, просто показывает некую зависимость. Но такая зависимость показывает непосредственную связь между выполняемыми действиями. Например, рассмотрим случай, когда для удаления некой записи, пользователю сначала необходимо открыть форму редактирования (просмотр записи), на которой он может либо исправить поля и нажать кнопку «сохранить» (редактировать запись), либо нажать кнопку «удалить» (удалить запись). В таком

случае между вариантами использования «просмотр записи» и «удаление записи» зависимости не будет. То есть последовательность действий на диаграмме вариантов использования с помощью зависимостей не отображается, как, впрочем, и никакой другой связью.

Включение вариантов использования показывает, что в некоторой точке базового варианта использования может использоваться функционал включаемого варианта использования. Пример см. на рисунке. Включаемый вариант использования не может существовать (выполняться) отдельно от базового. То есть на представленном примере ассоциация актёра с вариантом использования «просмотр фото» будет некорректным. Как правило, включаемый вариант использования описывает общую для нескольких других вариантов использования функциональность.

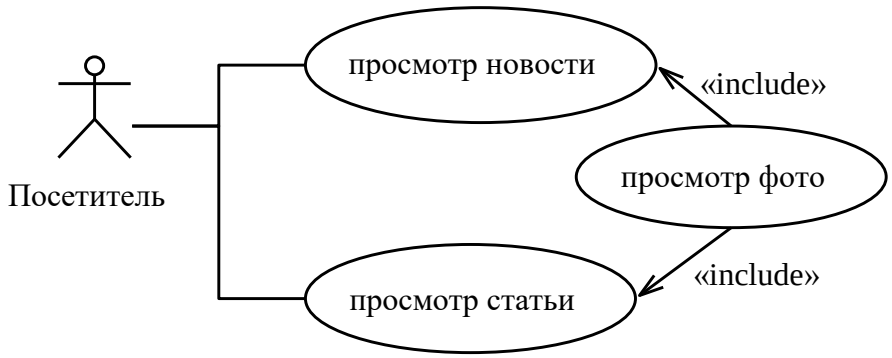

Рисунок 3.6. – Пример включения вариантов использования

Расширение вариантов использования подразумевает, что базовый вариант использования неявно содержит в указанной точке функционал расширяющего. Расширяющий вариант использования передаёт своё поведение базовому (возможно при каком-то условии). В отличие от включения, расширяющий вариант использования имеет самодостаточный смысл и может отдельно выполняться актёром, этим он похож на обобщение. Но, в отличие от обобщения, выполнение актёров базового варианта использования не предполагает, что вместо него может выполняться расширяющий вариант использования (только если он явно ассоциирован с актёром).

3.3. Диаграмма классов

Диаграмма классов предназначена для описания структуры программного обеспечения информационной системы на уровне классов и взаимосвязей их друг с другом. На диаграмме отображаются сами классы, их атрибуты (поля, переменные класса), их операции (методы, функции класса) и отношения (связи) между классами.

Диаграмма классов для всей информационной системы, содержащая все разрабатываемые классы, будет иметь слишком большой размер, учитывая, что некоторые информационные системы могут содержать до не-

скольких тысяч классов. Поэтому диаграмму классов могут создавать для решения одной из следующих задач.

– Общее концептуальное моделирование системы (как правило, на уровне интерфейсов и абстрактных классов верхнего уровня иерархии).

– Подробное моделирование для трансляции модели в программный код (как правило, для реализации конкретного требования или модуля).

– Моделирование предметной области.

Диаграмма классов может иметь статический или аналитический вид. Вначале рассмотрим аналитический вид диаграммы классов. В этом случае все классы маркируются одной из иконок, которая показывает назначение создаваемого класса. Возможные виды классов приведены в таблице 3.2.

Таблица 3.2 – Виды классов на диаграмме классов

Условное обозначение	**Описание**
 граничные классы	Такие классы обеспечивают взаимодействие с внешними относительно разрабатываемой системы факторами. Например, действия пользователя, времени или внешней системы (всё, что может являться актёром в контексте диаграммы вариантов использования). При этом класс может отвечать как за обработку действия (нажатие на кнопку, входящий запрос от внешней системы и т. д.), так и за представление (визуализацию) результата, например отображение формы с компонентом, визуализирующим данные, генерацию HTML-страницы в web-системах и т. д. Фактически, именно такие классы обеспечивают взаимодействие всей информационной системы со внешней средой, поэтому они и называются граничными.
 классы-обработчики	Классы-обработчики – это внутренние классы системы, которые отвечают за реализацию бизнес-логики приложения. К таким классам относятся все классы, отвечающие: за взаимодействие с источником данных; за обработку этих данных в соответствии с алгоритмами, которые необходимо реализовать в информационной системе; за проверку корректности этих данных и т.д. Фактически, все классы, не являющиеся граничными классами или классами-сущностями, являются классами-обработчиками.
 классы-сущности	Классы-сущности – это классы, которые хранят информацию о предметной области. Такие классы не выполняют никакой обработки информации. Хранимая информация содержится в полях класса, а все методы класса лишь предоставляют доступ к этой информации (так называемые методы get-теры и set-теры). При этом методы не должны выполнять даже проверку корректности данных, так как записывать информацию в объект с применением set-теров можно не только при вводе информации пользователем системы, но и при считывании информации из постоянного хранилища. При этом в большинстве информационных систем чтение данных из хранилища (которые уже должны быть корректны), происходит гораздо чаще, чем ввод данных пользователем (которые могут некорректными). Поэтому, с точки зрения производительности, выполнять даже элементарную проверку в классах-сущностях нецелесообразно. Такие классы используются для моделирования предметной области.

Примером диаграммы классов в аналитическом виде может быть диаграмма для некоторого требования, согласно которому в некой банковской системе кассир может перевести деньги с одного банковского счёта на другой (см. рисунок 3.7).

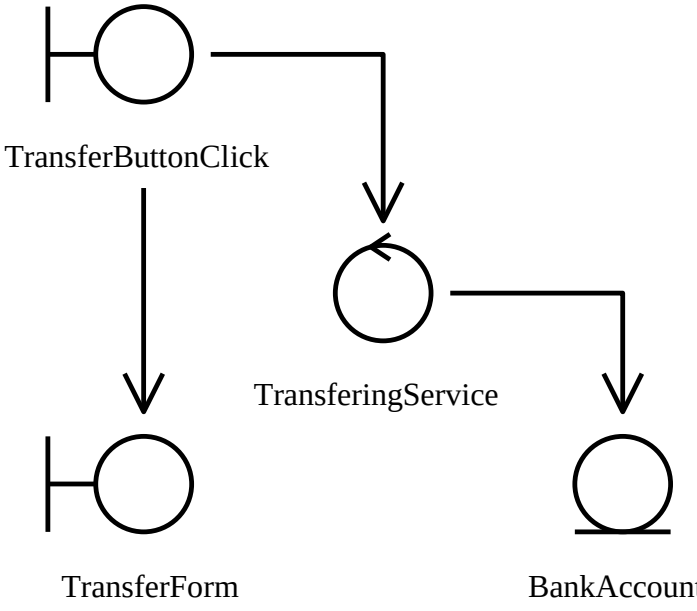

Рисунок 3.7 – Аналитический вид диаграммы классов

В данном примере граничный класс TransferButtonClick обрабатывает событие нажатия на кнопку «Перевести», считывает с полей формы, которую создаёт граничный класс TransferForm, и обращается к классу-обработчику TransferingService, который изменяет состояние объектов класса-сущности BankAccount и сохраняет их в постоянное хранилище.

Рассмотрим теперь диаграмму классов в статическом виде. Класс на диаграмме в таком виде представляется прямоугольником, разделённым на три секции: имя класса, поля класса, методы класса. В качестве примера рассмотрим класс BankAccount (см. рисунок 3.8).

BankAccount
-clientName: String -balance: Integer
+getClientName(): String +setClientName(name:String) +getBalance(): Integer +setBalance(balance:Integer)

Рисунок 3.8 – Класс на диаграмме классов статического вида

При отображении класса в статическом виде используются различные соглашения для отображения различных тонкостей, касающихся класса.

Так, например, области видимости помечаются значками возле описания поля и метода (знак «+» для public-членов, знак «#» для protected-членов, знак «–» для private-членов). Курсивом набираются имена абстрактных классов и интерфейсов, а также абстрактных методов. С помощью подчёркивания отображаются статические члена класса.

На диаграмме классов используются различные отношения между классами. Рассмотрим эти отношения.

Обобщение (наследование или расширение, Extend) – показывает, что один класс наследуется от другого класса. Отображается сплошной линией, заканчивающейся треугольной стрелкой с белой заливкой.

Реализация (Implement) – показывает, что класс реализует некоторый интерфейс. Отображается штриховой линией, заканчивающейся треугольной стрелкой с белой заливкой.

Ассоциация (Association) – признаком ассоциации является наличие в классе ссылки на объект другого класса. Отображается сплошной линией с обычной стрелкой.

Агрегация (Aggregation) – подвид ассоциации, моделирует связь «часть-целое». Отображается, как и ассоциация, но в начале линии ставится ромб с белой заливкой.

Композиция (Composition) – подвид агрегации, имеет привязку к времени жизни объектов. Дочерние объекты не могут существовать после уничтожения контейнера. Отображается, как и агрегация, но ромб в начале линии отображается с чёрной заливкой.

Зависимость (Dependency) – связь (всегда направленная), которая возникает в случае, если в классе есть метод, принимающий параметр-ссылку на объект другого класса, или если в классе есть метод, возвращающий ссылку на объект другого класса. Однако может существовать и неявно при создании и использовании объекта внутри некоторого метода класса. Отображается штриховой линией с обычной стрелкой.

4. РАЗРАБОТКА И ТЕСТИРОВАНИЕ

На этапе разработки производится детальное проектирование каждого модуля и его реализация с применением выбранных языков программирования, библиотек и технологий.

При разработке за реализацию информационной системы отвечают разработчики (developer) и ведущие разработчики (lead developer). Ведущие разработчики выполняются детальное проектирование модулей, участвуют в их реализации и координируют действия обычных разработчиков. Обычные разработчики занимаются непосредственной реализацией программных модулей, настройкой используемого стороннего программного обеспечения и библиотек.

Для управления процессом разработки используется список заданий (Task), за каждый из которых отвечает определённый разработчик. При составлении списка заданий, как правило, каждое задание обладает следующими характеристиками:

– описание (что нужно сделать, обычно указывается также пользовательское или функциональное требование, в рамках которого формулируется данное задание);

– планируемое время выполнение (указывается количество часов, за которое разработчик должен успеть выполнить это задание, при этом, как правило, если разработчик в это время не укладывается, так называемый overtime, то оплачивается разработчику только то время, которое было запланировано);

– статус (например, новое, принято к исполнению, завершено, отклонено, возобновлено; при этом для всех статусов, кроме новых, сохраняется информация о том, какой разработчик ответственен за дельнейшее выполнение задания);

– дата, когда задание было переведено в текущий статус (как правило, сохраняется вся история статусов заданий, чтобы можно было в течение некоторого периода отследить прогресс работы над программным обеспечением);

– предусловия (задания, которые должны быть выполнены до того, как разработчик сможет приступить к выполнению данного задания).

На этапе тестирования решаются следующие задачи:

– планирование процесса тестирования (на какой итерации и в каком объёме необходимо проводить тестирование);

– составление тестов (какой функционал подвергать тестированию и какие тесты выполнять для каждого из тестируемых функциональных требований);

– выполнение тестов (основное, что необходимо сделать специалисту по тестированию при выполнении тестов, это правильно оформить отчёт об ошибках, так называемый Bug Report, в котором обязательно указать,

какие данные вводились и, вообще, какие действия предпринимались, какой результат был достигнут, и какой результат должен был быть получен согласно функциональному требованию);

На этапе тестирования основными специалистами, отвечающими за выполнение работ, являются специалист по тестированию (Tester) и специалист по обеспечению качества (Quality Assurance Engineer – QA Engineer). Специалист по тестированию обеспечивает выполнение тестов и составление отчёта об ошибках. Специалист по обеспечению качества планирует процесс тестирования, составляет тесты и подготавливает итоговые отчёты о результатах итерации: сколько тестов было произведено, какой процент функционала был покрыт тестами (и какого рода тестами), каков общий результат проведённых тестов (сколько процентов тестов успешно пройдено, сколько не пройдено по каждой категории требований: критичных, средне-критичных и некритичных), также обязательно выполняется сравнение с результатами тестирования предыдущих этапов, на сколько улучшился функционал, сколько ошибок было исправлено, что позволяет менеджеру проекта оценить различные риски.

Основное отличие в работе этих двух специалистов заключается в их целеполагании. Основная цель специалиста по тестированию – найти как можно больше дефектов в программном обеспечении. Такая работа лишь декларирует, что наша система имеет некоторые проблемы, но мало способствует их устранению. Кроме того, такое целеполагание противопоставляет разработчика и специалиста по тестированию, ставит их в положение конкурентов, что мало способствует достижению целей проекта. Специалист по качеству же основной целью ставит обеспечение качества. Для него сам процесс тестирования не является самоцелью, а лишь средством выявления дефектов. А уже в дальнейшем его задача найти способы улучшения программного продукта, проанализировать и понять основные причины возникновения дефектов и помочь разработчикам устранить саму первопричину, а не просто исправить сами найденные дефекты.

СПИСОК ЛИТЕРАТУРЫ

1. Вигерс, К. Разработка требований к программному обеспечению / К. Вигерс, Дж. Битти. – Москва: Русская редакция, 2014. – 736 с.
2. Брауде, Э. Дж. Технология разработки программного обеспеченич. – Санкт-Петербург: Питер, 2004. – 655 с.
3. Коберн, А. Современные методы описания функциональных требований к системам. – Москва: Лори, 202. – 286 с.
4. IEEE Recommended Practice for Software Requirements Specifications – New York: The Institute of Electrical and Electronics Engineers, Inc., 1998. – 38 p.
5. Фаулер, М. Шаблоны корпоративных приложений / М. Фаулер и др. – Москва: Вильямс, 2010. – 544 с.
6. Фаулер, М. Архитектура корпоративных программных приложений / М. Фаулер и др. – Москва: Вильямс, 2007. – 544 с.
7. Шмуллер, Д. Освой самостоятельно UML за 24 часа / Д. Шмуллер. – Москва: Вильямс, 2005. – 405 с.
8. Буч, Г. Язык UML. Руководство пользователя / Г. Буч, Д. Рамбо, А. Джекобсон. – 2-е издание. – Санкт-Петербург: ДМК Пресс, 2004. – 432 с.
9. Фаулер, М. UML. Основы / М. Фаулер, К. Скотт. – Санкт-Петербург: Символ-Плюс, 2002. – 192 с.
10. Скотт, К. UML. Основные концепции / К. Скотт. – Москва: Вильямс, 2002. – 138 с.

Учебное издание

ЕРМОЧЕНКО Сергей Александрович

ОСНОВЫ БИЗНЕС-АНАЛИЗА

Курс лекций

Технический редактор *Г.В. Разбоева*

Компьютерный дизайн *А.В. Табанюхова*

Подписано в печать 2022. Формат 60x84 $^{1}/_{16}$. Бумага офсетная.

Усл. печ. л. 2,09. Уч.-изд. л. 1,66. Тираж экз. Заказ .

Издатель и полиграфическое исполнение – учреждение образования
«Витебский государственный университет имени П.М. Машерова».

Свидетельство о государственной регистрации в качестве издателя,
изготовителя, распространителя печатных изданий
№ 1/255 от 31.03.2014.

Отпечатано на ризографе учреждения образования
«Витебский государственный университет имени П.М. Машерова».
210038, г. Витебск, Московский проспект, 33.